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] 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 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:

<table>
<thead>
<tr>
<th>Interface</th>
<th>Clause</th>
<th>Requirements given in subclause</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device interface</td>
<td>5</td>
<td>5.1.2</td>
</tr>
<tr>
<td>Client interface</td>
<td>6</td>
<td>6.1.2</td>
</tr>
<tr>
<td>User interface</td>
<td>7</td>
<td>7.1.2</td>
</tr>
</tbody>
</table>

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


IEEE P1275.3, Standard for Boot (Initialization Configuration) Firmware—Supplement for VMEbus, D1, August 1994. 4


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 Varembo, 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 sequency 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:

\[
\text{driver-name@unit-address:device-arguments}
\]

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 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”
“reg”
“device_type”
“interrupts”
“model”
“address”
“compatible”

“status”

Standard property name to define the name of the package.
Standard property name to define the package’s registers.
Standard property name to specify the implemented interface.
Standard property name to define the interrupts used.
Standard property name to define a manufacturer’s model number.
Standard property name to define large virtual address region(s).
Standard property name to define alternate “name” property values.
Standard property name 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.

<table>
<thead>
<tr>
<th>Property name</th>
<th>Encoding</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>string</td>
<td>Name of system’s manufacturer and model number, e.g., “ABC,mat750”.</td>
</tr>
</tbody>
</table>

/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’.

<table>
<thead>
<tr>
<th>Property name</th>
<th>Encoding</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>string</td>
<td>“aliases”</td>
</tr>
</tbody>
</table>

/openprom

Describes the Open Firmware.

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

<table>
<thead>
<tr>
<th>Property name</th>
<th>Encoding</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>string</td>
<td>“openprom”</td>
</tr>
<tr>
<td>model</td>
<td>string</td>
<td>Describes manufacturer and revision level of firmware. See model property.</td>
</tr>
<tr>
<td>relative-addressing</td>
<td>(none)</td>
<td>Presence of this property indicates that the Open Firmware supports bus-relative physical addressing (early precursors to Open Firmware used a different addressing scheme).</td>
</tr>
</tbody>
</table>

/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.

<table>
<thead>
<tr>
<th>Property name</th>
<th>Encoding</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>string</td>
<td>“options”</td>
</tr>
</tbody>
</table>
/chosen

Has properties describing parameters chosen or specified at runtime.

<table>
<thead>
<tr>
<th>Property name</th>
<th>Encoding</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>string</td>
<td>&quot;chosen&quot;</td>
</tr>
<tr>
<td>stdin</td>
<td>int</td>
<td>Ihandle for the console input device. May be used with the read client interface function to read characters from that device.</td>
</tr>
<tr>
<td>stdout</td>
<td>int</td>
<td>Ihandle for the console output device. May be used with the write client interface function to write characters to that device.</td>
</tr>
<tr>
<td>bootpath</td>
<td>string</td>
<td>The device path 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.</td>
</tr>
<tr>
<td>bootargs</td>
<td>string</td>
<td>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).</td>
</tr>
<tr>
<td>memory</td>
<td>int</td>
<td>Ihandle for the package that describes the allocation status of physical memory. See 3.7.6.</td>
</tr>
<tr>
<td>mmu</td>
<td>int</td>
<td>Ihandle 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.</td>
</tr>
</tbody>
</table>

/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.

<table>
<thead>
<tr>
<th>Property name</th>
<th>Encoding</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>string</td>
<td>&quot;packages&quot;</td>
</tr>
</tbody>
</table>

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, device 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.

### 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.

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.

---

<table>
<thead>
<tr>
<th>Method</th>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>open</td>
<td>( -- okay? )</td>
<td>Prepare this device for subsequent use.</td>
</tr>
<tr>
<td>close</td>
<td>( -- )</td>
<td>Close this previously opened device.</td>
</tr>
<tr>
<td>decode-unit</td>
<td>( addr len -- phys.lo ... phys.hi )</td>
<td>Convert text unit-string to physical address.</td>
</tr>
<tr>
<td>encode-unit</td>
<td>( phys.lo ... phys.hi -- unit-str unit-len )</td>
<td>Convert physical address to text unit-string.</td>
</tr>
</tbody>
</table>

### 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.

### 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.
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 property name to define a device’s physical address.
"#address-cells"  Standard property name to define the package’s address format.
"#size-cells"  Standard property name 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** \( \{ \text{virt} \} \) size align -- base \) Allocate (claim) addressable resource
- **release** \( \text{virt size --} \) Free (release) addressable resource.
- **map** \( \text{phys.lo ... phys.hi virt size mode --} \) Create address translation
- **unmap** \( \text{virt size --} \) Invalidate existing address translation.
- **modify** \( \text{virt size mode --} \) Modify existing address translation.
- **translate** \( \text{virt -- false | phys.lo ... phys.hi mode true} \) Translate virtual address to physical address.

Additional requirements for the **claim** and **release** methods:
- The address format, \text{virt}, is a single-cell virtual address.
- The allocation length, \text{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 \text{virt} and \text{len} 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.

<table>
<thead>
<tr>
<th>“<strong>device_type</strong>” value</th>
<th>Example</th>
<th>Typical use</th>
</tr>
</thead>
<tbody>
<tr>
<td>“display”</td>
<td>bit-mapped frame-buffer</td>
<td>console output</td>
</tr>
<tr>
<td>“block”</td>
<td>hard disk</td>
<td>booting</td>
</tr>
<tr>
<td>“byte”</td>
<td>tape</td>
<td>booting</td>
</tr>
<tr>
<td>“network”</td>
<td>Ethernet interface</td>
<td>booting</td>
</tr>
<tr>
<td>“serial”</td>
<td>asynchronous serial line</td>
<td>console input and output</td>
</tr>
</tbody>
</table>
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 property name 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 “deblocker” 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.
IEEE Std 1275-1994
IEEE STANDARD FOR BOOT (INITIALIZATION CONFIGURATION) FIRMWARE:

read
( addr len -- actual )
Read device into memory buffer, return actual byte count.

(write)
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 “debblocker” 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. 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 property name to specify preassigned network address.

“mac-address” Standard property name to specify network address last used.

“address-bits” Standard property name to indicate network address length.

“max-frame-size” Standard property name 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:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>open</td>
<td>Prepare this device for subsequent use.</td>
</tr>
<tr>
<td>close</td>
<td>Close this previously opened device.</td>
</tr>
<tr>
<td>read</td>
<td>Read device into memory buffer; return actual byte count.</td>
</tr>
<tr>
<td>write</td>
<td>Write memory buffer to device; return actual byte count.</td>
</tr>
<tr>
<td>install-abort</td>
<td>Begin polling for a console abort sequence.</td>
</tr>
<tr>
<td>remove-abort</td>
<td>Cease polling for a console abort sequence.</td>
</tr>
<tr>
<td>restore</td>
<td>Restore device to usable state after unexpected reset.</td>
</tr>
<tr>
<td>ring-bell</td>
<td>Ring the bell.</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>claim</td>
<td>Allocate (claim) addressable resource.</td>
</tr>
<tr>
<td>release</td>
<td>Free (release) addressable resource.</td>
</tr>
</tbody>
</table>

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

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

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:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>open</td>
<td>Prepare this device for subsequent use.</td>
</tr>
<tr>
<td>close</td>
<td>Close this previously opened device.</td>
</tr>
<tr>
<td>load</td>
<td>Load a client program from device to memory.</td>
</tr>
<tr>
<td>offset</td>
<td>Convert partition-relative disk position to absolute position.</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>open</td>
<td>Prepare this device for subsequent use.</td>
</tr>
<tr>
<td>close</td>
<td>Close this previously opened device.</td>
</tr>
<tr>
<td>load</td>
<td>Load a client program from device to memory.</td>
</tr>
</tbody>
</table>

3.8.3 “deblocker” support package

The “deblocker” 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 “deblocker” support package defines the following methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>open</td>
<td>Prepare this device for subsequent use.</td>
</tr>
<tr>
<td>close</td>
<td>Close this previously opened device.</td>
</tr>
<tr>
<td>read</td>
<td>Read device into memory buffer; return actual byte count.</td>
</tr>
<tr>
<td>write</td>
<td>Write memory buffer to device; return actual byte count.</td>
</tr>
<tr>
<td>seek</td>
<td>Set device position for next read or write.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>block-size</td>
<td>Return “granularity” for accesses to this device.</td>
</tr>
<tr>
<td>max-transfer</td>
<td>Return size of largest possible transfer.</td>
</tr>
<tr>
<td>read-blocks</td>
<td>Read #blocks, starting at block#, from device into memory.</td>
</tr>
<tr>
<td>write-blocks</td>
<td>Write #blocks from memory into device, starting at block#.</td>
</tr>
</tbody>
</table>

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`.

```forth
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.

```forth
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.

```forth
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 ( n -- ) Insert n spaces to the right of the cursor.
delete-characters ( n -- ) Delete n characters to the right of the cursor.
insert-lines ( n -- ) Insert n blank lines at and below the cursor line.
delete-lines ( n -- ) Delete n 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 height of the display in pixels.
screen-width ( -- width ) Return total width 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 height of a font character in pixels.
char-width ( -- width ) Return the width 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:

left-split(string, “x”) -> initial, 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.

right-split(string, “x”) -> initial, 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(COMPARTMENT, ":") -> 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.

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

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.
5. Device interface

The *device interface* allows Open Firmware to identify and use *plug-in devices*. The interface is based on a byte-coded programming language known as *FCode*. The FCode language is evaluated by a Open Firmware component known as the *FCode evaluator*.

5.1 General

The Open Firmware *device interface* specifies the behavior of a *firmware* system so that, when compliant *devices* are added to a computer system whose firmware is compliant, the firmware may determine the characteristics of those devices and may use them for various purposes, such as text display and program *loading*.

5.1.1 Description

A standard *FCode evaluator* provides a defined environment for the execution of standard *FCode programs*. A standard FCode evaluator is typically a component of the boot *firmware* associated with a CPU board.

A standard FCode program is a program written in the *FCode* language (defined herein) that obeys prescribed rules for program structure and usage. Consequently, its behavior is predictable when executed by a standard FCode evaluator. A standard FCode program is typically resident on a *plug-in device*.

A common use of a standard FCode program is to implement a standard *package* that is relevant to the kind of device with which the FCode program is associated.

5.1.1.1 FCode basics

*FCode* is a way of representing a program in the Forth programming language by using machine-independent byte codes to represent a set of standard *Forth words*. FCode uses a dialect of Forth that is based on ANSI X3.215-1994 (however, FCode is not a Standard System as defined by ANSI X3.215-1994), with extensions appropriate for *firmware* requirements. An *FCode program* is a representation of a computer program in the FCode language.

FCode programs are processed by a software component known as an *FCode evaluator*. An FCode evaluator reads a sequence of bytes (the FCode program), performing a specified action for each byte.

Typically, an FCode program resides in a ROM attached to a *plug-in device*. The FCode program serves to identify and to provide a *firmware device driver* for that device. The FCode evaluator is typically a part of the firmware associated with a CPU board.

The means for invoking the FCode evaluator and for locating the FCode corresponding to particular devices depends on the set of buses and features supported by a particular Firmware implementation. Those means are described in machine-specific Open Firmware documents (see 2.1) and in clause 7.

Forth is a stack-based programming language with postfix syntax. *Forth source* code may be either interpreted “on-the-fly” or incrementally compiled for later execution. FCode is semantically similar to Forth source code, but the lexical tokens of Forth are space-delimited text strings, whereas the lexical tokens of FCode are binary bytes.

The basic action of a Forth *command interpreter* is to repeat the following sequence:

a) Collect a space-delimited string from the input buffer.
b) Find the corresponding name in a symbol table.
c) Either execute or compile the associated function.
An FCode evaluator replaces items a) and b) with “Read a byte” and “Use that byte as an index into an array,” respectively. The same executable functions that are associated with textual Forth words are associated with FCode numbers.

FCode programs are created from textual Forth source code by a program called a tokenizer. A tokenizer reads a sequence of textual Forth words and writes the corresponding sequence of FCode bytes. The specification for a preferred form of source code for generating FCode programs, and for the behavior of a tokenizer to process that form of source code, is given in annex C. The mapping from textual Forth words to FCode bytes is nearly one-to-one, and the preferred source format is very similar to a standard Forth program.

Since FCode functions are semantically identical to Forth words, the FCode execution environment is that of the Forth programming language. Forth words are passed input arguments and provide output results via a LIFO data stack. Each stack element is an integer. The maximum stack depth is implementation-dependent, but must be at least 64 elements.

5.1.1.2 Notation

This clause lists FCode functions using a short-form notation. The complete descriptions are given in annex A.

Each of the short-form descriptions in this clause gives the name of the FCode function (usually the same as the name of the corresponding Forth word), a stack diagram, the FCode number, and a brief description. The following is an example:

<table>
<thead>
<tr>
<th>Name</th>
<th>Stack diagram</th>
<th>FCode number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dup</td>
<td>( x -- x x )</td>
<td>0x47</td>
<td>Duplicate the top item on the stack.</td>
</tr>
</tbody>
</table>

A stack diagram documents the arguments that an FCode function removes from the top of the data stack and the results that the function places on the top of the data stack, as follows:

( argument1 argument2 ... -- result1 result2 ... )

The right-most item in each list represents the top-most item on the data stack. FCode functions that do not affect the stack are shown with the stack diagram:

( -- )

Some FCode functions, when evaluated, read one or more bytes from the FCode program. The description field for such functions takes the following form:

(F: n1 /FCode# name string/ -- n2 )

In this example, FCode# and name string represent bytes that are read from the FCode program when the FCode being described is first encountered. (n1 and n2 represent the stack effect at that time, if any.)

5.1.2 Specification

In order to be compliant with the Open Firmware device interface:
— The boot firmware associated with a main CPU device shall implement a standard FCode evaluator, the /packages standard system node, and the complete set of standard support packages. It should implement any additional standard packages that are relevant to the system environment. It may implement additional packages that are not defined in this specification. Packages in the device tree in the path from the root of the device tree to any package that can be opened with open-dev shall conform to the rules given in 3.6.1.
— A plug-in device shall have a standard FCode program packaged according to the rules for the particular expansion bus with which that device is used. That standard FCode program shall identify the device with at least a “name” property. It shall comply to any additional requirements imposed by the specification describing the application of Open Firmware to the expansion bus in question. It should implement any standard packages that are relevant to the particular device.

5.2 FCode evaluator

A standard FCode evaluator shall behave as described in 5.2.1 in processing the byte codes associated with an FCode program, and shall implement the set of FCode functions as described in 5.3.

5.2.1 FCode evaluation sequence

Once invoked, the FCode evaluator shall set the internal state variable fcode-end to false and shall repeat the following sequence of operations until fcode-end is true at the beginning of the sequence:

a) Read the next FCode#, denoting an FCode number, from the current FCode program.
b) Evaluate the FCode function associated with that FCode number.

The details of the reading process in the first step are bus-dependent and are specified in related documents describing the application of this standard to particular buses. An FCode# consists of either 1 or 2 bytes, as specified in 5.2.2.

The FCode evaluator has two states, “interpretation state” and “compilation state”, determining the way that it evaluates a particular FCode function. The execution of certain FCode functions causes transitions between these two states.

The details of step b) are as follows:

1) If the FCode function has explicit “FCode evaluation” semantics, perform the FCode function's “FCode evaluation” semantics.
2) Otherwise,
   i) If in interpretation state, perform the FCode function's execution semantics.
   ii) Otherwise (i.e., in compilation state), append the FCode function's execution semantics to the current definition.

Subclause 5.3 defines the association between particular FCode numbers and their corresponding FCode functions.

NOTE—The behavior for some FCode functions includes reading FCode bytes. Thus, some of the bytes in an FCode program are not read directly by the FCode evaluator but by those FCode functions instead.

5.2.2 Encodings of in-line data

The following data formats are used to encode FCode programs. At the top level, an FCode program consists of a sequence of FCode#. Certain individual FCode functions are followed by additional bytes in the sequence of bytes representing the FCode program. Those functions are recognized during the FCode evaluation process, and the bytes that follow are read from the FCode program and used as arguments to control the interpretation or compilation of the associated function. The encoding of such following bytes are described below.

In the following descriptions, the left-most byte in a printed sequence corresponds to the byte that appears first (either chronologically earlier or at a lower memory address, whichever is applicable) in the sequence of bytes constituting the FCode program, and so on from left to right. For binary values that are represented by more than one byte, bytes of greater significance precede those of lesser significance in the FCode program (i.e., big-endian byte ordering).
5.2.2.1 FCode#

Either

- byte (0x00 or 0x10 .. 0xFF) Encodes an FCode number less than 0x100.

or

- byte (0x01.. 0x0f) byte (0x00 .. 0xFF) Encodes an FCode number greater than or equal to 0x100.

5.2.2.2 FCode-offset

Either

- byte Encodes an 8-bit signed (two's complement) offset.

or

- byte.high byte.low Encodes a 16-bit signed (two's complement) offset.

A conditional or looping control transfer is represented by a pair of FCode functions. An FCode-offset specifies the number of bytes in the FCode program between two corresponding components of a control flow construct. The offset is calculated as the number of FCode bytes from the first byte of the offset to the byte just after the “target” of the control transfer. A positive offset corresponds to a transfer of control in the “forward” (towards the end of the FCode program) direction, and a negative offset corresponds to the “backward” (towards the beginning of the FCode program) direction.

The following control transfer pairs are meaningful, with “…” representing an arbitrary sequence of FCode bytes:

<table>
<thead>
<tr>
<th>FCode control transfer pair</th>
<th>Example source construct</th>
</tr>
</thead>
<tbody>
<tr>
<td>b(&lt;mark) ... bbranch FCode-offset</td>
<td>begin ... again</td>
</tr>
<tr>
<td>b(&lt;mark) ... b?branch FCode-offset</td>
<td>begin ... until</td>
</tr>
<tr>
<td></td>
<td>B</td>
</tr>
<tr>
<td>bbranch FCode-offset ... b(resolve)</td>
<td>... else ... then</td>
</tr>
<tr>
<td>b?branch FCode-offset ... b(resolve)</td>
<td>if ... then</td>
</tr>
<tr>
<td></td>
<td>T</td>
</tr>
<tr>
<td>b(do) FCode-offset1 ... b(loop) FCode-offset2</td>
<td>do ... loop</td>
</tr>
<tr>
<td>b(do) FCode-offset1 ... b(+loop) FCode-offset2</td>
<td>do ... +loop</td>
</tr>
<tr>
<td>b(?do) FCode-offset1 ... b(loop) FCode-offset2</td>
<td>?do ... loop</td>
</tr>
<tr>
<td>b(?do) FCode-offset1 ... b(+loop) FCode-offset2</td>
<td>?do ... +loop</td>
</tr>
</tbody>
</table>

The markers B and T show the “branch” and “target” locations used for the calculation of the value of FCode-offset. The value is the signed number of FCode bytes between B and T (positive if B is before T). B1/T1 are for FCode-offset1 and B2/T2 are for FCode-offset2.

NOTE—On some devices, FCode programs are stored with “gaps” between successive FCode bytes. For example, each FCode byte might be stored in the least significant byte of a separate quadlet, in which case it might be necessary to add four to the address to advance to the next FCode byte. This does not affect the calculation of an FCode-offset—the offset is in terms of the number of FCode bytes, independent of how those bytes are addressed.
The offset size (whether of 8 bits or 16 bits) is established at the beginning of the FCode program by the particular start code that begins the FCode program. version1 sets the offset size to 8 bits, and the other start codes (start0, start1, start2, and start4) set the offset size to 16 bits. The offset size may be changed from 8 bits to 16 bits by executing offset16.

In most cases (the exceptions are bbranch and b?branch in interpretation state), the FCode evaluator needs only the sign of the offset, not its numerical value. In these cases, the value of the offset is essentially redundant because control transfers are represented by pairs of FCode functions (a branching function and its target). The offset indicates the distance between the branch and its target, but that information can be derived during the FCode evaluation process without needing the offset value. However, standard FCode programs are required to have numerically correct offsets (as described in the above paragraph) for compatibility with existing practice.

### 5.2.2.3 FCode-num32

```
byte.high byte.highmid byte.lowmid byte.low
```

Encodes a 32-bit integer

### 5.2.2.4 FCode-string

```
byte.count byte.string1 ... byte.stringn
```

Encodes a text string. The first byte is the length of the string (0 to 255 bytes), not including the count byte. Subsequent bytes are the bytes of the string.

### 5.2.2.5 FCode-header

The FCode-header data type appears only at the beginning of an FCode program following one of the functions version1, start0, start1, start2, or start4. It contains information about the FCode program as a whole. That information is provided for the benefit of external software that may wish to characterize the FCode program. A standard FCode evaluator is permitted to skip and ignore the FCode-header information, or to use it to verify, in an implementation-dependent manner, that the FCode program is intact.

<table>
<thead>
<tr>
<th>Byte</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>format</td>
<td>The value 0x08 in this field indicates that this FCode program is intended to operate with boot firmware that complies with the device interface portion of this standard. The values 0x09 through 0xFF are reserved for future revisions of this standard. Values 0x00 through 0x07 indicate that this FCode program is intended to operate with boot firmware that does not comply with this standard.</td>
</tr>
<tr>
<td>2</td>
<td>checksum-high</td>
<td>High byte of the body checksum. Checksum is the doublet size sum of the bytes of the program body (i.e., excluding the header), calculated using two's complement addition and ignoring overflow.</td>
</tr>
<tr>
<td>3</td>
<td>checksum-low</td>
<td>Low byte of the body checksum.</td>
</tr>
<tr>
<td>4</td>
<td>length-high</td>
<td>Most significant byte of the program length. Program length is the quadlet size number of bytes in the program, including both the body and the header.</td>
</tr>
<tr>
<td>5</td>
<td>length-high-middle</td>
<td>High middle byte of the program length.</td>
</tr>
<tr>
<td>6</td>
<td>length-low-middle</td>
<td>Low middle byte of the program length.</td>
</tr>
<tr>
<td>7</td>
<td>length-low</td>
<td>Least significant byte of the program length.</td>
</tr>
</tbody>
</table>

### 5.3 FCode functions

The following subclause specify the set of predefined FCode functions and their associated FCode numbers. The evaluation of an FCode program may create additional FCode functions and associate them with FCode numbers.

The following subclauses give the names, FCode numbers, stack diagrams and brief descriptions of predefined FCode functions. The complete semantics of these FCode functions are specified in annex A.
5.3.1 FCode numbers (FCodes)

FCode numbers (sometimes called FCodes) are values between 0x0000 and 0x0FFF, inclusive. FCode numbers between 0x00 and 0xFF, inclusive, are represented in FCode programs by single bytes; all other FCode numbers are represented by pairs of bytes, with the high-order byte first. The FCode numbers between 0x01 and 0x0F, inclusive, are not used, thus eliminating the ambiguity between the one-byte and two-byte forms that would otherwise result. Another way to look at this is to think of the single-byte codes 0x01 through 0x0F as “escape” codes that are followed by another byte.

A summary of assigned FCode numbers is given in annex G.

5.3.1.1 System-defined FCode numbers

System-defined FCode functions are predefined by the firmware system and thus are available to any FCode program. They have FCode numbers in the range 0x0000 through 0x07FF, inclusive.

5.3.1.1.1 Historical FCode numbers

Historical FCode numbers correspond to FCode functions that are not defined by this standard, but that are or have been used by FCode evaluators that predate this standard. These numbers are reserved for the benefit of those pre-existing systems and are not available for reassignment by future revisions of this standard. The historical FCode numbers are interspersed within the range 0x000 through 0x2FF.

The historical FCode numbers are as follows:

0xA1 convert
0xB3 set-token
0xB4 set-table
0xBF b(code)
0xFE 4-byte-id
0x101 dma-alloc
0x104 memmap
0x106 >physical
0x10F my-params
0x118 driver
0x123 group-code
0x156 frame-buffer-busy?
0x170-17C fbl routines
0x190-0x196 Obsolete VMEbus support
0x1A0 return-buffer
0x1A1 xmit-packet
0x1A2 poll-packet
0x210 processor-type
0x211 firmware-version
0x212 fcode-version
0x229 adr-mask
0x238 probe
0x239 probe-virtual

These FCode numbers are reserved for the benefit of pre-existing FCode systems and shall not be used except for the purpose of compatibility with such pre-existing systems.
5.3.1.1.2 Defined FCode numbers

Defined FCode numbers correspond to FCode functions specified by this standard. Defined FCode numbers are interspersed within the range 0x000 through 0x5FF.

5.3.1.1.3 Reserved FCode numbers

Reserved FCode numbers are those in the range 0x10 through 0x5FF that are neither defined in this standard nor listed in 5.3.1.1.1. These FCode numbers are reserved for assignment by future versions of this standard.

5.3.1.1.4 Vendor-unique FCode numbers

Vendor-unique FCode numbers are all those in the range 0x600 through 0x7FF. These FCode numbers are reserved for vendor-specific use with built-in devices and will not be assigned by this standard or future versions thereof.

A standard FCode program shall not use vendor-unique FCode numbers.

NOTE—It is generally preferable to provide vendor-unique enhancements in the form of packages with methods, rather than as vendor-unique FCode functions.

5.3.1.2 Program-defined FCode numbers

Within a particular FCode program, new FCode functions may be created and assigned to FCode numbers in the range 0x0800 through 0x0FFF, inclusive. The assignment persists while that particular FCode program is being evaluated and becomes invalid thereafter.

The program-defined FCode number space is private to a particular FCode program; each FCode program may reuse these codes without conflicting with other FCode programs. Functions defined within an FCode program may be exported for external use via another mechanism that does not involve FCode numbers.

5.3.1.3 Undefined FCode numbers

All FCode numbers for which an implementation has not assigned a specific function shall be associated with the ferror function.

5.3.2 Forth FCode functions

5.3.2.1 Standard Forth FCode functions

The following FCode functions have behaviors identical to ANS Forth words (as specified by ANSI X3.215-1994) of the same names. They perform basic functions in the Forth programming language:

<table>
<thead>
<tr>
<th>FCode function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dup</td>
<td>( x -- x x ) 0x47 Duplicate the top item on the stack.</td>
</tr>
<tr>
<td>2dup</td>
<td>( x1 x2 -- x1 x2 x1 x2 ) 0x53 Duplicate the top two items on the stack.</td>
</tr>
<tr>
<td>?dup</td>
<td>( x -- 0</td>
</tr>
<tr>
<td>over</td>
<td>( x1 x2 -- x1 x2 x1 ) 0x48 Copy second stack item to top of stack.</td>
</tr>
<tr>
<td>2over</td>
<td>( x1 x2 x3 x4 -- x1 x2 x3 x4 x1 x2 ) 0x54 Copy second pair of stack items to top of stack.</td>
</tr>
<tr>
<td>pick</td>
<td>( xu ... x1 x0 u -- xu ... x1 x0 xu ) 0x4E Copy ath stack item to top of stack.</td>
</tr>
<tr>
<td>tuck</td>
<td>( x1 x2 -- x2 x1 x2 ) 0x4C Copy top stack item underneath the second stack item.</td>
</tr>
<tr>
<td>drop</td>
<td>( x -- ) 0x46 Remove top item from the stack.</td>
</tr>
<tr>
<td>2drop</td>
<td>( x1 x2 -- ) 0x52 Remove top two items from the stack.</td>
</tr>
<tr>
<td>nip</td>
<td>( x1 x2 -- x2 ) 0x4D Remove the second stack item.</td>
</tr>
<tr>
<td>roll</td>
<td>( xu ... x1 x0 u -- xu-1... x1 x0 xu ) 0x4F Rotate u+1 stack items as shown.</td>
</tr>
</tbody>
</table>
rot ( x1 x2 x3 -- x2 x3 x1 ) 0x4A Rotate top three stack items as shown.

-rot ( x1 x2 x3 -- x3 x1 x2 ) 0x4B Rotate top three stack items as shown.

2rot ( x1 x2 x3 x4 x5 x6 -- x3 x4 x5 x6 x1 x2 ) 0x56 Rotate three pairs of stack items as shown.

swap ( x1 x2 -- x2 x1 ) 0x49 Exchange top two stack items.

2swap ( x1 x2 x3 x4 -- x3 x4 x1 x2 ) 0x55 Exchange top two pairs of stack items.

> ( x -- ) 0x30 Move top stack item to the return stack.

> ( -- x ) 0x31 Move top return stack item to the stack.

> ( -- x ) 0x32 Copy top return stack item to the stack.

depth ( -- ) 0x51 Return count of items on the stack.

+ ( nu1 nu2 -- sum ) 0x1E Add nu1 to nu2.

- ( nu1 nu2 -- diff ) 0x1F Subtract nu2 from nu1.

* ( nu1 nu2 -- prod ) 0x20 Multiply nu1 by nu2.

/ ( n1 n2 -- quot ) 0x21 Divide n1 by n2; return quotient.

mod ( n1 n2 -- rem ) 0x22 Divide n1 by n2; return remainder.

/ ( n1 n2 -- quot ) 0x23 Divide n1 by n2; return quotient and remainder.

u/mod ( u1 u2 urem uquot ) 0x2B Divide n1 by n2, all unsigned.

abs ( n -- u ) 0x2D Return absolute value of n.

negate ( n1 -- n2 ) 0x2C Return negation of n1.

max ( n1 n2 -- n1|n2 ) 0x2F Return greater of n1 and n2.

min ( n1 n2 -- n1|n2 ) 0x2E Return lesser of n1 and n2.

bounds ( n cnt -- n+cnt n ) 0xAC Prepare arguments for do or ?do loop.

lshift ( x1 u -- x2 ) 0x27 Shift x1 left by u bit-places. Zero-fill low bits.

rshift ( x1 u -- x2 ) 0x28 Shift x1 right by u bit-places. Zero-fill high bits.

2* ( x1 -- x2 ) 0x29 Shift x1 left by one bit-place. Zero-fill low bit.

2/ ( x1 x2 -- x3 ) 0x2A Shift x1 right by one bit-place. High bit unchanged.

and ( x1 x2 -- x3 ) 0x2B Return bitwise logical "and" of x1 and x2.

xor ( x1 x2 -- x3 ) 0x2C Return bitwise logical "exclusive-or" of x1 and x2.

invert ( x1 -- x2 ) 0x2D Invert all bits of x1.

d+ ( d1 d2 -- d.sum ) 0xD8 Add d1 to d2 giving double-number d.sum.

d- ( d1 d2 -- d.diff ) 0xD9 Subtract d2 from d1 giving double-number difference d.diff.

um* ( u1 u2 -- d.prod ) 0x2E Unsigned multiply with double number product.

um/mod ( ud u-- urem uquot ) 0xD5 Divide unsigned double number ud by u.

char+ ( addr1 -- addr2 ) 0x2F Increment addr1 by the value of /c.

cell+ ( addr1 -- addr2 ) 0x30 Increment addr1 by the value of /n.

chars ( nu1 -- nu2 ) 0x66 Multiply nu1 by the value of /c.

cells ( nu1 -- nu2 ) 0x69 Multiply nu1 by the value of /n.

aligned ( n1 -- n1|a-addr ) 0xAE Increase n1 as necessary to give valid address boundary.

@ ( a-addr -- x ) 0x6D Fetch item x from cell at a-addr.

2@ ( a-addr -- x1 x2 ) 0x76 Fetch cell pair from a-addr.

c@ ( addr -- byte ) 0x71 Fetch byte from addr.

! ( x a-addr -- ) 0x72 Store item x to cell at a-addr.

2! ( x1 x2 a-addr -- ) 0x77 Store cell pair at a-addr.

+! ( nu a-addr -- ) 0x6C Add nu to cell at a-addr.

c! ( byte addr -- ) 0x75 Store byte to addr.

move ( src-addr dest-addr len -- ) 0x78 Copy len bytes from src-addr to dest-addr.

fill ( addr len byte -- ) 0x79 Set len bytes beginning at addr to the value byte.

key? ( -- pressed? ) 0x8D Return true if an input character available.

key ( -- char ) 0x8E Read a character from the console input device.

expect ( addr len -- ) 0x8A Get edited input line, storing it at addr.

span ( -- a-addr ) 0x88 variable holding number of characters received by expect.

bl ( -- 0x20 ) 0x9A ASCII code for space (blank) character.
emit ( char -- ) 0x8F  Display the given ASCII character.
type ( text-str text-len -- ) 0x90 Display text-len characters beginning at address text-str.
cr ( -- ) 0x92  Subsequent output goes to the next line.
count ( pstr --str len ) 0x84 Unpack a counted string to a text string.
base ( -- a-addr ) 0xA0  variable containing the number-conversion radix.
. ( u -- ) 0x9B  Display an unsigned number with a trailing space.
.r ( n size -- ) 0x9C  Display an unsigned number, right-justified.
.s ( ... -- ... ) 0x9F  Display entire stack contents, unchanged.
<# ( -- ) 0x96 Initialize pictured numeric output conversion.
# ( ud1 -- ud2 ) 0xC7 Convert a digit in pictured numeric output conversion.
#s ( ud -- 0 0 ) 0xC8 Convert remaining digits in pictured numeric output.
#> ( ud -- str len ) 0xC9 End pictured numeric output conversion.
hold ( char --  ) 0x95 Add char in pictured numeric output conversion.
sign ( n -- ) 0x98 If n < 0, insert "-" in pictured numeric output.
< ( n1 n2 -- less? ) 0x3A Return true if n1 is less than n2.
<> ( x1 x2 -- not-equal? ) 0x3D Return true if x1 is not equal to x2.
= ( x1 x2 -- equal? ) 0x3C Return true if x1 is equal to x2.
> ( n1 n2 -- greater? ) 0x3B Return true if n1 is greater than n2.
within ( n min max -- min<=n<max? ) 0x45 Return true if n is between min and max–1, inclusive.
0< ( n -- less-than-0? ) 0x36 Return true if n is less than zero.
0<> ( n -- not-equal-to-0? ) 0x35 Return true if n is not equal to zero.
0= ( n -- equal-to-0? ) 0x34 Return true if n is equal to zero.
0> ( n -- greater-than-0? ) 0x38 Return true if n is greater than zero.
u< ( u1 u2 -- unsigned-less? ) 0x40 Return true if ul is less than u2, unsigned.
u> ( u1 u2 -- unsigned-greater? ) 0x3E Return true if ul is greater than u2, unsigned.
i ( -- index ) (R: sys -- sys ) 0x19 Return current loop index value.
j ( -- index ) (R: sys -- sys ) 0x1A Return next outer loop index value.
unloop ( -- ) (R: sys -- sys ) 0x89 Discard loop control parameters.
evaluate ( ... str len -- ??? ) 0xCD Evaluate Forth text from the given string.
execute ( ... xt -- ??? ) 0x1D Execute the command whose execution token is xt.
exit ( -- ) (R: sys -- ) 0x33 Exit from the currently executing command.
abort ( ... -- ) (R: ... -- ) 0x216 Abort program execution; clear stacks.
catch ( ... xt -- ??? error-code | ??? false ) 0x217 Execute command indicated by xt. Return throw result.
throw ( ... error-code -- ??? error-code | ... ) 0x218 Transfer back to catch routine if error-code is nonzero.
here ( -- addr ) 0xAD Return current dictionary pointer.
c, ( byte -- ) 0xD0 Compile a byte into the dictionary.
, ( x -- ) 0xD3 Append x to data space.
compile, ( xt -- ) 0xDD Compile the behavior of the word given by xt.
state ( -- a-addr ) 0xDC variable containing true if in compilation state.
>body ( xt -- a-addr ) 0x86 Convert execution token to data field address.

5.3.2.2 Other simple Forth FCode functions

/c ( -- n ) 0x5A The number of address units to a byte: one.
/w ( -- n ) 0x5B The number of address units to a doublet: typically, two.
/l ( -- n ) 0x5C The number of address units to a quadlet: typically, four.
/n ( -- n ) 0x5D The number of address units in a cell.
ca+ ( addr1 index -- addr2 ) 0x5E Increment addr1 by index times the value of /c.
w+ ( addr1 index -- addr2 ) 0x5F Increment addr1 by index times the value of /w.
l+ ( addr1 index -- addr2 ) 0x60 Increment addr1 by index times the value of /l.
(continued)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\na+</td>
<td>Increment <code>addr1</code> by <code>index</code> times the value of <code>/n</code></td>
</tr>
<tr>
<td>\wal+</td>
<td>Increment <code>addr1</code> by the value of <code>/w</code></td>
</tr>
<tr>
<td>\lal+</td>
<td>Increment <code>addr1</code> by the value of <code>/l</code></td>
</tr>
<tr>
<td>/\n</td>
<td>Multiply <code>nu1</code> by the value of <code>/n</code></td>
</tr>
<tr>
<td>/\w</td>
<td>Multiply <code>nu1</code> by the value of <code>/w</code></td>
</tr>
<tr>
<td>/\l</td>
<td>Multiply <code>nu1</code> by the value of <code>/l</code></td>
</tr>
<tr>
<td>w@</td>
<td>Fetch doublet from <code>waddr</code></td>
</tr>
<tr>
<td>\w@</td>
<td>Fetch doublet from <code>waddr</code>, sign-extended</td>
</tr>
<tr>
<td>l@</td>
<td>Fetch quadlet from <code>qaddr</code></td>
</tr>
<tr>
<td>w!</td>
<td>Store doublet <code>w</code> to <code>waddr</code></td>
</tr>
<tr>
<td>l!</td>
<td>Store quadlet to <code>qaddr</code></td>
</tr>
<tr>
<td>w,</td>
<td>Compile a doublet into the dictionary, doublet-aligned</td>
</tr>
<tr>
<td>l,</td>
<td>Compile a quadlet into the dictionary, doublet-aligned</td>
</tr>
<tr>
<td>off</td>
<td>Store <code>false</code> to cell at <code>a-addr</code></td>
</tr>
<tr>
<td>on</td>
<td>Store <code>true</code> to cell at <code>a-addr</code></td>
</tr>
<tr>
<td>u#</td>
<td>Convert a digit in pictured numeric output conversion</td>
</tr>
<tr>
<td>u#s</td>
<td>Convert remaining digits in pictured numeric output conversion</td>
</tr>
<tr>
<td>u#&gt;</td>
<td>End pictured numeric output conversion</td>
</tr>
<tr>
<td>comp</td>
<td>Compare two arrays of length <code>len</code></td>
</tr>
<tr>
<td>lbsplit</td>
<td>Split a quadlet into four bytes</td>
</tr>
<tr>
<td>lwsplit</td>
<td>Split a quadlet into two doublets</td>
</tr>
<tr>
<td>wsplits</td>
<td>Split a doublet into two bytes</td>
</tr>
<tr>
<td>bljoin</td>
<td>Join four bytes to form a quadlet</td>
</tr>
<tr>
<td>bwjoin</td>
<td>Join two bytes to form a doublet</td>
</tr>
<tr>
<td>wljoin</td>
<td>Join two doublets to form a quadlet</td>
</tr>
<tr>
<td>wbflip</td>
<td>Swap the bytes within a doublet</td>
</tr>
<tr>
<td>wbflips</td>
<td>Swap the bytes within each doublet in the given region</td>
</tr>
<tr>
<td>lbf</td>
<td>Reverse the bytes within a quadlet</td>
</tr>
<tr>
<td>lbf</td>
<td>Reverse the bytes within each quadlet in the given region</td>
</tr>
<tr>
<td>lwflips</td>
<td>Swap the doubles within each quadlet</td>
</tr>
<tr>
<td>u2/</td>
<td>Shift <code>x1</code> right by one bit-place. Zero-fill high bit</td>
</tr>
<tr>
<td>between</td>
<td>Return <code>true</code> if <code>n</code> is between min and max, inclusive</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Return <code>true</code> if <code>n</code> is greater than or equal to <code>n2</code></td>
</tr>
<tr>
<td>&lt;=</td>
<td>Return <code>true</code> if <code>n</code> is less than or equal to <code>n2</code></td>
</tr>
<tr>
<td>0&gt;=</td>
<td>Return <code>true</code> if <code>n</code> is less than or equal to zero</td>
</tr>
<tr>
<td>0&lt;=</td>
<td>Return <code>true</code> if <code>n</code> is greater than or equal to zero</td>
</tr>
<tr>
<td>u&gt;</td>
<td>Return <code>true</code> if <code>ul</code> greater or equal to <code>u2</code>, unsigned</td>
</tr>
<tr>
<td>u&lt;&gt;</td>
<td>Return <code>true</code> if <code>ul</code> less or equal to <code>u2</code>, unsigned</td>
</tr>
<tr>
<td>&gt;&gt;a</td>
<td>Arithmetic shift <code>x1</code> right by <code>a</code> bit-places</td>
</tr>
<tr>
<td>body&gt;</td>
<td>Convert data field address to execution token</td>
</tr>
<tr>
<td>noop</td>
<td>Do nothing</td>
</tr>
<tr>
<td>lcc</td>
<td>Convert ASCII <code>char1</code> to lowercase</td>
</tr>
<tr>
<td>upc</td>
<td>Convert ASCII <code>char1</code> to uppercase</td>
</tr>
<tr>
<td>-1</td>
<td>Constant –1</td>
</tr>
<tr>
<td>0</td>
<td>Constant 0</td>
</tr>
<tr>
<td>1</td>
<td>Constant 1</td>
</tr>
<tr>
<td>2</td>
<td>Constant 2</td>
</tr>
</tbody>
</table>
5.3.3 FCode implementation functions

These FCode functions correspond only indirectly to Forth words (most other FCode functions correspond directly to identically named Forth words). In general, the names of these FCode functions do not appear in FCode source. Instead, certain FCode source constructs are translated by a tokenizer program into sequences of these FCode functions. Such constructs are indicated in glossary entries by the “T” type code (see A.1.2.3).

5.3.3.1 Defining new FCode functions

Program-defined FCode functions are created by executing a sequence of FCode functions of the following form:

\[
\text{[ instance ] token-type function-type}
\]

If present, instance modifies the behavior of the following FCode function definition so that it allocates instance-specific data storage instead of global data storage; instance only applies to FCode functions that allocate data storage, specifically, b(buffer:), b(defer), b(value), b(variable).

Token-type, one of the FCode functions new-token, named-token, or external-token, establishes the new function’s FCode number and possibly its externally visible name. The token-type portion of the FCode program includes an FCode-string (except in the case of new-token) and an FCode#. Any program-defined FCode function may be executed from within the FCode program that defines it, but only those functions with an externally visible name can be called from outside the FCode program (e.g., with $call-method).

Function-type, one of the FCode functions b(:), b(buffer:), b(constant), b(create), b(defer), b(field), b(value), or b(variable), establishes the general behavior of the new function.

\[
\begin{align*}
\text{instance} & \quad (\quad) \quad 0xC0 \quad \text{Mark next defining word as instance-specific.} \\
\text{new-token} & \quad (F: /FCode#/ \quad) \quad 0xB5 \quad \text{Create a new unnamed FCode function.} \\
\text{named-token} & \quad (F: /FCode-string FCode#/ \quad) \quad 0xB6 \quad \text{Create a new possibly named FCode function.} \\
\text{external-token} & \quad (F: /FCode-string FCode#/ \quad) \quad 0xB6 \quad \text{Create a new possibly named FCode function.} \\
\text{b(;) } & \quad (\quad) \quad 0xCA \quad \text{Create a new named FCode function.} \\
\text{b(;) } & \quad (\quad) \quad 0xC2 \quad \text{End an FCode colon definition.} \\
\text{b(;) } & \quad (\quad) \quad 0xB7 \quad \text{Defines type of new FCode function as “colon definition”}. \\
\text{b(buffer:)} & \quad (\text{size} \quad) \quad 0xBD \quad \text{Defines type of new FCode function as buffer:].} \\
\text{b(buffer:)} & \quad (\text{E: \quad \text{... \quad ???? \quad})} \\
\text{b(constant)} & \quad (n1 \quad) \quad 0xBA \quad \text{Defines type of new FCode function as constant.} \\
\text{b(constant)} & \quad (E: \quad \text{-- a-addr \quad}) \\
\text{b(create)} & \quad (\quad) \quad 0xBB \quad \text{Defines type of new FCode function as create word.} \\
\text{b(create)} & \quad (\quad) \quad 0xBC \quad \text{Defines type of new FCode function as defer word.} \\
\text{b(defer)} & \quad (\quad) \quad 0xC3 \quad \text{Defines type of new FCode function as defer word.} \\
\text{b(defer)} & \quad (\quad) \quad 0xC3 \quad \text{Defines type of new FCode function as defer word.} \\
\end{align*}
\]
IEEE
Std 1275-1994

IEEE STANDARD FOR BOOT (INITIALIZATION CONFIGURATION) FIRMWARE:

b(field) ( offset size -- offset+size ) 0xBE Defines type of new FCode function as field.
(E: addr -- addr+offset )

b(value) ( x -- ) 0xB8 Defines type of new FCode function as value.
(E: -- x )

b(variable) ( -- ) 0xB9 Defines type of new FCode function as variable.
(E: -- a-addr )

(is-user-word) ( name-str name-len xt -- ) 0x214 Create a named user interface command.
(E: ... -- ??? )

get-token ( fcode# -- xt immediate? ) 0xDA Convert FCode number to function execution token.

set-token ( xt immediate? fcode# -- ) 0xDB Assign FCode number to existing function.

5.3.3.2 Literals

Each of these functions reads a literal value from the FCode program and pushes the value on the data stack:

b(lit) ( -- n1 ) 0x10 Numeric literal FCode. Followed by FCode-num32.
b(') ( -- xt )0x11Function literal FCode. Followed by FCode#.
b(" ) ( -- str len ) 0x12 String literal FCode. Followed by FCode-string.

5.3.3.3 Controlling values and defers

b(to) sets the value of value and defer functions, reading the FCode# of the function whose value is to be set from the FCode program.

behavior ( defer-xt -- contents-xt ) 0xDE Retrieve execution behavior of a defer word.
b(to) ( params -- ) 0xC3 FCode for setting values and defers. Followed by FCode#.

5.3.3.4 Control flow

In various combinations, these functions implement control structures, such as conditionals and loops. Many of these functions are similar in most respects to ANS Forth words (as specified by ANSI X3.215-1994), but their behavioral descriptions account for the fact that they read FCode-offsets from the FCode program during FCode evaluation:

offset16 ( -- ) 0xCC Makes subsequent FCode-offsets use 16-bit (not 8-bit) form.
bbranch ( -- ) 0x13 Unconditional branch FCode. Followed by FCode-offset.
b?branch ( continue? -- ) 0x14 Conditional branch FCode. Followed by FCode-offset.
b(<mark) ( -- ) 0xB1 Target of backward branches.
b(>resolve) ( -- ) 0xB2 Target of forward branches.
b(loop) ( -- ) 0x15 End FCode do ... loop. Followed by FCode-offset.
b(+loop) ( delta -- ) 0x16 End FCode do ... +loop. Followed by FCode-offset.
b(do) ( limit start -- ) 0x17 Begin FCode do ... loop. Followed by FCode-offset.
b(?do) ( limit start -- ) 0x18 Begin FCode ?do ... loop. Followed by FCode-offset.
b(leave) ( -- ) 0x1B Exit from a do ... loop.
b(case) ( sel -- sel ) 0xC4 Begin a case (multiple selection) statement.
b(endcase) ( sel | <nothing> -- ) 0xC5 End a case (multiple selection) statement.
b(of) ( sel of-val -- sel | <nothing> ) 0x1C FCode for of in case statement. Followed by FCode-offset.
b(endof) ( -- ) 0xC6 FCode for endif in case statement. Followed by FCode-offset.

5.3.4 Package access

These functions manage the interface between packages, allowing packages to call each other’s methods and inspect each other’s properties.

56
5.3.4.1 Open/close packages

These functions find, open, and close packages:

find-package ( name-str name-len -- false | phandle true ) 0x204 Locate the support package named by name string.
open-package ( arg-str arg-len phandle -- ihandle | 0 ) 0x205 Open the package indicated by phandle.
$open-package ( arg-str arg-len name-str name-len -- ihandle | 0 ) 0x20F Open the support package named by name string.
close-package ( ihandle -- ) 0x206 Close the specified package instance.
my-self ( -- ihandle ) 0x203 Return the ihandle of the current instance.
my-parent ( -- ihandle ) 0x20A Return the ihandle of the parent of the current instance.
ihandle>phandle ( ihandle -- phandle ) 0x20B Return the phandle for the indicated ihandle.
next-property ( previous-str previous-len phandle -- false | name-str name-len true ) 0x23D Return the name of the property following previous of phandle.
peer ( phandle -- phandle.sibling ) 0x23C Return the phandle of the next sibling node.
child ( phandle.parent -- phandle.child ) 0x23B Return the phandle of the first child node of parent.

5.3.4.2 Call methods from other packages

These functions find and execute methods from other packages:

find-method ( method-str method-len phandle -- false | xt true ) 0x207 Find the method named method-string in the package phandle.
call-package ( ... xt ihandle -- ??? ) 0x208 Execute the method xt within the instance ihandle.
$call-method ( ... method-str method-len ihandle -- ??? ) 0x20E Execute the method named method-string in the instance ihandle.
$call-parent ( ... method-str method-len -- ??? ) 0x209 Execute the method named method-string in the parent instance.

5.3.4.3 Get local arguments

These functions return information about the current instance:

my-address ( -- phys.lo ... ) 0x102 Return low component(s) of device’s physical address.
my-space ( -- phys.hi ) 0x103 Return high component of device’s physical address.
my-unit ( -- phys.lo ... phys.hi ) 0x20D Return the unit address of the current instance.
my-args ( -- arg-str arg-len ) 0x202 Return the instance-argument string for this instance.
left-parse-string ( str len char -- R-str R-len L-str L-len ) 0x240 Split the string at first occurrence of delimiter char.
parse-2int ( str len -- val.lo val.hi ) 0x11B Convert a “hi,lo” string into a pair of values.

5.3.4.4 Mapping tools

These functions are shorthand versions of common sequences:

map-low ( phys.lo ... size -- virt ) 0x130 Map the specified region; return a virtual address.
free-virtual ( virt size -- ) 0x105 Destroy mapping and “address” property.
5.3.5 Property management

5.3.5.1 Property array encoding

These functions encode various data types into prop-encoded-arrays suitable for property values:

- `encode-int ( n -- prop-addr prop-len ) 0x111` Encode a number into a prop-encoded-array.
- `encode-string ( str len -- prop-addr prop-len ) 0x114` Encode a string into a prop-encoded-array.
- `encode-bytes ( data-addr data-len -- prop-addr prop-len ) 0x115` Encode a byte array into a prop-encoded-array.
- `encode-phys ( phys.lo ... phys.hi -- prop-addr prop-len ) 0x113` Encode a unit address into a prop-encoded-array.
- `encode+ ( prop-addr1 prop-len1 prop-addr2 prop-len2 -- prop-addr3 prop-len3 ) 0x112` Concatenate two prop-encoded-arrays into a single array.

5.3.5.2 Property array decoding

These functions decode various data types from prop-encoded-arrays:

- `decode-int ( prop-addr1 prop-len1 -- prop-addr2 prop-len2 n ) 0x21B` Decode a number from a prop-encoded-array.
- `decode-phys ( prop-addr1 prop-len1 -- prop-addr2 prop-len2 phys.lo ... phys.hi ) 0x128` Decode a unit address from a prop-encoded-array.
- `decode-string ( prop-addr1 prop-len1 -- prop-addr2 prop-len2 str len ) 0x21C` Decode a string from a prop-encoded-array.

5.3.5.3 Property declaration

These functions create, delete, and modify properties in the active package. `property` is the general-purpose function for creating properties. `delete-property` deletes a property. The other functions in this subclause are space-saving words for a common property, with behavior identical to the use of `property` with the indicated name string.

- `property ( prop-addr prop-len name-str name-len -- ) 0x110` Create a new property with the given name and value.
- `delete-property ( name-str name-len -- ) 0x21E` Delete the named property in the active package.
- `device-name ( str len -- ) 0x201` Create the “name” property; value is indicated string.
- `device-type ( str len -- ) 0x11A` Create “device_type” property; value is indicated string.
- `reg ( phys.lo ... phys.hi size -- ) 0x116` Create the “reg” property with the given values.
- `model ( str len -- ) 0x119` Create the “model” property; value is indicated string.

5.3.5.4 Property value access

The following functions retrieve property values:

- `get-package-property ( name-str name-len phandle -- true | prop-addr prop-len false ) 0x21F` Return value for name string property in package phandle.
- `get-inherited-property ( name-str name-len -- true | prop-addr prop-len false ) 0x21D` Return value for given property in the current instance or its parents.
- `get-my-property ( name-str name-len -- true | prop-addr prop-len false ) 0x21A` Return value for given property in this package.
5.3.6 Display device management

These functions assist in the implementation of console display devices, devices identified by the "display" device-type property.

5.3.6.1 Terminal emulator routines

The terminal emulator implements an ANSI X3.64 terminal using the display device driver for low-level screen manipulation operations.

The following values are used and set by the terminal emulator:

- `line#` (0x152) -- Return the current cursor line number.
- `column#` (0x153) -- Return the current cursor column number.
- `inverse?` (0x154) -- Indicates how to paint characters.
- `inverse-screen?` (0x155) -- Indicates how to paint the background.
- `#lines` (0x150) -- Return number of lines of text in text window.
- `#columns` (0x151) -- Return number of columns of text in text window.

The terminal emulator uses the following defer words to access display device driver routines:

- `draw-character` (char --) (0x157) -- Draw a character at the current cursor position.
- `reset-screen` (--) (0x158) -- Perform frame-buffer device initialization.
- `toggle-cursor` (--) (0x159) -- Toggle the state of the text cursor.
- `erase-screen` (--) (0x15A) -- Clear the screen.
- `blink-screen` (--) (0x15B) -- Flash the screen.
- `invert-screen` (--) (0x15C) -- Exchange the foreground and background colors.
- `insert-characters` (n --) (0x15D) -- Insert n spaces to the right of the cursor.
- `delete-characters` (n --) (0x15E) -- Delete n characters to the right of the cursor.
- `insert-lines` (n --) (0x15F) -- Insert n blank lines at and below the cursor line.
- `delete-lines` (n --) (0x160) -- Delete n lines at and below the cursor line.
- `draw-logo` (line# addr width height --) (0x161) -- Draw (at line#) the logo stored at location addr.

5.3.6.2 Frame-buffer support routines

These functions control the character font used to display characters. set-font may be used with the system-provided font described by the default-font or with a font provided by the FCode program.

- `default-font` (addr width height advance min-char #glyphs) (0x16A) -- Return the font parameters for the default system font.
- `set-font` (addr width height advance min-char #glyphs --) (0x16B) -- Set the current font as specified.
- `>font` (char -- addr) (0x16E) -- Return beginning address for char in the current font.

The following values are used internally by both the 1-bit and the 8-bit frame-buffer support routines:

- `frame-buffer-adr` (addr --) (0x162) -- Return current frame-buffer virtual address.
- `screen-height` (height --) (0x163) -- Return total height of the display in pixels.
- `screen-width` (width --) (0x164) -- Return total width of the display in pixels.
- `window-top` (border-height --) (0x165) -- Return window top border in pixels.
- `window-left` (border-width --) (0x166) -- Return window left border in pixels.
- `char-height` (height --) (0x16C) -- Return the height of a font character in pixels.
- `char-width` (width --) (0x16D) -- Return the width of a font character in pixels.
- `fontbytes` (bytes --) (0x16F) -- Return interval between entries in the font table.
5.3.6.3 Display device support

This subclause defines support routines used in FCode frame-buffer packages:

5.3.6.3.1 Frame-buffer package interface

- `is-install ( xt -- ) 0x11C`: Create `open` and other methods for this display device.
- `is-remove ( xt -- ) 0x11D`: Create `close` method for this display device.
- `is-selftest ( xt -- ) 0x11E`: Create `selftest` method for this display device.

5.3.6.3.2 Generic one-bit frame-buffer support (optional)

See annex H.

5.3.6.3.3 Generic eight-bit frame-buffer support

The “fb8” generic frame-buffer support package implements the display device interface for frame buffers with 8 bits per pixel.

- `fb8-install ( width height #columns #lines -- ) 0x18B`: Install all built-in generic 8-bit frame-buffer routines.
- `fb8-draw-character ( char -- ) 0x180`: Implement the “fb8” draw-character function.
- `fb8-reset-screen ( -- ) 0x181`: Implement the “fb8” reset-screen function.
- `fb8-toggle-cursor ( -- ) 0x182`: Implement the “fb8” toggle-cursor function.
- `fb8-erase-screen ( -- ) 0x183`: Implement the “fb8” erase-screen function.
- `fb8-blink-screen ( -- ) 0x184`: Implement the “fb8” blink-screen function.
- `fb8-invert-screen ( -- ) 0x185`: Implement the “fb8” invert-screen function.
- `fb8-insert-characters ( n -- ) 0x186`: Implement the “fb8” insert-characters function.
- `fb8-delete-characters ( n -- ) 0x187`: Implement the “fb8” delete-characters function.
- `fb8-insert-lines ( n -- ) 0x188`: Implement the “fb8” insert-lines function.
- `fb8-delete-lines ( n -- ) 0x189`: Implement the “fb8” delete-lines function.
- `fb8-draw-logo ( line# addr width height -- ) 0x18A`: Implement the “fb8” draw-logo function.

5.3.7 Other FCode functions

5.3.7.1 Peek/poke

The following functions attempt a read or write access at a possibly invalid address, returning a flag indicating whether or not an access error occurred:

- `cpeek ( addr -- false | byte true ) 0x220`: Attempt to fetch the byte at `addr`.
- `wpeek ( waddr -- false | w true ) 0x221`: Attempt to fetch the doublet `w` at `waddr`.
- `lpeek ( qaddr -- false | quad true ) 0x222`: Attempt to fetch the quadlet at `qaddr`.
- `cpoke ( byte addr -- okay? ) 0x223`: Attempt to store the byte to `addr`.
- `wpoke ( w waddr -- okay? ) 0x224`: Attempt to store the doublet `w` to `waddr`.
- `lpoke ( quad qaddr -- okay? ) 0x225`: Attempt to store the quadlet to `qaddr`.

5.3.7.2 Device-register access

The following functions are used to access device registers, providing a predictable access model in the presence of such effects as byte order differences across bus bridges, presence of write buffers, and so forth. Unlike the standard
Forth data-access words \texttt{c@}, \texttt{c!}, \texttt{@}, and \texttt{!}, these words are guaranteed to read or write with a single access operation.

\begin{verbatim}
rb@ \hfill ( addr -- byte ) 0x230  Fetch a byte from device register at addr.
rw@ \hfill ( waddr -- w ) 0x232  Fetch a doublet w from device register at waddr.
rl@ \hfill ( qaddr -- quad ) 0x234  Fetch a quadlet from device register at qaddr.
rb! \hfill ( byte addr -- ) 0x231  Store a byte to device register at addr.
rw! \hfill ( w waddr -- ) 0x233  Store a doublet w to device register at waddr.
rl! \hfill ( quad qaddr -- ) 0x235  Store a quadlet to device register at qaddr.
\end{verbatim}

5.3.7.3 Time

These functions provide basic real-time measurements and delays. The accuracy of time values is system-dependent.

\begin{verbatim}
get-msecs \hfill ( -- n ) 0x125  Return elapsed time in milliseconds.
ms \hfill ( n -- ) 0x126  Delay for at least n milliseconds.
alarm \hfill ( xt n -- ) 0x213  Execute xt repeatedly at intervals of n milliseconds.
user-abort \hfill ( ... -- (R: ... -- ) 0x219  After alarm routine is finished, abort program execution.
\end{verbatim}

5.3.7.4 System information

\begin{verbatim}
fcode-revision \hfill ( -- n ) 0x87  Return revision level of FCode interface.
mac-address \hfill ( -- mac-str mac-len ) 0x1A4  Return a sequence of bytes containing network address.
\end{verbatim}

5.3.7.5 FCode self-test

These functions are used primarily to implement selftest methods:

\begin{verbatim}
display-status \hfill ( n -- ) 0x121  Display the results of a device self-test.
memory-test-suite \hfill ( addr len -- fail? ) 0x122  Perform tests of memory, starting at addr for len bytes.
mask \hfill ( -- a-addr ) 0x124  variable to control bits tested with memory-test-suite.
diagnostic-mode? \hfill ( -- diag? ) 0x120  If true, boot from diag sources; perform longer self-tests.
\end{verbatim}

5.3.7.6 Start and end

These functions begin, end, and partition FCode programs. Spread is the distance in address units between consecutive bytes of the FCode program.

\begin{verbatim}
start0 \hfill ( -- ) 0xF0  Begin program with spread 0 followed by FCode-header.
start1 \hfill ( -- ) 0xF1  Begin program with spread 1 followed by FCode-header.
start2 \hfill ( -- ) 0xF2  Begin program with spread 2 followed by FCode-header.
start4 \hfill ( -- ) 0xF3  Begin program with spread 4 followed by FCode-header.
version1 \hfill ( -- ) 0xFD  Begin program with spread 1 followed by FCode-header.
end0 \hfill ( -- ) 0x00  Cease evaluating this FCode program.
end1 \hfill ( -- ) 0xFF  Cease evaluating this FCode program.
ferror \hfill ( -- ) 0xFC  Standard FCode number for undefined FCode functions.
suspend-fcode \hfill ( -- ) 0x215  Pause FCode evaluation if desired; can resume later.
new-device \hfill ( -- ) 0x11F  Start new package as child of active package.
finish-device \hfill ( -- ) 0x127  Finish this package; set active package to parent.
byte-load \hfill ( addr xt -- ) 0x23E  Evaluate FCode beginning at location addr.
set-args \hfill ( arg-str arg-len unit-str unit-len -- ) 0x23F  Set address and arguments of new device node.
\end{verbatim}
5.4 Standard FCode program

5.4.1 Overall structure

A standard FCode program
— shall begin with the FCode# for one of the following FCode functions:
  version1, start0, start1, start2, start4
— shall end with the FCode# for one of the following FCode functions:
  end0, end1
— may contain a sequence of other FCode#s between the beginning code and the ending code.

5.4.2 Usage rules

A standard FCode program shall comply with all of the following rules.
— Each FCode# that corresponds to an FCode function that reads from the FCode program shall be followed by the appropriate in-line data with the following exception: If the FCode# is itself the in-line data for some other FCode function (for example the FCode# that follows b(')), it shall not be followed by additional in-line data.
— During FCode evaluation, the compilation and control flow stack effects shall balance throughout the FCode program as a whole.
— For all possible execution paths, the contents of the stack shall be appropriate, at each execution of an FCode function, for that FCode function. The program may assume that the methods of this program are called from the outside with valid stack arguments, and that any external methods that are called perform according to their specifications.
— Any external methods that are called from within the program shall be called with valid stack arguments.
— Ambiguous conditions or undefined parameter ranges of FCode functions or external methods shall be avoided.
6. Client interface

The client interface allows client programs (programs that have been loaded and executed under the control of Open Firmware) to make use of services provided by Open Firmware. The interface consists of a set of software procedures and a mechanism for calling and passing arguments and results to and from those procedures.

6.1 General

The Open Firmware client interface specifies the behavior of a firmware system so that client programs (programs that are loaded into and execute from RAM) begin their execution with a predictable machine state and may use various Open Firmware facilities. The client interface consists of both the specification of the machine environment that exists when the client program begins execution and the set of services that Open Firmware provides for the program’s use.

6.1.1 Description

Client interface services are 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, and other services.

The client execution environment is the machine state that exists when a client program begins execution.

6.1.2 Specification

In order to be compliant with the Open Firmware client interface, the boot firmware associated with a main CPU device shall

— Implement the set of client interface services defined in this clause and provide the client execution environment specified in this clause.

— Implement these standard system nodes: /openprom, /options, /chosen, and “memory” (the node whose ihandle is given by the value of /chosen’s “memory” property).

6.1.3 Warning

The services provided herein may cease to be available if the client program does any of the following:

— Uses system memory not obtained from the client interface memory-allocation functions.

— Performs virtual-address-mapping operations, except by executing virtual-address-mapping client interface routines which may be provided as system-dependent extensions to Open Firmware.

— Directly modifies the state of any hardware device that is in use by the firmware (for example the console device). The list of such devices is system-dependent.

— Directly modifies the state of certain processor registers. The list of such registers is processor-dependent. Supplements to this document may specify such registers for particular processors.

6.2 Client program environment

The details of the client execution environment are ISA-dependent and are specified in Open Firmware ISA-specific supplements (see 2.1).
6.3 Client interface services

6.3.1 Access to the client interface functions

The Open Firmware client interface handler is 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. The data is transferred by means of an array of arguments and results whose address is provided to the client interface handler. The contents of that array is specified in this clause, but the detailed mechanism for transferring control and for providing the address of the array to the client interface handler is specified in the Open Firmware supplement for each ISA (see 2.1).

The argument array consists of the following sequence of cells:

<table>
<thead>
<tr>
<th>Cell Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>service</td>
<td>Address of a null-terminated string specifying the client interface service.</td>
</tr>
<tr>
<td>N-args</td>
<td>Number of input arguments to the client interface service.</td>
</tr>
<tr>
<td>N-returns</td>
<td>Number of return values from the client interface service.</td>
</tr>
<tr>
<td>arg1, …, argN</td>
<td>Input arguments to the client interface service.</td>
</tr>
<tr>
<td>ret1, …, retN</td>
<td>Returned values from the client interface service.</td>
</tr>
</tbody>
</table>

The argument service is the address of a null-terminated string that specifies which firmware service is to be invoked. The arguments N-args and N-returns specify the number of calling arguments and return values; they must agree with the number of arguments and return values expected by the particular service. Each argument arg1 through argN is either represented literally in the argument array (if it can fit into a cell) or is a pointer to the actual argument (if it requires more storage than a single cell); the return values are represented similarly. The form of these arguments and return values depends upon the particular service and may be constants, strings, data structures, or other arbitrary data.

In addition to the ret1 through retN values that are returned in the array, the client interface handler returns a single value as specified in the Open Firmware supplement for the appropriate ISA (see 2.1). That value indicates whether the transfer of control to the Open Firmware succeeded or failed. If the requested service is unavailable or if the control transfer failed, the client interface handler returns the value –1; otherwise, it returns the value zero.

Client interface service names shall be drawn from the character set “0-9 A-Z a-z , _ + -” and shall be at most 31 characters in length. Client interface service names as defined in this specification shall not include a “,”. Manufacturers may define proprietary client interface services; any services so defined shall contain the manufacturer’s name followed by a “,”, followed by the interface service name.

6.3.2 Client interface service definitions

In the following definitions, all arguments and return values are cells. The first item listed corresponds with arg1, the second with arg2, continuing through the nth item. The keyword none indicates that there is no argument or return value for this service. The modifier [string] indicates that this argument or return value is the address of a null-terminated string. The modifier [address] indicates that this argument or return value is an address.

6.3.2.1 Client interface
test
    IN:   [string] name
    OUT:  missing

    Missing is 0 if the service name exists, and –1 if it does not exist.
6.3.2.2  Device tree

peer
IN:  phandle
OUT: sibling-phandle

Sibling-phandle is either the identifier of the device node that is the next sibling of the device node identified by phandle or zero if there are no more siblings. If phandle is zero, sibling-phandle is the identifier of the root node.

child
IN:  phandle
OUT: child-phandle

Child-phandle is either the identifier of the device node that is the first child of the device node identified by phandle or zero if there are no children.

parent
IN:  phandle
OUT: parent-phandle

Parent-phandle is either the node identifier of the device node that is the parent of the device node identified by phandle or zero if phandle is the identifier of the root node.

instance-to-package
IN:  ihandle
OUT: phandle

Phandle is either the identifier corresponding to the instance identifier ihandle or –1 if there is no instance identifier ihandle.

If phandle is –1, Open Firmware was unable to translate ihandle. Open Firmware may, but is not required to, check the validity of an ihandle.

getproplen
IN:  phandle, [string] name
OUT: proplen

Proplen is either the length of the value associated with the property name in the device node identified by phandle, zero if the property name exists but has no corresponding value, or –1 if the property name does not exist.

getprop
IN:  phandle, [string] name, [address] buf, buflen
OUT: size

Copies a maximum of buflen bytes of the value of the property name in the device node identified by phandle into the memory pointed to by buf. Size is either the actual size of the property, or –1 if name does not exist.

nextprop
IN:  phandle, [string] previous, [address] buf
OUT: flag

Copies the name of the property following previous in the property list of the device node identified by phandle into buf, as a null-terminated string. Buf is the address of a 32-byte region of memory. If previous is zero or a pointer to a null string, copies the name of the device node’s first property. If there are no more properties after previous or if previous is invalid (i.e., names a property which does not exist in that device node), copies a null string. The return value flag is –1 if previous is invalid, zero if there are no more properties after previous, or 1 otherwise.
IEEE
Std 1275-1994
IEEE STANDARD FOR BOOT (INITIALIZATION CONFIGURATION) FIRMWARE:

setprop

IN:  phandle, [string] name, [address] buf, len
OUT:  size

Sets the property value of the property name in the device node identified by phandle to the value beginning at memory address buf and continuing for len bytes, attempting to create the property if it does not exist. Size is the actual length of the new value, or –1 if the property value could not be set or could not be created.

NOTE—There may be a length limitation on the property values of the “/options” node, which are stored as fields in nonvolatile RAM. In such cases, the property value could be truncated to fit the available space.

canon

IN:  [string] device-specifier, [address] buf, buflen
OUT:  length

This service converts the possibly ambiguous device-specifier to a fully qualified pathname, storing, at most, buflen bytes as a null-terminated string in the memory buffer starting at the address buf. If the length of the null-terminated pathname is greater than buflen, the trailing characters and the null terminator are not stored. Length is the length of the fully qualified pathname excluding any null terminator, or –1 if the pathname is invalid.

finddevice

IN:  [string] device-specifier
OUT:  phandle

Phandle is the identifier of the device node selected by device-specifier, as with find-device, or –1 if device-specifier cannot be matched. In either case, the active package is unaffected.

instance-to-path

IN:  ihandle, [address] buf, buflen
OUT:  length

This service returns the fully qualified pathname corresponding to the identifier ihandle, storing, at most, buflen bytes as a null-terminated string in the memory buffer starting at the address buf. If the length of the null-terminated pathname is greater than buflen, the trailing characters and the null terminator are not stored. Length is the length of the fully qualified pathname excluding any null terminator, or –1 if ihandle is invalid.

package-to-path

IN:  phandle, [address] buf, buflen
OUT:  length

Returns the fully qualified pathname corresponding to the node identifier phandle, storing, at most, buflen bytes as a null-terminated string in the memory buffer starting at the address buf. If the length of the null-terminated pathname is greater than buflen, the trailing characters and the null terminator are not stored. Length is the length of the fully qualified pathname excluding any null terminator, or –1 if phandle is invalid.
call-method
IN:  [string] method, ihandle, stack-arg1, ..., stack-argP
OUT:  catch-result, stack-result1, ..., stack-resultQ

Pushes two less than \( N\text{-args} \) items, \( stack-arg1, ..., stack-argP \), onto the Forth data stack, with \( stack-arg1 \) on top of the stack, and executes the package method named \( method \) in the instance \( ihandle \) as with \$call-method, guarded by catch. Pops the result returned by catch into catch-result. If that result is nonzero, restore the depth of the Forth data stack to its depth prior to the execution of call-method. If that result is zero, pops up to one less than \( N\text{-returns} \) items, \( stack-result1, ..., stack-resultQ \), from the Forth data stack into the returned values portion of the argument array, with \( stack-result1 \) corresponding to the top of the stack.

\( N\text{-args} \) and \( N\text{-returns} \) are stored in the argument array and may be different for different calls to call-method. If the number of items \( X \) left on the Forth data stack as a result of the execution of method is less than \( N\text{-returns} \), only \( stack-result1, ..., stack-resultX \) are modified; other elements of the returned values portion of the argument array are unaffected. If \( X \) is more than \( N\text{-returns} \), additional items are popped from the Forth data stack after setting \( stack-result1, ..., stack-resultQ \) so that, in all cases, the execution of call-method results in no net change to the depth of the Forth data stack.

An implementation shall allow at least six \( stack-arg \) and six \( stack-result \) items.

6.3.2.3 Device I/O

open
IN:  [string] device-specifier
OUT:  ihandle

Opens the package named by device-specifier as with open-dev, returning the instance identifier ihandle. Ihandle is zero if the operation fails.

The same package can be opened more than once if the particular package permits it, in which case a distinct ihandle will be returned each time.

close
IN:  ihandle
OUT:  none

Closes the instance identified by ihandle as with close-dev; subsequent use of that ihandle is invalid.

A client program should close instances it has opened after the instances are no longer needed, in order to release resources and to deactivate any associated devices.

read
IN:  ihandle, [address] addr, len
OUT:  actual

Executes the read method in the instance ihandle with arguments addr and len. Actual is either the value returned by that read method or –1 if that instance does not have a read method.

write
IN:  ihandle, [address] addr, len
OUT:  actual

Executes the write method in the instance ihandle with arguments addr and len. Actual is either the value returned by that write method or –1 if that instance does not have a write method.

seek
IN:  ihandle, pos.hi, pos.lo
OUT:  status

Executes the seek method in the instance ihandle with arguments pos.hi and pos.lo. Status is either the value returned by that seek method or –1 if that instance does not have a seek method.
6.3.2.4 Memory

claim
IN:  [address] virt, size, align
OUT:  [address] baseaddr
Allocates size bytes of memory. If align is zero, the allocated range begins at the virtual address virt. Otherwise, an aligned address is automatically chosen and the input argument virt is ignored. The alignment boundary is the smallest power of two greater than or equal to the value of align; an align value of 1 signifies 1-byte alignment. Base is the beginning address of the allocated memory (equal to virt if align was 0) or –1 if the operation fails (for example, if the requested virtual address is unavailable).

The range of physical memory and virtual addresses affected by this operation will be unavailable for subsequent mapping or allocation operations until freed by release.

release
IN:  [address] virt, size
OUT:  none
Frees size bytes of physical memory starting at virtual address virt, making that physical memory and the corresponding range of virtual address space available for later use. That memory must have been previously allocated by claim.

6.3.2.5 Control transfer

boot
IN:  [string] bootspec
OUT:  none
Exits the client program, resets the system (as with the command reset-all), and reboots the system with the device and arguments given by the null-terminated string bootspec. The string bootspec is interpreted in the same manner as the arguments of the command boot.

enter
IN:  none
OUT:  none
Enters the Open Firmware command interpreter (e.g., called by the operating system after a console input device abort). The client program may be resumed if the user continues execution with the go command.

exit
IN:  none
OUT:  none
Exits from the client program. The execution of the client program may not be resumed.

chain
IN:  [address] virt, size, [address] entry, [address] args, len
OUT:  none
Frees size bytes of memory starting at virtual address virt, then executes another client program beginning at address entry. The argument buffer args, len is copied into the Open Firmware memory and passed to the other program. The address of the arguments in the Open Firmware memory is the client program’s second argument, and their length is its third argument. chain is used to free any remaining memory for a secondary boot program and begin executing the booted program.

NOTE—The behavior of the chain client interface service includes the functions of init-program and go on behalf of the new client program, but does not include the functions of reading the client program into memory, parsing its header, or allocating its memory.
6.3.2.6 User interface

**interpret**

- **IN**: [string] cmd, stack-arg1, ..., stack-argP
- **OUT**: catch-result, stack-result1, ..., stack-resultQ

Pushes one less than \(N\)-args items, stack-arg1, ..., stack-argP, onto the Forth data stack, with stack-arg1 on top of the stack; executes the null-terminated string cmd as a Forth command line guarded by catch. Pops the result returned by catch into catch-result. If that result is nonzero, restore the depth of the Forth data stack to its depth prior to the execution of interpret. If that result is zero, pops up to one less than \(N\)-returns items, stack-result1, ..., stack-resultQ, from the Forth data stack into the returned values portion of the argument array, with stack-result1 corresponding to the top of the stack.

\(N\)-args and \(N\)-returns are stored in the argument array and may be different for different calls to interpret. If the number of items \(X\) left on the Forth data stack as a result of the execution of cmd is less than \(N\)-returns, only stack-result1, ..., stack-resultX are modified; other elements of the returned values portion of the argument array are unaffected. If \(X\) is more than \(N\)-returns, additional items are popped from the Forth data stack after setting stack-result1, ..., stack-resultQ so that, in all cases, the execution of interpret results in no net change to the depth of the Forth data stack.

An implementation shall allow at least six stack-arg and six stack-result items.

**set-callback**

- **IN**: [address] newfunc
- **OUT**: [address] oldfunc

Client programs may define a routine for handling the Open Firmware routines callback and sync. Newfunc is the address of the entry point of the callback routine. This service sets the callback handler to newfunc and returns as oldfunc the address of the entry point of the previously installed callback handler.

The Open Firmware shall use the same calling conventions specified in 6.3.1 for client interface services when calling the callback handler function. See callback and $callback glossary entries for details.

A client program callback handler shall return either a nonzero error code in the ret1 cell of the argument array if the service indicated by the service argument is unavailable, or zero otherwise. The client program callback handler shall return any additional results in the ret2 ... retN cells, setting \(N\)-returns to the total number of return values including the error code (or zero) that is in the ret1 cell. The handler shall not store more than \(M\) results, where \(M\) is the value that was in the \(N\)-returns cell when the handler was called, nor shall the returned value of \(N\)-returns exceed \(M\).
**set-symbol-lookup**

**IN:** [address] sym-to-value, [address] value-to-sym  
**OUT:** none

Sets the symbol table resolution **defer** words **sym>value** and **value>sym** so that they execute the client program callbacks whose addresses are given by the arguments **sym-to-value** and **value-to-sym**, respectively. If either argument is zero, the corresponding **defer** word is set to the action of **false**.

**sym-to-value** is called as follows:

**IN:** [string] symname  
**OUT:** error, symvalue

Searches for a symbol whose name is **symname**. If such a symbol is found, returns zero in **error** and the symbol’s value in **symvalue**. If no such symbol is found, returns –1 in **error** and zero in **symvalue**.

**value-to-sym** is called as follows:

**IN:** symvalue  
**OUT:** offset, [string] symname

Locates the symbol whose value is closest to but not greater than **symvalue** and returns **offset**, the non-negative offset from the value of that symbol to **symvalue**, and **symname**, the symbol name. If **symvalue** is less than the value of any known symbol, or is insufficiently close to any symbol value according to an implementation-dependent criterion, returns –1 in **offset** and the empty string in **symname**.

**set-symbol-lookup** is optional; it need be present only if the Open Firmware user interface is present and the Client Program Debugging **command group** (see 7.6) is implemented.

**6.3.2.7 Time**

**milliseconds**

**IN:** none  
**OUT:** ms

Returns a number that increases periodically, representing the passage of time in units of one millisecond. The granularity of this clock (i.e., the amount by which the number increases when it changes) is implementation-dependent.
7. User interface

The user interface allows a person to use Open Firmware services for such purposes as configuration management and debugging of hardware, software, and firmware. The interface consists of facilities for keyboard input, line editing, and display output, and an evaluator (the Forth command interpreter) for the Forth programming language.

7.1 General

The Open Firmware user interface specifies the behavior of a firmware system so that a human may interact with it for such purposes as configuration management, control of the booting process, and the debugging of hardware, client programs, device drivers, and the firmware itself.

7.1.1 Description

A standard command interpreter accepts and executes commands, typically entered interactively by a human, according to defined command editing, syntax, and semantic rules. A standard command interpreter is typically a component of the boot firmware associated with a CPU board.

A command group is a set of commands with defined behaviors, the group as a whole providing some particular capability (for example, one group of commands is concerned with client program debugging). Each command in the group may be executed via a standard command interpreter. This standard defines several such groups. Most such groups are optional. This clause lists the commands that comprise the individual command groups. The detailed specification of the commands themselves is given in annex A.

A standard program is a program, written in the language defined by the specification of the standard command interpreter in conjunction with the specification of one or more command groups, that obeys prescribed rules for program structure and usage. Consequently, its behavior is predictable when executed by a standard command interpreter. A standard program is typically either entered interactively by a human, downloaded from some storage device, or stored within the script (see 7.4.4.2).

7.1.2 Specification

In order to be compliant with the Open Firmware user interface, the boot firmware associated with a main CPU device shall implement a standard command interpreter that accepts user input as defined in this clause and one or more of the command groups defined herein (see figure 2). It should implement the Administration and the Forth Language command groups. It may implement additional commands that are not defined in this specification.

A command group implementation shall include all of the words and capabilities listed for that command group (except those words that are explicitly denoted as optional), and shall have behaviors as given.

* Command groups that should be implemented.
† The Forth Language command group shall be included in the FCode Debugging command group.
NOTE—Implementation of any one of the command groups shown is sufficient to claim compliance with the Open Firmware user interface.
Figure 2—Open Firmware user interface command groups

7.1.3 Warning

The Open Firmware user interface is not required to operate correctly after a client program has begun execution, because, in general, it is possible for a client program to modify system state in ways that are inconsistent with continued firmware operation.

7.2 Standard command interpreter

7.2.1 Command interpretation

Command lines are interpreted as specified by ANSI X3.215-1994 and 2.3.3, with the following clarifications:

— Although Open Firmware implementations are encouraged to leave the numeric conversion radix set to sixteen (hexadecimal) in normal operation, the implementation is not required to establish a particular radix before evaluating a Forth program or an FCode program, nor is it required to do so when executing device methods. Consequently, if a program requires a particular radix, it must explicitly set the radix (e.g., with decimal or hex).

— In accordance with ANSI X3.215-1994, an Open Firmware command interpreter treats numbers that end with a period (e.g., 123456.) as double numbers. Unlike ANSI X3.215-1994, which does not specify what happens with embedded periods, an Open Firmware command interpreter ignores “.” or “,” at other positions within numbers (e.g., 120.0000). Such embedded periods or commas may be used to make it easier for humans to read the numbers, but have no significance to the command interpreter. By convention, such periods or commas usually appear four digits from the right.

— At a given time, the process of searching for Forth words depends on whether or not there is an active package. If there is an active package, searching considers first the methods of that package, followed by globally visible Forth commands. If there is not an active package, searching considers only globally visible Forth commands.

7.2.2 Command-line editing

All keys typed by the user are echoed on the command line, except where noted. When the Return key (sometimes called the Enter key) is pressed, the edited line is presented to the command interpretation process.

The following keys edit the command line while it is being entered:

- Backspace: Erases the character before the cursor.
- Delete: Erases the character before the cursor.
- Control-U: Erases the entire line.
- Return (Enter): Finishes editing the line, making it available to the program.

7.2.3 Command-line editor extensions

The optional command-line editor extension provides additional command line editing capabilities.

These are user keystrokes, typed by the user while composing a command line. They allow a variety of convenient mechanisms for the user, including line editing, command history, and command completion.

The notation “^” means to hold down the Control key while typing the following character. “esc—” means to depress and release the “escape” key, then depress and release the following character.
7.2.3.1 Intraline editing

These command keystrokes alter the command line being typed. The editing cursor is considered to be between two adjacent characters. For displays that can only indicate the cursor position by highlighting a particular character, the character following the “true” cursor position should be highlighted.

Normal typed characters are inserted at the cursor position. Typing the Return key sends the entire visible line to be interpreted (regardless of the current cursor position).

Each of the following keystrokes shall perform the function of erasing the previous character: ^h, Delete key (if present), Backspace key (if present). If a system has both a Delete key and a Backspace key, each of these keys shall perform the “delete character” function.

<table>
<thead>
<tr>
<th>Keystroke</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>^b</td>
<td>Moves backward one character.</td>
</tr>
<tr>
<td>esc-b</td>
<td>Moves backward one word.</td>
</tr>
<tr>
<td>^f</td>
<td>Moves forward one character.</td>
</tr>
<tr>
<td>esc-f</td>
<td>Moves forward one word.</td>
</tr>
<tr>
<td>^a</td>
<td>Moves backward to beginning of line.</td>
</tr>
<tr>
<td>^e</td>
<td>Moves forward to end of line.</td>
</tr>
<tr>
<td>Delete</td>
<td>Erases previous character.</td>
</tr>
<tr>
<td>Backspace</td>
<td>Erases previous character.</td>
</tr>
<tr>
<td>^h</td>
<td>Erases previous character.</td>
</tr>
<tr>
<td>esc-h</td>
<td>Erases from beginning of word to just before the cursor, storing erased characters in a save buffer.</td>
</tr>
<tr>
<td>^w</td>
<td>Erases from beginning of word to just before the cursor, storing erased characters in a save buffer.</td>
</tr>
<tr>
<td>^d</td>
<td>Erases next character.</td>
</tr>
<tr>
<td>esc-d</td>
<td>Erases from cursor to end of the word, storing erased characters in a save buffer.</td>
</tr>
<tr>
<td>^k</td>
<td>Erases from cursor to end of line, storing erased characters in a save buffer.</td>
</tr>
<tr>
<td>^u</td>
<td>Erases entire line, storing erased characters in a save buffer.</td>
</tr>
<tr>
<td>^r</td>
<td>Retypes the line.</td>
</tr>
<tr>
<td>^q</td>
<td>Quotes next character (allows the insertion of control characters).</td>
</tr>
<tr>
<td>^y</td>
<td>Inserts the contents of the save buffer before the cursor.</td>
</tr>
</tbody>
</table>

7.2.3.2 Command-line history

These command keystrokes recall previously typed command lines. Once recalled, they may be edited and/or submitted for execution (by typing the Return key). At least eight previous command lines shall be saved.

<table>
<thead>
<tr>
<th>Keystroke</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>^p</td>
<td>Selects and displays the previous line for subsequent editing.</td>
</tr>
<tr>
<td>^n</td>
<td>Selects and displays the next line for subsequent editing.</td>
</tr>
<tr>
<td>^l</td>
<td>Displays the entire command history list.</td>
</tr>
</tbody>
</table>

7.2.3.3 Command completion

The command-completion function makes it easier for the user to enter long command names. After typing a portion of the desired word, typing the “completion” keystroke causes the system to search the dictionary of defined words, looking for word names beginning with the characters typed so far. If there is exactly one such word, the rest of the characters are filled in automatically. If there are several possibilities, the system fills in any additional characters that are common to all the candidates. If there are no defined word names starting with the given characters, characters are erased until there are candidates for the remaining characters.
Similarly, the “show” keystroke displays all words that begin with the letters supplied.

<table>
<thead>
<tr>
<th>Keystroke</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>^&lt;space&gt;</td>
<td>Complete this word.</td>
</tr>
<tr>
<td>^? or ^/</td>
<td>Show all possible matches.</td>
</tr>
</tbody>
</table>

## 7.3 Forth language command group

This clause does not attempt to describe how to use the Forth programming language; it simply lists the Forth commands that are required for conformance with this standard. More information about Forth may be found in the bibliography in ANSI X3.215-1994.

Commands in this subclause provide the basic Forth language structure. Many commands listed are also FCode functions. Unless stated otherwise, all commands shown may be used from the \( \text{ok} \) prompt within colon definitions and within downloaded Forth programs.

### 7.3.1 Stack

This subclause describes basic stack manipulation tools.

#### 7.3.1.1 Stack duplication

These commands duplicate stack items and have no other effect.

- **dup**
  \( (x -- x x) \) Duplicate the top item on the stack.
- **2dup**
  \( (x1 x2 -- x1 x2 x1 x2) \) Duplicate the top two items on the stack.
- **3dup**
  \( (x1 x2 x3 -- x1 x2 x3 x1 x2 x3) \) Duplicate three stack items.
- **?dup**
  \( (x -- 0 | x x) \) Duplicate top stack item if it is nonzero.
- **over**
  \( (x1 x2 -- x1 x2) \) Copy second stack item to top of stack.
- **2over**
  \( (x1 x2 x3 x4 -- x1 x2 x3 x4 x1 x2) \) Copy second pair of stack items to top of stack.
- **pick**
  \( (xu ... x1 x0 u -- xu ... x1 x0 xu) \) Copy \( u \)th stack item to top of stack.
- **tuck**
  \( (x1 x2 -- x2 x1 x2) \) Copy top stack item underneath the second stack item.

#### 7.3.1.2 Stack removal

These commands remove stack items and have no other effect.

- **clear**
  \( (...) -- \) Empty the stack.
- **drop**
  \( (x -- ) \) Remove top item from the stack.
- **2drop**
  \( (x1 x2 -- ) \) Remove top two items from the stack.
- **3drop**
  \( (x1 x2 x3 -- ) \) Remove top three items from the stack.
- **nip**
  \( (x1 x2 -- x2) \) Remove the second stack item.

#### 7.3.1.3 Stack rearrangement

These commands rearrange stack items and have no other effect.

- **roll**
  \( (xu ... x1 x0 u -- xu-1 ... x1 x0 xu) \) Rotate \( u+1 \) stack items as shown.
- **rot**
  \( (x1 x2 x3 -- x2 x3 x1) \) Rotate top three stack items as shown.
- **‐rot**
  \( (x1 x2 x3 -- x3 x1 x2) \) Rotate top three stack items as shown.
- **2rot**
  \( (x1 x2 x3 x4 x5 x6 -- x3 x4 x5 x6 x1 x2) \) Rotate three pairs of stack items as shown.
- **swap**
  \( (x1 x2 -- x2 x1) \) Exchange top two stack items.
- **2swap**
  \( (x1 x2 x3 x4 -- x3 x4 x1 x2) \) Exchange top two pairs of stack items.
### 7.3.1.4 Return stack

The return stack is a second stack used for system purposes. It may also be used as a temporary stack by the Forth programmer; however, there are strong restrictions on such usage as specified by ANSI X3.215-1994 (see 3.2.3.3).

- `>r` *(x --) (R:--x)* Move top stack item to the return stack.
- `r>` *(--x)(R:x--)* Move top return stack item to the stack.
- `r@` *(--x)(R:x--x)* Copy top return stack item to the stack.

### 7.3.1.5 Stack depth

The depth command is useful for error detection, by checking the total depth of the stack to find unintended stack effects.

- `depth` *(--u)* Return count of items on the stack.

### 7.3.2 Arithmetic

#### 7.3.2.1 Single-precision integer arithmetic

- `+` *(nu1 nu2--sum)* Add `nu1` to `nu2`.
- `-` *(nu1 nu2--diff)* Subtract `nu2` from `nu1`.
- `*` *(nu1 nu2--prod)* Multiply `nu1` by `nu2`.
- `u*` *(u1 u2--uprod)* Multiply `u1` by `u2` yielding `uprod`, all unsigned.
- `/` *(n1 n2--quot)* Divide `n1` by `n2`; return quotient.
- `*/` *(n1 n2 n3--quot)* Calculate `n1` times `n2` divided by `n3`.
- `mod` *(n1 n2--rem)* Divide `n1` by `n2`; return remainder.
- `/mod` *(n1 n2 n3--rem quot)* Divide `n1` by `n2`; return remainder and quotient.
- `*/mod` *(n1 n2 n3--rem quot)* Calculate `n1` times `n2` divided by `n3`.
- `u/mod` *(u1 u2--urem uquot)* Divide `n1` by `n2`, all unsigned.
- `1+` *(nu1--nu2)* Add 1 to `nu1`.
- `1-` *(nu1--nu2)* Subtract 1 from `nu1`.
- `2+` *(nu1--nu2)* Add 2 to `nu1`.
- `2-` *(nu1--nu2)* Subtract 2 from `nu1`.
- `abs` *(n--u)* Return absolute value of `n`.
- `negate` *(n1--n2)* Return negation of `n1`.
- `max` *(n1 n2--n1|n2)* Return greater of `n1` and `n2`.
- `min` *(n1 n2--n1|n2)* Return lesser of `n1` and `n2`.
- `bounds` *(n cnt--n+cnte n)* Prepare arguments for do or ?do loop.
- `even` *(n--n|n+n+1)* Round to nearest even integer _n_.

#### 7.3.2.2 Bitwise logical operators

- `lshift` *(x1 u--x2)* Shift `x1` left by `u` bit-places. Zero-fill low bits.
- `rshift` *(x1 u--x2)* Shift `x1` right by `u` bit-places. Zero-fill high bits.
- `>>a` *(x1 u--x2)* Arithmetic shift `x1` right by `u` bit-places.
- `<<` *(x1 u--x2)* Synonym for `lshift`.
- `>>` *(x1 u--x2)* Synonym for `rshift`.
- `2*` *(x1 --x2)* Shift `x1` left by one bit-place. Zero-fill low bit.
- `u2/` *(x1 --x2)* Shift `x1` right by one bit-place. Zero-fill high bit.
- `2/` *(x1 --x2)* Shift `x1` right by one bit-place. High bit unchanged.
- `and` *(x1 x2 -- x3)* Return bitwise logical “and” of `x1` and `x2`.
IEEE Std 1275-1994
IEEE STANDARD FOR BOOT (INITIALIZATION CONFIGURATION) FIRMWARE:

or

( x1 x2 -- x3 ) Return bitwise logical “inclusive-or” of x1 and x2.

xor

( x1 x2 -- x3 ) Return bitwise logical “exclusive-or” of x1 and x2.

invert

( x1 -- x2 ) Invert all bits of x1.

not

( x1 -- x2 ) Synonym for invert.

7.3.2.3 Double number arithmetic

Double numbers occupy two stack items. The most significant half of the double is always the topmost stack item.

s>d

( n1 -- d1 ) Convert a number to a double number.

d+

( d1 d2 -- d.sum ) Add d1 to d2 giving double number d.sum.

d−

( d1 d2 -- d.diff ) Subtract d2 from d1 giving double number d.diff.

um*

( u1 u2 -- ud ) Unsigned multiply with unsigned double number product.

m*

( n1 n2 -- d ) Signed multiply with double number product.

um/mod

( ud u -- urem uquot ) Divide ud by u.

fm/mod

( d n -- rem quot ) Divide d by n.

sm/rem

( d n -- rem quot ) Divide d by n, symmetric division.

7.3.2.4 Data type conversion

lbsplit

( quad -- b.lo b2 b3 b4.hi ) Split a quadlet into four bytes.

lwsplit

( quad -- w1.lo w2.hi ) Split a quadlet into two doublets.

wbsplit

( w -- b1.lo b2.hi ) Split a doublet into two bytes.

bljoin

( b1.lo b2 b3 b4.hi -- quad ) Join four bytes to form a quadlet.

bwjoin

( b.lo b.hi -- w ) Join two bytes to form a doublet.

wljoin

( w.lo w.hi -- quad ) Join two doublets to form a quadlet.

wbflip

( w1 -- w2 ) Swap the bytes within a doublet.

lbflip

( q1 -- q2 ) Reverse the bytes within a quadlet.

lwflip

( q1 -- q2 ) Swap the doublets within a quadlet.

7.3.2.5 Address arithmetic

/c

( -- n ) The number of address units to a byte: one.

/w

( -- n ) The number of address units to a doublet: typically, two.

/l

( -- n ) The number of address units to a quadlet: typically, four.

/n

( -- n ) The number of address units in a cell.

c+a

( addr1 index -- addr2 ) Increment addr1 by index times the value of /c.

w+a

( addr1 index -- addr2 ) Increment addr1 by index times the value of /w.

l+a

( addr1 index -- addr2 ) Increment addr1 by index times the value of /l.

n+a

( addr1 index -- addr2 ) Increment addr1 by index times the value of /n.

cal+

( addr1 -- addr2 ) Synonym for char+. A synonym for /c.

wal+

( addr1 -- addr2 ) Increment addr1 by the value of /w.

lal+

( addr1 -- addr2 ) Increment addr1 by the value of /l.

nal+

( addr1 -- addr2 ) Synonym for cell+. A synonym for /n.

c*/

( nu1 -- nu2 ) Multiply nu1 by the value of /c.

/w*

( nu1 -- nu2 ) Multiply nu1 by the value of /w.

/l*

( nu1 -- nu2 ) Multiply nu1 by the value of /l.

/n*

( nu1 -- nu2 ) Synonym for /n.

aligned

( n1 -- n|a-addr ) Increase n1 as necessary to give a var-aligned address.

char+

( addr1 -- addr2 ) Increment addr1 by the value of /c.

cell+

( addr1 -- addr2 ) Increment addr1 by index times the value of /n.

chars

( nu1 -- nu2 ) Multiply nu1 by the value of /c.

cells

( nu1 -- nu2 ) Multiply nu1 by the value of /n.

76
7.3.3 Memory control

7.3.3.1 Memory access

The following commands can be used to access memory. Devices can be accessed using the commands in 5.3.7.2. See 7.7.

@ (a-addr -- x) Fetch item x from address a-addr.
! (x a-addr --) Store item x to address a-addr.
2@ (a-addr -- x1 x2) Fetch two items from a-addr, item x2 from lower address.
2! (x1 x2 a-addr --) Store items x1 and x2 to a-addr, x2 at lower address.
c@ (addr -- byte) Fetch byte from addr.
c! (byte addr --) Store byte to addr.
w@ (waddr -- w) Fetch doublet w from waddr.
<w@ (waddr -- n) Fetch doublet w from waddr, sign-extended.
w! (w addr --) Store doublet w to waddr.
l@ (qaddr -- quad) Fetch quadlet from qaddr.
l! (quad qaddr --) Store quadlet to qaddr.
unaligned-w@ (addr -- w) Fetch doublet w from addr, any alignment is allowed.
unaligned-w! (w addr --) Store doublet w to addr, any alignment is allowed.
unaligned-l@ (addr -- quad) Fetch quadlet from addr, any alignment is allowed.
unaligned-l! (quad addr --) Store quadlet to addr, any alignment is allowed.
comp (addr1 addr2 len -- ?diff?) Compare two arrays of length len.
dump (addr len --) Display len bytes of memory starting at addr.
+! (nu a-addr --) Add nu to the number stored at address a-addr.
off (a-addr --) Store false at address a-addr.
on (a-addr --) Store true at address a-addr.
move (src-addr dest-addr len --) Copy len bytes from src-addr to dest-addr.
fill (addr len byte --) Set len bytes beginning at addr to the value byte.
blank (addr len --) Set len bytes beginning at addr to the value 0x20.
erase (addr len --) Set len bytes beginning at addr to zero.
wflips (waddr len --) Swap the bytes within each doublet in the given region.
lflips (qaddr len --) Reverse the bytes within each quadlet in the given region.
lwflips (qaddr len --) Swap the doublets within each quadlet in the given region.

7.3.3.2 Memory allocation

The following commands allocate and free regions of memory. The address can be used directly; it does not need to be mapped before use.

alloc-mem (len -- a-addr) Allocate len bytes of memory.
free-mem (a-addr len --) Free memory allocated by alloc-mem.

7.3.4 Text input and output

This subclause describes commands used for text input, output, and manipulation.

7.3.4.1 Text input

These commands parse text from the Forth input buffer. (See A.1.2.2 for the distinction between “text” and [text].)

( [text<] -- ) Ignore the immediately following text, up to closing “)”.
\ (rest-of-line<cr> -- ) Ignore the immediately following text on this line.
(continued)
7.3.4.2 Console input

These commands read characters from the console input device at execution time.

key? ( -- pressed? )  Return true if an input character is available.
key ( -- char )  Read a character from the console input device.
expect ( addr len -- )  Get an edited input line, storing it at addr.
span ( -- a-addr )  variable containing number of characters received by expect.
accept ( addr len1 -- len2 )  Get an edited input line, storing it at addr.

7.3.4.3 ASCII constants

These commands return the numeric values of particular characters.

bell ( -- 0x07 )  ASCII code for the Bell character.
bl ( -- 0x20 )  ASCII code for the Space (blank) character.
bs ( -- 0x08 )  ASCII code for the Backspace character.
carret ( -- 0x0D )  ASCII code for the Carriage-return character.
linefeed ( -- 0x0A )  ASCII code for the Linefeed character.
sic ( [text< >] -- char )  Generate ASCII code for the immediately following character.
char ( "text< >" -- char )  Generate ASCII code for the next character from input buffer.
[char] ( C: [text< >] -- char )  Generate ASCII code for the next character from input buffer.
control ( [text< >] -- char )  Generate control code for the immediately following character.

7.3.4.4 Console output

These commands display text on the console output device.

. " ( [text<"">] -- ) Display the immediately following text.
. ( [text<"] -- ) Display the immediately following text up to delimiting ")
emit ( char -- ) Display the given ASCII character.
type ( text-str text-len -- ) Display the text-len characters beginning at address text-str.

7.3.4.5 Output formatting

These commands control the positioning of displayed text on the console output device.

cr ( -- ) Subsequent output goes to the next line.
space ( -- ) Display a single space.
spaces ( cnt -- ) Display cnt spaces.
#line ( -- a-addr ) variable holding the output line number.
#out ( -- a-addr ) variable holding the output column number.
7.3.4.6 Display pause

This command is a tool that a routine can use to provide user-controlled pagination of its multiline output.

```lisp
exit? ( -- done? ) Return true when output should be terminated.
```

7.3.4.7 String literals

```
" ( [text">">] -- text-str text-len ) Gather the immediately following string or hex data.

s" ( [text"> ] -- text-str text-len ) Gather the immediately following string.
```

7.3.4.8 String manipulation

```
count ( pstr -- str len ) Unpack a counted string to a text string.
pack ( str len addr -- pstr ) Pack a text string into a counted string.
lcc ( char1 -- char2 ) Convert ASCII char1 to lowercase.
upc ( char1 -- char2 ) Convert ASCII char1 to uppercase.
-trailing ( str len1 -- str len2 ) Remove trailing spaces from string.
```

7.3.5 Numeric input and output

7.3.5.1 Numeric-base control

These commands control the numeric radix for input and output conversion, i.e., hex, decimal, etc.

```lisp
base ( -- a-addr ) variable containing the numeric conversion radix.
decimal ( -- ) Set numeric conversion radix to ten.
hex ( -- ) Set numeric conversion radix to sixteen.
octal ( -- ) Set numeric conversion radix to eight.
```

7.3.5.2 Numeric input

```
$number ( addr len -- true | n false ) Convert a string to a number.
>number ( d1 str1 len1 -- d2 str2 len2 ) Convert string to a number; add to d1.
digit ( char base -- digit true | char false ) Convert a character to a digit in the given base.
d# ( [number< >] -- n ) Interpret the following number as a decimal number (base ten).
h# ( [number< >] -- n ) Interpret the following number as a hexadecimal number (base sixteen).
o# ( [number< >] -- n ) Interpret the following number as an octal number (base eight).
```

7.3.5.3 Numeric output

```
. ( nu -- ) Display number, with a trailing space.
s. ( n -- ) Display a signed number, with a trailing space.
u. ( u -- ) Display an unsigned number, with a trailing space.
.r ( n size -- ) Display a signed number, right-justified.
u.r ( u size -- ) Display an unsigned number, right-justified.
.d ( n -- ) Display a signed number (and space) in decimal.
.h ( n -- ) Display a signed number (and space) in hex.
s ( ... -- ... ) Display entire stack contents, unchanged.
? ( a-addr -- ) Display the number at address a-addr.
```
7.3.5.4 Numeric output primitives

These commands give precise control of number output format.

- \( \text{(. \cdot n \rightarrow \text{str len})} \): Convert a number into a text string.
- \( \text{\( \text{(u\cdot u \rightarrow \text{str len})} \): Convert an unsigned number into a text string.
- \( \text{\# (ud1 -- ud2)} \): Initialize pictured numeric output conversion.
- \( \text{\# (ud -- \text{str len})} \): Convert a digit in pictured numeric output conversion.
- \( \text{\#> (ud -- \text{str len})} \): Convert remaining digits in pictured numeric output.
- \( \text{\text{hold (char --)}} \): Add char in pictured numeric output conversion.
- \( \text{\text{sign (n --)}} \): If \( n < 0 \), insert \"-\" in pictured numeric output.
- \( \text{\text{u# (u1 -- u2)}} \): Convert a digit in pictured numeric output conversion.
- \( \text{\text{u#s (u \rightarrow 0)}} \): Convert remaining digits in pictured numeric output.
- \( \text{\text{u#> (u \rightarrow \text{str len})}} \): End pictured numeric output conversion.

7.3.6 Comparison operators

- \( \text{\text{< (n1 n2 \rightarrow \text{less?})}} \): Return true if \( n1 \) is less than \( n2 \).
- \( \text{\text{\( (n1 n2 \rightarrow \text{less-or-equal?})} \): Return true if \( n1 \) is less than or equal to \( n2 \).
- \( \text{\text{\( (x1 x2 \rightarrow \text{not-equal?})} \): Return true if \( x1 \) is not equal to \( x2 \).
- \( \text{\text{\( (x1 x2 \rightarrow \text{equal?})} \): Return true if \( x1 \) is equal to \( x2 \).
- \( \text{\text{\( (n1 n2 \rightarrow \text{greater?})} \): Return true if \( n1 \) is greater than \( n2 \).
- \( \text{\text{\( (n1 n2 \rightarrow \text{greater-or-equal?})} \): Return true if \( n1 \) is greater than or equal to \( n2 \).
- \( \text{\text{\( (n \rightarrow \text{less-than-0?})} \): Return true if \( n \) is less than zero.
- \( \text{\text{\( (n \rightarrow \text{less-or-equal-to-0?})} \): Return true if \( n \) is less than or equal to zero.
- \( \text{\text{\( (n \rightarrow \text{not-equal-to-0?})} \): Return true if \( n \) is not equal to zero.
- \( \text{\text{\( (n \rightarrow \text{greater-than-0?})} \): Return true if \( n \) is greater than zero.
- \( \text{\text{\( (n \rightarrow \text{greater-or-equal-to-0?})} \): Return true if \( n \) is greater than or equal to zero.
- \( \text{\text{\( (u1 u2 \rightarrow \text{unsigned-less?})} \): Return true if \( u1 \) is less than \( u2 \), unsigned.
- \( \text{\text{\( (u1 u2 \rightarrow \text{unsigned-less-or-equal?})} \): Return true if \( u1 \) less or equal to \( u2 \), unsigned.
- \( \text{\text{\( (u1 u2 \rightarrow \text{unsigned-greater?})} \): Return true if \( u1 \) greater than \( u2 \), unsigned.
- \( \text{\text{\( (u1 u2 \rightarrow \text{unsigned-greater-or-equal?})} \): Return true if \( u1 \) greater or equal to \( u2 \), unsigned.

7.3.7 Flag constants

- \( \text{false (\rightarrow \text{false})} \): Return the value false (zero).
- \( \text{true (\rightarrow \text{true})} \): Return the value true (negative one).

7.3.8 Control-flow commands

This subclause describes commands which alter the program flow. This includes branches, loops, error handling, and other execution-control commands.

These commands can be used either within definitions or interactively.
7.3.8.1 Conditional branches

These commands provide a basic “if-then-else” branching capability.

```plaintext
if ( do-next? -- ) When flag is true, execute following code.
  (C: -- orig-sys )
else ( -- ) When if flag is false, execute following code.
  (C: orig-sys1 -- orig-sys2 )
then ( -- ) Terminate an if construct.
  (C: orig-sys -- )
```

7.3.8.2 Case statement

These commands provide an “n-way” branching capability.

```plaintext
case ( sel -- sel ) Begin a case (multiple selection) statement.
  (C: -- case-sys )
of ( sel of-val -- sel | <nothing> ) Begin of clause; execute through endof if params match.
  (C: case-sys1 -- case-sys2 of-sys )
endof ( -- ) Mark end of clause; jump to end of case if match.
  (C: case-sys1 of-sys -- case-sys2 )
endcase ( sel | <nothing> -- ) Mark end of a case statement.
  (C: case-sys -- )
```

7.3.8.3 Conditional loops

These commands loop until a specified condition is met.

```plaintext
begin ( -- ) Begin a conditional loop.
  (C: -- dest-sys )
until ( done? -- ) End a begin...until loop; exits loop if flag is true.
  (C: dest-sys -- )
again ( -- ) End an (infinite) begin...again loop.
  (C: dest-sys -- )
while ( continue? -- ) Conditional test within begin...while...repeat loop.
  (C: dest-sys -- orig-sys dest-sys )
repeat ( -- ) End a begin...while...repeat loop; jump to begin.
  (C: orig-sys dest-sys -- )
```

7.3.8.4 Counted loops

These commands loop for a specified number of iterations, maintaining an induction variable that may be read from within the loop.

```plaintext
do ( limit start -- ) (R: -- sys ) Start a counted loop; beginning index value is start.
  (C: -- destodest )
?do ( limit start -- ) (R: -- sys ) Similar to do, but do not execute loop if limit = start.
  (C: -- destodest )
loop ( -- ) (R: sys1 -- <nothing> | sys2 ) Add one to index; return to the previous do or exit the loop.
  (C: destodest -- )
+loop ( delta -- ) (R: sys1 -- <nothing> | sys2 ) Add delta to index; return to the previous do or exit the loop.
  (C: sys -- )
```

(continued)
7.3.8.5 Other control flow commands

- **eval**: Synonym for `evaluate`.
- **evaluate**: Interpret Forth text from the given string.
- **execute**: Execute the command whose execution token is `xt`.
- **exit**: Exit from the currently executing command.

7.3.8.6 Error handling

These commands can transfer control across multiple levels of procedure nesting.

- **quit**: Abort program execution.
- **abort**: Abort program execution; clear stacks.
- **abort"**: If flag is `true`, display text and call `abort`.
- **catch**: Execute command indicated by `xt`; return `throw` result.
- **throw**: Transfer back to `catch` routine if `error-code` is nonzero.

7.3.9 Forth dictionary

This section describes commands used to create and find Forth definitions and data. They are grouped into two broad categories: defining words and dictionary commands.

7.3.9.1 Defining words

Defining words are commands that create other Forth commands.

If a Forth command is created with the same name as an existing command, the new command will be created normally. A warning message "`xyz isn't unique`" may optionally be displayed. Previous uses of that command name will be unaffected. Subsequent uses of that command name will use the latest definition of that command name.

- **constant**: Create a named constant; `new-name` returns value `x`.
- **2constant**: Create a named two-number constant.
- **value**: Create a named variable; change with `to`.
- **variable**: Create a named variable; `new-name` returns address `a-addr`.
- **buffer**: Creates a named data buffer; `new-name` returns address.
CORE REQUIREMENTS AND PRACTICES

: ( "new-name< >" -- colon-sys) Begin creation of a colon definition.
    (E: ... -- ???)

; ( colon-sys -- ) End creation of a colon definition.

alias ( "new-name< >old-name< >" -- ) Create a new command equivalent to an existing command.
    (E: ... -- ???)

defer ( "new-name< >" -- ) Create a command with alterable behavior; alter with to.
    (E: ... -- ???)

struct ( -- 0 ) Start a struct...field definition.

field ( offset size "new-name< >" -- offset+size ) Create new field offset specifier, named new-name.
    (E: addr -- addr+offset )

create ( "new-name< >" -- ) Create a new command; behavior set by further commands.
    (E: ... -- ???)

does> (C: colon-sys1 -- colon-sys2 ) Specify run-time behavior of a created word.
    ( -- ) (R: sys1 -- )
    ( ... -- a-addr ) (R: -- sys2 )
    (E: ... -- ???)

$create ( name-str name-len -- ) Call create; new name specified by name string.

forget ( "old-name< >" -- ) Remove command old-name and all subsequent definitions.

7.3.9.2 Dictionary commands

Dictionary commands control various aspects of the Forth dictionary.

7.3.9.2.1 Data space allocation

These commands allocate and initialize memory at the top of the data space.

here ( -- addr ) Return current dictionary pointer.
allot ( len -- ) Allocate len bytes in the dictionary.
align ( -- ) Allocate dictionary bytes to leave top of dictionary var-aligned.
c, ( byte -- ) Compile a byte into the dictionary.
w, ( w -- ) Compile a doublet w into the dictionary (doublet-aligned).
l, ( quad -- ) Compile a quadlet into the dictionary (doublet-aligned).
, ( x -- ) Append x to data space.

7.3.9.2.2 Immediate words

Most Forth commands, when encountered within a colon definition, are compiled for later use. When the colon definition is later executed, the Forth commands compiled within are then executed. Immediate words, in contrast, are executed immediately, even when encountered within a colon definition.

immediate ( -- ) Declare the previous definition as “immediate”.
state ( -- a-addr ) variable containing true if in compile state.

(continued)
7.3.9.2.3 Dictionary search

These commands are all variations of the same idea: find a command in the dictionary (given its name) and return the execution token of the word and/or other information. The portions of the dictionary that are visible at a particular time are controlled by the search order (see 3.2.2.4 and 7.5.3.1 for additional commands).

\[ \text{[]} \text{[old-name< >]} \text{-- xt} \text{]} \text{Return execution token xt of a command.}' \\
\text{'} \text{[old-name< >]} \text{-- xt} \text{)} \text{Return execution token xt of a command, parsed later.'} \\
\text{find} \text{[pstr -- xt n | pstr false)} \text{Find command, return –1 (found), +1 (immediate), or 0 (not found).}

7.3.9.2.4 Miscellaneous dictionary

This subclause contains other dictionary-related commands.

to \text{[param [old-name< >]} \text{--)} \text{Change value or defer or machine register contents.} \\
behavior \text{[defer-xt -- contents-xt]} \text{Retrieve execution behavior of a defer word.} \\
>body \text{[xt -- a-addr]} \text{Convert execution token to data field address.} \\
body> \text{[a-addr -- xt]} \text{Convert data field address to execution token.} \\
noop \text{[--]} \text{Do nothing.} \\
recursive \text{[--]} \text{Make current definition visible, for recursive call.} \\
recurse \text{[... -- ???) \text{Compile recursive call to the command being compiled.} \\
forth \text{[--]} \text{Make Forth the context vocabulary.} \\
environment? \text{[str len -- false | value true]} \text{Return system information based on input keyword.}

7.3.9.3 Assembler

The assembler permits machine-level code definitions to be created and executed by the user at the interactive command interpreter level. A machine-code definition, once created, can be executed or used in subsequent definitions, just like any other command. The assembler mnemonics depend on the processor instruction set. The commands to invoke and exit the assembler are the same for all processors.

The assembler mnemonics are processor- and implementation-dependent and are not specified in this document. Creation of a machine-code definition also depends on specific knowledge of the details of the particular Forth implementation. Register usage, etc., by the Forth implementation is not specified in this document.

code \text{["new-name< >" -- code-sys)} \text{Begin creation of machine-code command called new-name.} \\
label \text{["new-name< >" -- code-sys)} \text{Begin machine-code sequence; leave addr on stack.} \\
c; \text{[code-sys --]} \text{End creation of machine-code command; will return to caller.} \\
end-code \text{[code-sys --]} \text{End creation of machine-code sequence.}
7.4 Administration command group

Commands in this subclause relate to booting and system configuration. These commands generally work from the ok prompt, but are not designed for use within Forth programs. An Open Firmware implementation that includes the Administration command group shall implement the /aliases and /options standard system nodes.

7.4.1 Help

The help function displays messages describing names and usages for various Open Firmware commands and configuration variables. This command may be invoked in three basic ways: general, by name, and by category.

Examples:

```
ok help<cr>
Enter "help command-name" or "help category-name" for more help
(Use ONLY the first word of a category description)
Examples: help select -or- help line
    Main categories are:
    File download and boot
    Resume execution
    Diag (diagnostic routines)
```

etc.

```
ok help +bp<cr>
+bp   ( addr -- )  add a breakpoint at the given address
```

The specific set of commands described by help is system-dependent. That set should include at least the most commonly used commands, and may include other commands as space permits.

```
help   ( "{name}<cr>" -- ) Provide information for category or specific command.
```

7.4.2 System start-up

Following is a quick summary of the Open Firmware start-up sequence after power-on or reset. All of the commands mentioned here are described in more detail later in this clause and in annex A.

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
h) Default boot (if auto-boot? is true)
i) Invoke the command interpreter (if the preceding step returns)

It is sometimes desirable to modify the sequence “probe-all install-console banner”. For example, commands that modify the characteristics of plug-in devices might need to be executed after the plug-in devices have been probed but before the console device has been selected. Such commands need to be executed between probe-all and install-console. Commands that display output on the console need to be placed after install-console or banner. This is accomplished by creating a custom script. To accommodate such customized script sequences, the sequence “probe-all install-console banner” is not executed if either banner or suppress-banner is executed from the script. This allows the use of probe-all,
install-console and banner inside the script, possibly interspersed with other commands, without having those commands re-executed after the script finishes.

7.4.3 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.

7.4.3.1 Overview

The booting process proceeds as follows. A device is selected for booting. A program is read from that device into memory, using a protocol that depends on the type of device, and is executed. Further behavior of that program may be controlled by an argument string that is made available to the program by the Open Firmware. 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 “/boot” program, which then might use the Network File System (NFS) protocol to load the operating system from a file named “/vmunix”.

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, it is important to recognize that from Open Firmware’s point of view the boot arguments are an opaque string that is passed uninterpreted to the boot program.

7.4.3.2 Device and argument selection

The automatic booting process is controlled by configuration variables as follows:

— If auto-boot? is false (its default value is true), automatic booting does not occur and the interactive command interpreter is invoked.

— Otherwise, the command specified by the boot-command configuration variable is executed. The default value of boot-command is the command boot with no command-line arguments. In that case, if diagnostic-mode? returns false, the default boot device is given by boot-device and the default boot arguments are given by boot-file; if diagnostic-mode? returns true, the default boot device is given by diag-device and the default boot arguments are given by diag-file.

The user can explicitly execute the boot command from the command interpreter, in which case the user can either supply explicit command-line arguments or omit them so that the default arguments will be used.

7.4.3.3 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.

7.4.3.4 Argument passing

The device path of the boot device is given by the value of the “bootpath” property in the /chosen node. This lets client programs determine the device from which they were booted.
The boot arguments are given by the “`bootargs`” property in the `/chosen` node.

### 7.4.3.5 User commands for booting

The syntax for the `boot` command is ambiguous because a `device` alias cannot be syntactically distinguished from the arguments. The ambiguity is resolved as follows:

- If the word following `boot` on the command line begins with a “/”, the word is a `device path` and, thus, a `device specifier`.
- Otherwise, if there is a device alias matching that word, the word is a device specifier.
- If that word is neither a device path nor a known alias, the default boot device is used and the word is included in arguments.

Assuming that `disk0` has been predefined as a device alias for some device path, the following are examples of valid boot commands:

```bash
ok boot<cr> \ default boot (values specified in configuration variables)
ok boot disk0<cr> \ boot from disk0; pass boot program default arguments
ok boot disk0 vmunix -asw<cr> \ boot from disk0; pass boot program “vmunix -asw”
ok boot vmunix -asw<cr> \ boot from default dev; pass boot program “vmunix -asw”
```

If the `boot` command is executed after a `client program` has already been run, the `boot` command may reset the machine as with `reset-all`, thus reinitializing the hardware state and Open Firmware’s data structures before proceeding with the booting process. This is necessary because the client program may have modified the machine’s state in ways that the Open Firmware cannot “undo” without a “hard reset” and the machine’s current state could prevent a boot from succeeding.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>boot</code></td>
<td>Load and execute a program specified by <code>param-text</code>.</td>
</tr>
<tr>
<td><code>diagnostic-mode?</code></td>
<td>If <code>true</code>, boot from diag sources; perform longer self-tests.</td>
</tr>
<tr>
<td><code>diag-switch?</code></td>
<td>If <code>true</code>, <code>diagnostic-mode?</code> returns <code>true</code>.</td>
</tr>
<tr>
<td><code>boot-device</code></td>
<td>Default <code>boot device-name</code> (<code>diagnostic-mode? false</code>).</td>
</tr>
<tr>
<td><code>boot-file</code></td>
<td>Default <code>boot arguments</code> (<code>diagnostic-mode? false</code>).</td>
</tr>
<tr>
<td><code>diag-device</code></td>
<td>Default <code>boot device-name</code> (<code>diagnostic-mode? true</code>).</td>
</tr>
<tr>
<td><code>diag-file</code></td>
<td>Default <code>boot arguments</code> (<code>diagnostic-mode? true</code>).</td>
</tr>
<tr>
<td><code>auto-boot?</code></td>
<td>If <code>true</code>, automatically execute <code>boot-command</code> after power-on or <code>reset-all</code>.</td>
</tr>
<tr>
<td><code>boot-command</code></td>
<td>Command executed if <code>auto-boot?</code> is <code>true</code>.</td>
</tr>
</tbody>
</table>

### 7.4.4 Nonvolatile memory

System nonvolatile memory is used to preserve system start-up and booting information. This information may be saved in EEPROM, battery-backed RAM, or some other device. The key features are that it can be accessed by the Open Firmware during system start-up, it can be changed by the user as desired, and its contents persist when power is off.

System nonvolatile memory is divided into two sections, one for the storage of `configuration variables`, and the other for the `script`.

#### 7.4.4.1 Configuration variables

`Configuration variables` are an optional feature of the Administrative `command group`.

A number of Open Firmware operating characteristics are controlled by configuration variables stored in nonvolatile memory. The value of a configuration variable can be a number, a string, a true/false flag, a selection
from a set of choices, or one of several other data types, depending on the particular variable. Most configuration variables have both a current value and a default value, with the default value stored in ROM. Open Firmware implementations can maintain a checksum of the nonvolatile memory used for configuration variable storage. If that memory becomes corrupted, the Open Firmware implementation can restore some of the configuration variables to their default values, leaving untouched those configuration variables without default values.

The nonvolatile memory locations where particular parameter values are stored can change from machine to machine and from revision to revision. Thus, they cannot be accessed at fixed locations, but instead must be accessed by name. Users can access them by name using `printenv` and `setenv`. `Client programs` can access them by name via `client interface` operations on the `/options device node`.

The configuration variable data types are given in the following list. Each data type is described in terms of its “Fundamental Data Type,” which is the kind of information that it represents, its “Stack Representation,” which is the way that information is presented on the Forth stack, its “Input Text Representation,” which is the human-writeable form as used with `setenv`, `$setenv`, and `setprop`, and its “Output Text Representation,” which is its human-readable form as used with `printenv` and `getprop`. The internal storage format is not specified.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Fundamental Data Type</th>
<th>Stack Representation</th>
<th>Input Text Representation</th>
<th>Output Text Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>number</td>
<td><code>n</code></td>
<td>decimal number or hexadecimal number beginning with “0x”</td>
<td>decimal number</td>
</tr>
<tr>
<td>bytes[n]</td>
<td>a sequence of <code>n</code> bytes</td>
<td><code>addr len</code></td>
<td>a sequence of <code>n</code> bytes</td>
<td>a sequence of <code>n</code> bytes</td>
</tr>
<tr>
<td>string[n]</td>
<td>text string capable of storing at least <code>n</code> characters</td>
<td><code>addr len</code></td>
<td>text</td>
<td>text</td>
</tr>
<tr>
<td>boolean</td>
<td>true/false flag</td>
<td><code>flag</code></td>
<td>one of the strings: <code>true</code>, <code>false</code>, <code>TRUE</code>, or <code>FALSE</code></td>
<td>one of the strings: <code>true</code> or <code>false</code></td>
</tr>
<tr>
<td>security-mode</td>
<td>enumerated type</td>
<td>none</td>
<td>one of the strings: <code>none</code>, <code>command</code>, or <code>full</code></td>
<td>one of the strings: <code>none</code>, <code>command</code>, or <code>full</code></td>
</tr>
</tbody>
</table>

Configuration variables are described in the following subclauses along with the features that require them. In implementations that do not support configuration variables, features that depend on particular configuration variables may use fixed default values for those variables.
The following commands inspect and modify configuration variables:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>setenv</code></td>
<td>(&quot;nv-param&lt; &gt;new-value&lt;eol&gt;&quot; -- ) Set the configuration variable <code>nv-param</code> to the indicated value.</td>
</tr>
<tr>
<td><code>$setenv</code></td>
<td>( data-addr data-len name-str name-len -- ) Set the configuration variable name string to new value.</td>
</tr>
<tr>
<td><code>printenv</code></td>
<td>(&quot;[param-name]&lt;eol&gt;&quot; -- ) Display current, default value of configuration variable (or all).</td>
</tr>
<tr>
<td><code>set-default</code></td>
<td>(&quot;param-name&lt;eol&gt;&quot; -- ) Set configuration variable to default value.</td>
</tr>
<tr>
<td><code>set-defaults</code></td>
<td>( -- ) Reset most configuration variables to their default values.</td>
</tr>
<tr>
<td><code>nodefault-bytes</code></td>
<td>( maxlen &quot;new-name&lt; &gt;&quot; -- ) Create custom configuration variable of size maxlen.</td>
</tr>
</tbody>
</table>

A summary list of all Open Firmware defined configuration variables is given in annex G. Note that implementations are free to define additional configuration variables as needed.

7.4.4.2 The script

The script is a section of nonvolatile memory that is reserved for storage of user-defined commands to be executed during the start-up sequence. The script may be used for various purposes, including installation-specific device configuration commands and device aliases, patches to correct Open Firmware bugs, or user-installed extensions. Commands in the script are stored in text form, just as the user would type them at the console.

The script is evaluated only if `use-nvramrc?` is `true`.

The commands `nvedit` and `nvstore` are used to edit the script. The intraline editing keystrokes are used within the script editor, with the following exceptions:

<table>
<thead>
<tr>
<th>Keystroke</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>^c</code></td>
<td>Exits the script editor, returning to the Open Firmware command interpreter. The temporary buffer is preserved but is not written back to the script. (Use <code>nvstore</code> afterwards to write it back.)</td>
</tr>
<tr>
<td><code>&lt;cr&gt;</code></td>
<td>Inserts a newline at the cursor position and advances to the next line.</td>
</tr>
<tr>
<td><code>^o</code></td>
<td>Inserts a newline at the cursor position and stays on the current line.</td>
</tr>
<tr>
<td><code>^k</code></td>
<td>If at the end of a line, joins the next line to the current line (i.e., deletes the newline).</td>
</tr>
<tr>
<td><code>^n</code></td>
<td>Moves to the next line of the script editing buffer.</td>
</tr>
<tr>
<td><code>^p</code></td>
<td>Moves to the previous line of the script editing buffer.</td>
</tr>
<tr>
<td><code>^l</code></td>
<td>Displays the entire contents of the editing buffer.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>nvramrc</code></td>
<td>( -- data-addr data-len )Contents of the script.</td>
</tr>
<tr>
<td><code>use-nvramrc?</code></td>
<td>( -- enabled? )If <code>true</code>, the script is evaluated at system start-up.</td>
</tr>
<tr>
<td><code>nvedit</code></td>
<td>( -- )Enter script editor (exit with <code>^c</code>).</td>
</tr>
<tr>
<td><code>nvstore</code></td>
<td>( -- )Copy contents of <code>nvedit</code> temporary buffer into the script.</td>
</tr>
<tr>
<td><code>nvquit</code></td>
<td>( -- )Discard contents of <code>nvedit</code> temporary buffer.</td>
</tr>
<tr>
<td><code>nvrecover</code></td>
<td>( -- )Attempt to recover lost script contents.</td>
</tr>
<tr>
<td><code>nvrun</code></td>
<td>( -- )Execute the contents of the <code>nvedit</code> temporary buffer.</td>
</tr>
</tbody>
</table>

7.4.5 I/O control

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.

After probing, the drivers for devices named by `input-device` and `output-device` are opened, and console input and output is directed to those devices. The `ihandle` s of the open input and output drivers are saved as the values of the "stdin" and "stdout" properties in the `/chosen` node, so that client programs may interact with the user through the console.
If either of the specified devices cannot be opened, system-dependent default devices may be used instead of the specified devices.

The console activation process is performed by the `install-console` command.

Normally, `install-console` is automatically executed during the Open Firmware start-up sequence just after probing, but it may be executed explicitly from the `script` if desired.

Before selecting the output device, `install-console` attempts to create a `device alias` named `screen`. If a device alias named `screen` does not exist, and if a device of type `display` was found during probing, `install-console` creates an alias `screen` representing the `device path` of the first device of type `display` that was found during probing. This feature provides a degree of autoconfiguration. Typically, a system will have only one display device. If the value of `output-device` is set to `screen`, the system will automatically locate that device and use it as the console output device.

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

The input and output devices may also be modified with the `input`, `output`, and `io` commands. These modifications take place immediately. These commands do not affect `input-device` and `output-device` and hence do not affect the choice of console after a subsequent `reset-all` or power cycle (unless present in the `script`). The action taken on failure to open an input or output device is system-dependent.
input-device ( -- dev-str dev-len ) Default console input device.
output-device ( -- dev-str dev-len ) Default console output device.
stdin ( -- a-addr ) variable containing the ihandle of the console input device.
stdout ( -- a-addr ) variable containing the ihandle of the console output device.
screen-#columns ( -- n ) Maximum number of columns on console output device.
screen-#rows ( -- n ) Maximum number of rows on console output device.
install-console ( -- ) Select and activate console input and output devices.

input ( dev-str dev-len -- ) Select the indicated device for console input.
output ( dev-str dev-len -- ) Select the indicated device for console output.
io ( dev-str dev-len -- ) Select the indicated device for console input and output.

7.4.6 Security

The security feature allows the system to be configured so that a password is required to access most commands from the interactive command interpreter’s ok prompt. Several levels of security (including “none”) are specified by the setting of security-mode. The password command is used to set the security password.

password ( -- ) Prompt user to set security password.
security-mode ( -- n ) Contains level of security access protection.
security-password ( -- password-str password-len ) Contains security password text string.
security-#badlogins ( -- n ) Contains total count of invalid security access attempts.

7.4.7 Reset

This command is used to initiate a system power-on reset, thus re-initializing the hardware state and Open Firmware’s data structures as if a power-on reset had occurred.

reset-all ( -- ) Reset the machine as if a power-on reset had occurred.

7.4.8 Self-test

Any device node (either for a built-in device or a plug-in device) may define its own selftest diagnostic routine as one of that device’s methods. Many devices have two levels of diagnostics. The simplest level is a very brief “sanity check” that could be automatically executed during initial probing or whenever that device is opened for use by Open Firmware. The more extensive selftest routine is executed only upon user command. Execution of a device’s selftest routine may be automated by storing the user command in the script.

test ( “device-specifier<cr>” -- ) Invoke the selftest routine for the specified device.
test-all ( “{device-specifier}<cr>” -- ) Invoke selftest routines at and below specified node.
selftest-#megs ( -- n ) Number of megabytes of memory to test.
diagnostic-mode? ( -- diag? ) If true, boot from diag sources, perform longer self-tests.

7.4.9 Client program callback

A client program may make available a set of application callback commands that the user may execute from the Open Firmware command interpreter. The client program declares the address of its callback procedure with set-callback. Application callback commands can be executed with the following Open Firmware commands:
callback ( “service-name< >” “arguments<cr>” -- ) Execute specified client program callback routine.
$callback ( argn ... arg1 nargs addr len -- retn ... ret2 Nreturns-1 ) Execute specified client program callback routine.
IEEE Std 1275-1994 IEEE STANDARD FOR BOOT (INITIALIZATION CONFIGURATION) FIRMWARE:

**sync**

Flush system file buffers, after a program interrupt.

### 7.4.10 Banner

These commands control the appearance of the *firmware* banner:

- **banner**
  
  Display the system power-on banner.

- **suppress-banner**
  
  Abbreviate system start-up sequence after the *script*.

- **oem-logo?**
  
  If true, *banner* displays custom logo in *oem-logo*.

- **oem-logo**
  
  Contains custom logo for *banner*, enabled by *oem-logo*.

- **oem-banner?**
  
  If true, *banner* displays custom message in *oem-banner*.

- **oem-banner**
  
  Contains custom *banner* text, enabled by *oem-banner*.

### 7.4.11 Device tree

The *show-devs* command is a useful tool to see the full *device path* names of all or part of the *device tree*. The following is an example of its output:

```
ok show-devs<cr>
/audio@1,f720100
/sbus@1,f8000000
/zs@1,f0000000
/packages@0,0
/sbus@1,f8000000/SUNW,le@0,c00000
/sbus@1,f8000000/SUNW,esp@0,80000
/sbus@1,f8000000/SUNW,esp@0,800000/sd@1,0
/sbus@1,f8000000/SUNW,esp@0,8000000/st@5,0
/sbus@1,f8000000/SUNW, cgthree@3,0
```

**show-devs**

```
("{device-specifier}<cr>" -- ) Show all devices beneath the indicated node.
```

#### 7.4.11.1 Device alias

- **devalias**
  
  Create device alias or display current alias(es).

- **nvalias**
  
  Create nonvolatile device alias; edit the *script*.

- **$nvalias**
  
  Create nonvolatile device alias; edit the *script*.

- **nvunalias**
  
  Delete nonvolatile device alias; edit the *script*.

- **$nvunalias**
  
  Delete nonvolatile device alias; edit the *script*.

- **“screen”**
  
  Standard string for alias created by *install-console*.

#### 7.4.11.2 Device tree browsing

*Device tree* browsing allows the user to examine and modify individual device tree nodes. The device tree browsing commands are similar to the UNIX commands for changing the working directory within the UNIX directory tree. The *active package* is the *device node* that the user may examine and modify with subsequent node examination commands. Selecting a device node makes it the active package.

When a node is the active package, user-created Forth commands are added to the list of *methods* for that *device*, new *properties* may be added to the list of properties for that device, dictionary search commands will operate on the node’s list of methods (followed by *system Forth words*), and that node’s methods can be executed as Forth words by typing their names.
Examples:

```
ok dev /zs@1,f0000000<cr>
ok .properties<cr>
name zs
reg 00 00 00 01 f0 00 00 00 00 00 00 08
intr 00 00 00 0c 00 00 00 00
device_type serial
keyboard
port-a-ignore-cd
port-b-ignore-cd
```

```
dev ( "device-specifier<cr>" -- ) Make the specified device node the active package.
find-device ( dev-str dev-len -- ) Make the device node dev-string the active package.
device-end ( -- ) Unselect the active package, leaving none selected.
pwd ( -- ) Display the device path that names the active package.
ls ( -- ) Display the names of the active package's children.
.properties ( -- ) Display names and values of properties of the active package.
```

7.4.11.3 Device probing

The `probe-all` command probes all available plug-in devices and adds to the device tree as appropriate. This command is normally invoked automatically during system start-up, but can be disabled by executing either `.banner` or `suppress-banner` from the script.

```
probe-all ( -- ) Probe for all available plug-in devices.
```

7.5 Firmware Debugging command group

This subclause describes commands that assist in the development of a Forth or FCode program. Most are interactive commands only and are not generally to be used within a Forth program.

7.5.1 Automatic stack display

This tool makes it easier for the user to see the stack effect of executed commands, by automatically displaying the entire stack just before each `ok` prompt. It is useful when developing Forth programs, as inadvertent stack errors will be more quickly spotted.

```
showstack ( -- ) Turn on automatic stack display.
noshowstack ( -- ) Turn off automatic stack display.
```

7.5.2 Serial download

This tool enables Forth source code to be downloaded and executed over a serial port.

```
dl ( -- ) Download and execute Forth text; end with ^d.
```

7.5.3 Dictionary

This subclause describes various productivity tools that involve the Forth dictionary. Most are useful for debugging Forth programs by providing tools for inspecting and altering the dictionary.
7.5.3.1 Dictionary search

Based on various parameters, these commands search the dictionary.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.calls</td>
<td>(xt --) Display all commands that use the execution token xt.</td>
</tr>
<tr>
<td>$sift</td>
<td>(text-addr text-len --) Display all command names containing text-string.</td>
</tr>
<tr>
<td>sifting</td>
<td>(&quot;text&lt; &gt;&quot; --) Display all command names containing text.</td>
</tr>
<tr>
<td>words</td>
<td>(--) Display the names of methods or commands.</td>
</tr>
</tbody>
</table>

7.5.3.2 Decompiler

The Forth decompiler allows the user to “see inside” any given routine in order to see how it is built. The decompiler displays the names of the component subwords in the order used to create the routine in question.

If the word being decompiled is an assembly-code definition, the disassembler will be invoked automatically (assuming the disassembler is present).

Unfortunately, many of the system routines might not decompile very well since the actual names of the component routines might have been previously discarded. For these cases, the address of the component routine will be shown instead of the name.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>see</td>
<td>(&quot;old-name&lt; &gt;&quot; --) Decompile the Forth command old-name.</td>
</tr>
<tr>
<td>(see)</td>
<td>(xt --) Decompile the Forth command whose execution token is xt.</td>
</tr>
</tbody>
</table>

7.5.3.3 Patch

These commands are used to change the definition of a command. Although the more typical method of changing the definition of a command is to edit some Forth source text, forget the old definition, and recompile the edited source, sometimes a quick patch is used to test a change first.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>patch</td>
<td>(new-name&lt; &gt;old-name&lt; &gt;word-to-patch&lt; &gt;&quot;&quot; --) Change contents of word-to-patch.</td>
</tr>
<tr>
<td>(patch)</td>
<td>(new-n1 num1? old-n2 num2? xt --) Change contents of command indicated by xt.</td>
</tr>
</tbody>
</table>

7.5.3.4 Forth source-level debugger

The Forth source-level debugger allows single-stepping and tracing of Forth programs; each “step” represents the execution of one Forth word.

In trace mode, the word marked for debugging is executed; the process continues with the next word called by the debugged word. In step mode (the default), the user controls the progress of the execution. Before the execution of each word called by the debugged word, the user is prompted for one of the following keystrokes:

<table>
<thead>
<tr>
<th>Keystroke</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;space&gt;</td>
<td>Executes the word just displayed and proceeds to the next word.</td>
</tr>
<tr>
<td>d</td>
<td>Goes down a level; i.e., marks for debugging the word whose name was just displayed and executes it.</td>
</tr>
<tr>
<td>u</td>
<td>Goes up a level; i.e., unmarks the word being debugged, marks its caller for debugging, and finishes executing the word that was previously being debugged.</td>
</tr>
<tr>
<td>c</td>
<td>Continues; i.e., switches from stepping to tracing, thus tracing the remainder of the execution of the word being debugged.</td>
</tr>
<tr>
<td>f</td>
<td>Starts a subordinate Forth interpreter. Forth commands may be executed normally. When the resume command is encountered, the interpreter exits and control-returns to the debugger at the place where the f keystroke was executed.</td>
</tr>
<tr>
<td>q</td>
<td>Quits; i.e., aborts the execution of the word being debugged and all its callers, returning to the command interpreter.</td>
</tr>
</tbody>
</table>
7.6 Client Program Debugging command group

This subclause describes various commands used to download and test machine-language client programs.

7.6.1 Registers display

Whenever a machine-language program execution is suspended, the complete state of the machine (particularly, all machine registers) is saved in the saved-program-state memory area. This serves two purposes: 1) the saved program state can be restored, allowing execution of the interrupted program to be resumed, and 2) the saved register values are available from the interpreter for inspection and/or modification for debugging purposes.

CPU registers may be examined and modified. Registers may be written or read individually or read as a group. For registers other than floating-point registers, the register access commands operate on memory copies of the register values instead of directly on the processor registers themselves. The contents of the processor registers are copied to the saved-program-state memory area when a running program transfers control to Open Firmware, typically from a user abort, a program breakpoint, or a severe system crash (resulting in a watchdog reset). When Open Firmware resumes execution of the suspended program (as with the go command), the processor registers are reloaded from the saved-program-state area and any modifications that the user has made prior to resumption of the program will then take effect.

Since normal operation of Open Firmware does not use or affect the values of floating-point registers, their values may be accessed “in-place,” rather than from memory copies, at the discretion of the implementor.

The names of the registers depend upon the CPU type. For a particular CPU type, the register names should be chosen to be obvious to a person familiar with that processor’s assembly language. However, the register names must not conflict with other Open Firmware word names. One way of preventing such name conflicts is to use register names beginning with “%”, which is not used within any other Open Firmware word names.

Register names are executable Forth words. Execution of a register name pushes the value contained in that register (or its memory copy) onto the stack. The value may be changed with the to command, as in the following:

```
( ff4 ) to %reg-name
```

See the processor-specific Open Firmware documents for examples of machine-specific register display commands.

7.6.2 Program download and execute

These functions allow a user to download a file containing assembly code, object code, Forth source code, or anything else. A typical use for these functions is to download some extended diagnostic test routines.
The syntax and behavior for `load` are similar to `boot` except that the program is loaded only and is not executed. Thus, the behavior of `boot` is almost equivalent to the following:

```
load device-specifier arguments<cr>
go
```

The difference is that `boot` will reset the machine before loading if a `client program` has been executed since the last reset, but `load` will not.

### 7.6.3 Abort and resume

The user may manually invoke the `command interpreter` by typing an abort sequence. The abort sequence is implementation-dependent. Typically, the abort sequence may be generated by a line break if the console is connected via an asynchronous serial line or by a specific key combination if the console is connected via a workstation keyboard/monitor, or by a hardware event such as a momentary switch. Virtual consoles connected via network devices may define other protocols to implement the abort sequence.

When a program is suspended by invoking the `command interpreter`, the contents of the processor registers are saved in the `saved-program-state` memory area. Execution of the suspended program may be resumed by using the `go` command.

<table>
<thead>
<tr>
<th>Keystroke</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;abort&gt;</td>
<td>Suspend the currently executing program, saving processor state in the <code>saved-program-state</code> memory area, and enter the Open Firmware command interpreter.</td>
</tr>
<tr>
<td>go</td>
<td>Execute or resume execution of a program in memory.</td>
</tr>
</tbody>
</table>

### 7.6.4 Disassembler

The `disassembler` takes a user-supplied memory address and produces an assembly-language interpretation of the contents of memory in a conventional format.

```
dis ( addr -- ) Begin disassembling at the given address.
+dis ( -- ) Continue disassembling where `dis` or `+dis` last stopped.
```

### 7.6.5 Breakpoints

This feature allows the user to set breakpoints in a `client program` and then use Open Firmware commands when one of the breakpoints is reached. After inspecting registers, memory, etc., the user can continue execution of the client program or choose from a variety of single-stepping options. The user may also cause any sequence of Open Firmware commands to be executed automatically when a breakpoint is reached, including automatic continuation.

To set breakpoints in a program, first `load` the program into memory or `boot` it. Re-enter the Open Firmware `command interpreter`, if necessary, with the user abort sequence. Set the desired breakpoints and continue or restart the program.

All of these commands are normally executed directly from the command interpreter, but may also be used within `colon definitions`. If a command that causes client program execution (`go`, `step`, `hop`, etc.) is executed within a colon definition, the remainder of the colon definition will not be executed.
One possible use of breakpoint commands within colon definitions is for complex breakpoint set-up sequences. Another possible use is to set-up conditional execution, by putting go or step commands inside an if statement within a command, and then loading that command into the .breakpoint or .step command.

7.6.6 Symbolic debugging

The symbolic debugger permits memory addresses to be displayed symbolically, i.e., with a program label and offset instead of simply a raw address. The program labels are taken from the string table entries of an appropriate executable file when the program is loaded by the secondary boot program. This secondary boot program can extract the symbol table from the executable file and use the client interface to initialize the Open Firmware symbol table.

After symbols are loaded, the disassembler displays symbolic addresses in addition to numeric addresses, and the Forth interpreter recognizes symbol names as though they were Forth words. Executing a symbol name pushes the symbol’s value on the stack (where it may be displayed with the Forth word “.”). The Forth interpreter searches for symbol names after attempting to recognize numbers; if there is a symbol name that is spelled the same as a Forth command or a number, the Forth interpreter will recognize the command or number instead of the symbol name. To alleviate this possible problem, the sym command is provided. Symbol name searches are case-sensitive, in contrast to searches for Forth commands.
7.7 FCode Debugging command group

The user interface versions of these FCode functions allow the user to debug FCode programs by providing named commands corresponding to FCode functions.

A system that implements the FCode Debugging command group shall implement commands corresponding to all of the FCode functions listed in 5.3.2 through 5.3.7, with the same names and semantics as those FCode functions, plus the commands given in the following list. Furthermore, a system that implements the FCode Debugging command group shall implement the Forth command group.

- `external` ( -- ) Newly created functions will be visible.
- `headerless` ( -- ) Newly created functions will be invisible.
- `headers` ( -- ) Newly created functions will be optionally visible.
- `fcode-debug?` ( -- names? ) If `true`, save names for FCodes with `headers`.
- `open-dev` ( dev-str dev-len -- ihandle | 0 ) Open device (and parents) named by given device specifier.
- `begin-package` ( arg-str arg-len reg-str reg-len dev-str dev-len -- ) Set up device tree before creating new node.
- `close-dev` ( ihandle -- ) Close device and all of its parents.
- `end-package` ( -- ) Close the device tree entry set up with `begin-package`.
- `execute-device-method` ( ... dev-str dev-len method-str method-len -- ??? ) Execute the named method in the package named `dev-string`.
- `apply` ( ... “method-name< >device-specifier< >” -- ??? ) Execute named method in the specified package.
Annex A
Open Firmware glossary
(normative)

A.1  Description

This annex describes the complete set of Forth words, FCode functions, methods, property names, and configuration variables that are defined by this standard. This annex defines the behavior of individual words, but it does not specify which of these words are required in order to claim conformance with one or more of the standard interfaces defined by this standard. That information is specified in other clauses.

A.1.1  Collating sequence

The words in this glossary are collated according to the following rules:
— if the word contains alpha characters ([a–z]) plus any numeric or punctuation characters, all non-alpha characters are ignored (removed) and the “alpha-only” word is ordered alphabetically;
— else, if the word contains only numeric ([0–9]) and punctuation characters, all punctuation characters are ignored and the resulting “numeric-only” word is ordered numerically;
— else, if the word contains only punctuation characters, the word is ordered according to its ASCII collating sequence.

Words consisting of only punctuation characters appear first in this glossary, “numeric-only” words appear second, and “alpha-only” words appear last.

A.1.2  Glossary entries

The general form of a glossary entry is as follows:

name ( stack comment ) Type Codes FCode# Brief description Full description

A.1.2.1  Name field

This field gives the name of the word that is being described. If there are multiple glossary entries with the same name, each name is followed by a clarifying comment, as shown in the following example:

draw-logo (FCode function)
draw-logo (package method)

A.1.2.2  Stack comment field

This field shows the effect of the word on the various Forth stacks and on other resources like the input buffer. The stack comment field is omitted for words to which it does not apply (e.g., property names).

ANS Forth conventions (as specified in ANSI X3.215-1994) are followed for stack comment descriptions, with the following differences and enhancements.
In the following table, “xyz” means an arbitrary sequence of characters, usually either a descriptive word (user-defined) or another existing standard stack abbreviation.

The prefixes “b”, “w”, and “q” are used to indicate 8-bit, 16-bit and 32-bit quantities. The terms “...” and “???” are used in place of the ANSI X3.215 terms “i*x” and “j*x”, respectively.

<table>
<thead>
<tr>
<th>Stack Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x x1 x2 etc.</td>
<td>Arbitrary stack items</td>
</tr>
<tr>
<td>n n1 n2 n3</td>
<td>Normal signed values</td>
</tr>
<tr>
<td>xyz</td>
<td>Arbitrary descriptive text</td>
</tr>
<tr>
<td>u uxyz</td>
<td>Unsigned value (_0)</td>
</tr>
<tr>
<td>nu nu1 etc.</td>
<td>Signed or unsigned value</td>
</tr>
<tr>
<td>char</td>
<td>8-bit value representing ISO_Latin-1 character</td>
</tr>
<tr>
<td>byte bxyz</td>
<td>Byte (8-bit value)</td>
</tr>
<tr>
<td>w wxyz</td>
<td>Doublet (16-bit value)</td>
</tr>
<tr>
<td>quad qwxyz</td>
<td>Quadlet (32-bit value)</td>
</tr>
<tr>
<td>dxyz</td>
<td>Double numbers (2 stack items; most significant on top of stack)</td>
</tr>
<tr>
<td>udxyz</td>
<td>Unsigned double numbers (2 stack items; most significant on top of stack)</td>
</tr>
<tr>
<td>xyz.lo</td>
<td>Low significant bits of a data item</td>
</tr>
<tr>
<td>xyz.hi</td>
<td>High significant bits of a data item</td>
</tr>
<tr>
<td>false</td>
<td>0 (false flag)</td>
</tr>
<tr>
<td>true</td>
<td>–1 (true flag)</td>
</tr>
<tr>
<td>xyz?</td>
<td>Flag (e.g., done? ok? error?); name indicates usage</td>
</tr>
<tr>
<td>?xyz?</td>
<td>Flag, but can generate “impure” values (besides 0 or –1)</td>
</tr>
<tr>
<td>addr virt</td>
<td>Address (32-bit virtual)</td>
</tr>
<tr>
<td>waddr</td>
<td>Doublet(16-bit)-aligned address</td>
</tr>
<tr>
<td>qaddr</td>
<td>Quadlet(32-bit)-aligned address</td>
</tr>
<tr>
<td>a-addr</td>
<td>Var-aligned address</td>
</tr>
<tr>
<td>phys</td>
<td>Physical address</td>
</tr>
<tr>
<td>phys.lo</td>
<td>Lower cell of physical address</td>
</tr>
<tr>
<td>phys.hi</td>
<td>Upper cell of physical address</td>
</tr>
<tr>
<td>len</td>
<td>Length (in bytes)</td>
</tr>
<tr>
<td>cnt</td>
<td>Count, number of operations</td>
</tr>
<tr>
<td>addr len</td>
<td>Address and length (2 associated stack items) for memory region</td>
</tr>
<tr>
<td>xyz-str xyz-len</td>
<td>Address and length (2 associated stack items) for string</td>
</tr>
<tr>
<td>xyz-pstr</td>
<td>Address of counted string (first byte contains length)</td>
</tr>
<tr>
<td>path-str path-len</td>
<td>String for device path</td>
</tr>
<tr>
<td>dev-str dev-len</td>
<td>String for device-specifier (device path or alias)</td>
</tr>
<tr>
<td>prop-addr prop-len</td>
<td>A property-encoded-array</td>
</tr>
<tr>
<td>xt</td>
<td>Execution token</td>
</tr>
<tr>
<td>phandle</td>
<td>Pointer (handle) for a package</td>
</tr>
<tr>
<td>ihandle</td>
<td>Pointer (handle) for an instance of a package</td>
</tr>
<tr>
<td></td>
<td>Separates two possible stack effects</td>
</tr>
<tr>
<td></td>
<td>Unspecified stack item(s). If encountered on both sides of stack comment, means same stack items on both sides</td>
</tr>
<tr>
<td>???</td>
<td>Unknown stack item(s)</td>
</tr>
<tr>
<td>&lt;nothing&gt;</td>
<td>Zero stack items, i.e., ( result</td>
</tr>
<tr>
<td>“text&lt;delim&gt;”</td>
<td>Text immediately following the same line as the command; parsed immediately. Text delimiter is enclosed in &lt;...&gt;</td>
</tr>
<tr>
<td>[text&lt;delim&gt;]</td>
<td>Text immediately following on the same line as the command; parsed immediately. Text delimiter is enclosed in &lt;...&gt;</td>
</tr>
<tr>
<td>/xyz/</td>
<td>In FCode evaluation, indicates FCode byte(s) to be read</td>
</tr>
<tr>
<td>&lt; &gt; &lt;space&gt;</td>
<td>Space delimiter. Leading spaces are ignored</td>
</tr>
<tr>
<td>&lt;eol&gt;</td>
<td>End-of-line delimiter</td>
</tr>
<tr>
<td>{text}</td>
<td>Optional text; causes default behavior if omitted</td>
</tr>
<tr>
<td>xyz-sys</td>
<td>Control-flow stack items; implementation-dependent</td>
</tr>
</tbody>
</table>
The topmost stack item is always shown on the right.

Flags are indicated in stack comments by “xyz?”, where xyz indicates the meaning of the flag. A true result agrees with the meaning of the flag. Examples are ok? (true if ok), done? (true if done), error? (true if error).

Return stack effect (if any) is shown on the same line as and after the normal stack effect (for either compile-time, run-time, or later-execution), with “R:” to distinguish the return stack, i.e.,

\[ r \]

\[ (x1 \rightarrow) (R: \rightarrow x1) \]

For words with different behaviors at compile time and run time, the run time behavior is shown after the compile-time behavior, with “C:” to distinguish the compile-time behavior, i.e.,

\[ \text{literal} \]

\[ (C: n1 \rightarrow) \]

\[ (\rightarrow n1) \]

For defining words, later behavior of the word that was created is shown before the stack effect for the defining word, with “E:” to distinguish the behavior of the word that was defined, i.e.,

\[ \text{constant} \]

\[ (E: \rightarrow n1) \]

\[ (n1 "\text{new-name}>" \rightarrow) \]

For Fcode words, behavior of the word during Fcode compilation is shown with “F:”, i.e.,

\[ \text{b(value)} \]

\[ (E: \rightarrow x) \]

\[ (F: x \rightarrow) \]

Here is a worst-case combination of some of the above:

\[ \text{does>} \]

\[ (E: \ldots \rightarrow ??) \] (created name execution)

\[ (C: \text{sys1} \rightarrow \text{sys2}) \] (compilation)

\[ (\rightarrow) (R: \rightarrow \text{sys1}) \] ("\text{does>}" execution)

\[ (\ldots \rightarrow \ldots \text{a-addr}) (R: \rightarrow \text{sys2}) \] (created name initiation)

Alternate individual stack items are separated by “[” without a space on either side, i.e., (addr len|0 result).
Alternate groups of stack items are separated by “[” with a space on either side, i.e., (addr len false | xt true).

For a text string on the stack, (name-str name-len) specifies the address and length, respectively, of the string. The body of the description sometimes refers to the entire string as simply “name string”.

For a “counted string” (also known as a “packed string”), in which the length is stored as the first byte of the string, the stack comment is (name-pstr).

The following paragraphs describe the syntax used to distinguish between the two forms of commands that parse following text. Both forms have identical results if encountered outside of a definition.

Commands that parse the input buffer immediately show the input buffer effect as [text<delimiter>] in the stack comment. For example, .( [text<]> -- ) could be used as either

\[ \text{.} \]

\[ (\text{print this}) \]

or

\[ : \text{foo} \]

\[ .(\text{print this}) ; \]

Commands that parse the input buffer upon later execution show the input buffer effect as “text<delimiter>” in the stack comment. For example, setenv ( "name< >value< >" -- ) could be used as either

\[ \text{setenv name value} \]

or

\[ : \text{foo} \]

\[ \text{setenv} ; \]
and later,

```
foo name value
```

“<eol>” denotes end-of-line. This is used for commands that consume the remainder of their input line. Unless specified otherwise, such commands remove leading and trailing white space from the arguments that they parse from the input line. For such commands, optional parameters are shown enclosed by { ... }, i.e.,

```
boot
  ( "{params}<eol>" -- )
```

### A.1.2.3 Type codes field

This field categorizes the word that is being defined. Some words fall into more than one category. Each category is denoted by a single uppercase letter. The field consists of a comma-separated list of such letters. The categories and their assigned letters are as follows:

- **A** ANS Forth word. In most cases, the description body for such words consists only of a one-line brief description. The full specification is given by ANSI X3.215-1994.
- **C** Compilation only. This command is not allowed outside of a colon definition.
- **F** This command is also a system-defined FCode function. The assigned FCode number follows.
- **M** Standard method name provided only in certain packages. Behavioral details may depend on the type of package in which it resides.
- **N** Configuration variable.
- **O** Obsolete. Refers to words that are not required for any Open Firmware implementation, but that have existed in some of Open Firmware’s precursors. The descriptions of those words are included to assist implementations that wish to be compatible with previous systems. For additional information, see annex E.
- **S** Standard string; not a command. Refers to certain “names” that have specified meanings. These names are used as string parameters to certain other commands but are not themselves Forth commands. Examples are package names, device types, property names, device aliases. In glossary listings, standard strings are spelled with leading/trailing double quotes (i.e., “string-name”) to distinguish them from Forth commands. The actual string name does not have double quotes. Standard strings do not have stack comments, because they are not Forth commands.
- **T** This command is commonly provided as a built-in macro in a tokenizer. See the glossary entries for the sequence of equivalent FCode primitives. Note that 0, 1, 2, and 3 represent the FCode commands named 0, 1, 2, and 3. Literal FCode bytes are represented by 00, 01, 02, etc.

### A.1.2.4 FCode# field

This field gives the FCode number assigned to this word. It is present only for words with the “F” type code.

### A.1.2.5 Description body

This field describes the semantics of the word. The first line of the description body is a brief description that gives an overview of the word’s behavior or purpose. Subsequent lines amplify that brief description to present the detailed specification of the word.

This document uses the word command to mean a Forth execution procedure (instead of the ANS Forth definition or named definition). In addition, this document uses the phrase command name to mean the text name of a command (instead of the ANS Forth word name).

Within a description body, subheadings are sometimes used, denoting the environment to which the following paragraphs apply. The subheadings are as follows:
— **Interpretation:** Defines the behavior of the word when it is encountered by the Forth interpreter when in interpretation state. This appears only for words whose interpretation behavior is different from their behavior when compiled inside a definition (mostly control flow words and literals).

— **Compilation:** Defines the behavior of the word when it is encountered by the Forth interpreter when in compilation state. Compilation semantics are given only for words whose compilation behavior is different from their behavior when compiled inside a definition (mostly control flow words and literals).

— **Run-time:** Defines the behavior of a word fragment when the definition into which it has been compiled is executed.

— **Execution:** Defines the behavior of the word when it is executed. This appears only for words that have either separately specified compilation or interpretation semantics.

The majority of *Forth words* do not have separately specified interpretation, compilation, and execution semantics; such words behave the same way both inside and outside of definitions, and that one behavior is given, without a subheading, in the description body.

— **FCode evaluation:** Defines the behavior of the word when it is encountered in interpretation state by the FCode evaluator. This appears only for FCode functions whose behavior is different when encountered by the FCode evaluator than when a definition in which they have been compiled is later executed. In most cases, these are FCode functions that read one or more following bytes from the FCode program, such as literals, control flow words, and defining words.

— **FCODE ONLY:** Indicates that the word is not required to be present as a user interface command even if the FCode Debugging command group is implemented.

— **Equivalent to:** Defines a possible implementation, in terms of other more primitive commands, for the command being described. It is permissible to use a different implementation if the effective behavior is identical.

— **Tokenizer equivalent:** Defines a sequence of FCode functions that, taken as a whole, is equivalent to the word being defined. Tokenizers typically generate that sequence when the word being defined is encountered in FCode source. This subheading appears for words with the “T” type code.

— **Tokenizer:** Defines the effect of the word on the subsequent behavior of a tokenizer. This is used for words that affect certain tokenizer modes, such as those controlling its numeric conversion radix, the size of generated branch offsets, and the visibility of names for program-defined functions.

— **ANS Forth/tokenizer difference:** Notes a difference between the ANS Forth behavior of the word and the suggested tokenizer behavior. Most such differences are inevitable because an FCode program has no textual input buffer.

— **ANS Forth note:** Notes a value-added difference where this standard specifies a behavior for the word that is a superset of the required ANS Forth behavior. Usually, the added value takes the form of a specification of the behavior for conditions under which ANS Forth declines to mandate a particular behavior (for example, the interpretation behavior of control flow words).

— **Example:** Gives an example of how the word might be used in a program, with explanatory text.

— **Used as:** Shows typical, representative usage of the command, but does not exclude other formulations. In some examples, the `ok` prompt is shown, emphasizing that this is the way the command is used at the Forth interpreter prompt, as opposed to the way it would be used when compiled inside another definition, e.g.,

```
ok  forget old-name
```
— **Usage restriction:** Indicates the conditions under which the word must behave as specified, thus imposing requirements on programs that wish to use the word correctly, but not on the word itself.

— **NOTE—** Gives additional information that is not part of the specification of the word, but that might be helpful in understanding how the word is used or how it can be implemented.

### A.2 Specification

A standard system or standard *package* that implements one or more of the following words shall implement them with the semantics given below:

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Semantics</th>
<th>Implementation Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>! (x a-addr --)</td>
<td><strong>Store item x to cell at a-addr.</strong></td>
<td><strong>See:</strong> ( \texttt{zl!} )</td>
</tr>
</tbody>
</table>
| " ([text]<><> -- text-str text-len) | **Gather the immediately following string or hex data.** | **Interpretation:** \([\texttt{text}<><>] -- \texttt{text-str text-len}\) \(\) Parse **text** delimited by """, with hex-sequence handling as described below. Store the resulting string **text-str** at the next available temporary location. The length of the temporary buffer is implementation-dependent but shall be no less than 80 characters. At least two such temporary buffers shall be provided, using the buffers alternately for successive uses of "" in interpretation state. **Compilation:** \([\texttt{text}<><>] --\) Parse **text** delimited by """, with hex-sequence handling as described below. Append the run-time semantics given below to the current definition. **Run-time:** \((-\ -- \texttt{text-str} \texttt{text-len})\) Return **text-str** and **text-len** that describe a string consisting of the characters **text**. A standard program shall not alter the contents of the string described by **text-str** and **text-len**. **Used as:** " text"<space> **Hex-sequence handling:** If parsing was terminated by a """" in the input buffer (as opposed to the exhaustion of the input buffer), take further action depending on the value of the next character in the input buffer as follows: If input buffer is exhausted: Take no further action If next character is white space: Consume that character and take no further action If next character is "": Consume that character, parse hexadecimal characters delimited by ")" as described below, append the characters denoted by those hexadecimal characters to **text**, and resume ""-delimited parsing to further extend **text**. (This feature is useful for including nonprintable characters in text strings.) Hexadecimal character treatment: Treat each pair of hexadecimal digits in the substring between "" (” and “)” as the numerical representation (0x00 .. 0xFF) of a character, ignoring nonhexadecimal characters between pairs of hexadecimal characters. Hexadecimal digits shall be recognized in either uppercase or lowercase. Otherwise: The result is implementation-dependent. **Used as:** " hello"(0102 ff 80,81)there" ( addr len ) **Tokenizer equivalent:** b(" len-byte xx-byte xx-byte ... xx-byte **NOTE—** is similar to **S** in ANS Forth, but with the addition of hex-sequence handling. The final delimiting """"**must** be followed by a white space character, in contrast to **S** which does not require a trailing space.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Semantics</th>
<th>Implementation Details</th>
</tr>
</thead>
<tbody>
<tr>
<td># (ud1 -- ud2)</td>
<td><strong>Convert a digit in pictured numeric output conversion.</strong></td>
<td></td>
</tr>
</tbody>
</table>
End pictured numeric output conversion.

'( "old-name< >" -- xt)A,T
Return execution token xt of a command, parsed later.

Tokenizer equivalent: b(') old-FCode#

ANS Forth/tokenizer difference: In FCode source, ' cannot be used inside a colon definition.'

( ( [text<]> ) -- ) A,T
Ignore the immediately following text up to closing “)”.

Tokenizer equivalent: <nothing>

( . ) ( n -- str len ) T
Convert a number into a text string.
Perform the conversion according to the value in base.

Tokenizer equivalent: dup abs <# u#s swap sign u#>

* ( nu1 nu2 -- prod ) A,F 0x20
Multiply nu1 by nu2.

*/ ( n1 n2 n3 -- quot ) A
Calculate n1 times n2 divided by n3.

+ ( nu1 nu2 -- sum ) A,F 0x1E
Add nu1 to nu2.

+! ( nu a-addr -- ) A,F 0x6C
Add nu to cell at a-addr.

, ( x -- ) A,F 0xD3
Append x to data space.

- ( nu1 nu2 -- diff ) A,F 0x1F
Subtract nu2 from nu1.

. ( nu -- ) A,F 0x9D
Display number (and trailing space).

." ( [text<"] ) -- ) A,T
Display the immediately following text.

** Interpretation:**
Parse text delimited by “” and display it.

Compilation:
Same as ANS Forth.

** Run-time:**
Same as ANS Forth.

Tokenizer equivalent: b(" len-byte xx-byte xx-byte ... xx-byte type

ANS Forth note: Usage also allowed while interpreting.
. ( [text<]> -- ) A,T
   Display the immediately following text up to delimiting "".

/ ( n1 n2 -- quot ) A,F 0x21
   Divide n1 by n2; return quotient.

"/"
   The root node of the device tree.
   See: 3.1 for a complete description.

: ( E: ... -- ???) A,T
   Begin creation of a colon definition.
   Tokenizer equivalent: new-token|named-token|external-token b(:)
   ANS Forth/tokenizer difference: In FCode source, : cannot be used inside another colon definition.

; ( colon-sys -- ) A,T,C
   End creation of a colon definition.
   Tokenizer equivalent: b(;)

< ( n1 n2 -- less? ) A,F 0x3A
   Return true if n1 is less than n2.

<# ( -- ) A,F 0x96
   Initialize pictured numeric output conversion.
   See: (.) and (u.) for examples of use.

<< ( x1 u -- x2 ) T
   Synonym for lshift.
   Tokenizer equivalent: lshift

<= ( n1 n2 -- less-or-equal? ) F 0x43
   Return true if n1 is less than or equal to n2.

<> ( x1 x2 -- not-equal? ) A,F 0x3D
   Return true if x1 is not equal to x2.

= ( x1 x2 -- equal? ) A,F 0x3C
   Return true if x1 is equal to x2.

> ( n1 n2 -- greater? ) A,F 0x3B
   Return true if n1 is greater than n2.

>= ( n1 n2 -- greater-or-equal? ) F 0x42
   Return true if n1 is greater than or equal to n2.

>> ( x1 u -- x2 ) T
   Synonym for rshift.
   Tokenizer equivalent: rshift
?  ( a-addr -- ) A,T
Display the number at address a-addr.

Tokenizer equivalent: @.

@  ( a-addr -- x ) A,F 0x6D
Fetch item x from cell at a-addr.
See: r1@

[  ( -- ) A,C
Enter interpretation state.

[ ’ ] ( [old-name< >] -- xt )A,T'
Return execution token xt of a command.

Interpretation:  ( [old-name< >] -- xt )
Skip leading space delimiters. Parse old-name delimited by a space. Find old-name and place its execution token xt on the
stack. An ambiguous condition exists if old-name is not found.

Compilation:  ( [old-name< >] -- )
Skip leading space delimiters. Parse old-name delimited by a space. Find old-name and append the run-time semantics
given below to the current definition. An ambiguous condition exists if old-name is not found.

Run-time:  ( -- xt )
Place old-name’s execution token xt on the stack.

Tokenizer equivalent: b(’) old-FCode#

ANS Forth note: Usage also allowed while interpreting.

\  ( [rest-of-line<eol>] -- ) A,T
Ignore the immediately following text on this line.

Tokenizer equivalent: <nothing>

]  ( -- ) A
Enter compilation state.

0  ( -- 0 ) F 0xA5
Constant 0.
This number has its own FCode value to save space in FCode binary form.

0<  ( n -- less-than-0? ) A,F 0x36
Return true if n is less than zero.

0<=  ( n -- less-or-equal-to-0? ) F 0x37
Return true if n is less than or equal to zero.

0<>  ( n -- not-equal-to-0? ) A,F 0x35
Return true if n is not equal to zero.

0=  ( nu|flag -- equal-to-0? ) A,F 0x34
Return true if nu|flag is equal to zero.
This command correctly inverts all flags, including “impure” flags.

0>  ( n -- greater-than-0? ) A,F 0x38
Return true if n is greater than zero.
0\geq n\geq \text{true if } n \text{ is greater than or equal to zero.}

1\quad (\text{-- 1})

Constant 1.

This number has its own FC code value to save space in FC code binary form.

1\quad (\text{nu1 -- nu2})

Add 1 to nu1.

Tokenizer equivalent: 1 +

1\quad (\text{nu1 -- nu2})

Subtract 1 from nu1.

Tokenizer equivalent: 1 -

-1\quad (\text{-- -1})

Constant -1.

This number has its own FC code value to save space in FC code binary form.

2\quad (\text{-- 2})

Constant 2.

This number has its own FC code value to save space in FC code binary form.

2\quad (\text{x1 x2 a-addr --})

Store cell pair at a-addr.

2\quad (\text{x1 -- x2})

Shift x1 left by one bit-place. Zero-fill low bit.

2\quad (\text{nu1 -- nu2})

Add 2 to nu1.

Tokenizer equivalent: 2 +

2\quad (\text{nu1 -- nu2})

Subtract 2 from nu1.

Tokenizer equivalent: 2 -

2\quad (\text{x1 -- x2})

Shift x1 right by one bit-place. High bit unchanged.

2\quad (\text{a-addr -- x1 x2})

Fetch cell pair from a-addr.

3\quad (\text{-- 3})

Constant 3.

This number has its own FC code value to save space in FC code binary form.
Arithmetic shift \( x1 \) right by \( u \) bit-places.
Copy high bits with the highest bit (i.e., sign-extend the high bit).

**abort**

\[ ( ... -- ) ( R: ... -- ) \]
Abort program execution; clear stacks.
The version of **ABORT** defined by the ANS Forth EXCEPTION wordset applies.

**abort"**

\[ ( C: [text" ] -- ) \]
If flag is nonzero, display text and call **abort**.
**Used while compiling as:**
\[ ( flag ) abort" text" \]
Equivalent to:
\[ -2 \text{ throw} \]

**abs**

\[ ( n -- u ) \]
Return absolute value of \( n \).

**accept**

\[ ( addr \text{ len1} -- \text{ len2} ) \]
Get an edited input line and store it at \( addr \).
**Tokenizer equivalent:**
\[ \text{span } @ \text{-rot expect span } @ \text{ swap span }! \]

**"address"**

Standard property name to identify device virtual address.

**prop-encoded-array:**
Arbitrary number of \( \text{virt-addr} \) values with each value encoded with **encode-int**.
Specifies the virtual addresses of one or more memory-mapped regions on this device. This property is typically used to report the virtual addresses of regions that the firmware has already mapped so that client programs can reuse those mappings.
The correspondence between declared addresses and the set of mappable regions for a particular device is device-dependent.
**Usage restriction:** A standard package should create an **"address"** property after virtual addresses have been assigned by mapping operations, and shall delete the **"address"** property when the corresponding virtual addresses are unmapped.

**NOTE**—The **"address"** property is particularly useful in the following cases:

a) When the mapped region is large, reuse of the virtual address conserves mapping resources.
b) For simple devices (for example, system interrupt control registers), using the firmware’s existing mapping prevents the client program from having to know about the mapping process.

**Used as:**
\[ ( \text{virt} ) \text{ encode-int} \]
" address" property
**See also:** **free-virtual**

**"address-bits"**

Standard property name to indicate number of network address bits.

**prop-encoded-array:**
Integer, encoded with **encode-int**.
This property, when declared in a **network** device, indicates the number of address bits needed to address this device on the physical layer of its network. The absence of this property indicates the default value of 48.

**Used as:**
\[ d\# 48 \text{ encode-int } " \text{address-bits}" \text{ property} \]
"#address-cells"  S
Standard property name to define the package’s address format.

prop-encoding-array:
  Integer, encoded with encode-int.
This property applies to packages that define a physical address space, i.e., those packages with “decode-unit” methods. The property value specifies the number of cells that are used to encode a physical address within that address space. The value of this property affects the other functions, commands, and methods that deal with physical addresses. In a package with a “decode-unit” method, a missing “#address-cells” property signifies that the number of address cells is two.
For a given bus, the value of this property should be the same on all machines for which that bus could possibly be used, even if those machines do not all have the same cell size. Consequently, the value of the property is determined in part by the smallest cell size among all the machines to which the bus can apply.
An Open Firmware implementation shall operate correctly with values of this property from one to four. An implementation may support larger values.
See also: map-in, map-low, decode-unit, my-address, my-space, my-unit, encode-phys, and decode-phys.

 adr ( addr -- )
Display symbolic form for the given address.
Display the symbol nearest to (but not greater than) the given address. The symbolic form of an address is usually a symbol name plus a non-negative numeric offset.
If value>sym returns false, display the address as a number.
Other aspects of the displayed value are ISA-dependent.
See also: value>sym

again (C: dest-sys -- ) A,T
End an (infinite) begin...again loop.

Compilation:
(C: dest-sys -- )
Perform the compilation semantics of ANS Forth AGAIN. Then, if the current definition is temporary and the depth of the control flow stack is the same as its depth when the temporary current definition was initiated, perform the compilation semantics of ; and execute the temporary current definition.

Run-time:
( -- )
Same as ANS Forth.
NOTE—An external event, such as a keyboard abort, is usually necessary to terminate a begin ... again loop.

Tokenizer equivalent: bbranch -offset

ANS Forth note: Also works outside of a definition.
alarm \ F 0x213
( xt n -- ) F 0x213
Execute \xt\ repeatedly, at intervals of \(n\) milliseconds (ms).

Arrange to periodically execute the package method \xt\ at intervals of \(n\) ms (to the best accuracy possible). If \(n\) is zero, stop the periodic execution of \xt\ within the current instance context (leaving unaffected any periodic execution of \xt\ that was established within a different instance).

Before each periodic execution of the method, the implementation shall set the current instance to be the same as the current instance at the time that \alarm\ was executed and shall restore the current instance to its previous value afterwards.

Usage restriction: \xt\ must be the execution token of a method whose stack diagram is ( -- ); i.e., it neither expects stack arguments nor leaves stack results.

Example: Assume the existence of a command named \fp-switch?\ that tests a momentary-contact front-panel switch, returning \true\ if the switch is activated. This example shows a way to signal a \user-abort\ when the switch is activated.

\begin{verbatim}
: fp-abort ( -- ) \ Abort if front-panel switch is activated
     \fp-switch? \ if
   begin \fp-switch? 0= until
     \user-abort
   then
;
['] \fp-abort \ d# 100 \alarm \ Test switch every tenth of a second
\end{verbatim}

alias \(\text{T} (\text{E} : ... -- ??? )\)
Create a new command equivalent to an existing command.

Create a new command \textit{new-name}, with the exact behavior of an existing command \textit{old-name}. The stack effect for execution of \textit{new-name} is the same as that of \textit{old-name}. Subsequently, when \textit{new-name} is found, e.g., with \texttt{"\textit{new-name}"}, the execution token returned will be that of \textit{old-name}; similarly, when \texttt{\textit{new-name}} is referenced during the compilation of a new de

Used as: \texttt{ok alias new-name old-name}

Tokenizer equivalent: <resolution of alias>
In FCode source, \texttt{alias} cannot be called from within a colon definition. During tokenization of FCode source, no FCode is generated when this command is encountered. Instead, the tokenizer will update its own lookup table of existing commands. Any occurrence of the new command will cause the assigned FCode of the old command to be generated. One implication is that the new command will not appear in the Forth dictionary after the FCode program is compiled. If this behavior is undesirable, use a colon definition instead. Note that this function is unrelated to \texttt{device aliases} (compare with \texttt{devalias})

```
"/aliases"
The node containing this system’s device alias list.

See: 3.5 for a complete description.
```

align \(\text{A ( -- )}\)
Allocate dictionary bytes to leave top of dictionary var-aligned.

aligned \(\text{F ( n1 -- n1|a-addr )}\)
Increase \(n1\) as necessary to give a var-aligned address.

The result is the same if \(n1\) is already a var-aligned address; otherwise, return the next larger var-aligned address.

alloc-mem \(\text{F ( len -- a-addr )}\)
Allocate \texttt{len} bytes of memory.

Return the virtual address \texttt{a-addr} of a buffer aligned to the most stringent requirements of the particular instruction set architecture. If the requested operation cannot be performed, a \texttt{throw} shall be called with an appropriate error message, as with \texttt{abort}.

NOTES—Out-of-memory conditions may be detected and handled properly in the code with \([']\) alloc-mem catch. Memory allocated with alloc-mem can be freed using free-mem. The memory allocated by alloc-mem is not suitable for DMA.
**allot**

allocate len-- A,T

Allocate len bytes in the dictionary.

If the requested operation cannot be performed, a throw shall be called with an appropriate error message, as with abort".

Tokenizer equivalent: 0 max 0 \?do 0 c, loop

NOTE—The "tokenizer equivalent" phrase does not handle negative arguments.

NOTE—Out-of-memory conditions may be detected and handled properly in the code with ['] allot catch.

**and**

(x1 x2-- x3) A,F 0x23

Return bitwise logical “and” of x1 and x2.

**apply**

(... "method-name" device-specifier" -- ???)

Execute named method in the specified package.

Perform the function of execute-device-method.

If the requested operation cannot be performed, a throw shall be called with an appropriate error message, as with abort".

Used as: apply set-tpe-test aliasname

NOTE—Error conditions may be detected and handled properly in the code with ['] apply catch.

**ascii**

([text>]-- char) T

Generate ASCII code for the immediately following character.

Interpretation: ([text>]-- char)

Skip leading space delimiters. Parse text delimited by a space. Put the integer value of the first character of text on the stack.

Compilation: ([text>]--)

Skip leading space delimiters. Parse text delimited by a space. Append the run-time semantics given below to the current definition.

Run-time: (-- char)

Place char, the integer value of the first character of text, on the stack.

Used as: ascii Boo (0x42)

ascii is similar to ANS Forth CHAR and CHAR, but has the same usage whether interpreting or compiling.

Tokenizer equivalent: b(lit) 00 00 00 xx-byte

**auto-boot?**

( -- auto? ) N

If true, automatically execute boot-command after power-on or reset-all.

As the next to last step of the Open Firmware start-up sequence, if auto-boot? is true, execute the command string specified by boot-command.

NOTE—In the usual case, the value of boot-command is boot. Usually boot transfers control to a client program, in which case the following step of entering the command interpreter is not performed.

Configuration variable type: Boolean. Suggested default value: true.

**available**

S

Standard property name to define available resources.

prop-encoded-array:

Arbitrary number of address, length pairs. Address is a phys.lo ... phys.hi list of integers, each integer encoded as with encode-int. Length (whose format depends on the package) is one or more integers, each encoded as with encode-int.

The value of this property defines resources, managed by this package, that are currently available for use by a client program. The use of claim and release affect the value of this property.

See also: claim, "existing", "reg", release
b ("") ( -- str len ) F 0x12

String literal FCode. Followed by FCode-string.

FCode evaluation: (F: /FCode-string/ -- )

Read an FCode-string from the current FCode program. If in interpretation state, copy the FCode-string to a temporary buffer if necessary and perform the run-time semantics given below. If in compilation state, append the run-time semantics given below to the current definition.

If a temporary buffer is used, at least two such buffers shall be provided, alternating between the buffers so that at least two distinct strings can be in use at any given time.

Run-time: ( -- str len )

Return str and len describing a string whose characters are the same as the FCode-string.

FCODE ONLY (Tokenized by ", .", and . ()

b (') ( -- xt ) F 0x11

Function literal FCode. Followed by FCode#.

FCode evaluation: (F: /FCode#/ -- )

Read an FCode# from the current FCode program. If in interpretation state, perform the run-time semantics given below. If in compilation state, append the run-time semantics given below to the current definition.

Run-time: ( -- xt )

Return the execution token xt of the FCode function corresponding to the FCode#.

FCODE ONLY (Tokenized by [ ' ] and ‘)

b (;) (E: ... , -- ???) F 0xB7

Define type of new FCode function as “colon definition”.

FCode evaluation: (F: -- colon-sys )

Define the behavior of the most recently created FCode function to be that of a Forth colon definition, with execution semantics as given below. Enter compilation state, initiating the current definition, which will be terminated by the execution of ;. Enter compilation state and start the current definition, thereby producing colon-sys. Append the initiation semantics given below to the current definition.

The execution semantics of the definition will be determined by the words compiled into the body of the definition.

Initiation: (R: -- sys )

Save implementation-dependent information sys about the calling definition.

Execution: (of defined word) ( ... -- ???)

Perform the body of the definition.

FCODE ONLY (Tokenized by ;)

b (;) ( -- ) C,F 0xC2

End an FCode colon definition.

FCode evaluation: (F: colon-sys -- )

Append the run-time semantics given below to the current definition, terminate the current definition, and enter interpretation state.

Run-time: ( -- ) (R: sys -- )

Return control to the caller of the definition containing ;. sys is produced by the corresponding b ( : ).

FCODE ONLY (Tokenized by ;)
banner ( -- )

Display the system power-on banner.

The banner is displayed at a system-dependent screen location (usually either at the top of the screen or at the current cursor position).

If the current output device has a device_type property whose value is "display", display a logo by executing the current output device's draw-logo method with the following arguments:

- The line# argument, at the system's discretion, is either 0 or the line number corresponding to the current cursor position.
- If oem-logo? is true, the addr argument is the address returned by oem-logo. Otherwise, it is the address of the system-dependent default logo.

The width and height arguments are both 64.

In any case, display additional information as follows:

- If oem-banner? is true, display the text given by the value of oem-banner.
- Otherwise, display implementation-dependent information about the system, for example, the machine type, serial number, firmware revision, network address, and hardware configuration.

If executed within the script, suppress automatic execution of the following Open Firmware start-up sequence:

probe-all install-console banner

See also: suppress-banner

base ( -- a-addr ) A,F 0xA0

variable containing the current numeric conversion radix.

Use this value for all subsequent numeric conversion, except where noted otherwise.

During tokenizing of FCode source, changes to this value do not affect evaluation of subsequent numeric input text.

ANS Forth/tokenizer difference: ANS Forth has no separate "tokenizing" behavior.

bbranch ( -- ) F 0x13

Unconditional branch FCode. Followed by FCode-offset.

FCode evaluation: (F: /FCode-offset/ -- )

Read FCode-offset, whose target is the matching b(<mark) or b(>resolve), from the current FCode program.

If FCode-offset is negative, corresponding to a backward branch: (C: dest orig1 .. orig1 -- orig1)

Append the run-time semantics given below to the current definition, resolving the backward reference dest. Then, if the current definition is temporary and the depth of the control flow stack is the same as its depth when the temporary current definition was initiated, perform the FCode evaluation semantics of b(;) and execute the temporary current definition.

If FCode-offset is non-negative, corresponding to a forward branch:

If in interpretation state:

Read and discard FCode-offset-2 bytes (if the FCode-offset size is 16 bits) or FCode-offset-1 byte (if the FCode-offset size is 8 bits) from the current FCode program and take no further action.

If in compilation state: (C: orig1 -- orig2 orig1)

Put the location of a new unresolved forward reference orig2 onto the control flow stack underneath orig1.

Append the run-time semantics given below to the current definition. The semantics will be incomplete until orig2 is resolved (e.g., by b(>resolve)).

Run-time: ( -- )

If FCode-offset is negative:

Continue execution at the location specified by dest.

If FCode-offset is non-negative:

Continue execution at the location given by the resolution of orig2.

NOTE—The FCode-offset negative case is used to implement the Forth words again and repeat. For repeat, the FCode number for b(>resolve) (resolving orig1) immediately follows bbranch and its offset. In either case, dest corresponds to the preceding b(<mark).

NOTE—The FCode-offset non-negative case is used to implement the Forth word else, in which case the FCode number for b(>resolve) (resolving orig1) immediately follows bbranch and its offset, and somewhat later another b(>resolve) (resolving orig2) follows the sequence corresponding to the Forth source code after else.

FCODE ONLY (Tokenized by again, repeat, and else)
Conditional branch FCode. Followed by FCode-offset.

FCode evaluation: 

Read FCode-offset, whose target is the matching `b(<mark)` or `b(>resolve)`, from the current FCode program.
If FCode-offset is negative, corresponding to a backward branch: 

- Append the run-time semantics given below to the current definition, resolving the backward reference dest. Then, if the current definition is temporary and the depth of the control flow stack is the same as its depth when the temporary current definition was initiated, perform the FCode evaluation semantics of `b(;)` and execute the temporary current definition.

If FCode-offset is non-negative, corresponding to a forward branch:
- If in interpretation state: 
  - If all bits of x are zero, read and discard FCode-offset-2 bytes (if the FCode-offset size is 16 bits) or FCode-offset-1 bytes (if the FCode-offset size is 8 bits) from the current FCode program and take no further action. Otherwise, take no further action.
- If in compilation state: 
  - Put the location of a new unresolved forward reference orig onto the control flow stack. Append the run-time semantics given below to the current definition. The semantics will be incomplete until orig is resolved (e.g., by `b(>resolve)`).

Run-time:

- If FCode-offset is negative: 
  - If all bits of x are zero, continue execution at the location specified by dest.
- If FCode-offset is non-negative: 
  - If all bits of x are zero, continue execution at the location specified by the resolution of orig.

NOTE—The FCode-offset negative case is used to implement the Forth word `until`.
NOTE—The FCode-offset non-negative case is used to implement the Forth words `if` and `while`.

FCODE ONLY (Tokenized by `until`, `while`, and `if`)

### b(buffer:)

(E: -- a-addr ) F 0xBD 
(F: size -- )

Defines type of new FCode function as `buffer:`.

FCode evaluation: 

If instance has been executed since the last execution of `b(buffer:)`, `b(variable)`, `b(value)`, or `b(defer)`, allocate size bytes of storage in the current package’s zero-filled data area; otherwise, allocate the storage in data space. Define the behavior of the most recently created FCode function to have the execution semantics given below.

Execution: 

- Return a-addr, the address of the storage associated with the defined word.

FCODE ONLY (Tokenized by `buffer:`)

### b(case)

(sel -- sel ) F 0xC4 
(F: -- )

Begin a case (multiple selection) statement.

FCode evaluation: 

Perform the interpretation or compilation semantics of case.

FCODE ONLY (Tokenized by `case`)
**b(constant)**

(E: -- n )  
(F: n -- )  

F 0xBA  

Defines type of new FCode function as constant.

**FCode evaluation:**

(F: n -- )  

Define the behavior of the most recently created FCode function to have the execution semantics given below.

**Execution:** (of defined word)  

Place n on the stack.

**FCODE ONLY (Tokenized by constant)**

---

**b(create)**

(E: -- a-addr )  
(F: -- )  

F 0xBB  

Defines type of new FCode function as create word.

**FCode evaluation:**

(F: -- )  

If the data space pointer is not aligned, reserve enough data space to align it. This address defines the most recently created word’s data field. **b(create)** does not allocate data space in the created word’s data field. Define the behavior of the most recently created FCode function to have the execution semantics given below.

**Execution:** (of defined word)  

Place a-addr, the address of the defined word’s data field, on the stack.

**FCODE ONLY (Tokenized by create)**

---

**b(defer)**

(E: ... -- ??? )  
(F: -- )  

F 0xBC  

Defines type of new FCode function as defer word.

**FCode evaluation:**

(F: -- )  

If instance has been executed since the last execution of **b(buffer:), b(variable), b(value), or b(defer)**, allocate sufficient storage for an execution token in the current package’s initialized data area; otherwise allocate the storage in data space. Set the initial value of that storage to the execution token for a definition that, if executed, will display a message indicating execution of an uninitialized defer word. Define the behavior of the most recently created FCode function to have the execution semantics given below.

**Execution:** (of defined word)  

Execute the definition currently associated with the defined word.

The definition associated with the defined word can be changed later by placing the execution token of the new definition on the stack and executing the FCode function corresponding to the sequence **b(to) FCode#, where FCode# is the FCode number of the defined word.**

**FCODE ONLY (Tokenized by defer)**

---

**b(do)**

( limit start -- )  
(F: /FCode-offset/ -- )  

F 0x17  

Begin FCode do .. loop. Followed by FCode-offset.

**FCode evaluation:**

(F: /FCode-offset/ -- )  

Read FCode-offset, whose target is the matching **b(loop) or b(+loop)**, from the current FCode program and perform the interpretation or compilation semantics of do.

**FCODE ONLY (Tokenized by do)**

---

**b(?do)**

( limit start -- )  
(F: /FCode-offset/ -- )  

F 0x18  

Begin FCode ?do .. loop. Followed by FCode-offset.

**FCode evaluation:**

(F: /FCode-offset/ -- )  

Read FCode-offset, whose target is the matching **b(loop) or b(+loop)**, from the current FCode program and perform the interpretation or compilation semantics of ?do.

**FCODE ONLY (Tokenized by ?do)**
begin

(C: -- dest-sys ) A,T

Begin a conditional loop.

Interpretation:

Enter compilation state, initiating a temporary current definition in a region of memory other than the data space. Then perform the compilation semantics of ANS Forth BEGIN.

Compilation:

Same as ANS Forth.

Run-time:

Same as ANS Forth.

Tokenizer equivalent: b (<mark )

ANS Forth note: Also works outside of a definition.

begin-package

(arg-str arg-len reg-str reg-len dev-str dev-len -- )

Set up device tree, before creating new node.

Perform the following:

— Open the parent device (and all higher parents) with open-dev and the parameter dev-string. If the call to open-dev returns a zero, terminate execution with an appropriate error message, as with abort".

— Set my-self to the new parent ihandle.

— Set the active package to the parent device.

— Call new-device to open the child device node.

— Call set-args to set arguments for the child by using the parameters arg-string and reg-string.

dev-string is the device path string (either a full device pathname, or a pre-defined device alias) of the parent of the child node about to be created.

reg-string is the unit address string (i.e., “3,1000”) and contains the text representation of the physical address of the child (within the address space of the parent device). The numerical representation of this physical address can be returned within the child with the my-address and my-space FCodes.

arg-string is the instance-arguments string and contains the value that can be returned within the child with the my-args FCode.

Used as: 0 0 " 3,2000" " /sbus " begin-package

behavior

(defer-xt -- contents-xt ) F 0xDE

Retrieve execution behavior of a defer word.

This command is used to obtain the execution contents of a defer word. A typical use would be to retrieve and save the execution behavior of the defer word, set the defer word to a new behavior, and then later restore the old behavior.

Used as: [ ’ ] defer-name behavior ( contents-xt )

bell

( -- 0x07 ) F 0xAB

ASCII code for “bell” character.

b (endcase)

( sel | <nothing> -- ) F 0xC5

End a case (multiple selection) statement.

FCode evaluation:

Perform the compilation semantics of endcase.

FCODE ONLY (Tokenized by endcase)
**b(endof)**

( -- ) F 0xC6

FCode for **endof** in **case** statement. Followed by **FCode-offset**.

**FCode evaluation:**

(F: /FCode-offset/ -- )

Read **FCode-offset**, whose target is the matching **b(endcase)**, from the current FCode program and perform the Compilation semantics of **endof**.

**FCODE ONLY** (Tokenized by **endof**)

**between**

( n min max -- min<=n<=max? ) F 0x44

Return **true** if n is between min and max, inclusive.

**b(field)**

(E: addr -- addr+offset) F 0xBE

(F: offset size -- offset+size )

Defines type of new FCode function as **field**.

**FCode evaluation:**

(F: offset size -- offset+size )

Define the behavior of the most recently created FCode function to have the execution semantics given below. Return the sum of offset and size.

**Execution:** (of defined word) ( addr -- addr+offset )

Return the sum of addr and offset.

**FCODE ONLY** (Tokenized by **field**)

**bl**

( -- 0x20 ) A,F 0xA9

ASCII code for “ ” (blank) character.

**blank**

( addr len -- ) A,T

Set len bytes beginning at addr to the value 0x20.

**Tokenizer equivalent:** bl fill

**b(leave)**

(F: -- ) F 0x1B

Exit from a **do ... loop**.

**FCode evaluation:**

(F: -- )

Append the execution semantics of **leave** to the current definition.

**FCODE ONLY** (Tokenized by **leave**)

**blink-screen**

( -- ) F 0x15B

defer, flash the screen.

**blink-screen** is one of the **defer** words of the display device interface. The terminal emulator package executes **blink-screen** when it has processed a character sequence that calls for ringing the console bell, but the console input device package has no “**ring-bell**” method.

Any standard package that uses the terminal emulator package shall, as part of the process of opening the terminal emulator package, set this **defer** word to a function with the following behavior:

  Cause some momentary discernible effect, afterwards leaving the screen in the same state as before.

**See also:** to. fb8-install
Numeric literal FCode. Followed by FCode-num32.

**FCode evaluation:**

Read an FCode-num32 from the current FCode program. If in interpretation state, perform the run-time semantics given below. If in compilation state, append the run-time semantics given below to the current definition.

**Run-time:**

Return n, the signed number with the same numerical value as the FCode-num32.

**FCODE ONLY** (Tokenized by numbers)

```
bljoin  ( b1.lo b2 b3 b4.hi -- quad ) F 0x7F
```

Join four bytes to form a quadlet.

The high bits of each of the four bytes must be zero.

“block”

Random-access, block-oriented device type.

Standard string value of the “device_type” property for disk (i.e., random-access, fixed-length block storage) devices. A standard package with this “device_type” property value shall implement the following methods.

- open, close, read, seek, load

A standard package with this “device_type” property value should implement the following method if the associated device is writeable.

- write

A standard package with this “device_type” property value may implement additional device-specific methods.

**NOTE**—Such packages often use the “deblocker” support package to implement the read, write, and seek methods and the “disk-label” support package to implement the load method.

```
block-size  ( -- block-len ) M
```

Return “granularity” for accesses to this device.

Return the “granularity” in bytes for accesses to this device. Perform all transfers to the device in multiples of this size. A returned value of 1 signifies that arbitrary transfer sizes are supported (up to the maximum specified by max-transfer).

```
b(loop)  ( -- ) F 0x15
```

End FCode do .. loop. Followed by FCode-offset.

**FCode evaluation:**

Read FCode-offset, whose target is the matching b(do) or b(?do), from the current FCode program and perform the compilation semantics of loop. Then, if the current definition is temporary and the depth of the control flow stack is the same as its depth when the temporary current definition was initiated, perform the FCode evaluation semantics of b(;) and execute the temporary current definition.

**FCODE ONLY** (Tokenized by loop)

```
b(+loop)  ( delta -- ) F 0x16
```

End FCode do .. +loop. Followed by FCode-offset.

**FCode evaluation:**

Read FCode-offset, whose target is the matching b(do) or b(?do), from the current FCode program and perform the compilation semantics of +loop. Then, if the current definition is temporary and the depth of the control flow stack is the same as its depth when the temporary current definition was initiated, perform the FCode evaluation semantics of b(;) and execute the temporary current definition.

**FCODE ONLY** (Tokenized by +loop)
b(<mark) (F: --) F 0xB1
Target of backward bbranch or b?branch.

FCode evaluation: (F: --)
Perform the interpretation or compilation semantics of begin.

FCODE ONLY (Tokenized by begin)

body> (a-addr -- xt) F 0x85
Convert data field address to execution token.

>bbody (xt -- a-addr) A,F 0x86
Convert execution token to data field address.

b(of) (sel of-val -- sel | <nothing>) F 0x1C
FCode for of in case statement. Followed by FCode-offset.

FCode evaluation: (F: /FCode-offset/ --)
Read FCode-offset, whose target is the matching b(endof), from the current FCode program and perform the compilation semantics of of.

FCODE ONLY (Tokenized by of)

boot ("{param-text}<eol>" --)
Load and execute a program, specified by param-text.

Perform whatever system-dependent steps are necessary to ensure a suitable state for booting, in the event that user actions have interrupted the normal start-up procedure. Then perform the function of load in order to load a client program from the device specified by the command line arguments.

If the loading process succeeds, perform the function of go to execute the client program.

Used as:
ok boot
ok boot device-specifier
ok boot arguments
ok boot device arguments

boot-command ( -- addr len) N
Command executed if auto-boot? is true.

The value of this configuration variable is a string that is evaluated as with evaluate.
Configuration variable type: string[32]. Suggested default value: boot.

boot-device ( -- dev-str dev-len) N
Default device-name for boot, if diagnostic-mode? is false.

dev-string is a device-specifier or a list of device-specifiers, as described in load.
Configuration variable type: string[32]. Suggested default value: disk.

boot-file ( -- arg-str arg-len) N
Default arguments for boot, if diagnostic-mode? is false.

Configuration variable type: string[32]. Suggested default value: an empty string.

"bootargs" S
Standard property name containing the chosen boot command arguments.

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

This property appears in the /chosen node if a boot or load command has been issued since the firmware was last reset. Its value is the arguments field of the most recent boot command.
"bootpath"  
Standard property name containing the chosen boot device path.

prop-encoded-array:  
Text string, encoded with `encode-string`.

This property appears in the `/chosen` node if a `boot` or `load` command has been issued since the firmware was last reset. Its value is the complete device path to which the `device-specifier` of that command was resolved.

`bounds`  
Prepare arguments for `do` or `?do` loop.

Equivalent to: `over + swap` 

`+bp`  
Add the given address to the breakpoint list.

`-bp`  
Remove the breakpoint at the given address.

`--bp`  
Remove most recently set breakpoint (repeat if desired).

`.bp`  
Display a list of all locations that are breakpoints.

`bpoff`  
Remove all breakpoints from the breakpoint list.

`.breakpoint`  
Action performed when breakpoint occurs.

Execute this command whenever a breakpoint occurs. The default behavior is the `.instruction` command.

`.breakpoint` is a `defer` command, alterable with the `to` command. For example, the following example shows how to display registers at every breakpoint.

Use as: `['] .registers to .breakpoint`

`b(>resolve)`  
Target of forward `bbranch` or `b?branch`.

FCCode evaluation:

Perform the compilation semantics of `then`. Then, if the current definition is temporary and the depth of the control flow stack is the same as its depth when the temporary current definition was initiated, perform the FCCode evaluation semantics of `b();` and execute the temporary current definition.

FCODE ONLY (Tokenized by `else`, `then`, and `repeat`)

`bs`  
ASCII code for “backspace” character.
**b(to)**

\[(x \rightarrow)\quad F\quad 0xC3\]

FCode for setting **values** and **defers**. Followed by **FCode#**.

**FCode evaluation:**

\[(F: /FCode#/ \rightarrow)\]

Read **FCode#** from the current FCode program.

If in interpretation state:

- Perform the execution semantics given below.

If in compilation state:

- Append the execution semantics given below to the current definition.

**Execution:**

\[(x \rightarrow)\]

Set the value associated with the definition corresponding to **FCode#** to \(x\). The interpretation of \(x\) and the method of storage depend on the type (e.g., **value** or **defer**) of the definition being stored to.

**FCODE ONLY** (Tokenized by \(\text{to}\))

**buffer:**

\[(E: -- a-addr)\quad T\quad (\text{len "new-name" } > \rightarrow)\]

Creates a named data buffer. \(\text{new-name}\) returns address.

Creates a data variable named \(\text{new-name}\) and allocates a data buffer of \(\text{len}\) bytes (using \(\text{alloc-mem}\)). Upon later execution of \(\text{new-name}\), return the address \(a-addr\) of the first byte of the buffer.

**Used as:** \(\text{ok 100 buffer: new-name}\)

**Later used as:** \(\text{55 new-name 20 + c}\!\!1\)

**Tokenizer equivalent:** \(\text{new-token|named-token|external-token b(buffer:)}\)

In FCode source, **buffer:** cannot be used inside a colon definition.

**NOTE**—The memory allocated by **buffer:** is not suitable for DMA.

**b(value)**

\[(E: -- x)\quad F\quad 0xB8\]

\[(F: x \rightarrow)\]

Defines type of new FCode function as **value**.

**FCode evaluation:**

\[(F: x \rightarrow)\]

If **instance** has been executed since the last execution of \(\text{b(buffer:)\}, \text{b(variable)\), b(value)\), or \(\text{b(defer)}\), allocate one cell of storage in the current package’s initialized data area; otherwise, allocate the storage in data space. Set the initial value of that cell to \(x\). Define the behavior of the most recently created FCode function to have the execution semantics given below.

**Execution:** (of defined word)

\[(\rightarrow x)\]

Return the value \(x\) associated with the defined word.

The value associated with the defined word can be changed later with **b(to)**.

**FCODE ONLY** (Tokenized by **value**)  

**b(variable)**

\[(E: -- a-addr)\quad F\quad 0xB9\]

\[(F: \rightarrow)\]

Defines type of new FCode function as **variable**.

**FCode evaluation:**

\[(F: \rightarrow)\]

If **instance** has been executed since the last execution of \(\text{b(buffer:)\}, \text{b(variable)\), b(value)\), or \(\text{b(defer)}\), allocate one cell of storage in the current package’s initialized data area; otherwise, allocate the storage in data space. Set the initial value of that cell to zero. Define the behavior of the most recently created FCode function to have the execution semantics given below.

**Execution:** (of defined word)

\[(\rightarrow a-addr)\]

Place \(a-addr\), the address of the allocated cell, on the stack.

**FCODE ONLY** (Tokenized by **variable**)  

**bwjoin**

\[(b.lo b.hi \rightarrow w)\quad F\quad 0xB0\]

Join two bytes to form a doublet.

The high bits of each of the two bytes must be zero.
**“byte”**
Sequential-access, record-oriented device type.

Standard string value of the “**device_type**” property for tape (i.e., sequential-access, record-oriented storage) devices.

A standard package with this “**device_type**” property value shall implement the following methods.

- **open**, **close**, **read**, **seek**, **load**

A standard package with this “**device_type**” property value should implement the following method if the associated device is writable.

- **write**

A standard package with this “**device_type**” property value may implement additional device-specific methods.

**Additional Requirements for the** **seek** **method:**
 Seek to the byte numbered **pos.lo** within the tape file **pos.hi**. If **pos.lo** and **pos.hi** are both zero, rewind the tape.
Return **false** if successful, **true** if not successful.

**Additional Requirements for the** **load** **method:**
 Read a client program from the tape file specified by the value of the **instance-argument** text string (as returned by **my-args**). That value is the string representation of a decimal integer. If the **instance-argument** string is empty, use tape file zero.
Place the program at memory address **addr** and return its length **len**.

**NOTE**—Such packages often use the “**deblocker**” support package to implement the **read**, **write**, and **seek** methods.

---

**byte-load**

( addr xt -- ) F 0x23E

Interpret FCode beginning at location **addr**.

Save the state of the FCode evaluator, including the location of the next byte to be interpreted, the internal state variable **fcode-end**, the size of **FCode-offsets**, the assignments of FCode numbers to program-defined FCode functions, the **spread** value, and the specification of the current FCode access procedure.

Set the internal state variable **fcode-end** to false. Set **spread**, the initial distance between successive FCode bytes, to one.

If **xt** is one, set the FCode access procedure to **rb@**. Otherwise, set the FCode access procedure to the definition whose execution token is **xt**.

Assign the FCode function **ferror** to all FCode numbers in the program-defined range. Evaluate the FCode program, reading successive FCode bytes by repeated execution of the FCode access procedure as described below, continuing evaluation until **fcode-end** becomes true (e.g., as a result of the execution of **end0**).

The stack effect of the FCode access procedure is ( **addr1 -- byte** ) where **addr1** is a number that selects the FCode byte **byte**; its precise meaning depends on the FCode access procedure. The first time that a particular invocation of **byte-load** executes the FCode access procedure, **addr1** is the same as **addr**. Each subsequent time, **addr1** exceeds the previous value of **addr1** by the current value of **spread**.

When evaluation of this FCode program ceases, or if a **throw** that is not caught at a lower level is executed during the FCode evaluation, restore the state of the FCode evaluator to the saved values.

**NOTE**—**byte-load** does not itself create a new device node as a “container” for any properties and methods defined by the FCode program that **byte-load** evaluates. If, as is commonly the case, it is desirable to create such a device node, that must be done as a separate step, for example by executing **new-device** and **set-args** before executing **byte-load**, and by executing **finish-device** afterwards. If **byte-load** is to be executed as a user interface command, instead of as an FCode function, additional setup is usually necessary before executing **new-device**; see **begin-package** for more information.

---

**c!**

( byte addr -- ) A,F 0x75

Store byte to **addr**.

See: **rb!**.

**c,**

( byte -- ) A,F 0xD0

Compile a byte into the dictionary.

**c;**

( code-sys -- )

End creation of machine-code command; will return to caller.

Assemble code so that the created machine-code command, when executed, returns control to the calling routine. **code-sys** is balanced by the corresponding **code** or **label**.
The number of address units to a byte: one.

Synonym for `chars`.  
**Tokenizer equivalent:** `chars`

Fetch byte from `addr`.  
See: `rb@`.

Increment `addr1` by `index` times the value of `/c`.

Synonym for `char+`.  
**Tokenizer equivalent:** `char+`

Execute specified client program callback routine.

<table>
<thead>
<tr>
<th>Cell Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>service</td>
<td>The address of a null-terminated string containing service-name.</td>
</tr>
<tr>
<td>N-args</td>
<td>1</td>
</tr>
<tr>
<td>N-returns</td>
<td>1</td>
</tr>
<tr>
<td>arg1</td>
<td>The address of a null terminated string containing arguments.</td>
</tr>
<tr>
<td>ret1</td>
<td>&lt;One uninitialized return value cell.&gt;</td>
</tr>
</tbody>
</table>

When the handler returns, `throw` the value returned in the `ret1` cell.

**NOTE**—With an argument of zero, `throw` is effectively a no-op; thus, to return successfully, the application callback handler should return a zero in the `ret1` cell.
$callback ( argv ... arg1 nargs addr len -- retn ... ret2 Nreturns-1 )

Execute specified client program callback routine.

If a client program callback handler has not been installed (as with the set-callback client interface service), signal an error by executing throw with a system-dependent nonzero argument. Otherwise, call the client program callback handler with an argument array containing the following items:

<table>
<thead>
<tr>
<th>Cell Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>service</td>
<td>The address of a null-terminated string containing the application callback service-name defined by addr, len</td>
</tr>
<tr>
<td>N-args</td>
<td>nargs</td>
</tr>
<tr>
<td>N-returns</td>
<td>&lt;An integer with minimum value 6. See below.&gt;</td>
</tr>
<tr>
<td>arg1, ..., argN</td>
<td>arg1 ... argn</td>
</tr>
<tr>
<td>ret1, ..., retN</td>
<td>&lt;A minimum of 6 uninitialized return value cells. See below.&gt;</td>
</tr>
</tbody>
</table>

The argument array shall have at least six cells available for return values, and N_returns shall be set to the number of such cells before calling the handler.

When the handler returns, throw the value contained in the ret1 cell. If throw returns (i.e., if ret1 contained zero), push onto the data stack the return values retN (pushed first) through ret2 (pushed last), followed by the number of return values, which is one less than the value contained in the N_returns cell.

NOTE—The value in N_returns after the handler returns will not necessarily be the same as the value that was placed there prior to calling the handler. Prior to calling the handler, it indicates the number of available cells for return values, and after calling the handler, it indicates the number of return cells actually returned by the handler.

$call-method ( ... method-str method-len ihandle -- ??? ) F 0x20E

Execute the method named method-string in the instance ihandle.

Save the value of my-self, set my-self to ihandle (thus making ihandle the current instance), execute the indicated method, and restore my-self to the saved value. If the called package has no such method, signal an error with throw.

call-package ( ... xt ihandle -- ??? ) F 0x208

Execute the method xt within the instance ihandle.

Save the value of my-self, set my-self to ihandle (thus making ihandle the current instance), execute the method xt, and restore my-self to the saved value.

xt is typically obtained with find-method.

$call-parent ( ... method-str method-len -- ??? ) F 0x209

Execute the method named method-string in the parent instance.

Equivalent to: my-parent $call-method

If the called package has no such method, signal an error with throw.

.calls ( xt -- )

Display all commands which use the execution token xt.

Used as: ['] test-name .calls

NOTE—Only direct usages are found. Thus, if name2 calls name1 and name3 calls name2, then:

['] name1 .calls

will list name2 but not name3.

carret ( -- 0x0D ) T

ASCII code for “carriage-return” character.

Tokenizer equivalent: b(lit) 00 00 00 0x0D
case

Begin a case (multiple selection) statement.

**Interpretation:**

Enter compilation state, initiating a temporary current definition in a region of memory other than the data space. Then perform the compilation semantics of ANS Forth **CASE**.

**Compilation:**

Same as ANS Forth.

**Run-time:**

Same as ANS Forth.

**Tokenizer equivalent:** `b(case)`

ANS Forth note: Also works outside of a definition.

---

catch

Execute command indicated by `xt`. Return `throw` result.

The value of `my-self` shall be included within the exception frame.

ANS Forth note: Also saves `my-self`.

cell+

Increment `addr1` by the value of `/n`.

cells

Multiply `nu1` by the value of `/n`.

char

Generate ASCII code for next character from input buffer.

char+

Increment `addr1` by the value of `/c`.

[char]

Generate ASCII code for next character from input buffer.

---

**character-set**

Standard property name to specify the character set for this device.

prop-encoded array:

Text string, encoded with `encode-string`.

This standard property applies to packages implementing “device_type”, “serial”, or “display”. The value of this property defines the character set for this device. A typical value is "ISO8859-1". The character set names are as defined by the X Registry for use with the X Window System.

**Used as:** "ISO8859-1" `encode-string" character-set" property

For more information about the X Registry contact:

Bob Scheifler
Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139
**char-height** ( -- height ) F 0x16C

*char-height* is a *value* that is used by the “fb1” and “fb8” frame-buffer support packages. It denotes the height of a character in pixels.

Any standard package that uses one of the frame-buffer support packages shall set this value prior to executing either `fb1-install` or `fb8-install`. That is typically done by executing `set-font`.

**chars** ( nu1 -- nu2 ) A,F 0x66

Multiply nu1 by the value of `/c`.

**char-width** ( -- width ) F 0x16D

*char-width* is a *value* that is used by the “fb1” and “fb8” frame-buffer support packages. It denotes the width of a character in pixels (the frame-buffer support packages use fixed-width fonts, thus all characters are the same width).

Any standard package that uses one of the frame-buffer support packages shall set this value prior to executing either `fb1-install` or `fb8-install`. That is typically done by executing `set-font`.

**child** ( phandle.parent -- phandle.child ) F 0x23B

Return the phandle of the first child node of parent.

*Phandle.child* is the node identifier of the node that is the first child of the device node *phandle.parent*, or zero if there are no children.

“/chosen” S

The node containing run-time choices

See: 3.5 for a complete description.

**claim** ( [ addr ... ] size ... align -- base ... ) M

Allocate (claim) addressable resource.

Allocates size ... (whose format depends on the package) bytes of the addressable resource managed by the package containing this method. If align is zero, the allocated range begins at the address addr ... (whose format depends on the package). Otherwise, addr ... is not present, and an aligned address is automatically chosen. The alignment boundary is the smallest power of two greater than or equal to the value of align; an align value of 1 signifies one-byte alignment.

Base ... (whose format is the same as addr ...) is the address that was allocated (equal to addr ... if align was 0).

If the operation fails, uses `throw` to signal the error.

*Claim* does not automatically create an address translation for the allocated resource. See 3.6.5.

**NOTE**—This method provides fine-grained control over the allocation of addressable resources. In general, such control is needed only by system-specific programs. General-purpose memory allocation can be accomplished in a portable fashion by `alloc-mem`.

See also: `alloc-mem`, “available”, `free-mem`, `release`.

**clear** ( ... -- )

Empty the stack.

This command is useful as a development tool. However, it is almost always inappropriate to use this command in a program.

**close** ( -- ) M

Close this previously opened device.

Restore the device (which has been previously opened) to its “not-in-use” state. Typical behavior is to turn off the device, unmap it, and deallocate any resources that were allocated by `open`.

Any standard package that has an `open` method shall also have a `close` method.

**NOTE**—When closing an instance chain, a particular instance’s `close` method is executed before its parent instances are closed, so the parent’s methods can still be used during the execution of `close`.
**close-dev**

( ihandle -- )

Close device and all of its parents.

**close-package**

( ihandle -- ) F 0x206

Close the specified package instance.

Close the instance identified by *ihandle* by calling the package’s `close` method and then destroying the instance.

**code**

(E: ... -- ??? ) A

( "new-name< >" -- code-sys )

Begin creation of machine-code command called *new-name*.

Interpret commands which follow as assembler mnemonics.

Note that if the assembler is not installed, `code` is still present, except that machine-code must be entered into the dictionary explicitly by value, i.e., with `c`, `w`, `l`, and `,`.

The machine-code command is terminated by the `c;` or `end-code` commands.

**Used as:**

```
ok  code new-name
ok  { assembler mnemonics}
ok  c;
```

**Later used as:**

```
new-name  ( execute machine-code)
```

code-sys is balanced by the corresponding `c;` or `end-code` commands.

**column#**

( -- column# ) F 0x153

`value`, return the current cursor column number.

Return the current horizontal position of the text cursor.

**NOTE**—A value of zero represents the leftmost cursor position of the *text window*, not the leftmost pixel of the frame-buffer.

See: `window-left` for more details.

**#columns**

( -- columns ) F 0x151

`value`, return number of columns of text in *text window*.

**#columns** is a `value` that is part of the display device interface. The terminal emulator package uses it to determine the width (number of character columns) of the text region that it manages. The “fb1” and “fb8” frame-buffer support packages also use it for a similar purpose.

Any standard package that uses the terminal emulator package shall, during the execution of its “open” method (see `is-install`), set this `value` to the desired width of the text region (perhaps, but not necessarily, by executing `fb1-install` or `fb8-install`).

See also: `to`, `fb8-install`.

**comp**

( addr1 addr2 len -- n ) F 0x7A

Compare two strings of length *len*.

Compare the string specified by `addr1` and `len` to the string specified by `addr2` and `len`. The strings are compared, beginning at the given addresses, character by character up to the length `len`, or until a difference is found. If the two strings are identical, then `n` is zero. Otherwise, `n` is negative one if the first unmatching character in the string beginning at `addr1` has a lesser numeric value than the corresponding character in the string beginning at `addr2`, whereas `n` is one if the first unmatching character in the string beginning at `addr1` has a greater numeric value than the corresponding character in the string beginning at `addr2`. 
“compatible”  
Standard property name to define alternate "name" property values.

prop-encoded-array:  
The concatenation, with encode+, of an arbitrary number of text strings, each encoded with encode-string.  
Specifies a list of devices with which this device is compatible. This is used by a client program when it is trying to 
determine the appropriate driver for this device, in case the client program has no driver matching the value of the 
"name" property. 
The text strings of which this property value is composed follow the same conventions and limitations as the value of the 
"name" property.

A client program, when searching for an operating system driver for the device represented by a device node containing 
this property, should do the following: 
First look for a driver matching this device’s "name" property. 
If no match is found, look for a driver matching the first text string in the "compatible" property. Failing that, try 
the second text string, and so on. 
Used as:
" XYZCO,dev-name"  encode-string  
" ABCCO,my-dev"  encode-string  encode+  
" RST,dev21-type4"  encode-string  encode+  
" compatible"  property

NOTE—The “compatibility” of a client program’s device driver with a specific piece of hardware is ultimately 
determined by the manufacturers of the client program software and the hardware device, not by Open Firmware. When an 
FCode program “registers” a “compatible” property with Open Firmware, the manufacturer of that hardware is 
sending a “hint” to client program software saying that this specific piece of hardware is either substantially similar to or 
identical with the hardware device(s) named by the “compatible” property. This might be done to help the client 
program software choose a device driver when it would not recognize the contents of the "name" property as a supported 
device, but would recognize an alternate "name" within the “compatible” property.

compile
Compile following command at run time. 
Used within a colon definition. When the colon definition is later executed, append the execution semantics that 
immediately follow those of compile within the definition that contains compile to the current definition.

compile,  
Compile the behavior of the word given by xt.

[compile]
Compile the immediately following command.

constant
Create a named constant. new-name returns value x.

Tokenizer equivalent: new-token|named-token|external-token b(constant)  
ANS Forth/tokenizer difference: In FCode source, constant cannot appear inside a colon definition.

2constant
Create a named two-number constant.
**control**

( [text< >] -- char )
T

Generate control-code for the immediately following character.

Generate control-codes by calculating the ASCII code for the following character and setting all but the lower five bits to zero. For example, either of control B or control b will leave the value 0x02 on the stack.

**Interpretation:**
( [text< >] -- char )

Skip leading space delimiters. Parse text delimited by a space. Put the integer value of the least significant 5 bits of the first character of text on the stack.

**Compilation:**
( [text< >] -- )

Skip leading space delimiters. Parse text delimited by a space. Append the run-time semantics given below to the current definition.

**Run-time:**
( -- char )

Place char, the integer value of the least significant 5 bits of first character of text, on the stack.

**Used as:** control Boo ( 0x02 )

**Tokenizer equivalent:** b(lit) 00 00 00 xx-byte

---

**count**

( pstr -- str len )
A,F 0x84

Unpack a counted string to a text string.

**cppeek**

( addr -- false | byte true )
F 0x220

Attempt to fetch the byte at addr.

Return the data and true if the access was successful. A false return indicates that a read access error occurred.

**cpoke**

( byte addr -- okay? )
F 0x223

Attempt to store the byte to addr.

Return true if the access was successful. A false return indicates that a write access error occurred.

**cr**

( -- )
A,F 0x92

Subsequent output goes to the next line.

Terminate a line on the display and move the cursor to the beginning of the next line. The actual control codes issued are implementation-dependent.

**cr**

( -- )
F 0x91

Output the carriage-return character, or carret (0x0D).

**NOTE**—The most common use for (cr) is for reporting the progress of a test that has many steps. By using (cr) instead of cr, the progress report appears on a single line instead of scrolling.

**create**

( E: -- a-addr )
A,T

Create a new command; behavior set by further commands.

**NOTE**—Since FCode has no function that is equivalent to ANS Forth’s does>, create cannot be used in conjunction with does> in FCode source. However, it is still useful for arrays of initialized data, in which case it is typically followed by sequences of functions like c, w, and , taking care to ensure proper address alignment.

**Tokenizer equivalent:** new-token|named-token|external-token b(create)

**ANS Forth/tokenizer difference:** In FCode source, create cannot appear inside a colon definition.

**$create**

( E: -- a-addr )

Call create, new name specified by name string.

---

130
**ctrace**  
( -- )  
Display saved call stack, showing subroutines calls and arguments.

Display the subroutine call stack that was in effect when the program state was saved (i.e., when the program was suspended). The format of the display is implementation-dependent.

**d#**  
( [number< >] -- n )  
T

Interpret the following number as a decimal number (base ten).

**Interpretation:**  
( [number< >] -- n )

Skip leading space delimiters. Parse number delimited by a space. Convert the string number to an integer n using a conversion radix of ten. Put n on the stack. An ambiguous condition exists if the conversion fails.

**Compilation:**  
( [number< >] -- )

Skip leading space delimiters. Parse number delimited by a space. Convert the string number to an integer n using a conversion radix of ten. Append the run-time semantics given below to the current definition. An ambiguous condition exists if the conversion fails.

**Run-time:**  
( -- n )

Place n on the stack.

The number is interpreted in decimal regardless of the current value in base. The value of base is unchanged.

**Used as:** d# 1001 ( 1001 )

**Tokenizer equivalent:** b(lit) xx-byte xx-byte xx-byte xx-byte

**d+**  
( d1 d2 -- d.sum )  
A,F 0xD8

Add d1 to d2 giving double-number d.sum.

**d−**  
( d1 d2 -- d.diff )  
A,F 0xD9

Subtract d2 from d1 giving double-number d.diff.

**.d**  
( n -- )  
T

Display a signed number (and space) in decimal.

Ignore the value in base and leave it unchanged. Also display a single trailing space.

**Tokenizer equivalent:** base @ swap 10 base ! . base !

```
"deblocker"
```

Support package; handles byte I/O for block-oriented devices.

See: 3.8.3.

**debug**  
( "old-name< >" -- )  

Mark the command old-name for debugging.

**Used as:** ok debug old-name

Subsequent attempts to execute that command enter the Forth source-level debugger. A standard system that implements this feature may allow several commands to be marked for debugging simultaneously, but only one is required.

During the execution of a debugged command, before the execution of each command called by the debugged command, display the contents of the stack followed by the name of the command that is about to be executed.

Debugging occurs in either “step mode” or “trace mode”, controlled by the stepping and tracing commands. “Step mode” allows the user to control the progress of execution, whereas “trace mode” causes execution to continue automatically (but with calling information displayed).

**NOTE**—Debug mode can be turned off with the debug-off command.

**NOTE**—The system does not necessarily operate at full speed when one or more commands are marked for debugging.

Debugging basic Forth commands (which could have been used in the implementation of debug) is not recommended. The system may ignore requests to debug words that are “unsafe” to debug.
(debug ( xt -- )
  Mark the command indicated by xt for debugging.
  See: debug for more information.

debug-off ( -- )
  Turn off the Forth source-level debugger.

decimal ( -- ) A,T
  Set numeric conversion radix to ten.
  Tokenizer: If decimal is encountered in FCode source outside a definition, set the tokenizer’s numeric conversion
  radix to ten. If decimal is encountered in FCode source inside a definition, append the following sequence to the FCode
  program that is being created.
  Tokenizer equivalent: 10 base !
  ANS Forth/tokenizer difference: ANS Forth has no separate “tokenizing” behavior.

decode-bytes ( prop-addr1 prop-len1 data-len -- prop-addr2 prop-len2 data-addr data-len )T
  Decode a byte array from a prop-encoded-array.
  Return the remainder of the array prop-addr2 prop-len2 and the decoded byte array data-addr data-len.
  Tokenizer equivalent: >r over r@ + swap r@ - rot r>

decode-int ( prop-addr1 prop-len1 -- prop-addr2 prop-len2 n ) F 0x21B
  Decode a number from a prop-encoded-array.
  Decode a quadlet number from the beginning of the array prop-addr1 prop-len1, return the remainder of the array prop-
  addr2 prop-len2 and the decoded number n.

decode-phys ( prop-addr1 prop-len1 -- prop-addr2 prop-len2 phys.lo ... phys.hi ) F 0x128
  Decode a unit address from a prop-encoded-array.
  The unit address is a list of cells denoting a physical address, encoded as defined in encode-phys. decode-phys
  decodes that list from the beginning of the prop-encoded array denoted by prop-addr1 prop-len1, returning the remainder
  of the array prop-addr2 prop-len2 and the list of address components phys.lo ... phys.hi.
  The number of cells in the list phys.lo ... phys.hi is determined by the value of the #address-cells property of the
  parent node.

decode-string ( prop-addr1 prop-len1 -- prop-addr2 prop-len2 str len ) F 0x21C
  Decode a string from a prop-encoded-array.
  Decode a (null-terminated) string from the beginning of the array prop-addr1 prop-len1, return the remainder of the array prop-
  addr2 prop-len2 and the decoded string str len. len reflects the length of the decoded string not including the null
  terminator.

decode-unit ( addr len -- phys.lo ... phys.hi ) M
  Convert text unit-string to physical address.
  Convert unit-string, the text string representation, to phys.lo ... phys.hi, the numerical representation of a physical address
  within the address space defined by this device node. The number of cells in the list phys.lo ... phys.hi is determined by
  the value of the #address-cells property of this node.
  decode-unit is a static method.
  Used as: " 3,4000" decode-unit { ff004000 d4000003 }

default-font ( -- addr width height advance min-char #glyphs ) F 0x16A
  Return the font parameters for the default system font.
  Used as: default-font ( ... ) set-font
  See also: set-font
**defer**

Create a command with alterable behavior; alter with **to**.

Skip leading space delimiters. Parse `new-name` delimited by a space. Create a definition for `new-name` with the execution semantics defined below, initially associating with it a definition that displays a message indicating execution of an uninitialized **defer** word.

The definition associated with `new-name` can be changed later by placing the execution token of the new definition on the stack and executing the phrase **to new-name**.

`new-name` is referred to as a **defer** word.

**Used as:** ok defer new-name ( create new command)  
**Execution:** (of `new-name`) ( ... -- ???)

Execute the definition currently associated with `new-name`.

**Later used as:**

`[' old-name to new-name  ( load with old command)  
`new-name  ( execute new command, behavior is old command)`

In FCode source, **defer** cannot appear inside a colon definition.  
**Tokenizer equivalent:** new-token|named-token|external-token b(defer)

**delete-characters**

**defer**, delete `n` characters to the right of the cursor.

**delete-characters** is one of the **defer** words of the display device interface. The terminal emulator package executes **delete-characters** when it has processed a character sequence that requires the deletion of characters to the right of the cursor.

Any standard package that uses the terminal emulator package shall, as part of the process of opening the terminal emulator package, set this **defer** word to a function with the following behavior:

Move the remainder of the line to the left to fill the deleted positions and erase the `n` rightmost columns in the line without moving the cursor.

**See also:** **to**, **fb8-install**

**delete-lines**

**defer**, delete `n` lines at and below the cursor line.

**delete-lines** is one of the **defer** words of the display device interface. The terminal emulator package executes **delete-lines** when it has processed a character sequence that requires the deletion of lines of text below the line containing the cursor.

Any standard package that uses the terminal emulator package shall, as part of the process of opening the terminal emulator package, set this **defer** word to a function with the following behavior:

Move the following lines up to fill the deleted portions and erase the `n` lines at the bottom without moving the cursor.

**See also:** **to**, **fb8-install**

**delete-property**

Delete the named property in the **active package**.

If the named property does not exist in the **active package**, take no action.

**depth**

Return count of items on the stack.

**dev**

Make the specified device node the **active package**.

Parse `device-specifier` delimited by end of line. Perform the equivalent of **find-device** with `device-specifier` as its argument.

**Used as:** ok dev device-specifier <eol>
devalias

( "{alias-name}< >{device-path}<eol>" -- )
Create device alias or display current alias(es).
If alias-name and device-path are specified, create a device alias named alias-name representing device-path.
If an alias with the same name already exists, the new value supersedes the old.
Used as: ok devalias alias-name /full/pathname
If only alias-name is specified, display the device-path corresponding to alias-name (if this alias exists). If nothing is specified after devalias, display all current existing device aliases.

device-end

( -- )
Unselect the active package, leaving none selected.

device-name

( str len -- ) F 0x201
Create the “name” property, value is indicated string.
Shorthand command to create a property in the active package whose property name is “name”.
Equivalent to: encode-string " name" property
Used as: " XYZCO,my-dev-name" device-name

device-type

( str len -- ) F 0x11A
Create “device_type” property, value is indicated string.
Shorthand command to create a property in the active package whose property name is “device_type”.
Equivalent to: encode-string " device_type" property
Used as: " block" device-type
See: “device_type” glossary entry for more information.
NOTE—There is a spelling difference between the name of the property (“device_type”) and the name of the command that can be called to create it (“device-type”).

“device_type”

S
Standard property name to specify the implemented interface.
prop-encoded-array:
- Text string encoded with encode-string.
See: 3.4.5 for more information and for a list of supported string values for this property and their meanings.
NOTE—This property can be created with property or with device-type. Note the spelling difference between the property name (“device_type”) and the command name “device-type”. This is an historical accident. The property name should have been “device-type” for consistency with the naming conventions generally used herein, but changing the property name would have resulted in compatibility problems for little payback.
This standard defines the following device types: “block” “byte” “display” “network” “serial”.
Used as: " network" encode-string " device_type" property

diag-device

( -- dev-str dev-len ) N
Default device-name for boot, if diagnostic-mode? is true.
dev-string is a device-specifier or a list of device-specifiers, as described in load.
Configuration variable type: string[32]. Suggested default value: net.

diag-file

( -- arg-str arg-len ) N
Default arguments for boot, if diagnostic-mode? is true.
Configuration variable type: string[32]. Suggested default value: diag.
diagnostic-mode? ( -- diag? ) F 0x120

If true, boot from diag sources and perform longer self-tests.

**diagnostic-mode?** controls several aspects of machine function, described next. During booting, **diagnostic-mode?** controls the choice of boot device and boot file, if not specified in the **boot** command. If **diagnostic-mode?** is true, the default boot device is specified by **diag-device** and the default boot file is specified by **diag-file**. If **diagnostic-mode?** is false, the default boot device is specified by **boot-device** and the default boot file is specified by **boot-file**. During machine power-on, **diagnostic-mode?** controls the extent of system self-test and controls the amount of informative messages displayed. If **diagnostic-mode?** is true, more extensive tests are performed and more messages are displayed. The details of self-test, however, are implementation-dependent.

FCode programs can use **diagnostic-mode?** to control the extent of the self-tests performed. While the specifics of use are controlled by the FCode program itself, the recommended use is described in the preceding paragraph. In other words, if **diagnostic-mode?** is true, more extensive tests are performed and more messages are displayed.

**diagnostic-mode?** will return true if any of the following conditions are met:
- diag-switch? is true
- machine diagnostic switch (system-dependent) is ON
- other system-dependent indicators request extensive diagnostics

**NOTE**—diag-switch? true implies diagnostic-mode? true, but diag-switch? false does not necessarily imply diagnostic-mode? false. Other system-dependent mechanisms can cause diagnostic-mode? to be true.

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

diag-switch? ( -- diag? ) N

If true, **diagnostic-mode?** returns true.

**NOTE**—diag-switch? true implies diagnostic-mode? true, but diag-switch? false does not necessarily imply diagnostic-mode? false. Other system-dependent mechanisms can cause diagnostic-mode? to be true.

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

digit ( char base -- digit true | char false ) F 0xA3

Convert a character to a digit in the given base. If the character is invalid, leave the character on the stack. The flag indicates the success or failure of the operation.

dis ( addr -- )

Begin disassembling at the given address.

The format of the disassembly, and the conditions for stopping disassembly, are ISA-dependent.

See also: 7.6.6.

+dis ( -- )

Continue disassembling where dis or +dis last stopped.

See: dis for more information.

“disk-label” S

Support package, interprets disk partitioning information.

See: 3.8.1 for more information.
“display” S

Graphic output display device type.

Standard string value of “device_type” property for user output devices with randomly addressable pixels.

“display” devices can be used for console output.

A standard package with this “device_type” property value shall implement the following methods.

open, close, write, draw-logo

A standard package with this “device_type” property value should implement the following method if an unexpected system reset can cause the display to become invisible (e.g., the video is turned off) and the display can be restored to visibility without performing memory mapping or memory allocation operations:

restore

Additional Requirements for the write method:

Display the sequence of len characters beginning at addr, interpreting escape sequences as described in the terminal emulator section.

A standard package with this “device_type” property value may implement additional device-specific methods.

See: 3.8.4; character-set.

NOTE—Such packages typically use the terminal emulator support package to process ANSI X3.64 escape sequences for the write method. “Dumb” frame-buffer devices typically use either the “fb1” or the “fb8” support package to implement the “Character Map” defer words interface. More complicated display devices, such as those with hardware acceleration, typically implement that interface directly.

display-status ( n -- ) F,O 0x121

Show the result of a device self-test.

n is a device-dependent number indicating the status of the device or the progress of the self-test.

The method of showing the result is system-dependent, but is intended to use some device that is likely to be available at an early phase of system start-up, even if much of the system is not operational. For example, diagnostic LEDs are often used.

dl ( -- )

Download and execute Forth text; end with ^D.

Receive text from the current input source and store it in a buffer until an EOT (0x04, or control-D) character is received.

Do not store the EOT character. Evaluate the contents of the buffer as with the eval command. The buffer size shall be at least 4096 characters.

NOTE—This is typically used with a serial line as the current input source. After issuing the dl command, the user typically issues commands to another computer to cause the desired Forth text (such as a text file) to be sent over the serial line, followed by the EOT (0x04, or control-D) character.

dma-alloc ( ... size -- virt ) M

Allocate a memory region for later use.

Allocate size bytes of memory, contiguous within the direct-memory-access address space of the device bus, suitable for direct memory access by a “bus master” device. Return the virtual address virt. That virtual address is suitable for CPU access to the allocated region, but, in general, dma-map-in must be used to convert it to an address suitable for direct memory access by the bus-master device.

Allocate the memory according to the most stringent alignment requirements for the bus.

Some hierarchical devices may require additional mapping space parameters.

See also: dma-map-in, dma-free

If the requested operation cannot be performed, a throw shall be called with an appropriate error message, as with abort”.

NOTE—Out-of-memory conditions may be detected and handled properly in the code with [‘] dma-alloc catch.

dma-free ( virt size -- ) M

Free memory allocated with dma-alloc.

Free size bytes of memory at virtual address virt, previously allocated by the dma-alloc method.
dma-map-in

Convert virtual address to device bus DMA address.

Convert the virtual address range virt size, previously allocated by the dma-alloc method, into an address suitable for DMA on the device bus. Return this address devaddr.

dma-map-in can also be used to map application-supplied data buffers for DMA use, if possible on the bus.

If the flag cacheable?, is nonzero, the caller wishes to make use of caches for the DMA buffer if they are available.

Immediately after dma-map-in has been executed, the contents of the address range as seen by the processor (the processor’s “view”) is the same as the contents as seen by the device that performs the DMA (the device’s “view”). After the DMA device has performed DMA or the processor has performed a write to the range in question, the contents of the address range as seen by the processor (the processor’s “view”) is not necessarily the same as the contents as seen by the device that performs the DMA (the device’s “view”). The two views can be made consistent by executing dma-map-out or dma-sync.

Some hierarchical devices may require additional mapping space parameters.

If the requested operation cannot be performed, a throw shall be called with an appropriate error message, as with abort".

NOTE—Out-of-memory conditions may be detected and handled properly in the code with ['] dma-map-in catch.

dma-map-out

Free DMA mapping set up with dma-map-in.

Free the DMA mapping specified by virt devaddr size, previously created with the dma-map-in method.

This will also have the effect of flushing all caches (with dma-sync) associated with that mapping.

dma-sync

Synchronize (flush) DMA memory caches.

Flush any memory caches associated with the DMA mapping virt devaddr size.

do

Start a counted loop; beginning index value is start.

Enter compilation state, initiating a temporary current definition in a region of memory other than the data space. Then perform the compilation semantics of ANS Forth DO.

Same as ANS Forth.

Same as ANS Forth.

Same as ANS Forth.

Same as ANS Forth.

Also works outside of a definition.

?do

Similar to do, but do not execute loop if limit = start.

Enter compilation state, initiating a temporary current definition in a region of memory other than the data space. Then perform the compilation semantics of ANS Forth ?DO.

Same as ANS Forth.

Same as ANS Forth.

Also works outside of a definition.
does> (C: colon-sys1 -- colon-sys2) A
   ( -- ) (R: sys1 -- )
   ( ... -- ... a-addr ) (R: -- sys2)
   (E: ... -- ???)

Set run-time behavior of a create...does> construct.

draw-character (char --) F 0x157
defer: draw a character at the current cursor position.

draw-character is one of the defer words of the display device interface. The terminal emulator package executes
when it has processed a character sequence that calls for the display of a printable character
(subsequently, the terminal package advances the cursor to the next character position).

Any standard package that uses the terminal emulator package shall, as part of the process of opening the terminal
emulator package, set this defer word to a function with the following behavior:

- Draw char on the display device at the cursor position. Other character positions on the line are unaffected.

See also: to, fb8-install

draw-logo (FCode function) (line# addr width height -- ) F 0x161
defer: draw (at line#) the logo stored at location addr.

draw-logo is one of the defer words of the display device interface. is-install creates a “draw-logo” method
whose behavior is to execute the draw-logo defer word. banner executes the “draw-logo” method of the
console output device, if that device has such a method.

Any standard package that uses the terminal emulator package shall, as part of the process of opening the terminal
emulator package, set this defer word to a function with the following behavior:

- Draw a logo whose upper-left corner coincides with the upper-left corner of the character position at the beginning of
text line line#. The logo can be either a device specific logo or the one-bit-per-pixel logo bit-map image specified by
addr, width, and height. The format of that bit-map is as follows:
addr is the starting address of the bit-map. width and height are its dimensions in pixels. Each bit of the bit-map
represents one pixel. The most significant bit of the first byte controls the upper-left-corner pixel. The next bit
controls the next pixel to the right and so on. A zero bit represents the background color, and a one bit represents the
foreground color.

See also: to, fb8-install

draw-logo (package method) (line# addr width height -- ) M

Draw a logo on an output device.

The arguments and semantics of this method are identical to those of the draw-logo FCode function.

NOTE—is-install automatically creates an implementation of this method that executes the draw-logo defer
word.

See also: “display”, banner, draw-logo (FCode function)

driver (addr len --) F,O 0x118
Creates the “name” property.

Removes the manufacturer name prefix from the string addr len, then creates the “name” property from the remainder of
the string. Previous versions of SBus firmware have implemented the process of removing the manufacturer name prefix
in inconsistent ways, thus there is no single definition of driver that will ensure backwards compatibility in all cases.

NOTE—SBus [B2] developers were advised to avoid the use of this FCode function when the inconsistency was
discovered, and the committee believes that its use has largely been eliminated.

drop (x --) A,F 0x46
Remove top item from the stack.

2drop (x1 x2 --) A,F 0x52
Remove top two items from the stack.
3drop \( \text{drop} \ x_1 \ x_2 \ x_3 \rightarrow \) 
Remove top three items from the stack.

**Tokenizer equivalent:** \text{drop} \ 2\text{drop}

dump \( \text{dump} \ ( \text{addr} \ \text{len} \rightarrow ) \) 
Display \text{len} bytes of memory starting at \text{addr}.

dup \( \text{dup} \ ( \ x \rightarrow \ x \ x \ x \ ) \) 
Duplicate the top item on the stack.

2dup \( \text{2dup} \ ( \ x_1 \ x_2 \rightarrow \ x_1 \ x_2 \ x_1 \ x_2 \ ) \) 
Duplicate the top two items on the stack.

3dup \( \text{3dup} \ ( \ x_1 \ x_2 \ x_3 \rightarrow \ x_1 \ x_2 \ x_3 \ x_1 \ x_2 \ x_3 \ ) \) 
Duplicate three stack items.

**Tokenizer equivalent:** 2 \text{pick} \ 2 \text{pick} \ 2 \text{pick}

?dup \( \text{?dup} \ ( \ x \rightarrow \ 0 \ | \ x \ ) \) 
Duplicate top stack item if it is nonzero.

else \( \text{else} \ ( \ C: \ \text{orig-sys1} \rightarrow \ \text{orig-sys2} \ ) \) 
When \text{if} flag was \text{false}, execute following code.

**Tokenizer equivalent:** \text{bbranch} \ +\text{offset} \ b(>\text{resolve})

**ANS Forth note:** Also works outside of a definition.

emit \( \text{emit} \ ( \ \text{char} \rightarrow ) \) 
Display the given character.

encode+ \( \text{encode+} \ ( \ \text{prop-addr1} \ \text{prop-len1} \ \text{prop-addr2} \ \text{prop-len2} \rightarrow \ \text{prop-addr3} \ \text{prop-len3} \ ) \) 
Concatenate two \text{prop-encoded-arrays} into a single array.

The property encoding of the result is the first array immediately followed by the second array. Consequently, \text{prop-len3} is equal to \text{prop-len1} plus \text{prop-len2}.

**Usage restriction:** The first array \text{prop-addr1} \ \text{prop-len1} must have been created just before the second array \text{prop-addr2} \ \text{prop-len2}, with no intervening dictionary allocation or other \text{prop-encoded-arrays} having been created.

In a typical implementation, \text{prop-addr3} is the same as \text{prop-addr1}.

**Used as:**

" some-text" \text{encode-string} \ ( \ \text{prop-addr1} \ \text{len1} ) 
5000 \text{encode-int} \ ( \ \text{prop-addr1} \ \text{len1} \ \text{prop-addr2} \ \text{len2} ) 
\text{encode+} \ ( \ \text{prop-addr} \ \text{len1+2} )

encode-bytes \( \text{encode-bytes} \ ( \ \text{data-addr} \ \text{data-len} \rightarrow \ \text{prop-addr} \ \text{prop-len} \ ) \) 
Encode a byte array into a \text{prop-encoded-array}.

The property encoding of a byte array is the sequence of bytes itself, with no additional bytes.

Contrast to \text{encode-string}, which appends a null byte to the end.

**NOTE**—In order to decode the encoded byte array, some additional method must be provided to know the length of the array. This could be by previous agreement or by encoding the length value first.

**Used as:** ( \ \text{data-addr} \ \text{data-len} ) \text{encode-bytes} ( \ \text{prop-addr} \ \text{prop-len} )
**encode-int**  
(n -- prop-addr prop-len)  
F 0x111

Encode a number into a prop-encoded-array.

The property encoding of a (quadlet) number is a sequence of 4 bytes, with the most significant byte first (i.e., at the smallest address).

No alignment is implied; the sequence of 4 bytes begins at the first available location.

**Used as:**  
5000 encode-int ( prop-addr prop-len )

**encode-phys**  
(phys.lo ... phys.hi -- prop-addr prop-len)  
F 0x113

Encode a unit address into a prop-encoded-array.

The property encoding of the unit address (a list of cells denoting a physical address) is the property encoding (as with **encode-int**) of phys.hi component, followed by the encoding of the component that appears on the stack below phys.hi, and so on, ending with the encoding of the phys.lo component.

The number of cells in the list phys.lo ... phys.hi is determined by the value of the *address-cells* property of the parent node.

**Used as:**  
(phys.lo ... phys.hi) encode-phys (prop-addr prop-len)

**encode-string**  
(str len -- prop-addr prop-len)  
F 0x114

Encode a string into a prop-encoded-array.

The property encoding of a string is the bytes of the string, followed by a null (binary 0) byte. The length identified by prop-len includes the null byte; thus prop-len is one more than len.

Contrast to **encode-bytes**, which does not append a null byte.

**Used as:**  
"some-text" encode-string (prop-addr prop-len)

**encode-unit**  
(phys.lo ... phys.hi -- unit-str unit-len)  
M

Convert physical address to text unit-string.

Convert phys.lo ... phys-high, the numerical representation, to unit-string, the text string representation of a physical address within the address space defined by this device node. The number of cells in the list phys.lo ... phys.hi is determined by the value of the *address-cells* property of this node.

**encode-unit** is a static method.

**end0**  
(user interface)  
( -- )

Cease interpreting this program.

**Interpretation:**  
( -- )

Perform the execution semantics given below.

**Compilation:**  
( -- )

Perform the execution semantics given below.

**Execution:**  
( -- )

Cause the command interpreter to ignore the remainder of the input buffer and all subsequent lines from the same input source.

**NOTE**—The optional user interface semantics of this command duplicate the purpose, but not the detailed behavior, of the FCode semantics. The detailed behavior differs because the user interface command interpreter processes text, while the FCode evaluator processes byte-encoded FCode programs.
end0 (FCode function) ( -- ) F 0x00

Cease interpreting this program.

FCode evaluation: (F: -- )

Set the internal state variable fcode-end to true, which will cause the FCode evaluator to cease evaluating the current FCode program immediately after this function is interpreted.

FCode Execution: ( -- )

Set the internal state variable fcode-end to true, which will cause the FCode evaluator to cease evaluating the current FCode program after the FCode function whose evaluation resulted in the execution of this function finishes.

NOTE—Normally, end0 appears at the end of an FCode program. However, it can be used within the body of an FCode program, for example, to terminate FCode evaluation based upon a conditional test. To do so, it is necessary to postpone the execution of end0 to prevent the FCode evaluator from exiting as soon as end0 is encountered. That can be accomplished with the phrase [''] end0 execute, usually within an if ... then construct.

end1 ( -- ) F 0xFF

Cease interpreting this program.

Same as end0.

NOTE—end1 is not intended to appear in source code. It is defined as a guard against mis-programmed ROMs. Unprogrammed regions of ROM usually appear as all ones or all zeroes. Having both 0x00 and 0xff defined as end codes stops the FCode interpreter if it enters an unprogrammed region of a ROM.

endcase (C: case-sys -- ) A,T

( sel | <nothing> -- )

Mark end of a case statement.

Compilation: (C: case-sys -- )

Perform the compilation semantics of ANS Forth ENDCASE. Then, if the current definition is temporary and the depth of the control flow stack is the same as its depth when the temporary current definition was initiated, perform the compilation semantics of ; and execute the temporary current definition.

Run-time: ( sel | <nothing> -- )

Same as ANS Forth.

Tokenizer equivalent: b(endcase)

ANS Forth note: Also works outside of a definition.

dend ( code-sys -- )

End creation of machine-code sequence.

No additional assembly language code is assembled.

code-sys is balanced by the corresponding code or label.

deof (C: case-sys1 of-sys -- case-sys2 ) A,T

( -- )

Mark end of clause; jump to end of case if match.

Tokenizer equivalent: b(endof) +offset

ANS Forth note: Also works outside of a definition.

dend-package ( -- )

Close the device tree entry set up with begin-package.

Perform the following:

Call finish-device to close the child device node.

Set the working vocabulary to Forth.

Call close-dev.

environment? ( str len -- false | value true ) A

Return system information based on input keyword.

The exact set of recognized keyword strings is implementation-dependent.
erase ( addr len -- ) A,T
  Set len bytes beginning at addr to zero.

  Tokenizer equivalent: 0 fill

erase-screen ( -- ) F 0x15A
defer, clear the screen.

erase-screen is one of the defer words of the display device interface. The terminal emulator package executes erase-screen when the console device is first activated, and also when it has processed a character sequence that calls for the screen to be cleared.

Any standard package that uses the terminal emulator package shall, as part of the process of opening the terminal emulator package, set this defer word to a function with the following behavior:
  Clear the entire screen (not just the text window), setting it to the background color.

See also: to, fb8-install

eval ( ... str len -- ??? ) T
  Synonym for evaluate.

  Tokenizer equivalent: evaluate

evaluate ( ... str len -- ??? ) A,F 0xCD
  Interpret Forth text from the given string.

even ( n -- n | n+1 )
  Round to nearest even integer _ n.

execute ( ... xt -- ??? ) A,F 0x1D
  Execute the command whose execution token is xt.

execute-device-method ( ... dev-str dev-len method-str method-len -- ... false | ??? true)
  Execute the named method in the package named dev-string.

  dev-string is a device-specifier. Returns false if the method could not be executed (i.e., the device-specifier is invalid, or that device has no method with the given name, or execution of that method resulted in an abort or throw). Otherwise, returns true above whatever results were placed on the stack by the execution of the method.

  The process is as defined in 4.3, using the rules given for execute-device-method.

  See also: apply

“existing” S
  MMU package property name to define existing virtual address resources.

  prop-encoded-array:
    Arbitrary number of virtual-address, len pairs. Virtual-address may be one or more integers, each encoded as with encode-int. Len may be one or more integers encoded as with encode-int.

  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 virtual-address and len are MMU-specific.

  See also: “available”, map, modify, “reg”, translate, unmap

exit ( -- ) (R: sys -- ) A,F 0x33
  Exit from the currently executing command.
exit? ( -- done? )

Return true when output should be terminated.

Handle output pagination and user control thereof. Return true if the user has requested the cessation of output from the current command.

exit? is used inside loops that might send many lines of output to the console. It is typically called once for each line of output.

The precise behavior is implementation-dependent; a suggested behavior follows:

If the value contained in the #line variable is greater than a predetermined value (typically returned by a word named lines/page) prompt the user with the message:

More [ space, cr, q ] ?

and wait for a character to be typed on the console. If that character is “q”, return true. If that character is “cr” (carriage return), arrange for the next call to exit? to prompt the user, and return false. If the character is neither “q” nor “cr”, set the contents of #line to zero and return false.

If a “q” character has been typed on the console input device since the last time that exit? was called return true.

If any other character has been typed, prompt for what to do next, as shown above, and return false.

The typical behavior described above has the following features (assuming that output-generating commands call exit? once per line of output):

— Output pauses at the end of each page of output, allowing the user to either stop further output (“q”), get one more line output before pausing again (“cr”), or continue with the next page of output (“space”).

— The user can stop further output at any time by typing “q”.

— The user can cause a pause before the end of a page by typing a character other than “q”.

expect ( addr len -- ) A,F 0x8A

Get and display input keyboard line, storing it at addr.

external ( -- ) T

Newly created functions will be visible.

Arrange for subsequently created definitions to have permanent names that persist after the active package is finished, so that those definitions will be visible package methods. In implementations that lack the ability to make temporary names, this may be a no-operation. The mode established by external persists until changed by headers or headerless.

Tokenizer: Arrange for subsequently created FCode functions to use external-token, so that those functions will be visible package methods.

external-token ( -- ) F 0xCA

Create a new named FCode function.

FCode evaluation:

Read an FCode-string, then an FCode#, from the current FCode program. Create a new FCode function, associating with it the FCode number FCode#. The new function’s execution semantics are initially undefined; they will be determined later by the execution of either b(:), b(create), b(defer), b(constant), b(buffer:), b(field), b(variable), or b(value).

Associate the new function with the name given by FCode-string, thus making it a method of the current node. The new method can later be executed with, for example, $call-method.

FCODE ONLY (Tokenized by defining words in external mode)

false ( -- false ) A,T

Return the value false (zero).

Tokenizer equivalent: 0
**fb8-blink-screen**  
Implement the “fb8” blink-screen function.

*Typically implemented as:* fb8-invert-screen fb8-invert-screen

**NOTE**—Typical generic implementations of this function are likely to be quite slow, since they probably will access each pixel on the screen four times. For most devices, there is a device-specific implementation for the blink-screen function that is much faster, for example, disabling video output for about 20 ms. It is recommended that such device-specific implementations be used instead of the generic fb8-blink-screen function.

**fb8-delete-characters**  
Implement the “fb8” delete-characters function.

**fb8-delete-lines**  
Implement the “fb8” delete-lines function.

**fb8-draw-character**  
Implement the “fb8” draw-character function.

**fb8-draw-logo**  
Implement the “fb8” draw-logo function.

The logo is painted with a pixel value of 0x01.

**fb8-erase-screen**  
Implement the “fb8” erase-screen function.

**fb8-insert-characters**  
Implement the “fb8” insert-characters function.

**fb8-insert-lines**  
Implement the “fb8” insert-lines function.

**fb8-install**  
Install all built-in generic 8-bit frame-buffer routines.

Install the “fb8” generic 8-bit frame-buffer routines into the display device interface `defer` words, configuring the “fb8” routines for a frame-buffer `height` pixels high, with successive scan lines `width` pixels apart.

`#columns` and `#lines` indicate the maximum number of text columns and lines that the device is capable of supporting (``#columns` and `#lines` usually depend upon the width and height of the font to be used, among other things).

`width` is the difference between the starting memory addresses of two consecutive scan lines in the frame-buffer. For frame-buffers where all memory locations correspond to displayable pixels, this is the same as the width of the screen in pixels.

`height` is the height of the display in scan lines.

Set `screen-width` to the `width` argument, `screen-height` to the `height` argument, `#columns` to the minimum of `#columns` and `screen-#columns`, and `#lines` to the minimum of `#lines` and `screen-#rows`.

Set `window-top` and `window-left` to center the text region on the screen (the calculation typically involves `#columns`, `#lines`, `char-width`, `char-height`, `screen-width`, and `screen-height`). The calculation assumes that `width` pixels per scan line are displayable. If some are not (for example, some number of pixels at the right of the display), it is the responsibility of the display driver to adjust `window-left` to locate the text region in an appropriate place after `fb8-install` returns.

**Usage restriction:** `char-width` and `char-height` must be set before `fb8-install` is executed; otherwise, the centering is likely to be incorrect.

**See also:** `set-font`

**fb8-invert-screen**  
Implement the “fb8” invert-screen function.
fb8-reset-screen  ( -- )  F  0x181
Implement the “fb8” reset-screen function.
This routine is usually implemented as a no-op.

fb8-toggle-cursor  ( -- )  F  0x182
Implement the “fb8” toggle-cursor function.

fcode-debug?  ( -- names? )  N
If true, save names for FCodes with headers.
This flag is used during the evaluation of an FCode program. If this flag is true, preserve the names of local FCodes created with named-token in the Forth dictionary. If this flag is false, discard those names fields.
Configuration variable type: Boolean. Suggested default value: false.

fcode-revision  ( -- n )  F  0x87
Return revision level of device interface.
The human-readable representation of the revision level is a string of the form “major.minor”, where “major” and “minor” are decimal numbers. The fcode-revision returns a single number representation whose value is given by the formula (major*65536 + minor). For example, if the release number were 2.12, the return value would be 0x0002.000C.
The revision level of the device interface described by this standard is “3.0”, therefore fcode-revision shall return 0x0003.0000.

ferror  ( -- )  F  0xFC
Standard FCode number for undefined FCode functions.
Set the internal state variable fcode-end to true, which will cause the FCode evaluator to cease interpreting the current FCode program after the FCode function whose evaluation resulted in the execution of this function finishes. Additional semantics are implementation-dependent (typically, such additional semantics include displaying an error messages indicating that an undefined FCode number has been executed).
In addition to this function’s assigned FCode number, undefined FCode numbers (those that are not explicitly assigned to other functions) are associated with ferror.
NOTE—ferror’s assigned FCode number provides a way to detect at run-time whether or not a particular FCode number is defined. This can be accomplished by comparing the execution token of the function in question with ferror’s execution token, e.g.,
77 get-token drop   ( xt )
['] ferror =        ( undefined? )
See: b(').

field  (E: addr -- addr+offset )  T
( offset size "new-name< >" -- offset+size )
Create new field offset specifier, named new-name.
Skip leading delimiters. Parse new-name delimited by a space. Create a definition for new-name with the execution semantics defined below.
Execution: (of new-name)  (E: addr -- addr+offset )
Return the sum of addr and offset.
field is used to create a named offset into a data structure. When a definition created by field is executed, its addr argument is typically the base address of the data structure.
In FCode source, field cannot be called from within a colon definition.
Example: The sequence:
( 24 ) 10 field >name  ( 34 )
creates a command called >name with the same behavior as the following:
: >name  24 + ;
Used as:
Assume a data structure 18 bytes long, organized as:
- size - bytes 0-1
- flag - bytes 2
- name - bytes 3-17
- first - bytes 3-9
- last - bytes 10-17
- ok struct ( 0 )
- ok 2 field >size ( 2 )
- ok 1 field >flag ( 3 )
- ok 0 field >name ( 3 )
- ok 7 field >first ( 10 )
- ok 8 field >last ( 18 )
- ok constant record-size

Later used as:
- ok record-size buffer: my-record
- ok my-record >name ( name-addr )
- ok my-record >flag ( flag-addr )

Tokenizer equivalent:
- new-token|named-token|external-token b(field)

fill ( addr len byte -- ) A,F 0x79
Set len bytes beginning at addr to the value byte.

find ( pstr -- xt n | pstr 0 ) A
Find command, return -1 (found), +1 (immediate), or 0 (not found).

$find ( name-str name-len -- xt true | name-str name-len false ) F 0xCB
Find the command named name string in the dictionary.
If found, return true and xt. Otherwise, return false and leave name string on the stack.

find-device ( dev-str dev-len -- )
Make the device node dev-string the active package.
If dev-string is the string "..", set the active package to the parent of the currently active package. Otherwise, set the active package as described by 4.3 in find-device context, using dev-string as the device-specifier.
If the specified device is not found, execute abort.
NOTE—find-device is similar to dev, except that its argument is a string on the stack instead of text parsed from the input buffer, allowing find-device to be used within a definition, with a literal string argument that is compiled into the definition.

find-package ( name-str name-len -- phandle true ) F 0x204
Locate the support package named name string.
If the package can be located, return its phandle and true; otherwise, return false.
Interpret the name in name string relative to the "packages" device node. If there are multiple packages with the same name (within the "packages" node), return the phandle for the most recently created one.
finish-device  

( -- ) F 0x127

Finish this package, set active package to parent.

Complete a device node that was created by new-device, as follows: If the device node has no “name” property, remove the device node from the device tree. Otherwise, save the current values of the current instance’s initialized data items within the active package for later use in initializing the data items of instances created from that node. In any case, destroy the current instance, make its parent instance the current instance, and select the parent node of the device node just completed, making the parent node the active package again.

fm/mod  

( d n -- rem quot ) A

Divide d by n.

>font  

( char -- addr ) F 0x16E

Return beginning address for char in the current font.

set-font must have been previously called for this command to be valid.

For any characters not in the font table, return a font entry for a valid, unspecified standard-width character.

fontbytes  

( -- bytes ) F 0x16F

value, return interval between entries in the font table.

fontbytes is a value that is used by the “fb1” and “fb8” frame-buffer support packages. It denotes the distance in bytes between the successive scan lines of a glyph in the current font.

Any standard package that uses one of the frame-buffer support packages shall set this value prior to executing either fb1-install or fb8-install. That is typically done by executing set-font.

forget  

( "old-name< >" -- ) A

Remove command old-name and all subsequent definitions.

Leave the dictionary pointer at the value which it had just before the command old-name was defined. If there are multiple commands in the dictionary with the name old-name, remove the most recent definition.

forth  

( -- ) A

Make Forth the context vocabulary.

frame-buffer-adr  

( -- addr ) F 0x162

value, return current frame-buffer virtual address.

Must be set using to, when the frame-buffer package is opened.

free-mem  

( a-addr len -- ) F 0x8C

Free memory allocated by alloc-mem.

The values a-addr and len must be the same as used in a previous alloc-mem command.

free-virtual  

( virt size -- ) F 0x105

Destroy mapping and “address” property.

If the package associated with the current instance has an “address” property whose first value encodes the same address as virt, delete that property. In any case, then execute the parent instance’s map-out method with virt size as its arguments.

.fregisters  

( -- )

Display floating-point registers (if present).

A standard system may either access the registers “in-place” or access copies of their values saved as part of the saved program state. The exact set of registers displayed, and the format, is ISA-dependent.
**get-inherited-property** ( name-str name-len -- true | prop-addr prop-len false ) F 0x21D

Return value for given property in the current instance or its parents.

Locate, within the package associated with the current instance or any of its parents, the property whose property name matches the value name string. If the property exists, return the associated value as the prop-encoded-array prop-addr prop-len and false. Otherwise, return true.

---

**get-msecs** ( -- n ) F 0x125

Return elapsed time, in milliseconds.

Return a free-running clock value, unreferenced to any specific time value. It is not assumed to maintain correctness during power-down of the system.

Return the value to the best available accuracy and precision. There is no minimum specification for either accuracy or precision.

---

**get-my-property** ( name-str name-len -- true | prop-addr prop-len false ) F 0x21A

Return value for given property in this package.

Locate, within the package associated with the current instance, the property whose property name matches the value name string. If the property exists, return the associated value as the prop-encoded-array prop-addr prop-len and false. Otherwise, return true.

---

**get-package-property** ( name-str name-len phandle -- true | prop-addr prop-len false ) F 0x21F

Return value for name string property in package phandle.

Locate, within the package phandle, the property whose property name matches the value name string. If the property exists, return the associated value as the prop-encoded-array prop-addr prop-len and false. Otherwise, return true.

---

**get-token** ( fcode# -- xt immediate? ) F 0xDA

Convert FCode number to function execution token.

Return the execution token xt of the word associated with FCode number fcode#, and a flag immediate? that is true if and only if that word will be executed (rather than compiled) when the FCode evaluator encounters its FCode number while in compilation state.

---

**go** ( -- )

Execute or resume execution of a program in memory.

Restore the processor state from the saved-program-state memory area and begin/resume execution of the machine-code program.

Resume execution at the address saved in the saved-program-state program counter register. This will normally contain the initial value for a newly loaded program or the resumption address for a suspended program. However, the saved program counter register can be altered by the user, causing the program to resume (when go is executed) from an arbitrary address.

This command has no effect unless state-valid contains true.

**go** can be used in conjunction with other commands in one of several ways:

After load (which also initializes the saved-program-state), go executes the program just downloaded.

After a program is suspended by entering the implementation-dependent “abort-sequence” (which saves the processor state in saved-program-state), go resumes execution of the suspended program.

When testing a program with breakpoints, and after a breakpoint has been encountered (which saves the processor state in saved-program-state), go resumes execution of the program being tested.

---

**gos** ( n -- )

Execute go n times.
**h#**

( [number< >] -- n )

Interpret the following number as a hexadecimal number (base sixteen).

**Interpretation:**

( [number< >] -- n )

Skip leading space delimiters. Parse number delimited by a space. Convert the string number to an integer n using a conversion radix of sixteen. Put n on the stack. An ambiguous condition exists if the conversion fails.

**Compilation:**

( [number< >] -- )

Skip leading space delimiters. Parse number delimited by a space. Convert the string number to an integer n using a conversion radix of sixteen. Append the run-time semantics given below to the current definition. An ambiguous condition exists if the conversion fails.

**Run-time:**

( -- n )

Place n on the stack.

The number is interpreted in hexadecimal regardless of the current value in base. The value of base is unchanged.

**Used as:** h# 1001 ( decimal 4097 )

**Tokenizer equivalent:** b(lit) xx-byte xx-byte xx-byte xx-byte

.h

( n -- )

Display a signed number (and space) in hex.

Ignore the value in base and leave it unchanged. Also display a single trailing space.

**Tokenizer equivalent:** base ® swap 16 base ! . base !

**headerless**

( -- )

Newly created functions will be invisible.

Arrange for subsequently created definitions to have temporary names that disappear when the active package is finished, so those definitions will be invisible thereafter. In implementations that lack the ability to make temporary names, this may be a no-operation.

The mode established by headerless persists until changed by headers or external.

**Tokenizer:** Arrange for subsequently created FCode functions to use new-token, so those functions will be invisible (internal to their package).

**headers**

( -- )

Newly created functions will be optionally visible.

Arrange for subsequently created definitions to have, at the user’s discretion, either permanent names as with external or temporary names as with headerless. In implementations that lack the ability to make temporary names, this may be a no-operation. Otherwise, the means for controlling whether names are permanent or temporary shall be the same as that used by named-token.

The mode established by headers persists until changed by external or headerless.

**Tokenizer:** Arrange for subsequently created FCode functions to use named-token, so those functions will be either visible or invisible at the user’s discretion (see fcode-debug?). This is the default behavior.

**help**

( "[name]<eol>" -- )

Provide information for category or specific command.

If name is a specific command, list help for that command, if available. Otherwise, display an implementation-dependent message.

**Used as:** ok help command-name

If name is a category, list all help messages for commands in that category, or a list of subcategories.

**Used as:** ok help category-name

If name is omitted, provide general help and a list of available categories.

Commands should be grouped into categories so that the help messages for a category occupy no more than twenty-three output lines. Categories may be divided into subcategories. The number and names of categories are implementation-dependent.

**here**

( -- addr )

Return current dictionary pointer.
**hex** (tokenizer)  
( -- )  
A,T  
Set numeric conversion radix to sixteen.

If **hex** is encountered in FCode source outside a definition, set the tokenizer’s numeric conversion radix to sixteen. If **hex** is encountered in FCode source inside a definition, append the following sequence to the FCode program that is being created:

**Tokenizer equivalent:** `16 base !`

**ANS Forth/tokenizer difference:** ANS Forth has no separate “tokenizing” behavior.

**hold**  
( char -- )  
A,F 0x95  
Add char in pictured numeric output conversion.

**hop**  
( -- )  
Execute single instruction, or entire subroutine call.

Same as **step**, except that if the instruction to be executed is a subroutine call, execute the entire subroutine before stopping, instead of just the call instruction.

If the execution of that subroutine results in encountering another breakpoint, the result is implementation-dependent.

**hops**  
( n -- )  
Execute **hop** n times.

**i**  
( -- index )  
(R: sys -- sys )  
A,F 0x19  
Return current loop index value.

**ANS Forth note:** Also works outside of a definition.

**if**  
(C: -- orig-sys )  
A,T  
( do-next? -- )  
If flag is true, execute following code.

**Interpretation:**  
(C: -- orig-sys )  
Enter compilation state, initiating a temporary current definition in a region of memory other than the data space. Then perform the compilation semantics of ANS Forth **IF**.

**Compilation:**  
(C: -- orig-sys )  
Same as ANS Forth.

**Run-time:**  
( do-next? -- )  
Same as ANS Forth.

**Tokenizer equivalent:** `b?branch +offset`

**ANS Forth note:** Also works outside of a definition.

**ihandle>phandle**  
( ihandle -- phandle )  
F 0x20B  
Return the **phandle** for the indicated **ihandle**.

**immediate**  
( -- )  
A  
Declare the previous definition as “immediate”.

**>in**  
( -- a-addr )  
A  
**variable** containing offset of next input buffer character.

**init-program**  
( -- )  
Initialize **saved-program-state**.

Set **saved-program-state** to the ISA-dependent initial program state required for beginning the execution of a client program.
input ( dev-str dev-len -- )
Select the indicated device for console input.

Search for a device node matching the pathname or device-specifier given by dev-str dev-len. The search process is as defined in 4.3, using the rules given for find-device, but restore the active package to its previous package afterwards.

If such a device is found, search for its read method.
If the read method is found, open the device, as with open-dev.
If the open succeeds, execute the device’s install-abort method, if any.
If any of these steps fails, display an appropriate error message and return without performing the steps following the one that failed.

If there is a console input device, as indicated by a nonzero value in the stdin variable, execute the console input device’s remove-abort method and close the console input device. Set stdin to the ihandle of the newly opened device, making it the new console input device.
Used as: " device-alias" input

input-device ( -- dev-str dev-len ) N
Default console input device.

Indicates the device to be established as the console input device by install-console. dev-string is a device-specifier.
Used as: ok setenv input-device device-alias <eol>
Configuration variable type: string[32]. Suggested default value: keyboard.

insert-characters ( n -- ) F 0x15D
defer, insert n spaces to the right of the cursor.

insert-characters is one of the defer words of the display device interface. The terminal emulator package executes insert-characters when it has processed a character sequence that calls for opening space for characters to the right of the cursor.

Any standard package that uses the terminal emulator package shall, as part of the process of opening the terminal emulator package, set this defer word to a function with the following behavior:

Move the remainder of the line to the right, thus losing the n rightmost characters in the line, without moving the cursor. Fill the vacated character positions with the background color.
See also: to, fb8-install

insert-lines ( n -- ) F 0x15F
defer, insert n blank lines at and below the cursor line.

insert-lines is one of the defer words of the display device interface. The terminal emulator package executes insert-lines when it has processed a character sequence that calls for opening space for lines of text below the cursor.

Any standard package that uses the terminal emulator package shall, as part of the process of opening the terminal emulator package, set this defer word to a function with the following behavior:

Move the cursor line and all following lines down, thus losing the n bottom lines, without moving the cursor. Fill the n vacated lines with the background color.
See also: to, fb8-install

install-abort ( -- ) M
Begin polling for a keyboard abort sequence.

Instruct the device driver to begin periodic polling for a keyboard abort sequence. If a keyboard abort sequence is subsequently encountered, abort is executed.
This command is executed when the device is selected as the console input device.
install-console

Select and activate console input and output devices.

Activate the console function and select input and output devices as follows:

a) Activate the console so that subsequent input (e.g., key) and output (e.g., emit) will use the devices selected by input and output.
b) Attempt to create a screen alias as described in 7.4.5.
c) Execute output with the value returned by output-device.
d) Execute input with the value returned by input-device.
e) If the previous code failed and there is a fallback device to be used for console functions, select that device as the console device.

install-console may take other system-dependent actions to ensure that a console device is available in the event of a failure, and may display messages indicating that such action has been taken.

instance

Mark next occurring defining word as instance-specific.

Modify the next occurrence of value, variable, defer, or buffer: to create instance-specific data instead of static data. Re-allocate the data each time a new instance of this package is created.

Used as: ok 30 instance value new-name

.instruction

Display next pending address and instruction.

Display the address where the last breakpoint occurred and the instruction that would have executed next if the breakpoint had not been there.

The instruction-display format is ISA-specific.

“interrupts” S

Standard property name to define the interrupts used.

prop-encoded-array:

Arbitrary number of interrupt specifiers (bus-specific), each typically encoded with encode-int.

Specifies the interrupt level(s) used by this device and possibly other appropriate information (such as interrupt vectors).

The level given is the bus-specific (local) level, not the CPU level. The actual format of the data is bus-specific; see the appropriate Open Firmware machine-specific document for details.

See also: intr

intr ( sbus-interrupt# vector -- ) F,O 0x117

Creates the “intr” property.

See the description of the “intr” property for more details.

“intr” S

Standard property name, defines SBus interrupt level(s).

prop-encoded-array:

n (CPU_level,intr_vector) pairs, each value encoded with encode-int.

CPU_level created by: SBus_level sbus-intr>cpu encode-int

Vector created by: intr_vector encode-int

If, at the time the “intr” property is created, the active package does not have an “interrupts” property, create an “interrupts” property in addition to the “intr” property, with the following property value:

prop-encoded-array:

n SBus_levels, each encoded with encode-int, corresponding to the n CPU_levels of the “intr” property value.

This property, with its semantics of creating an “interrupts” property, is included as a concession to existing FCode programs. It should be used only by those FCode programs that require compatibility with older SBus systems. It should not be used by FCode programs for non-SBus devices. The specification of this property is included here, rather than in an SBus-specific supplement, because of the possibility that, even on systems that nominally do not support SBus, SBus devices might be used via a bus-to-bus bridge.
One of two forms is used:

First form: $5 \ 0 \ \text{intr}$

This is the most common usage. The \text{intr} FCode translates the \text{sbus-interrupt#} (5) to the appropriate \text{CPU_level} (with \text{sbus-intr>cpu}), and then creates the \text{“intr”} property with the CPU interrupt value. The \text{intr_vector} value is always 0.

Second form:

$5 \ \text{sbus-intr>cpu} \ \text{encode-int} \ 0 \ \text{encode-int} \ \text{encode+}$
$7 \ \text{sbus-intr>cpu} \ \text{encode-int} \ \text{encode+} \ 0 \ \text{encode-int} \ \text{encode+}$

This form is generally only used when multiple-interrupt levels must be declared. (Multiple-levels cannot be declared with the first form.)

\text{inverse?} \quad (\text{-- white-on-black?}) \quad F \quad 0x154

\text{value}, indicates how to paint characters.

This \text{value} is part of the display device interface.

The \text{terminal emulator} package shall set \text{inverse?} to \text{true} when the escape sequences that it has processed have indicated that subsequent characters are to be shown with foreground and background colors exchanged, and to \text{false}, indicating normal foreground and background colors, otherwise.

The \text{“fb1”} and \text{“fb8”} frame-buffer support packages shall draw characters with foreground and background colors exchanged if \text{inverse?} is \text{true}, and with normal foreground and background colors is \text{false}. If \text{inverse?} is neither \text{true} nor \text{false}, the result is undefined.

Standard packages that use the \text{terminal emulator} package should draw characters with foreground and background colors exchanged if \text{inverse?} is \text{true}, and with normal foreground and background colors if \text{false}.

\text{inverse?} affects the character display operations \text{draw-character}, \text{insert-characters}, and \text{delete-characters}, but not the other operations such as \text{insert-lines}, \text{delete-lines}, and \text{erase-screen}.

\text{inverse-screen?} \quad (\text{-- black?}) \quad F \quad 0x155

\text{value}, indicates how to paint the background.

This \text{value} is part of the display device interface.

The \text{terminal emulator} package shall set \text{inverse-screen?} to \text{true} when the escape sequences that it has processed have indicated that the foreground and background colors are to be exchanged for operations that affect the background, and to \text{false}, indicating normal foreground and background colors, otherwise.

The \text{“fb1”} and \text{“fb8”} frame-buffer support packages shall perform screen drawing operations other than character drawing operations with foreground and background colors exchanged if \text{inverse-screen?} is \text{true}, and with normal foreground and background colors is \text{false}. If \text{inverse-screen?} is neither \text{true} nor \text{false}, the result is undefined.

Standard packages that use the \text{terminal emulator} package should perform screen drawing operations other than character drawing operations with foreground and background colors exchanged if \text{inverse-screen?} is \text{true}, and with normal foreground and background colors is \text{false}. \text{inverse-screen?} affects background operations such as \text{insert-lines}, \text{delete-lines} and \text{erase-screen}, but not character display operations such as \text{draw-character}, \text{insert-characters} and \text{delete-characters}.

\text{NOTE—When \text{inverse-screen?} and \text{inverse?} are both \text{true}, the colors are exchanged over the entire screen, and subsequent characters are not highlighted with respect to the (inverse) background. For exchanged screen colors and highlighted characters, the settings are \text{inverse-screen? true} and \text{inverse? false}.}

\text{invert} \quad (x1 \ -- \ x2) \quad A,F \quad 0x26

Invert all bits of \text{x1}.
invert-screen

(defer, exchange the foreground and background colors.

invert-screen is one of the defer words of the display device interface. The terminal emulator package executes invert-screen when it has processed a character sequence that calls for exchanging the foreground and background colors (e.g., changing from black-on-white to white-on-black).

Any standard package that uses the terminal emulator package shall, as part of the process of opening the terminal emulator package, set this defer word to a function with the following behavior:

- Change all pixels on the screen so that pixels of the foreground color are given the background color, and vice versa, leaving the colors that will be used by subsequent text output unaffected.

See also: to, fb8-install

io

((dev-str dev-len -->)

Select the indicated device for console input and output.

Execute input followed by output with dev-str dev-len as arguments in both cases.

Used as: " device-alias" io

is-install

((xt -->) F 0x11C

Create open, other methods for this display device.

Create methods for accessing the display device driver in the active package.

Used as: ['] my-open-routine is-install

Create the following methods:

open
- When later called, execute the display driver’s “my-open-routine” (whose execution token is xt) and initialize the terminal emulator.

write
- When later called, pass its argument string to the terminal emulator for interpretation.

draw-logo
- When later called, execute the display driver’s “my-draw-logo” procedure which was installed into the defer word draw-logo by the driver’s “my-open-routine”.

restore
- When later called, execute the display driver’s “my-reset-screen” procedure which was installed into the defer word reset-screen by the driver’s “my-open-routine”.

is-remove

((xt -->) F 0x11D

Create close method for this display device.

Used as: ['] my-close-routine is-remove

The created close method, when later called, executes the display driver’s “my-close-routine” procedure (whose execution token is xt).

is-selftest

((xt -->) F 0x11E

Create selftest method for this display device.

Used as: ['] my-selftest-routine is-selftest

The created selftest method, when later called, executes the display driver’s “my-self-test-routine” procedure (whose execution token is xt).

(is-user-word)

((E: ..., -- ???) F 0x214

(name-str name-len xt -->)

Create new command named by string, behavior is xt.

Create a Forth command whose name is given by name string. The behavior of the new command is given by the execution token xt, which must refer to a static method.

Used as: " new-name" ['] old-name (is-user-word)
Return next outer loop index value.

ANS Forth note: Also works outside of a definition.

key

Read a character from the console input device.

If no key has been typed since the last key or expect commands, key will wait until a new character is typed. Use key? to determine if a character is available.

key?

Return true if an input character is available from the console input device.

key? is non-destructive; the keyboard character is not consumed.

l!

Store quadlet to qaddr.

See: r1!

1,

Compile a quadlet into the dictionary (doublet-aligned).

Allocate 4 bytes (with allot) at the current top of the dictionary and store the value quad into that space. The dictionary pointer must have been doublet-aligned.

l@

Fetch quadlet from qaddr.

See: r1@

/1

The number of address units to a quadlet; typically, four.

/1*

Multiply nu1 by the value of /1.

la+

Increment addr1 by index times the value of /1.

lal+

Increment addr1 by the value of /1.
**label**

(E: -- addr )

("new-name< >" -- code-sys )

Begin machine-code sequence, leave addr on stack.

Begin creation of an machine-code sequence called new-name. Interpret the following commands as assembler mnemonics. Commands created by label leave the address of the code on the stack when executed.

As with code, label is present even if the assembler is not installed. In this case, machine-code must be entered into the dictionary explicitly by value, i.e., with c, w, l, or . The machine-code sequence is terminated by the c; or end-code commands.

**Used as:**

- ok label new-name
- ok (assembler mnemonics)
- ok end-code

**Later used as:**

- new-name ( machine-code-addr )
- code-sys is balanced by the corresponding c; or end-code.

**lbflip**

( quad1 -- quad2 ) F 0x227

Reverse the bytes within a quadlet.

**lbflips**

( qaddr len -- ) F 0x228

Reverse the bytes within each quadlet in the given region.

The region begins at qaddr and spans len bytes. The behavior is undefined if len is not a multiple of /l.

**lbsplit**

( quad -- b.lo b2 b3 b4.hi ) F 0x7E

Split a quadlet into 4 bytes.

The high bits of the 4 bytes are zero.

**lcc**

( char1 -- char2 ) F 0x82

Convert ASCII char1 to lowercase.

Convert input values between 0x41 and 0x5A (ASCII A-Z) to 0x61 through 0x7A (ASCII a-z). All other input values are unchanged.

**leave**

( -- ) (R: sys -- ) A,T

Exit this do or ?do loop immediately.

**Tokenizer equivalent:** b{leave}

**ANS Forth note:** Also works outside of a definition.

**?leave**

( exit? -- ) (R: sys -- ) T

If flag is nonzero, exit this do or ?do loop immediately.

If exit? is nonzero, discard the current loop control parameters. An ambiguous condition exists if they are unavailable. Continue execution immediately following the innermost syntactically enclosing do ... loop or do ... ?loop.

**Tokenizer equivalent:** if leave then

**See:** ANS Forth return stack restrictions

**left-parse-string**

( str len char -- R-str R-len L-str L-len ) F 0x240

Split the string at first occurrence of delimiter char.

R-string is the string after, and L-string is the string before, the first occurrence of the delimiter. Neither string includes that first occurrence of the delimiter, although R-string might contain other later occurrences of that character.

If the delimiter does not appear in the argument string, R-len is zero and L-string is the same as the argument string.
line# ( -- line# )
value, return the current cursor line number.
Return the current vertical position of the text cursor.
NOTE—A value of zero represents the topmost line of the text window, not the topmost pixel of the frame-buffer.
See: window-top for more details.

#line ( -- a-addr )
variable containing the number of output lines.
a-addr is the address of a cell containing the number of output lines since the last user interaction.
#line is set to zero when the command interpreter prompts for a new command line (whenever the ok prompt is displayed). Its value is increased by one when cr is executed.
See: exit? for other conditions in which #line might be set to zero.

linefeed ( -- 0x0A )
ASCII code for “linefeed” character.
Tokenizer equivalent: b(lit) 00 00 00 0x0A

#lines ( -- rows )
value, return number of lines of text in text window.
#lines is a value that is part of the display device interface. The terminal emulator package uses it to determine the height (number of rows of characters) of the text region that it manages. The “fb1” and “fb8” frame-buffer support packages also use it for a similar purpose.
Any standard package that uses the terminal emulator package shall, as part of the process of opening the terminal emulator package, set this value to the desired height of the text region. That width shall not exceed the value of screen-#rows.
See also: to fb8-install.

literal (C: x1 -- ) A,C
( -- x1 )
Compile a number; later, leave it on the stack.

load (user interface) ( "{params}<eol>" -- )
Load a program, specified by params.
Skip leading space delimiters. Parse first-arg delimited by a space. If first-arg is the empty string, set device-specifier to the default device and arguments to the default arguments as specified below, and proceed with the loading process as specified below. Otherwise, handle first-arg as follows:
If first-arg begins with the “/” character, or if it the name of a defined devalias, set device-specifier to first-arg, then skip leading space delimiters, set arguments to the remainder of the command line.
Otherwise, set device-specifier to the default device and arguments to the portion of the command line beginning at first-arg and continuing to the end of the line (including first-arg itself).
If arguments is the empty string, replace it with the default arguments.
Proceed as follows.
Loading Process:
If the client interface is implemented, save arguments and the device-path corresponding to device-specifier so they may be retrieved later via the client interface.
Open, as with open-dev, the package specified by device-specifier, thus obtaining an ihandle. If unsuccessful, execute the equivalent of abort, thus stopping the loading process. Otherwise, execute, as with $call-method, the load method of that ihandle, passing the system-dependent default load address to “load” as its argument. Then close, as with close-dev, that ihandle.
If the “load” method succeeds, and the beginning of the loaded image is a valid client program header for the system, allocate memory at the address and of the size specified in that header, move the loaded image to the address, and perform the function of init-program to initialize saved-program-state so that a subsequent go command will begin execution of that program.
Default device and default arguments:
The default arguments are given by the value of \texttt{boot-file} if \texttt{diagnostic-mode?} is \texttt{false}, otherwise by the
value of \texttt{diag-file}.

The default device is determined by the value of \texttt{boot-device} if \texttt{diagnostic-mode?} is \texttt{false}, otherwise by the
value of \texttt{diag-device}. Either \texttt{boot-device} or \texttt{diag-device} may contain a list of \texttt{device-specifiers} separated
by spaces. If that list contains only one entry, that entry is the default device. If that list contains more than one entry, the
system attempts to open, as with \texttt{open-dev}, each specified device in turn, beginning with the first entry in the list and
proceeding to the next-to-last entry. If an open succeeds, the device is closed, as with \texttt{close-dev}, and that \texttt{device-}
specifier becomes the default device (it will be subsequently opened again by the loading process). If the last entry is
reached without any prior successful opens, the last entry becomes the default device, without having been opened as part
of the default device selection process.

\textbf{See also:} \texttt{boot}.

\textbf{Used as:} \texttt{ok load device-specifier arguments <eol>}

\texttt{load} (package method) \hspace{1cm}  \texttt{( addr -- len )} \hspace{1cm} M

Load a client program from device to memory.

Load a client program from the device into memory beginning at address \texttt{addr}, returning \texttt{len}, the size in bytes of the
program that was loaded.

If the device can contain several such programs, the \texttt{instance-arguments} (as returned by \texttt{my-args}) can be used in a
device-dependent manner to select the particular program.

\textbf{Usage restriction:} The package containing the \texttt{load} method must be open before the \texttt{load} method is executed.

\texttt{local-mac-address`x} \hspace{1cm} S

Standard \texttt{property name} to specify preassigned network address.

\texttt{prop-encoded-array}:

Array of six bytes encoded with \texttt{encode-bytes}.

Specifies the 48-bit IEEE 802.3-style Media Access Control (MAC) (as specified in ISO/IEC 8802-3 : 1993 [B3]) address
assigned to the device represented by the package, of \texttt{device type “network”}, containing this property. The absence of
this property indicates that the device does not have a permanently assigned MAC address.

\textbf{Used as:}

create my-mac-address 8 c, 0 c, 20 c, 0 c, 14 c, 5e c,
my-mac-address 6 encode-bytes " local-mac-address" property

\textbf{NOTE—}In many systems, the MAC address is not associated with the individual network devices, but instead with the
system itself. In such cases, the system-wide MAC address applies to all the network interfaces on that system, and
individual network device nodes might not have mac-address properties. In other cases, especially with plug-in network
interface cards that are intended for use on a variety of different systems, the manufacturer of the card assigns a MAC
address to the card, which is reported via the \texttt{“local-mac-address”} property. A system is not obligated to use that
assigned MAC address if it has a system-wide MAC address.

\textbf{See also:} \texttt{“network”}, \texttt{“mac-address”} and \texttt{mac-address}.

\texttt{loop} \hspace{1cm} \texttt{(C: dodest-sys -- )} \hspace{1cm} A,T

Add one to index, then return to the previous \texttt{do} or \texttt{?do} or exit the loop.

\textbf{Compilation:}

(C: dodest-sys -- )

Perform the compilation semantics of ANS Forth \texttt{LOOP}. Then, if the current definition is temporary and the depth of the
control flow stack is the same as its depth when the temporary current definition was initiated, perform the compilation
semantics of \texttt{;} and execute the temporary current definition.

\textbf{Run-time:}

(C: dodest-sys -- )

Same as ANS Forth.

\textbf{Tokenizer equivalent:} \texttt{b(loop) -offset}

\textbf{ANS Forth note:} Also works outside of a definition.
Add delta to index, then return to the previous do or exit the loop.

**Compilation:**
Perform the compilation semantics of ANS Forth `+loop`. Then, if the current definition is temporary and the depth of the control flow stack is the same as its depth when the temporary current definition was initiated, perform the compilation semantics of ; and execute the temporary current definition.

**Run-time:**
Same as ANS Forth.

**Tokenizer equivalent:** `b(+loop) ^offset`

**ANS Forth note:** Also works outside of a definition.

---

**lpeek**
( `qaddr -- false | quad true` )
F 0x222

Attempt to fetch the quadlet at `qaddr`.

Return the data and `true` if the access was successful. A `false` return indicates that a read access error occurred.

**lpoke**
( `quad qaddr -- okay?` )
F 0x225

Attempt to store the quadlet to `qaddr`.

Return `true` if the access was successful. A `false` return indicates that a write access error occurred.

**ls**
---

Display the names of the active package’s children.

**lshift**
( `x1 u -- x2` )
A,F 0x27

Shift `x1` left by `u` bit-places. Zero-fill low bits.

**lwflip**
( `quad1 -- quad2` )
F 0x226

Swap the doublets within a quadlet.

**lwflips**
( `qaddr len --` )
F 0x237

Swap the doublets within each quadlet in the given region.

The region begins at `qaddr` and spans `len` bytes. The behavior is undefined if `len` is not a multiple of `/1`.

**lwsplit**
( `quad -- w1.lo w2.hi` )
F 0x7C

Split a quadlet into two doublets.

The high bits of the two doublets are zero.

**m**
( `n1 n2 -- d.prod` )
A

Signed multiply with double-number product.

**mac-address**
( `-- mac-str mac-len` )
F 0x1A4

Return a sequence of bytes containing network address.

The address is the Media Access Control (MAC) address to be used by a network device. The MAC address is denoted by a sequence of `mac-len` binary bytes beginning at `mac-str`. For example, for the 48-bit IEEE 802.3-style MAC address whose human-readable representation is "8:20:15:12:3e:45", `mac-address` would return an array of 6 bytes containing 0x08, 0x20, 0x15, 0x12, 0x3E, 0x45.

The method by which the MAC address is determined is system-dependent. For example, in some systems, `mac-address` returns a system-wide address stored in a system-dependent location, and in other systems, the return value is derived from the "`local-mac-address" property (if any) of the package corresponding to the current instance. Other systems might select their `mac-address` from a configuration table.
"mac-address"

  Standard property name to specify network address last used.

  prop-encoded-array:
    Array of 6 bytes encoded with encode-bytes.

  Specifies the 48-bit IEEE 802.3-style Media Access Control (MAC) address that was last used by the device represented by the package, of device type "network", containing this property. This property is created by the open method of a network device.

  **NOTE**—This property is typically used by client programs that need to determine which network address was used by the network interface from which the client program was loaded.

map

  ( phys.lo ... phys.hi len ... mode -- ) M

Create address translation.

Create an address translation associating virtual addresses beginning at virt and continuing for len ... (whose format depends on the package) bytes with consecutive physical addresses beginning at phys.lo ... phys.hi. Mode is an MMU-dependent parameter (typically, but not necessarily, one cell) denoting additional attributes of the translation, for example access permissions, cacheability, mapping granularity, etc. If mode is –1, an implementation-dependent default mode is used. If there are already existing address translations within the region delimited by virt and len ..., the result is undefined.

If the operation fails for any reason, uses throw to signal the error.

See also: claim, modify, release, translate

map-in

  ( phys.lo ... phys.hi size -- virt ) M

Map the specified region; return a virtual address.

Create a mapping associating the range of physical addresses beginning at phys.lo ... phys.hi and extending for size bytes within this device’s physical address space with a processor virtual address. Return that virtual address virt.

The number of cells in the list phys.lo ... phys.hi is determined by the value of the #address-cells property of the node containing map-in.

If the requested operation cannot be performed, a throw shall be called with an appropriate error message, as with abort".

**NOTE**—Out-of-memory conditions may be detected and handled properly in the code with ['] map-in catch.

map-low

  ( phys.lo ... size -- virt ) F 0x130

Map the specified region; return a virtual address.

Create a mapping associating the range of physical addresses beginning at phys.lo ... my-space and extending for size bytes within this device’s physical address space with a processor virtual address. Return that virtual address virt.

Equivalent to: my-space swap " map-in" $call-parent

The number of cells in the list phys.lo ... is one less than the number determined by the value of the #address-cells property of the parent node..

If the requested operation cannot be performed, a throw shall be called with an appropriate error message, as with abort".

**NOTE**—Out-of-memory conditions may be detected and handled properly in the code with ['] map-low catch.

See also: map-out

map-out

  ( virt size -- ) M

Destroy mapping from previous map-in.

Destroy the mapping set up by map-in at virtual address virt, of length size bytes.

See also: free-virtual

mask

  ( -- a-addr ) F 0x124

variable to control bits tested with memory-test-suite.

The contents of this variable control which bits out of every cell will be tested with memory-test-suite. To test all bits, set mask to all ones. To test only the low-order byte of each cell, set just the lower bits of mask.

Used as: 000000ff  mask  !
max  ( n1 n2 -- n1|n2 )  A,F  0x2F
Return greater of n1 and n2.

"max-frame-size"  S
Standard property name to indicate maximum allowable packet size.

Prop-encoded-array:
Integer, encoded with encode-int.

This property, when declared in "network" devices, indicates the maximum packet length (in bytes) that the physical layer of the device can transmit at one time. This value can be used by client programs to allocate buffers of the appropriate length.

Used as: 4000 encode-int " max-frame-size" property

max-transfer  ( -- max-len )  M
Return size of largest possible transfer.
Return the size in bytes of the largest single transfer that this device can perform, rounded down to a multiple of block-size.

"memory"  S
Random access memory device type.

Standard string of the "device_type" property for memory devices.
A standard package with this "device_type" property value shall implement the following methods:

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

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 address format is phys.lo ... phys.hi, a physical address of the form defined by the parent bus. The allocated resource is a region of random-access memory.
The allocation length parameter size consists of one or more cells depending on the parent bus. (See "#size-cells").

A standard package with this "device_type" property value shall implement the following properties:

"reg" The property values are as defined for the standard "reg" format, with physical addresses of the form required by the parent bus. The regions of physical address space denote the physical memory that is installed in the system, without regard to whether or not that memory is currently in use.

"available" The property values are as defined for the standard "reg" format, with physical addresses of the form required by the parent bus. The regions of physical address space denote the physical memory that is currently unallocated by the Open Firmware and is available for use by client programs.

See also: "available", claim, "reg", release, "#size-cells"

memory-test-suite  ( addr len -- fail? )  F  0x122
Perform tests of memory, starting at addr for len bytes.
Return true if any of the tests fail and display a failure message on a system-dependent diagnostic output device.
The actual tests performed are machine-specific and often vary depending on whether diagnostic-mode? is true or false.
If diagnostic-mode? is true, send a message to the console output device giving the name of each test.
The value stored in mask controls whether only some or all data lines are tested.

min  ( n1 n2 -- n1|n2 )  A,F  0x2E
Return lesser of n1 and n2.
mod ( n1 n2 -- rem ) A,F 0x22
Divide n1 by n2; return remainder.

*/mod ( n1 n2 n3 -- rem quot ) A
Calculate n1 times n2; divided by n3.

/mod ( n1 n2 -- rem quot ) A,F 0x2A
Divide n1 by n2; return remainder and quotient.

model ( str len -- ) F 0x119
Create the "model" property; value is indicated string.
Shorthand command to create a property in the active package whose property name is "model".
Equivalent to: encode-string " model" property
Used as: " XYZCO,1416-02" model
See: "model" glossary entry for more information.

"model" S
Standard property name to define a manufacturer’s model number.

prop-encoded-array:
Text string, encoded with encode-string.
A manufacturer-dependent string that generally specifies the model name and number (including revision level) for this device. The format of the text string is arbitrary, although in conventional usage the string begins with the name of the device’s manufacturer as with the "name" property.
Although there is no standard interpretation for the value of the "model" property, a specific device driver might use it to learn, for instance, the revision level of its particular device.
See also: property, model.
Used as: " XYZCO,1416-02" encode-string " model" property

modify ( virt len ... mode -- ) M
Modify existing address translation.
Modifies the existing address translations for virtual addresses beginning at virt and continuing for len ... (whose format depends on the package) bytes to have the attributes specified by mode, as with map.
If the operation fails for any reason, uses throw to signal the error.
See also: claim, map, release, translate, unmap

move ( src-addr dest-addr len -- ) A,F 0x78
Copy len bytes from src-addr to dest-addr.

ms ( n -- ) A,F 0x126
Delay for at least n milliseconds.
Return low component(s) of device’s probe address.

### my-address

( -- phys.lo ... )

F 0x102

Return the instance-argument string for this instance.

### my-args

( -- arg-str arg-len )

F 0x202

Return the ihandle of the parent of the current instance.

### my-parent

( -- ihandle )

F 0x20A

Return the ihandle of the instance that opened the current instance.

### my-self

( -- ihandle )

F 0x203

Return high component of device’s probe address.

### my-space

( -- phys.hi )

F 0x103

Return the unit address of the current instance.

### my-unit

( -- phys.lo ... phys.hi )

F 0x20D

The number of address units in a cell.

### /n

( -- n )

F 0x5D
IEEE Std 1275-1994

IEEE STANDARD FOR BOOT (INITIALIZATION CONFIGURATION) FIRMWARE:

/n* ( nu1 -- nu2 ) T

Synonym for cells.

Tokenizer equivalent: cells

na+ ( addr1 index -- addr2 ) F 0x61

Increment addr1 by index times the value of /n.

na1+ ( addr1 -- addr2 ) T

Synonym for cell+.

Tokenizer equivalent: cell+

"name" S

Standard property name to define the name of the package.

prop-encoded-array:

Text string, encoded with encode-string.

Represents the name of this package. The string consists of a sequence of 1 to 31 letters, digits, and punctuation characters from the set ",_.+". The string shall contain, at most, one comma. Uppercase and lowercase letters are considered distinct.

For plug-in devices, the value string shall begin with a company name string in one of the following forms, followed by a comma ("."):"

"0NNNNNN", where NNNNNN is a sequence of 6 uppercase hexadecimal digits representing the company’s 24-bit Organizationally Unique Identifier (OUI) assigned by the IEEE Registration Authority Committee (RAC). To obtain an OUI, contact:

Registration Authority Committee
The Institute of Electrical and Electronic Engineers, Inc.
445 Hoes Lane
Piscataway, NJ 08855-1331
USA
(908) 562-3815

This is the recommended form of company name, as it is guaranteed to be unique worldwide.

"VWXYZ", where VWXYZ is a sequence of from one to five uppercase letters representing the stock symbol of the company on any stock exchange whose symbols do not conflict with the symbols of the New York Stock Exchange and the NASDAQ exchange. (In practice, all United States stock exchanges comply with this rule, but other stock exchanges worldwide do not necessary coordinate their symbols with NYSE and NASDAQ.)

This form of company name is allowed as a concession to existing practice.

"pdqxyz", where pdqxyz is any string that cannot be confused for one of the above forms (perhaps by containing characters that are not allowed by those forms such as lowercase letters, or by being longer than five letters).

This form of company name is permitted, but discouraged, because of the possibility that two different companies might choose the same name.

A standard package shall define this property.

Used as:

" XYZQ,devname" encode-string " name" property

See also: property, device-name

164
named-token

Create a new possibly named FCode function.

FCode evaluation:

Read an FCode-string, then an FCode#, from the current FCode program. Create a new FCode function, associating with it the FCode number FCode#. The new function's execution semantics are initially undefined; they will be determined later by the execution of the following FCode methods:

- `b(\_)`, `b(create)`, `b(defer)`, `b(constant)`, `b(buffer:)`, `b(field)`, `b(variable)`, or `b(value)`.

At the system's discretion (typically controlled by `fcode-debug?`), either leave the new function unnamed or associate it with the name given by FCode-string. If the function is unnamed, the only way to refer to it later is via its associated FCode number; it cannot be accessed by name from the user interface or via other mechanisms like `$call-method`. If the function is named, it becomes a method of the current node. That method can later be executed with, for example, `$call-method`.

FCODE ONLY (Tokenized by defining words in headers mode)

negate

Return negation of n1.

Equivalent to: 0 swap -

“network”

Packet-oriented network device type.

Standard string value of the “device_type” property for network devices with IEEE 802 packet formats.

A standard package with this “device_type” property value shall implement the following methods.

- `open`, `close`, `read`, `write`, `load`

Additional requirements for the `open` method:

- Execute `mac-address` and create a ‘mac-address’ property with that value.

Additional requirements for the `read` method:

- Receive (non-blocking) a network packet, placing at most the first `len` bytes of that packet into memory starting at `addr`. Return the number of bytes actually received (not the number copied into memory), or zero if no packet is currently available.

- Discard packets containing hardware-detected errors, as though they were not received.

Additional requirements for the `write` method:

- Transmit the network packet of length `len` bytes from the memory buffer beginning at `addr`. Return the number of bytes actually transmitted.

- The packet to be transmitted begins with an IEEE 802 Media Access Control (MAC) header.

Usage restriction: The caller must supply the complete header; the source hardware address will not necessarily be “automatically inserted” into the outgoing packet.

Additional requirements for the `load` method:

- Read the default client program into memory, starting at `addr`, using the default network booting protocol.

A standard package with this “device_type” property value may implement additional device-specific methods.

A standard package with this “device_type” property value shall implement the following property if the associated device has a preassigned IEEE 802.3-style MAC (network) address:

- “local-mac-address”

NOTE—Such packages often use the “obp-tftp” support package to implement the “load” method.

See also: “address-bits”, “max-frame-size”

new-device

Start new package, as child of active package.

Create a new device node as a child of the active package and make the new node the active package. Create a new instance and make it the current instance; the instance that invoked new-device becomes the parent instance of the new instance.

Subsequently, newly defined Forth words become the methods of the new node and newly defined data items (such as types `variable`, `value`, `buffer:`, and `defer`) are allocated and stored within the new instance.
new-token

(F: /FCode#/ -- ) F 0xB5

Create a new unnamed FCode function. Followed by FCode#.

FCode evaluation:

(F: /FCode#/ -- )

Read an FCode# from the current FCode program. Create a new FCode function, associating with it the FCode number FCode#. The new function’s execution semantics are initially undefined; they will be determined later by the execution of either b( ); b(create), b(defer), b(constant), b(buffer:), b(field), b(variable), or b(value).

The new function is unnamed, thus the only way to refer to it later is via its associated FCode number; it cannot be accessed by name from the user interface or via other mechanisms like $call-method.

Usage restriction: A standard FCode program shall use new-token only with FCode numbers in the program-defined range.

FCODE ONLY (Tokenized by defining words in headerless mode)

next-property

( previous-str previous-len phandle -- false | name-str name-len true ) F 0x23D

Return the name of the property following previous of phandle.

Name is a null-terminated string that is the name of the property following previous in the property list for device phandle. If previous is zero or points to a zero-length string, name is the name of the phandle’s first property. If there are no more properties after previous or if previous is invalid (i.e., names a property that does not exist in phandle), name is a pointer to a zero-length string.

nip

( x1 x2 -- x2 ) A,F 0x4D

Remove the second stack item.

nodefault-bytes

(E: -- addr len )

( maxlen "new-name< >" -- ) F 0x23E

Create custom configuration variable of size maxlen.

If the requested operation cannot be performed, a throw shall be called with an appropriate error message, as with abort".

NOTE—Out-of-memory conditions may be detected and handled properly in the code with ['] nodefault-bytes catch.

Users can create new configuration variables with the nodefault-bytes command. Although the values of user-created configuration variables persist across system resets, in order for them to be accessed Open Firmware must be “reminded” of their existence after every system reset. Furthermore, the nodefault-bytes commands creating them must be executed in the same order each time. For these reasons, nodefault-bytes is usually executed from the script.

nodefault-bytes creates a configuration variable whose data is of type byte-array. As with other built-in byte-array configuration variables, these user-created configuration variables can be set with setenv (restricted to printable characters) or $setenv and can be displayed with the printenv command. However, set-default and set-defaults have no effect on user-created configuration variables.

Used as: ok 100 nodefault-bytes new-name

Later used as: ( data-addr data-len ) " new-name" $setenv

Later used as: new-name ( data-addr data-len )

noop

( -- ) F 0x7B

Do nothing.

NOTE—noop is primarily used for debugging or testing purposes, such as a placeholder for patching in other commands or to provide short delays for debugging device-timing problems.

noshowstack

( -- ) A,F 0x23F

Turn off showstack (automatic stack display).

The system default is noshowstack.

See: showstack.
**not**

\[(x_1 \rightarrow x_2)\] T

Synonym for *invert*.

Tokenizer equivalent: *invert*

---

**$number**

\[(\text{addr} \rightarrow \text{true} | \text{n false})\] F 0xA2

Convert a string to a number.

Perform the conversion according to the current value in *base*. Return *true* if an inconvertible character is encountered.

---

**$nvalias**

\[(\text{name-str} \rightarrow \text{name-len} \rightarrow \text{dev-str} \rightarrow \text{dev-len})\]

Create nonvolatile device alias; edit the *script*.

Create the following command line in the *script*:

```
  devalias alias-name device-specifier
```

If the *script* already contains a *devalias* line with the same *alias* name, delete that entry and replace it with the new entry at the same location in the *script*. Otherwise, place the new entry at the beginning of the *script*.

If there is insufficient space in the *script* for the new *devalias* command, display a message to that effect and abort without modifying the *script*.

If the *script* was successfully modified, execute the new *devalias* command immediately, creating a new memory-resident alias.

If the *script* is currently being edited (i.e., *nvedit* has been executed, but has not been completed with either *nvstore* or *nvquit*), abort with an error message before taking any other action.

If the *script* was successfully modified, but *use-nvramrc?* is *false*, set *use-nvramrc?* to *true*.

**Used as:** `ok nvalias alias-name /full/pathname <eol>`

---

**nvquit**

\[(--)\]

Discard contents of *nvedit* temporary buffer.

Prompt for confirmation of the user’s intent to carry out this function. If confirmation is obtained, discard the *nvedit* temporary buffer. Otherwise, take no further action.
Contents of the script.

The size of the script region is system-dependent.
While it is possible to alter the contents of the script with setenv or the $setenv command, use of the script editor is preferred.
The contents of the script are cleared by set-defaults. Under some circumstances cleared contents can be recovered with nvrecover.
The commands in the script are interpreted during system start-up, but only if use-nvramrc? is true.
Configuration variable type: string[?]. Suggested default value: an empty string.
See: nvedit for more details on altering its contents.

Attempt to recover lost script contents.
Attempt to recover the contents of the script if they have been lost as a result of the execution of set-default or set-defaults. Enter the the script editor as with the nvedit command. In order for nvrecover to succeed, nvedit must not have been executed between the time that the script contents were lost and the time that nvrecover is executed.

Execute the contents of the nvedit temporary buffer.

Copy contents of nvedit temporary buffer into the script.
The nvedit temporary buffer is then cleared. Used after nvedit to save the results of an editing session into the script.

Delete nonvolatile device alias from the script.
If the script contains a devalias command line with the same name as alias-name, delete that command line from the script. Otherwise, leave the script unchanged. If the script is currently being edited (i.e., nvedit has been executed, but has not been completed with either nvstore or nvquit), abort with an error message before taking any other action.
Used as: ok nvunalias alias-name

Delete nonvolatile device alias from the script.
Perform the same function as nvunalias, except that the alias name is specified by name string.
Used as: ok " alias-name" $nvunalias

Interpret the following number as an octal number (base eight).

Interpretation:
Skip leading space delimiters. Parse number delimited by a space. Convert the string number to an integer n using a conversion radix of eight. Put n on the stack. An ambiguous condition exists if the conversion fails.
Compilation:
Skip leading space delimiters. Parse number delimited by a space. Convert the string number to an integer n using a conversion radix of eight. Append the run-time semantics given below to the current definition. An ambiguous condition exists if the conversion fails.
Run-time:run-time
Place n on the stack.
Interpret the number in octal regardless of the current value in base. The value in base is unchanged.
Used as: o# 1001 ( 513 )
Tokenizer equivalent: b(lit) xx-byte xx-byte xx-byte xx-byte
“obp-tftp”
Support package, implements TFTP protocol.
See: 3.8.2 for more information.

octal
( -- ) T
Set numeric conversion radix to eight.

Tokenizer: If octal is encountered in FCode source outside a definition, set the tokenizer’s numeric conversion radix to eight. If octal is encountered in FCode source inside a definition, append the following sequence to the FCode program that is being created.

Tokenizer equivalent: 8 base !

oem-banner
( -- text-str text-len ) N
Contain custom banner text, enabled by oem-banner?.
Configuration variable type: string[80]. Suggested default value: an empty string.

oem-banner?
( -- custom? ) N
If true, banner displays custom message in oem-banner.
If true, banner displays a custom message instead of the normal system-dependent messages.
If false, banner displays the normal system-dependent messages.
Configuration variable type: Boolean. Suggested default value: false.

oem-logo
( -- logo-addr logo-len ) N
Contain custom logo for banner, enabled by oem-logo?.
This logo is displayed by the banner command if oem-logo? is true.
The logo is a 512-byte field, representing a 64x64-bit logo bit map. Each bit controls one pixel. The most significant bit of the first byte controls the upper-left corner pixel. The next bit controls the next pixel to the right and so on.
oem-logo cannot receive arbitrary data with setenv, but $setenv can be used to set its value.
oem-logo is unaffected by set-default or set-defaults.
Used as: ( logo-addr logo-len ) " oem-logo" $setenv
Configuration variable type: bytes[512]. Suggested default value: all zeroes.

oem-logo?
( -- custom? ) N
If true, banner displays custom logo in oem-logo.
If true, banner will display the custom logo instead of the normal system-dependent logo.
If false, banner will display the normal system-dependent logo.
Configuration variable type: Boolean. Suggested default value: false.

of
(C: case-sys1 -- case-sys2 of-sys ) A,T
Begin of clause, execute through endif if params match.
Used within a case statement.
Tokenizer equivalent: b(of) +offset
ANS Forth note: Also works outside of a definition.

off
( a-addr -- ) F 0x6B
Store false to cell at a-addr.
offset ( d.rel -- d.abs ) M

Convert partition-relative disk position to absolute position.

This is a method of the disk label support package. d.rel is a double-number disk position, expressed as the number of bytes from the beginning of the partition that was specified in the arguments when the support package was opened. d.abs is the corresponding double-number disk position, expressed as the number of bytes from the beginning of the disk. If no partition was specified when the support package was opened, a system-dependent default partition is used. If the disk label support package does not support disk partitioning, d.abs is equal to d.rel.

offset16 ( -- ) F 0xCC

All further branches use 16-bit (not 8-bit) offsets.

Sets the FCode-offset size to 16 bits. Within the current FCode program, subsequent FCode functions that read an FCode-offset read the 16-bit form.

Tokenizer: Execution (once only) of offset16 causes the tokenizer to use the 16-bit form, rather than the 8-bit form, for all subsequent FCode-offsets within this FCode program.

A tokenizer may automatically insert offset16 at the beginning of an FCode program.

Once offset16 is executed, the offset size remains 16 bits for the duration of the FCode program; it cannot be set back to 8 bits. Multiple calls of offset16 have no additional effect. offset16 is only useful within an FCode program that begins with version1. All other starting tokens (start0, start1, start2, and start4) automatically set the offset size to 16 bits.

FCODE ONLY

on ( a-addr -- ) F 0x6A

Store true to cell at a-addr.

open ( -- okay? ) M

Prepare this device for subsequent use.

Typical behavior is to allocate any special resource requirements it needs, map the device into virtual address space, initialize the device and perform a brief “sanity test” to ensure that the device appears to be working correctly.

Return true if this open method was successful, false if not.

When a device’s open method is called, that device’s parent has already been opened (and so on, up to the root node, which has no parent), so this open method can call its parent’s methods, for instance to create mappings within the parent’s address space.

Several device types require the existence of an open method, particularly those with read and write methods.

open-dev ( dev-str dev-len -- ihandle | 0 )

Open device (and parents) named by given device-specifier.

Open the device specified by dev-string. Return ihandle if successful, or 0 if not. Open each node of the device tree in turn, starting at the top. The current instance and the active package are not changed.

The opening process is as defined in 4.3, using the rules given for open-dev.

Used as: " device-alias" open-dev

open-package ( arg-str arg-len phandle -- ihandle | 0 ) F 0x205

Open the package indicated by phandle.

Create an instance of the package identified by phandle, save in that instance the instance-argument specified by arg-string and invoke the package’s open method.

Return the instance handle ihandle of the new instance, or 0 if the package could not be opened. This could occur either because that package has no open method, or because its open method returned false, indicating an error.

The parent instance of the new instance is the instance that invoked open-package. The current instance is not changed.
$open-package ( arg-str arg-len name-str name-len -- ihandle | 0 ) F 0x20F
Open the support package named by name string.
Similar to find-package ... open-package, except that if find-package fails, returns 0 immediately, without calling open-package.
Equivalent to: find-package if open-package else 2drop false then

"/openprom" S
The node describing this Open Firmware implementation.
See: 3.5 for a complete description.

"/options" S
The node containing this system’s configuration variables.
See: 3.5 for a complete description.

or ( x1 x2 -- x3 ) A,F 0x24
Return bitwise logical “inclusive-or” of x1 and x2.

#out ( -- a-addr ) F 0x93
variable holding the output column number.
Increment the value when a character is displayed and reset to zero when cr is executed.

output ( dev-str dev-len -- )
Select the indicated device for console output.
Search for a device node matching the pathname or device-specifier given by dev-str dev-len. The search process is as defined in 4.3, using the rules given for find-device, but restore the active package to its previous package afterwards.
If such a device is found, search for its write method.
If the write method is found, open the device, as with open-dev.
If any of these steps fails, display an appropriate error message and return without performing the steps following the one that failed.
If there is a console output device, as indicated by a nonzero value in the stdout variable, close the console output device.
Set stdout to the ihandle of the newly opened device, making it the new console output device.
Used as: " device-alias" output

output-device ( -- dev-str dev-len ) N
Default console output device.
Indicates the console output device to be established by install-console. dev-string is a device-specifier, containing either a full device-path or a pre-defined device alias.
Used as: ok setenv output-device device-alias <eol>
Configuration variable type: string[32]. Suggested default value: screen.

over ( x1 x2 -- x1 x2 x1 ) A,F 0x48
Copy second stack item to top of stack.

2over ( x1 x2 x3 x4 -- x1 x2 x3 x4 x1 x2 ) A,F 0x54
Copy second pair of stack items to top of stack.

pack ( str len addr -- pstr ) F 0x83
Pack a text string into a counted string.
Store the string str,len as a counted string beginning at the address addr, returning the result pstr, which is the same address as addr. At most 256 characters, including the count byte, shall be stored in the array at addr.
"/packages"  S
The node containing all standard support packages.
For example, the "disk-label" support package is located in the device tree at "/packages/disk-label".
See also: 3.8.

parse   ( delim "text<delim>" -- str len )  A
Parse text from the input buffer, delimited by delim.

parse-2int   ( str len -- val.lo val.hi )  F  0x11B
Convert a "hi,lo" string into a pair of values.
In the string, val.hi is first, separated from val.lo by a comma.
Used as:  " 33,555"  parse-2int  ( 555 33 )
Perform the conversion according to the current value in base.
If the string does not contain a comma, val.lo is zero and val.hi is the result of converting the entire string. If either
component contains non-numeric characters, according to the value in base, the result is undefined.

parse-word   ( "text< >" -- str len )
Parse text from the input buffer, delimited by white space.
Skip leading spaces and parse name delimited by a space. str is the address (within the input buffer) and len is the length
of the selected string. If the parse area was empty, the resulting string has a zero length.

password   ( -- )
Prompt user to set security password.
Prompt the user (twice) to enter a new password, terminated by end-of-line. Do not echo the password on the screen as it
is typed. The password length is zero to eight characters in length. Ignore any additional characters (more than eight).
If the entered password is the same both times, store the new password string in security-password. Note that
security-mode must be set to enable password protection.

patch   ( "new-name< >old-name< >word-to-patch< >" -- )
Change contents of word-to-patch.
In the compiled definition of word-to-patch, change the first occurrence of old-name to new-name. Works properly even if
old-name and/or new-name are numbers.
Used as:  ok patch 555 test patch-me
to edit the definition of patch-me, replacing the command test with the literal value 555.
Implementation note:
When replacing a command with a number, an implementation might need to automatically create a named constant value
for the replacement number. (The reason is that Forth commands often compile into a smaller memory space than literal
numbers, so patching a number in place of an existing command is a problem.) A suggested name format is h#--.--, i.e.,
the number 555 (hex) would be named "h#555". A name containing only digits (i.e., 555 constant 555) is not
recommended, since changing base would cause incorrect evaluation of subsequent uses of that named value.

(patch)   ( new-n1 num1? old-n2 num2? xt -- )
Change contents of command indicated by xt.
In the compiled definition of the command indicated by xt, change the first occurrence of old-n2 to new-n1. n1 and n2 can
each be either an execution token or a literal number. The flag num1?, if true, indicates that new-n1 is a literal number.
If false, it indicates that new-n1 is an execution token. The flag num2? is interpreted similarly.
Used as:  ['] new-name false 555 true ['] patch-me (patch)
to edit the definition of patch-me, replacing the value 555 with the command new-name.
See: patch for more information.
**peer**

( phandle -- phandle.sibling )  F  0x23C

Return the phandle of the next sibling node.

*Phandle.sibling* is the node identifier of the node that is the next sibling of the device described by *phandle*, or zero if there are no more siblings. If *phandle* is zero, *phandle.sibling* is the node identifier of the root node.

**pick**

( xu ... x1 x0 u -- xu ... x1 x0 xu )  A,F  0x4E

Copy uth stack item to top of stack.

Remaining stack items are unchanged.

For example:

```
0 pick <=> dup
1 pick <=> over
```

**postpone**

( C: [old-name< >] -- )  A,C
( ... -- ??? )

Delay execution of the immediately following command.

**printenv**

( "{param-name}<eol>" -- )

Display current, default value of configuration variable (or all).

Parse *param-name*, delimited by end-of-line. If *param-name* is missing, display the current and default values of all configuration variables. Otherwise, display the current and default values of the configuration variable given whose name is *param-name*.

The values are displayed in their output text representation form. The overall display format is implementation-dependent. A Open Firmware implementation may, at its option, display an abbreviated form of some of the values and may suppress the display of nonprintable characters.

**probe-all**

( -- )

Probe for all available plug-in devices.

Search for plug-in devices on the system-dependent set of expansion buses, creating device nodes for devices that are located.

**NOTE**—Undesirable results, such as duplicate device nodes for the same device, might occur if *probe-all* is executed more than once. It is normally executed automatically during system start-up following the evaluation of the *script*, but this automatic execution is disabled if *banner* or *suppress-banner* is executed from the *script*.

**probe-self**

( arg-str arg-len reg-str reg-len fcode-str fcode-len -- )  M

Evaluate FCode as a child of this node.

*fcode-string* is a unit address text string, representing the location of the FCode program for the child device.

*reg-string* is a probe-address text string, representing the location of the child device itself.

*arg-string* is a instance-arguments text string, providing the arguments for the child (which can be retrieved within the child's FCode program with the *my-args* FCode.)

First check to see if there is an FCode program at the indicated location (perhaps by mapping the device and using *cpeek* to ensure that the device is present and that the first byte is a valid FCode start byte). If so, perform the function of *new-device* (thus creating a new device node), then interpret the FCode program, then perform the function of *finish-device*.

If a valid FCode program cannot be located at the indicated address, do not create a new device node.

A typical implementation of *probe-self* might execute *byte-load* and *set-args*.

**NOTE**—If the FCode program is a standard package, successful completion of *probe-self* will be indicated by the presence of a new device node containing a "name" property. If the evaluation of the FCode program fails in some way, the new device node might be empty (containing no properties or methods.)

**.properties**

( -- )

Display names and values of properties of the active package.
property ( prop-addr prop-len name-str name-len -- ) F 0x110
Create a new property with the given name and value.
If there is a current instance, create a property in the package from which the current instance was created. Otherwise, if there is an active package, create a property in the active package. If there is neither a current instance nor an active package, the result is undefined. The new property’s property name is given by name string and its value is given by the prop-encoded-array prop-addr prop-len.
If a property with that property name already exists in the package in which the property would be created, replace its value with the new value.
See the specifications of individual property for any additional requirements.
Used as: 55 encode-int " my-property-name" property

pwd ( -- )
Display the device-path that names the active package.

quit ( -- ) (R: ... -- ) A
Abort program execution.

r> ( -- x ) (R: x -- ) A,F 0x31
Move top return stack item to the stack.
Usage restrictions: See ANS Forth return stack restrictions.
ANS Forth note: Usage also allowed while interpreting.

r@ ( -- x ) (R: x -- x ) A,F 0x32
Copy top return stack item to the stack.
Usage restrictions: See ANS Forth return stack restrictions.
ANS Forth note: Usage also allowed while interpreting.

.r ( n size -- ) A,F 0x9E
Display a signed number, right-justified.

>r ( x -- ) (R: -- x ) A,F 0x30
Move top stack item to the return stack.
Usage restrictions: See ANS Forth return stack restrictions.
ANS Forth note: Usage also allowed while interpreting.
“ranges” S

Standard property name to define a device’s physical address.

Buses such as SBus and VMEbus, whose children can be accessed with CPU load-and-store operations (as opposed to buses such as SCSI or IPI, whose children are accessed with a command protocol), require a way to define the relationship between the physical address spaces of the parent and child nodes. The “ranges” property provides this capability.

The value of the “ranges” property describes the correspondence between the physical address space defined by a bus node (the “child address space”) and the physical address space of that bus node’s parent (the “parent address space”). The “ranges” property value is a sequence of

child-phys parent-phys size

specifications. Child-phys is an address, encoded as with encode-phys, in the child address space. Parent-phys is an address (likewise encoded as with encode-phys) in the parent address space. Size is a list of integers, each encoded as with encode-int, denoting the length of the child’s address range. The number of integers in each size entry is determined by the value of the #size-cells property of this node (the node in which the ‘ranges’ property appears) or 1 if the #size-cells property is absent. The interpretation of size is bus-dependent.

Each specification defines a one-to-one correspondence between the child addresses in the range child-phys..child-phys+size–1 and the parent addresses in the range parent-phys..parent-phys+size–1. The address ranges thus described might (and often will) define a sparse address space, i.e., the address ranges need not be consecutive in either the child address space or the parent address space. Successive specifications are encoded one after another. It is recommended, but not required, that the specifications be sorted in ascending order of child-phys.

If a “ranges” property exists but has a zero-length property value, the child address space is identical to the parent address space.

The absence of a “ranges” property for a bus node indicates that there is no direct correspondence between the child address space and the parent address space, e.g., the bus is a command protocol bus such as SCSI.

Example: Suppose that a 4-slot, 28-bit SBus is attached to a machine whose physical address space consists of a 32-bit memory space (phys.hi=0) and a 32-bit i/o space (phys.hi=1). The SBus slots are numbered 0, 1, 2, and 3, and appear in i/o space at addresses 0x80000000, 0x90000000, 0xA0000000, and 0xB0000000, respectively. The “ranges” property for this “sbus” device node would contain the encoded form of the following sequence of numbers:

<table>
<thead>
<tr>
<th>Child Address</th>
<th>Parent Address</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>phys.hi phys.lo</td>
<td>phys.hi phys.lo</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

See also: “#size-cells”.

rb! (FCode function) ( byte addr -- )

Store a byte to device register at addr.

Data is stored with a single access operation and flushes any intervening write buffers, so that the data reaches its final destination before the next FCode function is executed.

Register is stored with identical bit ordering as the input stack item.

rb! (user interface) ( byte addr -- )

Store a byte to device register at addr.

Compilation:

Perform the equivalent of the phrase:

h# 231 get-token if execute else compile, then

Interpretation:

Perform the equivalent of the phrase:

h# 231 get-token drop execute

NOTE—A bus device can substitute (see set-token) a bus-specific implementation of rb! for use by its children. This is sometimes necessary to correctly implement its semantics with respect to bit-order and write-buffer flushing. The given user interface semantics of rb! ensure that such substitutions are visible at the user interface level.
IEEE
Std 1275-1994
IEEE STANDARD FOR BOOT (INITIALIZATION CONFIGURATION) FIRMWARE:

\texttt{rb@} (FCode function) \hfill \texttt{rb@} (user interface)
\begin{verbatim}
( addr -- byte ) \hfill ( addr -- byte )
\end{verbatim}

Fetch a byte from device register at \texttt{addr}.

Data is read with a single access operation.
Result has identical bit ordering as the original register data.

Compilation:
\begin{verbatim}
( -- )
\end{verbatim}

Perform the equivalent of the phrase:
\begin{verbatim}
(h# 230 get-token if execute else compile, then)
\end{verbatim}

\textbf{NOTE}—A bus device can substitute (see \texttt{set-token}) a bus-specific implementation of \texttt{rb@} for use by its children. This is sometimes necessary to correctly implement its semantics with respect to bit-order and write-buffer flushing. The given user interface semantics of \texttt{rb@} ensure that such substitutions are visible at the user interface level.

\textbf{Compilaton:}
\begin{verbatim}
( -- )
\end{verbatim}

Perform the equivalent of the phrase:
\begin{verbatim}
h# 230 get-token drop execute
\end{verbatim}

\texttt{read} \hfill \texttt{read}
\begin{verbatim}
( addr len -- actual ) \hfill ( addr len -- actual )
\end{verbatim}

Read device into memory buffer; return actual byte count.

Read at most \texttt{len} bytes from the device into the memory buffer beginning at \texttt{addr}. Return \texttt{actual}, the number of bytes actually read. If \texttt{actual} is zero or negative, the read operation did not succeed.

Some standard device types impose additional requirements on their \texttt{read} methods; see the descriptions of various device types (e.g., "network") for more information.

For some devices, the \texttt{seek} method sets the position for the next \texttt{read}.

\texttt{read-blocks} \hfill \texttt{read-blocks}
\begin{verbatim}
( addr block# #blocks -- #read ) \hfill ( addr block# #blocks -- #read )
\end{verbatim}

Read \texttt{#blocks}, starting at \texttt{block#}, from device into memory.

Read \texttt{#blocks} records of length \texttt{block-size} bytes from the device (starting at device block \texttt{block#}) into memory (starting at \texttt{addr}). Return \texttt{#read}, the number of blocks actually read.

If the device is not capable of random access (e.g., a sequential access tape device), \texttt{block#} is ignored.

\textbf{Compilaton:}
\begin{verbatim}
( -- )
\end{verbatim}

Allow the current definition to be found in the dictionary.

\textbf{NOTE}—Normally, when a colon definition is being compiled, its name is not visible in the dictionary until the definition is completed. This way a call to that same name finds the previous version, not itself. \texttt{recursive} makes the current definition visible so that subsequent uses of its name compile recursive calls to itself.
Create the "reg" property with the given values.

Shorthand command to create a property in the active package whose property name is "reg" for buses whose "#size-cells" value is one.

Equivalent to:

\[ r \ ( \text{phys.lo} \ldots \text{phys.hi} \text{ size } ) \quad \text{encode-phys } \ ( \text{addr} \text{ len } ) \]
\[ r> \ ( \text{addr1 len1 size } ) \quad \text{encode-int } \ ( \text{addr1 len1 addr2 len2 } ) \]

"reg" property

The \texttt{reg} function creates a "reg" property with a single \texttt{phys.lo} \ldots \texttt{phys.hi size} specification. It is not appropriate to use \texttt{reg} if the parent specifies a "#size-cells" value other than one. To create a "reg" property with multiple specifications, the \texttt{property} command must be used.

\textbf{Used as: my-address 8000 + my-space 40 reg}

\textbf{See also: "#address-cells", "reg", "#size-cells".}

"reg"

Standard property name to define the package’s registers.

\texttt{prop-encoded-array:}

- Arbitrary number of (\texttt{phys-addr size}) pairs.
- \texttt{phys-addr} is a (\texttt{phys.lo} \ldots \texttt{phys.hi}) list, encoded with \texttt{encode-phys}.
- \texttt{size} is a list of integers, each encoded with \texttt{encode-int}.

Specifies the range of addressable regions on the device. The "reg" property represents the physical address, within its parent node’s address space, of the device associated with the node and also the amount of physical address space consumed by that device. In general, the "reg" property of a node can contain several \texttt{phys.lo} \ldots \texttt{phys.hi size} specifications representing several disjoint ranges of physical address space.

The number of integers in each \texttt{size} entry is determined by the value of the "#size-cells" property in the parent node.

If the parent node has no such property, the value is one. The interpretation of the \texttt{size} entries is dependent on the parent bus.

\texttt{phys.lo} \ldots \texttt{phys.hi} is typically obtained by modifying the values obtained from \texttt{my-address my-space}, perhaps by adding the offset of device registers to some component of the base address obtained from \texttt{my-address my-space}.

\textbf{Used as (assuming "#size-cells" is one):}

\begin{align*}
\texttt{my-address} & \ 8000 \ + \ \texttt{my-space} \ \texttt{encode-phys} \\
& \ 40 \ \texttt{encode-int} \ \texttt{encode+} \\
\texttt{my-address} & \ c000 \ + \ \texttt{my-space} \ \texttt{encode-phys} \ \texttt{encode+} \\
& \ 8 \ \texttt{encode-int} \ \texttt{encode+} \\
& \ " \texttt{reg}" \ \texttt{property} \\
\end{align*}

\textbf{NOTE—}The first specified \texttt{phys-addr} becomes the default \texttt{unit address} for subsequent instances of this package. This value will be used when a precise specification of this card is required (i.e., device tree specifications). For example, in a device-path of: \texttt{../.../XYZ,devname@3,2000}, the "3,2000" is the text representation of the first \texttt{phys.lo phys.hi} specification in this package’s "reg" property declaration.

\textbf{See also: "#address-cells", \texttt{my-unit.property.reg}, "#size-cells"}

\textbf{.registers}

Display saved register values.

Display the register values that were in effect when the program state was saved (i.e., when the program was suspended). The exact set of registers displayed, and the format, is ISA-dependent.
“relative-addressing”
Standard property name to indicate firmware addressing style.

prop-encoded-array:
None; presence or absence of the property conveys the information.

The presence of this property indicates that each device node address is relative, i.e., local to the address space defined by
the node’s parent. The absence of the property indicates that device node addresses are absolute addresses within the
system-wide address space.

This property shall be present within the /openprom node (because this specification requires relative device node
addresses).

release ( addr ... len ... -- ) M
Free (release) addressable resource.
Frees len ... (whose format depends on the package) bytes of the addressable resource managed by the package containing
this method, beginning at the address addr ... (whose format depends on the package), making it available for subsequent
use.
See also: claim, alloc-mem, ”available”, free-mem

remove-abort ( -- ) M
Cease polling for a keyboard abort sequence.
Instruct the device driver to cease periodic polling for a keyboard abort sequence. Executed when the console input device
is changed from this device to another.

repeat (C: orig-sys dest-sys -- ) A,T
Mark end of a begin...while...repeat loop. Jump to begin.

Compilation:
(C: orig-sys dest-sys -- )
Perform the compilation semantics of ANS Forth REPEAT. Then, if the current definition is temporary and the depth of
the control flow stack is the same as its depth when the temporary current definition was initiated, perform the
compilation semantics of ; and execute the temporary current definition.
Run-time:
( -- )
Same as ANS Forth.
Tokenizer equivalent: bbranch -offset b(>resolve)
ANS Forth note: Also works outside of a definition.

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

The definition of “quiescent” is device-specific. This method is used primarily for permanently installed devices (which
are therefore not probed) that do not automatically assume a quiescent state after a system reset.
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-all ( -- )
Reset the machine as if a power-on reset had occurred.

This command is used to initiate a system power-on reset, thus re-initializing the hardware state and Open Firmware’s
data structures as if a power-on reset had occurred.
Perform frame-buffer device initialization.

**reset-screen** (--) F 0x158

**reset-screen** is one of the **defer** words of the display device interface. The terminal emulator package executes **reset-screen** when it has processed a character sequence that calls for resetting the display device to its initial state. The open method that is automatically created by **is-install** executes **reset-screen** after executing **erase-screen** as part of the process of initializing the terminal emulator. **is-install** automatically creates a “**restore**” method that executes **reset-screen**.

Any standard package that uses the terminal emulator package shall, as part of the process of opening the terminal emulator package, set this **defer** word to a function with the following behavior:

Put the display device into a state in which display output is visible (e.g., enable video).

**See also:** **to**, **fb8-install**.

**restore** (--) M

Restore device to useable state after unexpected reset.

On some systems, unexpected system errors result in a bus reset that turns off some devices, but does not necessarily destroy the machine state necessary for debugging the error. In such systems, the system-dependent firmware handler for that reset condition may execute the **restore** methods of the console input and output devices, in order to re-enable those devices for user interaction and subsequent debugging.

**NOTE**—**is-install** automatically creates an implementation of this method whose behavior is to execute the **reset-screen** **defer** word.

**resume** (--) M

Exit from a “subordinate interpreter” back to the stepper.

This command is used after the f keystroke was used with the stepper.

**return** (--) M

Execute until return from this subroutine.

**ring-bell** (--) M

Ring the bell.

Cause the device to emit a brief audible sound (beep).

**See also:** **blink-screen**.

**rl!** (FCode function) (quad qaddr --) F 0x235

Store a quadlet to device register at **qaddr**.

Data is stored with a single access operation and flushes any intervening write buffers, so that the data reaches its final destination before the next FCode function is executed.

Register is stored with identical bit ordering as the input stack item.

**rl!** (user interface) (quad qaddr --)

Store a quadlet to device register at **qaddr**.

**Compilation:** (--)

Perform the equivalent of the phrase:

```
  h# 235 get-token if execute else compile, then
```

**Interpretation:** (quad qaddr --)

Perform the equivalent of the phrase:

```
  h# 235 get-token drop execute
```

**NOTE**—A bus device can substitute (see **set-token**) a bus-specific implementation of **rl!** for use by its children. This is sometimes necessary to correctly implement its semantics with respect to bit-order and write-buffer flushing. The given user interface semantics of **rl!** ensure that such substitutions are visible at the user interface level.
**rl@** (FCode function) 
(qaddr -- quad) F 0x234
Fetch a quadlet from device register at qaddr.
Data is read with a single access operation.
Result has identical bit ordering as the original register data.

**rl@** (user interface) 
(qaddr -- quad)
Fetch a quadlet from device register at qaddr.

**Compilation:**
Perform the equivalent of the phrase:
```
 h# 234 get-token if execute else compile, then
```

**Interpretation:**
Perform the equivalent of the phrase:
```
 h# 234 get-token drop execute
```

**NOTE—**A bus device can substitute (see set-token) a bus-specific implementation of rl@ for use by its children.
This is sometimes necessary to correctly implement its semantics with respect to bit-order and write-buffer flushing. The given user interface semantics of rl@ ensure that such substitutions are visible at the user interface level.

**roll** 
(xu ... x1 x0 u -- xu-1 ... x1 x0 xu) A,F 0x4F
Rotate u+1 stack items as shown.

**For example:**
1 roll <=> swap
2 roll <=> rot

**rot** 
(x1 x2 x3 -- x2 x3 x1) A,F 0x4A
Rotate top three stack items as shown.

**-rot** 
(x1 x2 x3 -- x3 x1 x2) F 0x4B
Rotate top three stack items as shown.

**2rot** 
(x1 x2 x3 x4 x5 x6 -- x3 x4 x5 x6 x1 x2) A,F 0x56
Rotate three pairs of stack items as shown.

**rshift** 
(x1 u -- x2) A,F 0x28
Shift x1 right by u bit-places. Zero-fill high bits.

**rw!** (FCode function) 
(w waddr -- ) F 0x233
Store a doublet w to device register at waddr.
Data is stored with a single access operation and flushes any intervening write buffers, so that the data reaches its final destination before the next FCode function is executed.
Register is stored with identical bit ordering as the input stack item.
\textbf{rw!} (user interface) \hspace{1cm} ( w waddr -- )

Store a doublet \textit{w} to device register at \textit{waddr}.

\textbf{Compilation:} \hspace{1cm} ( -- )

Perform the equivalent of the phrase:

\begin{verbatim}
  h# 233 get-token if execute else compile, then
\end{verbatim}

\textbf{Interpretation:} \hspace{1cm} ( w waddr -- )

Perform the equivalent of the phrase:

\begin{verbatim}
  h# 233 get-token drop execute
\end{verbatim}

\textbf{NOTE—}A bus device can substitute (see \texttt{set-token}) a bus-specific implementation of \texttt{rw!} for use by its children. This is sometimes necessary to correctly implement its semantics with respect to bit-order and write-buffer flushing. The given user interface semantics of \texttt{rw!} ensure that such substitutions are visible at the user interface level.

\textbf{rw@} (FCode function) \hspace{1cm} ( waddr -- w ) \hspace{1cm} F \hspace{1cm} 0x232

Fetch a doublet \textit{w} from device register at \textit{waddr}.

Data is read with a single access operation.

Result has identical bit ordering as the original register data.

\textbf{rw@} (user interface) \hspace{1cm} ( waddr -- w )

Fetch a doublet \textit{w} from device register at \textit{waddr}.

\textbf{Compilation:} \hspace{1cm} ( -- )

Perform the equivalent of the phrase:

\begin{verbatim}
  h# 232 get-token if execute else compile, then
\end{verbatim}

\textbf{Interpretation:} \hspace{1cm} ( waddr -- w )

Perform the equivalent of the phrase:

\begin{verbatim}
  h# 232 get-token drop execute
\end{verbatim}

\textbf{NOTE—}A bus device can substitute (see \texttt{set-token}) a bus-specific implementation of \texttt{rw@} for use by its children. This is sometimes necessary to correctly implement its semantics with respect to bit-order and write-buffer flushing. The given user interface semantics of \texttt{rw@} ensure that such substitutions are visible at the user interface level.

\textbf{s"} \hspace{1cm} ( [text" ] -- text-str text-len ) \hspace{1cm} A,T

Gather the immediately following string.

The description of \textit{S"} in the ANS Forth “File-Access” wordset applies to this standard. At least two temporary buffers, used alternately, shall be provided.

\textbf{Usage restriction:} Since an implementation is only required to provide two temporary buffers, a standard program cannot depend on the system’s ability to simultaneously maintain more than two distinct interpreted strings. Compiled strings do not have this limitation, since they are not stored in the temporary buffers.

\textbf{Tokenizer equivalent:} \begin{verbatim} b(" ) len-byte xx-byte xx-byte ... xx-byte \end{verbatim}

\textbf{s.} \hspace{1cm} ( n -- ) \hspace{1cm} T

Display a signed number, with a trailing space.

Display the number according to the current value in \texttt{base}, with a leading minus sign if necessary.

\textbf{Tokenizer equivalent:} \begin{verbatim} (.) type space \end{verbatim}

\textbf{#s} \hspace{1cm} ( ud -- 0 0 ) \hspace{1cm} A,F \hspace{1cm} 0xC8

Convert remaining digits in pictured numeric output.

\textbf{.s} \hspace{1cm} ( ... -- ... ) \hspace{1cm} A,F \hspace{1cm} 0x9F

Display entire stack contents, unchanged.
Converts SBus interrupt level to CPU interrupt level.
For systems with one built-in SBus, return the CPU interrupt level \( cpu-intr\) corresponding to the SBus interrupt level
\( sbus-intr\) for that built-in SBus. For other systems, return \( cpu-intr\) equal to \( sbus-intr\).

\( sbus-intr\) is used by some existing FCode drivers for devices that interrupt on more than one SBus interrupt level,
both computing the value of the “intr” property. That property has been replaced by the “interrupts” property,
which specifies the SBus interrupt level directly, without requiring that it be converted to the corresponding CPU interrupt level.
It is not always possible for an SBus node to know the mapping from the SBus level to the CPU level, especially in cases where the SBus node results from a plug-in bridge from some other bus to SBus.

This FCode function should be used only by those FCode programs that require compatibility with older SBus systems. It
should not be used by FCode programs for non-SBus devices. The specification of this property is included here, rather
than in an SBus-specific supplement, because of the possibility that, even on systems that nominally do not support SBus,
SBus devices might be used via a bus-to-bus bridge.

\textbf{screen-\#columns} \quad \text{( -- n ) \ N}

Maximum number of columns on console output device.

Standard display packages use this value to determine the width in characters of their text region. If the device is
incapable of displaying that many columns, the device restrictions prevail.

Configuration variable type: \textit{integer}. Suggested default value: \texttt{80}.

\textbf{screen-height} \quad \text{( -- height ) \ F \ 0x163}

\textit{value}, return total height of the display in pixels.

\textbf{screen-width} \quad \text{( -- width ) \ F \ 0x164}

\textit{value}, return total width of the display in pixels.

\textbf{s>d} \quad \text{( n1 -- d1 ) \ A}

Convert a number to a double number.

\textbf{security-\#badlogins} \quad \text{( -- n ) \ N}

Contain total count of invalid security access attempts.

This counter is incremented by one, whenever a bad password is entered when attempting to enter the command
interpreter while \texttt{security-mode} is set (to either \texttt{command} mode or \texttt{full} mode).

The value in \texttt{security-\#badlogins} is not affected by the \texttt{set-default} or \texttt{set-defaults} commands.

Configuration variable type: \textit{integer}. There is no suggested default value.
security-mode   ( -- n )        N

Contain level of security access protection.

Security mode requires user knowledge of a password to allow use of most commands through the command interpreter.

Used as: ok  setenv security-mode  full

The following keywords denote the security levels:

none No security, no password required.

command Requires password entry to execute any command except for go, boot (default device and default file), or automatic boot after system power-on or boot call.

full Requires password entry to execute any command except for go command. Automatic booting is disabled, machine will not automatically reboot after a power failure.

The value of security-mode is not affected by the set-default or set-defaults commands.

Configuration variable type: security-mode. There is no suggested default value.

NOTE—It is not possible to determine the level of security protection from within a program, as the value n returned by this command cannot be related unambiguously to the security level keywords.

security-password   ( -- password-str password-len )        N

Contain security password text string.

The value of security-password shall not be displayed when printenv is executed. The value of security-password is not affected by the set-default or set-defaults commands. If the value is set to a value that contains fewer characters than its prior value, the remaining characters of the prior value should be set to zero to prevent accidental discovery of a prior password.

Configuration variable type: string[8]. There is no suggested default value.

NOTE—The value of security-password is normally set with the password command, although setenv can also be used.

see   ( "old-name" -- )        A

Decompile the Forth command old-name.

Used as: ok  see old-name

(see)   ( xt -- )

Decompile the Forth command whose execution token is xt.

Used as: ['] old-name  (see)

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

Set device position for next read or write.

Set the device position at which the next read or write will take place. The position is specified by a pair of numbers pos.lo pos.hi, whose interpretation depends on the device type. Return –1 if the operation fails and either zero or one if it succeeds.

NOTE—The return value one (1) is meant as a concession to existing practice. Programs that use the seek method should treat either of the status values 0 or 1 as an indication of success.

selftest   ( -- 0 | error-code )        M

Perform self-test for this device.

Return 0 if successful, a device-specific nonzero error number if an error is detected. The complexity of this test will typically be much greater than that of the test performed when open is called.

This method is typically invoked by the user commands test or test-all, via execute-device-method.

Consequently, the package’s open method has not necessarily been executed before selftest is invoked.

execute-device-method does not call open, but it is possible for the device to have already been previously opened. selftest should leave the device in a state similar to that before selftest was executed. Therefore, selftest is responsible for establishing any device state necessary to perform its function prior to starting the tests and for releasing any resources that were allocated during the process after completing the tests.

The extent of the testing performed by selftest may depend on the value returned by diagnostic-mode?; if so, more extensive testing shall be performed when diagnostic-mode? return true.
selftest-#megs \( (-- n) \) \( N \)
Number of megabytes of memory to test.

The `selftest` routine of the "memory" node (the node whose `ihandle` is given by the value of `/chosen`'s "memory" property) is a memory test. In most systems that memory test is automatically executed after the secondary diagnostics. (Some smaller portion of memory is usually tested by POST, as well.) `selftest-#megs` controls the extent of memory `selftest`. If `diagnostic-mode?` is true, the system may ignore the value of `selftest-#megs`.


```
"serial" \( S \)
Byte-oriented device type such as a serial port.

Standard string value of the "device_type" property for serial devices.
A standard package with this "device_type" property value shall implement the following methods.

`open`, `close`, `read`, `write`, `install-abort`, `remove-abort`
A standard package with this "device_type" property value should implement the following method if an unexpected system reset can cause the display to become invisible (e.g., the video is turned off) and the display can be restored to visibility without performing memory mapping or memory allocation operations:

`restore`
A standard package with this "device_type" property value should implement the following method if the device has an audible annunciator that requires some action other than sending an ASCII BEL character in order to make it emit a beep:

`ring-bell`
A standard package with this "device_type" property value may implement additional device-specific methods.

Additional requirements for the `read` method:
Receive a number of bytes equal to the minimum of `len` and the number of bytes available for immediate reception, from the device into memory starting at `addr`. Return `actual`, the number of bytes read, or -2 if no bytes are currently available from that device.

See also: `character-set`
```
set-args \( (arg-str \arg-len \unit-str \unit-len \--) \) \( F \) \( 0x23F \)
Set address and arguments of new device node.
`unit-string` is a text string representation of a physical address within the address space of the parent device. Translate `unit-string` to the equivalent numerical representation by executing the parent instance's "decode-unit" method. Set the current instance's `probe-address` (i.e., the values returned by `my-address` and `my-space`) to that numerical representation.

Copy the string `arg-string` to instance-specific storage, and arrange for `my-args` to return the address and length of that copy when executed from the current instance.

**NOTE**—`set-args` is typically used just after `new-device`. `new-device` creates and selects a new device node, and `set-args` sets its `probe-address` and arguments. Subsequently, the device node's properties and methods are created by interpreting an FCode program with `byte-load` or by interpreting Forth source code.

**NOTE**—The empty string (any string of zero length) is commonly used as the arguments for a new device node, for example: `0 0 " 3,2000" set-args`

set-default \( ("param-name<eol>" \(--\)) \)
Set configuration variable to default value.

**Used as:** `ok set-default nv-name <eol>`
Some configuration variables are unaffected by `set-default`, as noted in individual configuration variable command descriptions.

set-defaults \( (\--) \)
Reset most configuration variables to their default values.

Some configuration variables are unaffected by `set-defaults`, as noted in individual configuration variable command descriptions.
setenv
  ( "nv-param< >new-value<eol>" -- )
  Set the configuration variable nv-param to the indicated value.
  
  Skip leading space delimiters. Parse nv-param delimited by a space. Parse new-value as the remainder of the input buffer
  minus leading and trailing white space.
  
  If new-value is the empty string, display an error message and return.
  Otherwise, perform the equivalent of $setenv, with string arguments denoting new-value and new-value.
  
  See Also: $setenv, 7.4.4.1.
  
  Used as:
  
  ok setenv auto-boot? true <eol>
  ok setenv selftest-#megs 55 <eol>
  ok setenv oem-banner This.Is also22 more-text4 <eol>
  
  The stored string in this case is "This.Is also22 more-text4".

$setenv
  ( data-addr data-len name-str name-len -- )
  Set the configuration variable name string to new value.
  
  data-addr,len is a string of characters or bytes representing the new value for the configuration variable whose name is
given by name-str,len. Interpret that new value according to the input text representation of that configuration variable’s
configuration variable data type. If the given value string is not suitable for that data type, display an error message.
Otherwise, set the configuration variable to that new value, truncating it to fit the available space if necessary, and then
display the output text representation of that configuration variable’s value.
  
  See Also: setenv, 7.4.4.1.
  
  Used as: " new-value" " nv-name" $setenv

set-font
  ( addr width height advance min-char #glyphs -- ) F 0x16B
  Set the current font as specified.
  
  The font is used by the “fb1” and “fb8” frame-buffer support packages.
  
  set char-height to height, char-width to width, and fontbytes to advance. Configure the subsequent behavior
  of >font to access the font described by the arguments, according to the font data structure specified below.
  
  The font is a fixed width monochrome character font. The glyphs in the font are represented as follows: A glyph is
  represented by a series of horizontal scan line images, from top to bottom. The sets of scan line images representing the
glyphs for successive characters are placed one after the other in character order.
  
  addr is the address of the font. width is the glyph width, in pixels. height is the glyph height, in scan lines.
  advance is the distance in bytes between the successive scan lines of a glyph. Typically, it is (((width + 15)/16) * 2), i.e.,
  the storage area for a scan line is padded out to a doublet boundary.
  
  min-char is the character number of the first glyph in the font.
  
  #glyphs is the number of glyphs represented in the font.
  
  The number of scan lines images in each glyph is one less than the height. The missing scan line image is assumed to
  contain all zeroes. The top scan line image for each glyph must contain all zeroes.

set-token
  ( xt immediate? fcode# -- ) F 0xDB
  Assign FCode number to existing function.
  
  Assign the FCode number fcode# to the FCode function whose execution token is xt, with compilation behavior specified
  by immediate? as follows. If immediate? is zero, then the FCode evaluator will execute the function's execution semantics
  if it encounters that FCode number while in interpretation state, or append those execution semantics to the current
  definition if it encounters that FCode number while in compilation state. If immediate? is nonzero, the FCode evaluator
  will execute the functions’s FCode evaluation semantics anytime it encounters that FCode number.

show-devs
  ( "[device-specifier]<eol>" -- )
  Show all devices beneath the indicated node.
  
  Skip leading space delimiters. Parse device-specifier delimited by a space. Discard the remainder of the command line.
  Show the full device path for each device in the subtree of the device tree underneath the specified node. The search
  process by which the specified node is located is as defined in 4.3, using the rules given for find-device. If device-
  specifier is the empty string (i.e., there is nothing on the command line following show-devs), show all system devices.
  
  Used as: ok show-devs device-alias <eol>
**showstack**

```
showstack
```

Turn on automatic stack display.

Display entire stack, with a format similar to the `.s` command, just before each `ok` prompt.

This feature is be turned off with the `noshowstack` command. The system default is `noshowstack`.

**Typical implementation:** `[''] .s is status`

See: `noshowstack`.

**$sift**

```
$sift
```

Display all command-names containing `text-string`.

Search the current vocabulary and display the names of all commands which include the specified `text-string` as part of the command-name. Upper and lowercase distinctions are ignored. This command is useful for finding all commands of a particular "type", or for finding any command where the name is only partially known.

**Used as:** `" ing" $sift`

**sifting**

```
sifting
```

Display all command-names containing `text`.

**Used as:** `ok sifting text<space>`

See: `$sift` for more information.

**sign**

```
sign
```

If `n < 0`, insert “-” in pictured numeric output.

```
\[ A,F \quad 0x98 \]
```

**#size-cells**

Standard property name to define the package’s address size format.

```
prop-encoded-array: Number encoded as with `encode-int`.
```

This property applies to bus nodes. The property value specifies the number of cells that are used to encode the size field of a child’s “reg” format property. A missing “#size-cells” property signifies the default value of one. Plug-in devices shall use the value specified for that bus, and if unspecified, shall use the default value of one.

For a given bus, the value of this property should be the same on all machines for which that bus could possibly be used, even if those machines do not all have the same cell size. Consequently, the value of the property is determined in part by the smallest cell size among all the machines to which the bus can apply.

**sm/rem**

```
sm/rem
```

Divide `d` by `n`, symmetric division.

```
\[ A \]
```

**source**

```
source
```

Return the location and size of the input buffer.

```
\[ A \]
```

**space**

```
space
```

Display a single space.

**Tokenizer equivalent:** `bl emit`

**spaces**

```
spaces
```

Display `cnt` spaces.

**Tokenizer equivalent:** `0 max 0 ?do space loop`

**span**

```
span
```

Variable containing number of characters received by `expect`.

```
\[ A,F \quad 0x88 \]
```
**start0**  
( -- )  
F 0xF0

Begin program with *spread 0*. Followed by *FCode-header*.

Set the *spread* value to zero, thus causing the FCode evaluator to read all bytes of the current FCode program from the same address. Set the size of *FCode-offsets* to 16 bits. Read an *FCode-header* from the current FCode program and either discard it or use it to verify the integrity of the current FCode program in an implementation-dependent manner.

*FCODE ONLY* (the first byte of an FCode program)

**start1**  
( -- )  
F 0xF1

Begin program with *spread 1*. Followed by *FCode-header*.

Set the *spread* value to one, thus causing the FCode evaluator to read bytes of the current FCode program from locations one address unit apart. Set the size of *FCode-offsets* to 16 bits. Read an *FCode-header* from the current FCode program and either discard it or use it to verify the integrity of the current FCode program in an implementation-dependent manner.

*FCODE ONLY* (the first byte of an FCode program)

**start2**  
( -- )  
F 0xF2

Begin program with *spread 2*. Followed by *FCode-header*.

Set the *spread* value to two, thus causing the FCode evaluator to read bytes of the current FCode program from locations two address unit apart. Set the size of *FCode-offsets* to 16 bits. Read an *FCode-header* from the current FCode program and either discard it or use it to verify the integrity of the current FCode program in an implementation-dependent manner.

*FCODE ONLY* (the first byte of an FCode program)

**start4**  
( -- )  
F 0xF3

Begin program with *spread 4*. Followed by *FCode-header*.

Set the *spread* value to four, thus causing the FCode evaluator to read bytes of the current FCode program from locations four address unit apart. Set the size of *FCode-offsets* to 16 bits. Read an *FCode-header* from the current FCode program and either discard it or use it to verify the integrity of the current FCode program in an implementation-dependent manner.

*FCODE ONLY* (the first byte of an FCode program)

**state**  
( -- a-addr )  
A,F 0xDC

*variable* containing *true* if in compilation state.

**state-valid**  
( -- a-addr )  
*variable* containing *true* if *saved-program-state* is valid.

Contains *true* if *saved-program-state* is valid, *false* otherwise. Set to *true* by the *init-program* command and by actions that result in the saving of program state.

*saved-program-state* must be valid in order for execution with *go* to perform properly.

**status**  
( -- )

*defer* word that can be used to modify the user interface prompt.

This is a *defer* word, initially vectored to *noop*, which can be used to display whatever additional information the user wishes to see with each prompt.

**Example:**

The following example illustrates a way to show the current numeric base, in parentheses, on the prompt line.

```plaintext
: showbase ( -- ) ." ( " base $ .d ." ) " ;
['] showbase to status
```

To return to the default condition use the following:

```plaintext
['] noop to status
```
"status" S
Standard property name to indicate the operational status of this device.

prop-encoded-array:
Text string, encoded with encode-string.
If this property is present, the value is a string indicating the status of the device, as follows:

"okay"
The device is believed to be operational.

"disabled"
The device represented by this node is not operational, but it might become operational in the future (e.g., an external switch is turned off, or something isn’t plugged in.)

"fail"
The device represented by this node is not operational because a fault has been detected, and it is unlikely that the device will become operational without repair. No additional failure details are available.

"fail-xxx"
The device represented by this node is not operational because a fault has been detected, and it is unlikely that the device will become operational without repair. "xxx" is additional human-readable information about the particular fault condition that was detected.

The absence of this property means that the operational status is unknown or okay.

stdin ( -- a-addr )
variable containing the ihandle of the console input device.

"stdin" S
Standard property name containing the ihandle of the console input device.

prop-encoded-array:
Integer, encoded with encode-int.
This property appears in the /chosen node.

stdout ( -- a-addr )
variable containing the ihandle of the console output device.

"stdout" S
Standard property name containing the ihandle of the console output device.

prop-encoded-array:
Integer, encoded with encode-int.
This property appears in the /chosen node.

step ( -- )
Executes a single machine-code instruction.
Resume client program execution as with go, but only execute one instruction. The effect is as if breakpoints were established at the possible successors to that instruction and then automatically removed when the breakpoint is handled.

.step ( -- )
Action performed when a single step occurs.
Execute this command whenever a single step occurs. The default behavior is the .instruction command.
.step is a defer command, alterable with the to command. For example, to display registers at every single step.
Use as: [''] .registers to .step
stepping ( -- )
Set “step mode” (default) for Forth source-level debugging.
This mode allows interactive step-by-step execution of the command being debugged. “Step mode” is the default.
While in “step mode”, before the execution of each command called by the debugged command, prompt the user for one of a number of keystrokes. See 7.5.3.4 for a list of these keystrokes.
See: debug for more information.

steps ( n -- )
Execute step n times.

struct ( -- 0 ) T
Start a struct...field definition.
Initialize a structure definition, for use with field commands. Leave a zero on the stack to define the initial offset.
Tokenizer equivalent: 0
See: field for more information.

suppress-banner ( -- )
Abbreviate system start-up sequence in the script.
If executed within the script, suppress automatic execution of the following Open Firmware start-up sequence:
probe-all install-console banner
supress-banner is useful for creating custom banners with commands in the script, as it suppresses the default system banner.
See also: banner.

suspend-fcode ( -- ) F 0x215
Pause FCode evaluation if desired; can resume later.
Advise the FCode evaluator that the device identification properties for the active package have been declared and that it is safe to postpone the evaluation of the remainder of the package.
If the FCode evaluator chooses to postpone (suspend) evaluation, it saves the state of the evaluation process necessary for later resumption of the process.
Usage restriction: “name”, “reg” and “device_type” properties must exist in the active package before this command is executed.
This feature is intended to save memory space and reduce the system start-up time by preventing the compilation of FCode drivers that are not actually used.

swap ( x1 x2 -- x2 x1 ) A,F 0x49
Exchange top two stack items.

2swap ( x1 x2 x3 x4 -- x3 x4 x1 x2 ) A,F 0x55
Exchange top two pairs of stack items.

sym ( "name<>" -- n )
Return value of client program symbol “name”.
Parse name delimited by a space. If sym>value returns false, perform the function of abort. Otherwise, return the symbol value n corresponding to that symbol name.
sym>value ( addr len -- addr len false | n true )
defer word to resolve symbol names.

This defer command is executed when the symbolic debugger needs to translate a symbol name into its corresponding value. If sym>value is present, the Forth interpreter attempts to perform such translation if a word is neither found in the normal dictionary search nor recognized as number. The translation is also attempted by sym.

If a symbol whose name matches the string given by addr len is defined, sym>value returns the corresponding symbol value and true. Otherwise, sym>value returns its addr len arguments and false.

The default action for sym>value, when no symbol table is present, is the action of false. A program can provide a symbol table and use to to install a command to search that symbol table into sym>value.

test ( "device-specifier<eol>" -- )
Invoke the selftest routine for the specified device.

If the device node specified by device-specifier has a selftest method, invoke it with execute-device-method. Otherwise display an error message.

NOTE—The self-test routine that is executed might display device-specific error messages.

Used as: ok test device-alias <eol>

test-all ( "{device-specifier}<eol>" -- )
Invoke selftest routines at and below specified node.

For each node in the subtree of the device tree at and below the specified node, or the root node if no node is specified: If the node has a selftest method, invoke it with execute-device-method.

NOTE—The selftest routines that are executed might display device-specific error messages.

The system may choose not to test certain active devices that it believes are “unsafe” to test while active.

Used as: ok test-all device-alias <eol>

then (C: orig-sys -- ) A,T

Terminate an if statement.

Compilation:

Perform the compilation semantics of ANS Forth THEN. Then, if the current definition is temporary and the depth of the control flow stack is the same as its depth when the temporary current definition was initiated, perform the compilation semantics of ; and execute the temporary current definition.

Run-time:

Same as ANS Forth.

ANS Forth note: Also works outside of a definition.

throw ( ... error-code -- ??? error-code | ... ) A,F 0x218

Transfer back to catch routine if error-code is nonzero.

The value of my-self shall be restored from the exception frame.

ANS Forth note: also restores my-self.

till ( addr -- )

Execute until the given address.

Equivalent to: +bp go
Change `value` or `defer` or machine register contents.

**Tokenizer equivalent:** `b(to) old-FCode#`

**ANS Forth note:** `to` can be used with other word types (`defer` words and machine register names) in addition to those specified in ANS Forth.

**Used as:** `55 to #lines`

`toggle-cursor` is one of the `defer` words of the display device interface. The terminal emulator package executes `toggle-cursor` when it is about to process a character sequence that might involve screen drawing activity, and executes it again after it has finished processing that sequence. The first execution removes the cursor from the screen so that any screen drawing will not interfere with the cursor, and the second execution restores the cursor, possibly at a new position, after the drawing activity related to that character sequence is finished. `toggle-cursor` is also called once during the terminal emulator initialization sequence.

Any standard package that uses the terminal emulator package shall, as part of the process of opening the terminal emulator package, set this `defer` word to a function with the following behavior:

- If the text cursor is on, turn it off. If the text cursor is off, turn it on. (On a bit-mapped display, a typical implementation of this function inverts the pixels of the character cell to the right of the current cursor position.)
- If the display device hardware has internal state (for example color map settings) that might have been changed by external software without firmware’s knowledge, that hardware state should be re-established to the state that the firmware device driver requires when the cursor is toggled to the off state (which indicates that firmware drawing operations are about to begin). This situation can occur, for example, when an operating system is using a display device, but that operating system uses firmware text output services from time to time, e.g., for critical warning messages.

See also: `to`, `fb8-install`.

Set “trace mode” for Forth source-level debugging.

This mode causes execution of the word being debugged to be traced, showing the name and stack contents for each command called by the debugged command.

Continue tracing until `stepping` is executed or a system reset takes place.

See: `debug` for more information.

Remove trailing spaces from string.

Translate virtual address to physical address.

If a valid virtual to physical translation exists for the virtual address `virt`, return the physical address `phys.lo ... phys.hi`, the translation mode `mode`, and `true`. Otherwise return `false`. The physical address format is the same as that of the “memory” node (the node whose `ihandle` is given by the value of `/chosen`’s “memory” property). The interpretation of `mode` is implementation dependent.

Return the value `true` (negative one).

`tuck` Copy top stack item underneath the second stack item.

Display `text-len` characters beginning at address `text-str`.
u# ( u1 -- u2 ) F 0x99

Convert a digit in pictured numeric output conversion.
Divide \textit{u1} by \textit{base}. Leave the quotient on the stack as \textit{u2}. Convert the remainder digit to a printable character representation and add it to the text string with the \texttt{hold} command.
See: (.) and (u.) for examples of use.

u#> ( u -- str len ) F 0x97

End pictured numeric output conversion.
Leave the text string on the stack, suitable for use with \texttt{type}.
See: (.) and (u.) for examples of use.

u#s ( u1 -- u2 ) F 0x9A

Convert remaining digits in pictured numeric output.
Repeat the \texttt{u#} operation until the quotient is zero.
See: (.) and (u.) for examples of use.

u* ( u1 u2 -- uprod )

Multiply \textit{u1} by \textit{u2} yielding \textit{uprod}, all unsigned.

u. ( u -- ) A,F 0x9B

Display an unsigned number, with a trailing space.

u< ( u1 u2 -- unsigned-less? ) A,F 0x40

Return \texttt{true} if \textit{u1} is less than \textit{u2}, unsigned.

u<= ( u1 u2 -- unsigned-less-or-equal? ) F 0x3F

Return \texttt{true} if \textit{u1} less or equal to \textit{u2}, unsigned.

u> ( u1 u2 -- unsigned-greater? ) A,F 0x3E

Return \texttt{true} if \textit{u1} is greater than \textit{u2}, unsigned.

u>= ( u1 u2 -- unsigned-greater-or-equal? ) F 0x41

Return \texttt{true} if \textit{u1} greater or equal to \textit{u2}, unsigned.

(u.) ( u -- str len ) T

Convert an unsigned number into a text string.
Perform the conversion according to the value in \textit{base}.
Tokenizer equivalent: \langle u#s u# \rangle

u2/ ( x1 -- x2 ) F 0x58

Shift \textit{x1} right by one bit-place. Zero-fill high bit.

um* ( u1 u2 -- ud.prod ) A,F 0xD4

Unsigned multiply with unsigned double-number product.

um/mod ( ud u -- urem uquot ) A,F 0xD5

Divide \textit{ud} by \textit{u}. 
u/mod ( u1 u2 - urem uquot ) F 0x2B

Divide u1 by u2, both unsigned.

unaligned-l! ( quad addr -- )

Store quadlet to addr, any alignment is allowed.

unaligned-l@ ( addr -- quad )

Fetch quadlet from addr, any alignment is allowed.

unaligned-w! ( w addr -- )

Store doublet w to addr, any alignment is allowed.

unaligned-w@ ( addr -- w )

Fetch doublet w from addr, any alignment is allowed.

unloop ( -- ) (R: sys -- ) A,F 0x89

Discard loop control parameters.

ANS Forth note: Also works outside of a definition.

unmap ( virt len ... -- ) M

Invalidate existing address translation.

Invalidate any existing address translation for the region of virtual address space beginning at virt and continuing for len ... (whose format depends on the package) bytes. unmap does not free either the virtual address space (as with the release standard method) or any physical memory that was associated with virt.

If the operation fails for any reason, uses throw to signal the error.

until (C: dest-sys -- ) A,T

( done? -- )

End a begin...until loop. Exit loop if flag is nonzero.

Compilation:

(C: dest-sys -- )

Perform the compilation semantics of ANS Forth UNTIL. Then, if the current definition is temporary and the depth of the control flow stack is the same as its depth when the temporary current definition was initiated, perform the compilation semantics of ; and execute the temporary current definition.

Run-time:

( done? -- )

Same as ANS Forth.

Tokenizer equivalent: b?branch -offset

ANS Forth note: Also works outside of a definition.

upc ( char1 -- char2 ) F 0x81

Convert ASCII char1 to uppercase.

Convert input values between 0x61 and 0x7A (ASCII a-z) to 0x41 through 0x5A (ASCII A-Z). All other input values are unchanged.

u.r ( u size -- ) A,F 0x9C

Display an unsigned number, right-justified.

use-nvramrc? ( -- enabled? )

If true, the script is evaluated at system start-up.

If false, the script is not evaluated.

Configuration variable type: Boolean. Suggested default value: false.
IEEE Std 1275-1994  
IEEE STANDARD FOR BOOT (INITIALIZATION CONFIGURATION) FIRMWARE:

**user-abort**  
\((...--)\)  
\((R:--)\)  
\(F\)  
\(0x219\)

After alarm routine is finished, abort program execution.

Used within an alarm routine to signify that the user has typed an abort sequence. When the alarm routine finishes, instead of returning to the program that was interrupted by the execution of the alarm routine, enter the command interpreter by calling the **abort** command.

**value**  
\((E:--x)\)  
\(A,T\)  
\((x"new-name<>"--)\)

Create a named variable, change with **to**.

*Tokenizer equivalent: new-token|named-token|external-token b(value)*

*ANS Forth/tokenizer difference: In FCode source, value cannot appear inside a colon definition.*

**value>sym**  
\((n1--false|n2 addr len true)\)

Defer word to resolve symbol values.

This **defer** command is executed when the symbolic debugger needs to translate a symbol value into its corresponding name. If **value>sym** is present, the disassembler attempts to perform such translation to display symbolic representations of the addresses that it displays. The translation is also attempted by **.adr**.

If the symbol table contains a symbol whose value is sufficiently close to, but not greater than, the value \(n1\), **value>sym** returns the string \(addr\ len\) representing the name of that symbol, the non-negative difference \(n2\) between the symbol value and \(n1\), and **true**. Otherwise, **value>sym** returns its \(n1\) argument and **false**.

The default action for **value>sym**, when no symbol table is present, is the action of **false**. A program can provide a symbol table and use **to** to install a command to search that symbol table into **value>sym**.

**variable**  
\((E:--a-addr)\)  
\(A,T\)  
\("new-name<>"--)\)

Create a named variable. **new-name** returns address \(a-addr\).

*Tokenizer equivalent: new-token|named-token|external-token b(variable)*

*ANS Forth/tokenizer difference: In FCode source, variable cannot appear inside a colon definition.*

**version1**  
\(--\)  
\(F\)  
\(0xFD\)

Begin program with **spread 1**. Followed by **FCode-header**.

Set the **spread** value to one, thus causing the FCode Evaluator to read bytes of the current FCode program from locations one address unit apart. Set the size of **FCode-offsets** to 8 bits. Read an **FCode-header** from the current FCode program and either discard it or use it to verify the integrity of the current FCode program in an implementation-dependent manner.

**FCODE ONLY**

\(w!\)  
\((w waddr--)\)  
\(F\)  
\(0x74\)

Store doublet \(w\) to \(waddr\).

See: **rw!**

\(w,\)  
\((w--)\)  
\(F\)  
\(0xD1\)

Compile a doublet \(w\) into the dictionary (doublet-aligned).

Allocate two bytes (with **allot**) at the current top of the dictionary and store the value \(w\) into that space. The dictionary pointer must have been doublet-aligned.

\(w@\)  
\((waddr--w)\)  
\(F\)  
\(0x6F\)

Fetch doublet \(w\) from \(waddr\).

See: **rw@**

\(/w\)  
\(--n\)  
\(F\)  
\(0x5B\)

The number of address units to a doublet: typically, two.
Multiply \( nu1 \) by the value of \( /w \).

Fetch doublet from \( waddr \), sign-extended.

Increment \( addr1 \) by \( index \) times the value of \( /w \).

Increment \( addr1 \) by the value of \( /w \).

Swap the bytes within a doublet.

The high bits of the input doublet must be zero.

Swap the bytes within each doublet in the given region.

The region begins at \( waddr \) and spans \( len \) bytes. The behavior is undefined if \( len \) is not a multiple of \( /w \).

Split a doublet \( w \) into two bytes.

The high bits of each of the two bytes are zero.

Mark first clause of a \texttt{begin...while...repeat} loop.

\texttt{while} is a value that is used by the \texttt{fbl} and \texttt{fb8} frame-buffer support packages. It denotes the width in pixels of the border at the left of the screen, before the first pixel that is part of the text region.

\texttt{window-left} is set automatically by the execution of either \texttt{fbl-install} or \texttt{fb8-install}, so as to center the text region on the screen. A standard package that uses one of those frame-buffer support packages can subsequently alter its value in order to move the text region to some screen location other than its normal centered position.

\texttt{window-top} is a value that is used by the \texttt{fbl} and \texttt{fb8} frame-buffer support packages. It denotes the number of display scan lines in the border at the top of the screen, before the first scan line that is part of the text region.

\texttt{window-top} is set automatically by the execution of either \texttt{fbl-install} or \texttt{fb8-install}, so as to center the text region on the screen. A standard package that uses one of those frame-buffer support packages can subsequently alter its value in order to move the text region to some screen location other than its normal centered position.

Return \texttt{true} if \( n \) is between \( min \) and \( max-1 \), inclusive.

\texttt{while} (C: dest-sys -- orig-sys dest-sys ) A,T
( continue? -- )

\texttt{Tokenzer equivalent: b?branch +offset}

\texttt{ANS Forth note: Also works outside of a definition.}
wljoin  ( w.lo w.hi -- quad )  F  0x7D
Join two doublets to form a quadlet.
The high bits of each of the two doublets must be zero.

write  ( addr len -- actual )  M
Write memory buffer to device; return actual byte count.
Write len bytes to the device from the memory buffer beginning at addr. Return actual, the number of bytes actually written. If actual is less than len, the write did not succeed.
For some devices, the seek method sets the position for the next write.

write-blocks  ( addr block# #blocks -- #written )  M
Write #blocks records of length block-size bytes from memory (starting at addr) to the device (starting at device block block#). Return #written, the number of blocks actually written.
If the device is not capable of random access (e.g., a sequential access tape device), block# is ignored.

word  ( delim "<delims>text<delims>" -- pstr )  A
Parse text from the input buffer, delimited by delim.

words  ( -- )  A
Display the names of methods or commands.
If there is an active package, display the names of its methods. Otherwise, display an implementation-dependent subset (preferably the entire set) of the globally visible Forth commands. In either case, the order of display is to display more recently defined names before less recently defined names.

wpeek  ( waddr -- false | w true )  F  0x221
Attempt to fetch the doublet w at waddr.
Return the data and true if the access was successful. A false return indicates that a read access error occurred.

w poke  ( w waddr -- okay? )  F  0x224
Attempt to store the doublet w to waddr.
Return true if the access was successful. A false return indicates that a write access error occurred.

xor  ( x1 x2 -- x3 )  A,F  0x25
Return bitwise logical “exclusive-or” of x1 and x2.
Annex B
Open Firmware terminal emulator control sequences
(normative)

B.1 Introduction

The terminal emulator support package implements an ANSI X3.64 terminal (as specified in ANSI X3.64-1979).

The terminal emulator support package shall interpret the command sequences in the required subclause, B.2. It should interpret the command sequences in the recommended subclause, B.3. It may interpret and implement additional implementation-dependent command sequences. It shall ignore syntactically valid ANSI X3.64 escape sequences whose behavior it does not implement. It shall display any printable character that is not part of an escape sequence using draw-character, then shall advance the cursor to the next column as follows:

If the value of column# is less than the value of columns, add one to column#.

Otherwise, perform the functions of “Return” and “Line-feed” as described below.

Notation:

“ESC” represents the “Escape” character (0x1B).
“#” represents an optional numeric parameter.
Other characters represent themselves.

B.2 Required command sequences

The definitions of the following sequences are as given in ANSI X3.64-1979. The terminal emulator state variables that are affected and the display device low-level interfaces that are used to implement each command are listed after the command.

<table>
<thead>
<tr>
<th>Sequence</th>
<th>ANSI X3.64 Mnemonic</th>
<th>Affected Words</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESC[#A</td>
<td>Cursor up</td>
<td>Affects: line#</td>
</tr>
<tr>
<td>ESC[#B</td>
<td>Cursor down</td>
<td>Affects: line#</td>
</tr>
<tr>
<td>ESC[#C</td>
<td>Cursor forward</td>
<td>Affects: column#</td>
</tr>
<tr>
<td>ESC[#D</td>
<td>Cursor backward</td>
<td>Affects: column#</td>
</tr>
<tr>
<td>ESC[#E</td>
<td>Cursor next line</td>
<td>Affects: line#</td>
</tr>
<tr>
<td>ESC[#1;#2f</td>
<td>Cursor position</td>
<td>Affects: line# and column#</td>
</tr>
<tr>
<td>ESC[#1;#2H</td>
<td>Cursor position</td>
<td>Affects: line# and column#</td>
</tr>
<tr>
<td>ESC[J</td>
<td>Erase in display</td>
<td>Uses: delete-characters and delete-lines</td>
</tr>
<tr>
<td>ESC[K</td>
<td>Erase in line</td>
<td>Uses: delete-characters</td>
</tr>
<tr>
<td>ESC[#L</td>
<td>Insert line</td>
<td>Uses: insert-lines</td>
</tr>
<tr>
<td>ESC[#M</td>
<td>Delete line</td>
<td>Uses: delete-lines</td>
</tr>
<tr>
<td>ESC[#8</td>
<td>Insert character</td>
<td>Uses: insert-characters</td>
</tr>
<tr>
<td>ESC[#p</td>
<td>Delete character</td>
<td>Uses: delete-characters</td>
</tr>
<tr>
<td>ESC[#m</td>
<td>Select graphic rendition</td>
<td>Affects: inverse?</td>
</tr>
</tbody>
</table>

* If the numeric parameter is omitted, the default value is zero. At least the following two graphic renditions shall be implemented:
   — 0, which means normal rendition
   — 7, which means negative (reverse) image
If negative image is on, subsequent characters are displayed with the foreground and background colors exchanged (reversed).
The following special characters shall also be defined:

<table>
<thead>
<tr>
<th>Character</th>
<th>Mnemonic</th>
<th>Affected Words</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTRL-G (0x7)</td>
<td>Bell (BEL)</td>
<td>Uses: <em>blink-screen</em> and &quot;ring-bell&quot; method</td>
<td>An audible indicator sounds or a visible indication is given.</td>
</tr>
<tr>
<td>CTRL-H (0x8)</td>
<td>Backspace (BS)</td>
<td>Affects: <em>column#</em></td>
<td>The cursor moves one position to the left on the current line. If it is already at the left edge of the screen, nothing happens.</td>
</tr>
<tr>
<td>CTRL-I (0x9)</td>
<td>Tab (TAB)</td>
<td>Affects: <em>column#</em></td>
<td>The cursor moves right on the current line to the next tab stop. The tab stops are fixed at every multiple of eight columns. If the cursor is already at the right edge of the screen, nothing happens; otherwise, the cursor moves right a minimum of one and a maximum of eight character positions.</td>
</tr>
<tr>
<td>CTRL-J (0xA)</td>
<td>Line-feed (LF)</td>
<td>Affects: <em>line#</em></td>
<td>The cursor moves down one line, remaining at the same character position on the line. If the cursor is already at the bottom line, the screen scrolls up before the cursor is moved down.</td>
</tr>
<tr>
<td>CTRL-K (0xB)</td>
<td>Reverse line-feed (VT)</td>
<td>Affects: <em>line#</em></td>
<td>The cursor moves up one line, remaining at the same character position on the line. If the cursor is already at the top line, nothing happens.</td>
</tr>
<tr>
<td>CTRL-L (0xC)</td>
<td>Form-feed (FF)</td>
<td>Affects: <em>line#</em> and <em>column#</em></td>
<td>The cursor is positioned to the &quot;home&quot; position (upper-left corner) and the entire screen is cleared.</td>
</tr>
<tr>
<td>CTRL-M (0xD)</td>
<td>Return (CR)</td>
<td>Affects: <em>column#</em></td>
<td>The cursor moves to the leftmost character position on the current line.</td>
</tr>
</tbody>
</table>

### B.3 Recommended control sequences

The following escape sequences are not defined by ANSI X3.64. Their descriptions should be as given:

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Mnemonic</th>
<th>Affected Words</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESC[p</td>
<td>Normal text colors</td>
<td>Affects: <code>inverse-screen?</code> and <code>inverse?</code> Uses: <code>invert-screen</code></td>
<td>If <code>inverse-screen?</code> is false, does nothing. If <code>inverse-screen?</code> is true, sets it to false, changes the value of <code>inverse?</code> (to its opposite value (i.e., from true to false or vice versa), and executes <code>invert-screen</code>. The effect of this is to establish the default foreground and background colors for the entire screen.</td>
</tr>
<tr>
<td>ESC[q</td>
<td>Inverse text colors</td>
<td>Affects: <code>inverse-screen?</code> and <code>inverse?</code> Uses: <code>invert-screen</code></td>
<td>If <code>inverse-screen?</code> is true, does nothing. If <code>inverse-screen?</code> is false, sets it to true, changes the value of <code>inverse?</code> (to its opposite value (i.e., from true to false or vice versa), and executes <code>invert-screen</code>. The effect of this is to establish inverted foreground and background colors for the entire screen (i.e., the screen background uses the default foreground color, and vice versa).</td>
</tr>
<tr>
<td>ESC[s</td>
<td>Reset display device</td>
<td>Uses: <code>reset-screen</code></td>
<td>Resets the display device associated with the terminal emulator.</td>
</tr>
</tbody>
</table>
Annex C
The tokenizer

(informative)

C.1 Introduction

This standard precisely defines the behavior of FCode binary, but it does not specify either the FCode text format or the means to convert that text into FCode binary.

Nonetheless, it is useful to have an agreed-upon format for FCode text and the means to convert that text into FCode binary. This annex documents a recommended format for FCode text and a recommended behavior for a tokenizer program to convert that text into FCode binary.

C.2 Recommended tokenizer behavior

The tokenizer program may be implemented on any convenient system of choice. Its input is a standard text file (FCode text). Its output is a binary file (FCode binary) in a format suitable for the development system being used.

FCode text is substantially similar in appearance to normal Forth text. Certain additional tokenizer commands are also recognized. Certain Forth commands are not recognized.

The tokenizer’s behavior is to read words, one at a time, from the FCode text file and take appropriate action. Appropriate actions are as follows:
— If the word read is an existing FCode name (with an assigned FCode#), generate the appropriate FCode# and append that number to the FCode binary file.
— If the word read is a standard tokenizer macro (indicated by a type-code of “T”), generate the appropriate series of FCode#s as specified in the description of the command. Some macros will cause more words to be read from the input FCode text file. These usually have a stack comment including […], i.e., [text<delim>].
— If the word that was read is a tokenizer-only command (these are listed later), perform the appropriate action as specified.
— Otherwise, print an error message that the particular word is not recognized.

Tokenizing behavior continues until the end of the FCode text file is encountered. If there were no errors, the FCode binary file is created.

Standard tokenizer macros (indicated by a type-code of “T”) will generate a stream of FCodes to perform the equivalent function. If the macro is relatively long (more than two to four FCodes) and is used frequently, this could have an adverse effect on the total size of the FCode binary and could make the compiled Forth dictionary larger. To avoid this problem, create a duplicate colon definition in the FCode, as follows:

```
: 3dup 3dup ;
```

C.3 Tokenizer-only commands

The following commands are recognized by the tokenizer and cause special actions to be performed by the tokenizer. In contrast to most other Forth commands, these commands are only meaningful in the context of an FCode text file being “tokenized.”
C.3.1 Manual tokenizer output

These commands allow the user to generate arbitrary sequences of FCode bytes. This is useful, for example, if using an older tokenizer that does not support some new FCode features.

Within “tokenizer-escape” mode, the basic command is `emit-byte` to manually output a specified FCode byte. Some tokenizers may support additional commands as well. For example, a tokenizer written in Forth may allow Forth calculations, do-loops, and even `colon definitions` within “tokenizer-escape” mode, in order to create complex sequences of FCode bytes. Tokenizers written in other languages may choose to support similar extensions.

```
tokenizer[ ( -- ) T

Enter tokenizer-escape mode, allowing manual FCode generation.
Save the current tokenizer numeric conversion radix, and set the radix to sixteen (hexadecimal). Enter tokenizer escape mode so that the tokenizer will interpret all subsequent commands as direct tokenizer commands, not as FCodes. The emit-byte command may be used while in this mode, to output specified FCode byte(s). Other commands may also be used, but these are not specified in this document. No FCode is generated by this command.
Tokenizer-escape mode is exited by ]tokenizer.
FCODE ONLY command.
```

```
]tokenizer ( -- ) T

Exit tokenizer-escape mode, resumes FCode interpretation.
Restore the tokenizer numeric conversion radix to the value saved by tokenizer[ and exit tokenizer escape mode, thus resuming the tokenizer’s normal behavior of converting words of FCode source to corresponding FCode numbers.
No FCode is generated by this command.
See tokenizer[ for more information.
FCODE ONLY command.
```

```
emit-byte ( FCode# -- ) T

Output given FCode#, only in tokenizer-escape mode.
Only valid while in tokenizer-escape mode. Adds the specified FCode# to the FCode program that the tokenizer is creating.
Used as:

tokenizer[ 244 emit-byte ]tokenizer

See: tokenizer[ for more information.
FCODE ONLY command.
```

C.3.2 File inclusion

This command allows FCode text programs to be factored and shared. It behaves similarly to the `#include` statement in the C language. Arbitrary nesting of included files is allowed.

```
fload ( [filename<cr>] -- ) T

Insert the specified file at this point.
Used as:
fload filename <cr>

Save the specification of the current FCode text file. Begin tokenizing the FCode text file specified by filename. When the specified file is exhausted, restore the saved file and resume tokenizing it.
fload commands may be nested arbitrarily. In other words, the file just loaded may contain its own `fload` commands, as well as normal FCode text commands.
The behavior of `fload` inside a definition in FCode source text is unspecified.
The syntax for `filename` is dependent on the system on which the tokenizer is running.
FCODE ONLY command.
```
Annex D

Sun4c™ bus specifics

(informative)

D.1 Overview and references

This annex provides an example of the application of the Open Firmware specification to a particular system architecture. The structure of this annex is typical of the structure of documents that “bind” the Open Firmware standard to particular buses, and can be used as a template for other such documents.

The architecture described in this annex is the Sun4c system architecture, which is the first system on which OpenBoot, the ancestor of Open Firmware, was deployed. The Sun4c system architecture, designed for use with SPARC microprocessors, defines a memory management unit (MMU), its associated physical address format, and a set of I/O devices to support a uniprocessor multitasking operating system with demand-paged virtual memory.

D.1.1 Definitions of terms

bus node: A device node that represents the characteristics of a Sun4c bus.

child node: A device node that represents a device attached to a Sun4c bus.

D.1.2 References

The characteristics of the Sun4c system architecture can be inferred from the specifications of chip sets implementing that architecture. The aspects of the Sun4c architecture that are relevant to this annex are summarized within this annex.

D.2 Bus characteristics

D.2.1 Physical address formats and representations

The main physical address bus on Sun4c machines consists of a 1-bit type field, representing either memory space (RAM) or I/O space (SBus [B2] and other devices), and a 30-bit offset. Conventionally, this 30-bit range is considered to be centered around zero, so the offset values range from zero to 0x1FFF.FFFF and from 0xE000.0000 to 0xFFFF.FFFF.

This physical address is represented numerically by the type number (zero for memory, one for I/O) in the phys.hi number and the offset in the phys.lo number. The text string representation is “t,offset”, where “t” is either the character “0” or the character “1”, and “offset” is the ASCII representation of a hexadecimal number.

D.2.2 Bus-specific configuration variables

None.

---

™ Sun4c is a trademark of Sun Microsystems, Inc.

9 Such chip sets are available from LSI Logic Corporation and perhaps from other vendors.
D.2.3 Format of a probe list

None. This bus cannot accept plug-in devices. However, most Sun4c implementations have an SBus [B2] subordinate to the root node.

D.2.4 Interrupt specification format

The Sun4c architecture uses the SPARC processor’s interrupts directly; thus, interrupts are specified as for that processor. Specifically, interrupt priority levels range from one to fifteen and are not vectored.

D.2.5 FCode interpretation semantics

None. This bus cannot accept plug-in devices.

D.3 Bus nodes

In Sun4c systems, the Sun4c bus node is the root node of the device tree, since the Sun4c bus is connected directly to the MMU, and thus is the main physical address bus of the machine. In the earliest OpenBoot implementations, the root node of the device tree represented not only the main physical address bus but also the characteristics of the CPU and its associated MMU (if any). Subsequent experience shows that it is better to represent the CPU and MMU characteristics in separate nodes subordinate to the root node. Representing CPU information in separate nodes allows individual description of the various CPUs of a multiprocess system and provides a logical separation between the bus information and the CPU information.

This annex is historically accurate, representing the CPU characteristics in the root node, but that should not be construed as a recommendation for that technique.

D.3.1 Properties

D.3.1.1 Open Firmware-defined properties for bus nodes

Since the Sun4c bus node is the root node of the device tree and thus has no parent address space, there is neither a “reg” property nor a “ranges” property.

The following standard properties, as defined in Open Firmware, have special meanings or interpretations for Sun4c.

“device_type”

Standard property name to specify the implemented interface.

prop-encoded array:

Text string, encoded with encode-string.

The historical value for this property for Sun4c machines is the string “cpu”, indicating that the node in question represents not only the Sun4c physical address bus, but also the characteristics of the main system CPU. This double-use of the node is an historical accident and is not recommended as a precedent for the future.
D.3.1.2 Bus specific properties for bus nodes

**mips-off**

_property name_ to specify the CPU performance.

_prop-encoded-array:_

   Integer, encoded with _encode-int_.

The value of this property indicates the approximate speed of the processor, in millions of instructions per second, when the cache is turned off. Its primary intended use is for calculating the number of iterations needed by short time-delay loops.

**mips-on**

_property name_ to specify the CPU performance.

_prop-encoded-array:_

   Integer, encoded with _encode-int_.

The value of this property indicates the approximate speed of the processor, in millions of instructions per second, when the cache is turned on. Its primary intended use is for calculating the number of iterations needed by short time-delay loops.

**mmu-nctx**

_property name_ to specify the number of MMU contexts.

_prop-encoded-array:_

   Integer, encoded with _encode-int_.

The value of this property indicates the number of contexts implemented by the system MMU.

**mmu-npmg**

_property name_ to specify the number of MMU PMEGs.

_prop-encoded-array:_

   Integer, encoded with _encode-int_.

The value of this property indicates the number of Page Map Entry Groups implemented by the system MMU.

**vac-hwflush**

_property name_ to indicate the presence of cache-flushing hardware.

_prop-encoded-array:_

   None; the information is conveyed by the presence or absence of the property.

The presence of this property indicates that the virtually addressed cache has hardware support for cache flushing. The absence of this property indicates that cache flushing must be done with low-level software operations.

**vac-linesize**

_property name_ to indicate the cache line size.

_prop-encoded-array:_

   Integer, encoded with _encode-int_.

The value of this property is the number of bytes in each line of the virtually addressed cache.

**busmaster-regval**

_property name_ to specify the Ethernet chip hardware configuration.

_prop-encoded-array:_

   Integer, encoded with _encode-int_.

The value of this property is the value that should be written to the CSR3 register of the AMD 7990 Ethernet chip that is a standard part of the Sun4c system architecture. A typical value is 0x07, indicating that the BSWP, ACON, and BCON bits of that register should be set to ones.
**busrerr-type**

*property name* to specify the type of error registers.

*prop-encoded-array:*

- Integer, encoded with `encode-int`.

If this property is absent, or if it is present with a value of zero, the system bus error register layout is that of a SPARCstation-1 machine. If the property is present with a value of one, the system bus error register layout is that of a SPARCstation-2 machine.

**idprom**

*property name* to specify the IDPROM contents.

*prop-encoded-array:*

- Byte array, encoded with `encode-bytes`.

The 32-byte value of this property is the verbatim contents of the Sun IDPROM structure, which contains the machine’s serial number, Ethernet address, and other information.

**clock-frequency**

*property name* to specify the CPU clock frequency.

*prop-encoded-array:*

- Integer, encoded with `encode-int`.

The value of this property is the frequency in megahertz of the CPU’s external clock.

**get-unum**

*property name* to specify the address of an address-to-name translation routine.

*prop-encoded-array:*

- Integer, encoded with `encode-int`.

The value of this property is the virtual address of a subroutine that translates a Sun4c physical address to a human-readable null-terminated string telling the location on the CPU board of the field-replacable part that contains that address. The subroutine is called with standard SPARC subroutine calling conventions. The primary use of the subroutine is to help the operating system display the location of a defective memory module.

**NOTE—**The use of a property to report the address of a machine language subroutine is not a recommended technique. The register usage and calling conventions for such subroutines often varies from compiler to compiler, thus the use of machine language subroutines creates a dependency on a particular compiler.

### D.3.2 Methods

#### D.3.2.1 Open Firmware-defined methods for bus nodes

Sun4c bus nodes implement the following standard *methods* as defined in Open Firmware, with physical address representations as specified in D.2.1.

- **decode-unit** `(addr len -- phys.lo ... phys.hi)` Convert text unit-string to physical address.
- **map-out** `(virt size --)` Destroy mapping from previous **map-in**.
- **map-in** `(phys.lo ... phys.hi size -- virt)` Map the specified region; return a virtual address.
- **close** `( -- )` Close this previously *opened* device.
- **open** `( -- okay? )` Prepare this device for subsequent use.
D.3.2.2 Bus-specific methods for bus nodes

\texttt{decode-space (addr len -- phys.hi)} \quad M

Convert text string to \textit{phys.hi}.

Convert the address space named by the string \texttt{addr,len} to the numerical representation of the high component of the physical address \textit{phys.hi}. Either of the strings \texttt{"mem"} or \texttt{"obmem"} is converted to the numerical value 0, and either of the strings \texttt{"io"} or \texttt{"obio"} is converted to the numerical value 1.

D.4 Child nodes

D.4.1 Properties

D.4.1.1 Open Firmware-defined properties for child nodes

The following standard properties, as defined in Open Firmware, apply to Sun4c child nodes, with physical address representations and interrupt formats as specified in D.2.1.

\begin{itemize}
  \item \texttt{"reg"} \quad \textit{Standard property name} to define the package’s registers.
  \item \texttt{"interrupts"} \quad \textit{Standard property name} to define the interrupts used.
\end{itemize}

D.4.1.2 Bus-specific properties for child nodes

None.

D.4.2 Methods

There are no special requirements on the \textit{methods} of Sun4c child nodes beyond those described in Open Firmware.

D.5 User interface extensions

This section describes commands that are only applicable to Sun4c systems or have special meaning when used with Sun4c systems.

D.5.1 Bus-specific interpretations of standard commands

None.

D.5.2 Bus-specific FCodes

None.

D.5.3 Bus-specific FCodes

An Open Firmware-compliant user interface on a Sun4c system should implement the following Sun4c-specific user interface commands.

\texttt{obio ( -- space )}

\begin{itemize}
  \item \textit{The on-board I/O address space; for mapping.}
\end{itemize}
obmem ( -- space )
The on-board memory address space; for mapping.

sbus ( -- space )
The SBus address space; for mapping.

pgmap@ ( virt -- pme )
The page map entry pme corresponds to the virtual address virt.

pgmap? ( virt -- )
Display the page map entry corresponding to the virtual address virt.

pgmap! ( pme virt -- )
Store a page map entry pme for the virtual address virt.

smap@ ( virt -- pmeg )
The segment map entry pmeg corresponds to the virtual address virt.

smap? ( virt -- )
Display the segment map entry for the virtual address virt.

smap! ( pmeg virt -- )
Store a segment map entry for the virtual address virt.

map? ( virt -- )
Display the multi-level MMU mappings corresponding to the virtual address virt.

cache-off ( -- )
Disable the CPU cache.

cache-on ( -- )
Enable the CPU cache.

cacheable ( space -- cache-space )
Adjust the address space space so the subsequent address mapping is cacheable.

cdata@ ( offset -- data )
data is at offset from the start of the CPU cache.

cdata! ( data offset -- )
Store data at offset from the start of the CPU cache.

clear-cache ( -- )
Invalidate all the entries in the cache.

cetag@ ( offset -- value )
The cache tag value is at offset from the beginning of the CPU cache.

cetag! ( value offset -- )
Store value at the offset from the start of the CPU cache.
aerr! ( data -- )
Write the asynchronous error register.

aerr@ ( -- data )
Read the asynchronous error register.

averr! ( data -- )
Write the asynchronous virtual address register.

averr@ ( -- data )
Read the asynchronous virtual address register.

aux! ( data -- )
Write the auxiliary register.

aux@ ( -- data )
Read the auxiliary register.

context! ( data -- )
Write the context register.

context@ ( -- data )
Read the MMU context register.

dcontext@ ( -- data )
Read the cache context register.

enable! ( data -- )
Write the system enable register.

enable@ ( -- data )
Read the system enable register.

idprom@ ( offset -- data )
Read the idprom byte at offset from the start of the idprom.

interrupt-enable! ( data -- )
Write the interrupt enable register.

interrupt-enable@ ( -- data )
Read the interrupt enable register.

serr! ( data -- )
Write the synchronous error register.

serr@ ( -- data )
Read the synchronous error register.

sverr! ( data -- )
Write the synchronous error virtual address register.
sverr@ ( -- data )
Read the synchronous error virtual address register.

pagesize ( -- size )
The page size of the CPU MMU.

segmentsize ( -- size )
The segment size of the CPU MMU.

map-page ( phys space virt -- )
Create a mapping for physical address phys space to virtual address virt. Both phys and virt, if not already on a page boundary, are truncated to the next lower page boundary.

map-pages ( phys space virt size -- )
Performs consecutive calls to map-pages to map the range of physical addresses beginning at phys space and extending for size bytes to a range of virtual addresses beginning at virt.

map-segments ( smentry virt size -- )
Map a memory region with smap!.

D.6 Client execution environment

NOTE—The following information would typically be specified in a document describing the application of Open Firmware to a particular system. Such information would usually be considered inappropriate for a document describing the application of Open Firmware to a standard bus. The information is included here because this annex applies to the entire Sun4c system architecture, not just to the Sun4c bus.

When a client begins execution, PMEGs 0xFF, 0xFE, 0xFD, and 0xFC, ... (as needed) are already in use. Only context 0 is used. PMEG 0xFF is used as the invalid PMEG and is filled with invalid PTEs. Unused virtual addresses are invalidated. All unused segments are set to the invalid PMEG. All unused memory pages are set to the invalid PTE.

All memory is scrubbed (either set to 0 or read/written to clean out start-up parity error artifacts). Parity is turned off. All cache tags are cleared before use. Some pages are marked cacheable, and the cache is turned on.

The trap table is set so that a routine to save the state of the CPU is installed in all trap vectors except for window-overflow (trap 0x05), window-underflow (trap 0x06), and interrupt-level-14 (trap 0x1E). The interrupt-control register is set to enable level-14 interrupts for a counter, and to allow any interrupts. The processor-interrupt level is set to 13. The counter-limit value is set to interrupt every 10 ms.
Annex E
SCSI host adapter package class

(informative)

E.1 Overview and references

This annex describes the application of Open Firmware to the Small Computer Systems Interface (SCSI) bus and addresses nodes representing SCSI host adapters.

E.1.1 Definitions of terms

bus node: A device node that represents the interface, or “host adapter,” between a SCSI bus and its parent (which may be another bus).

child node: A device node that represents an SCSI “target” device.

E.1.2 References

SCSI (Small Computer Systems Interface) is a peer-to-peer I/O bus defined by ISO/IEC 10288: . . . [B4], which will supersede ISO/IEC 9316: 1989 when it is approved and published.

E.2 Bus characteristics

E.2.1 Physical address formats and representations

SCSI devices are addressed with a 4-bit “target” number and a 3-bit “unit” number.

The numerical representation on an SCSI bus physical address consists of the target number in the high number and the unit number in the low number. The text string representation is target,unit, where target and unit are both hexadecimal numbers. Future SCSI extensions have proposed target address spaces greater than 4 bits and unit address spaces larger than 3 bits. The given address representation allows for up to 32 bits for each of the target and unit addresses.

SCSI is not a memory-mapped bus. Operations on target devices are performed by executing a transaction consisting of multiple phases, including selecting a particular target device, sending a multibyte command to the target, possibly transferring multiple data bytes to or from the target, and returning status.

E.2.2 Bus-specific configuration variables

None.

E.2.3 Format of a probe list

None.

E.2.4 Interrupt specification format

None (SCSI has no interrupts).
E.2.5 FCode interpretation semantics

None (SCSI has no provision for device identification via FCode.)

E.3 Bus nodes

E.3.1 Properties

E.3.1.1 Open Firmware-defined properties for bus nodes

The following standard property, as defined in Open Firmware, has special meaning or interpretation for SCSI:

“device_type”

Standard prop-name to specify the implemented interface.

The meaning of this property is as defined in Open Firmware. A package conforming to this specification and corresponding to a device that implements an SCSI Bus shall implement this property with the string value “scsi-2”.

E.3.1.2 Bus-specific properties for bus nodes

None.

E.3.2 Methods

E.3.2.1 Open Firmware-defined methods for bus nodes

A package implementing the “scsi-2” device type shall implement the following standard methods as defined in Open Firmware, with physical address representations as specified in E.2.1:

open ( -- okay? ) M

Prepare this device for subsequent use.

close ( -- ) M

Close this previously openned device.

dma-alloc (... size -- virt ) M

Allocate a memory region for later use.

dma-free ( virt size -- ) M

Free memory allocated with dma-alloc.

decode-unit ( addr len -- phys.lo ... phys.hi ) M

Convert text unit-string to physical address.

E.3.2.2 Bus-specific methods for bus nodes

A package implementing the “scsi-2” device type shall implement the following bus-specific methods.

max-transfer ( -- n )

Returns the maximum DMA transfer length supported by the hardware.

set-address ( unit# target# -- )

Sets the SCSI target number (0x0..0xf) and unit number (0..7) to which subsequent commands apply.
**set-timeout** ( msecs -- )

Sets the maximum length of time in milliseconds that the driver will wait for the completion of a command. The default value of zero means to wait indefinitely. A hardware error result is reported for a command that times out.

**show-children** ( -- )

Searches the SCSI bus for attached target devices and their associated units. Displays the information that the SCSI “Inquiry” command reports for those devices.

**execute-command** ( buf-addr buf-len dir cmd-addr cmd-len -- hw-err? | statbyte 0 )

Executes the SCSI command, which is stored in memory at cmd-addr and whose length is cmd-len. Dir is true if the data transfer phase of the SCSI command will transfer data from the device to memory, and false otherwise. buf-addr is the address of the memory buffer to be used for the data transfer phase, and buf-len is the expected maximum length of the data transfer phase. The memory buffer must be contained within a DMA-accessible region that was returned by a previous execution of *dma-alloc*. If buf-len is zero, indicating that the command is not expected to have a data transfer phase, both buf-addr and dir are ignored. Hw-err?, the returned hardware error status, is nonzero if the command could not be executed at all (perhaps due to the device not responding to the selection attempt). If hw-err? is zero, statbyte is the status byte returned by the status phase of the command.

**retry-command** ( buf-addr buf-len dir cmd-addr cmd-len #retries -- 0 | hw-err? stat | sensebuf 0 stat )

Executes a SCSI command, automatically retrying under certain conditions. retry-command is similar to *execute-command* except that retry-command automatically retries under certain failure conditions and automatically executes the “request sense” SCSI command as necessary. #retries is the maximum number of times that the command will be retried; if #retries is –1, the command will be retried indefinitely. retry-command returns 0 if the command eventually succeeds. Otherwise, it returns the status byte returned by the last attempted command on top of the stack (–1 if the command failed due to a hardware error). The second number on the stack (hw-err?) indicates whether or not the extended sense information is available. If hw-err? is zero, the third number on the stack (sensebuf) is the address of a memory buffer containing the extended sense information returned by the “request sense” command that was executed after the last attempt to execute the desired command. The criteria for whether or not to retry the command are as follows:

a) If the requested number of retries have already been performed, do not retry.
b) If the failure is due to a hardware error, do not retry.
c) If the failure was due to a “device busy” condition reported in the status byte, retry.
d) Otherwise, execute the “get extended status” command and attempt to determine whether or not the failure could be retried based on the data in the returned sense buffer, as follows:

1) Unknown error class (not 7) is not retryable.
2) Filemark is not retryable.
3) End of media is not retryable.
4) Illegal length indicator is not retryable.
5) sense key = No Sense is retryable.
6) sense key = Recoverable error is retryable.
7) sense key = Not Ready is retryable.
8) sense key = Unit Attention is retryable.
9) Transaction aborted due to Incoming SCSI Bus reset is retryable
10) Otherwise, the error is not retryable.

**no-data-command** ( cmd-addr -- error? )

Executes a simple SCSI command, automatically retrying under certain conditions. cmd-addr is the address of a 6-byte command buffer containing an SCSI command that does not have a data transfer phase. Executes the command, retrying indefinitely with the same retry criteria as retry-command. error? is nonzero if an error occurred, zero otherwise.

NOTE—no-data-command is a convenience function. It provides no capabilities that are not present in retry-command, but for those commands that meet its restrictions, it is easier to use.
short-data-command ( data-len cmd-addr cmd-len -- error? | data-adr 0 )
Executes a simple SCSI command, automatically retrying under certain conditions.

`cmd-addr` is the address and `cmd-len` the length of a command buffer containing an SCSI command whose data transfer phase is expected to transfer less than 256 bytes in an incoming direction. `data-len` is the expected length (1..255) of the data transfer. Executes the command, retrying indefinitely with the same retry criteria as `retry-command`. `error?` is nonzero if an error occurred, zero otherwise. If `error?` is zero, `data-adr` is the address of a buffer containing the data transferred by the execution of the command.

**NOTE**—`short-data-command` is a convenience function, eliminating the need for allocating a DMA buffer. It is primarily intended for use with “informational” SCSI commands like “read block limits” and “inquiry”.

diagnose ( -- error-code | 0 )
Performs a simple self-test for a generic SCSI device.
Perform an SCSI “test-unit-ready” command on the currently selected target and unit (see `set-address`). If that fails, display a message indicating the details of the failure and return a nonzero error code. Otherwise, perform an SCSI “send-diagnostic” command, returning zero if it succeeds or a nonzero error code if it fails.

### E.4 Child nodes

#### E.4.1 Properties

**E.4.1.1 Open Firmware-defined properties for child nodes**

None.

**E.4.1.2 Bus-specific properties for child nodes**

None.

#### E.4.2 Methods

**E.4.2.1 Open Firmware-defined methods for child nodes**

None.

**E.4.2.2 Bus-specific methods for child nodes**

None.

### E.5 User interface extensions

None.
E.6 Sample driver code

This subclause contains the source code for an FCode program implementing a driver for a hypothetical SCSI host adapter device. This source code can be processed by a tokenizer program that behaves as described in annex C.

E.6.1 overall.fth

\ FCode driver for hypothetical SCSI host adapter

hex

" XYZI,scsi" name \ Name of device node
" XYZI,12346-01" model \ Manufacturer's model number
" scsi-2" device-type \ Device implements SCSI-2 method set
3 0 intr \ Device interrupts on level 3, no vector

external

\ These routines may be called by the children of this device.
\ This card has no local buffer memory for the SCSI device, so it
\ depends on its parent to supply DMA memory. For a device with
\ local buffer memory, these routines would probably allocate from
\ that local memory.

: dma-alloc ( n -- vaddr ) " dma-alloc" $call-parent ;
: dma-free ( vaddr n -- ) " dma-free" $call-parent ;
: dma-sync ( vaddr devaddr n -- ) " dma-sync" $call-parent ;
: dma-map-in ( vaddr n cache? -- devaddr ) " dma-map-in" $call-parent ;
: dma-map-out ( vaddr devaddr n -- ) " dma-map-out" $call-parent ;
: max-transfer ( -- n )
  " max-transfer" ['] $call-parent catch if 2drop h# 7fff.ffff then
  \ The device imposes no size limitations of its own; if it did, those
  \ limitations could be described here, perhaps by executing:
  \ my-max-transfer min
;  

fload scsiha.fth

fload hacom.fth

new-device
  fload scsidisk.fth \ scsidisk.fth also loads scsicom.fth
finish-device

new-device
  fload scsitape.fth \ scsitape.fth also loads scsicom.fth
finish-device

end0

E.6.2 scsiha.fth

\ Example FCode driver for a hypothetical SCSI bus interface device

hex

\ The following structure defines the registers for the SCSI device.
\ This hypothetical device is designed for ease of programming. It
\ has a separate register for each function (no bit packing). All
\ registers are both readable and writeable. The device has a random-
access buffer large enough for a maximum-length SCSI command block.

To execute a SCSI command with this device, write the appropriate information into the registers named ">cmd-adr" through ">input?", write a 1 to the ">start" register, and wait for the ">start" register to change to 0. Then read the ">phase" register to determine whether or not the command completed all phases (">phase" reports 0 on success, h# fd for incoming reset, h# ff for other hardware error).

If so, ">status" contains the SCSI status byte, and ">message-in" contains the command-complete message byte.

struct ( scsi-registers )
0c field >cmd-adr       \ Up to 12 command bytes
 4 field >cmd-len       \ Length of command block
 4 field >data-adr      \ Base address of DMA data area
 4 field >data-len      \ Length of data area
 1 field >host-selectid \ Host's selection ID
 1 field >target-selectid \ Target's selection ID
 1 field >input?       \ 1 for data output; 0 for data input
 1 field >message-out  \ Outgoing message byte
 1 field >start        \ Write 1 to start. Reads as 0 when done.
 1 field >phase        \ Reports the last transaction phase
 1 field >status       \ Returned status byte
 1 field >message-in   \ Incoming message byte
 1 field >intena       \ Write 1 to enable interrupts.
 1 field >reset-bus    \ Write 1 to reset the SCSI bus.
 1 field >reset-board  \ Write 1 to reset the board.
constant /scsi-regs

Now that we have a symbolic name for the size of the register block, we can declare the "reg" property.

Registers begin at offset 800000 and continue for "/scsi-regs" bytes.

my-address 80.0000 + my-space /scsi-regs reg

-1 instance value regs       \ Virtual base address of device registers
0 instance value my-id       \ host adapter's selection ID
0 instance value his-id      \ target's selection ID
0 instance value his-lun     \ target's unit number

Map device registers
: map ( -- )
  my-address 80.0000 + my-space /scsi-regs ( addr-low addr-high size )
  " map-in" $call-parent to regs ()
;
: unmap ( -- )
  regs /scsi-regs " map-out" $call-parent -1 to regs
;
create reset-done-time 0,
create resetting false,

5 seconds appears to be about the right length of time to wait after a reset, considering a variety of disparate devices.
d# 5000 value scsi-reset-delay
: reset-wait ( -- )
  resetting @ if
    begin get-msecs reset-done-time @ - 0>= until
      resetting off then
  ;

: reset-scsi-bus ( -- )
  1 regs >reset-board rb! \ Reset the controller board.
  0 regs >intena rb! \ Turn off interrupts.
  1 regs >reset-bus rb! \ Reset the SCSI bus.

\ After resetting the SCSI bus, we have to give the target devices
\ some time to initialize their microcode. Otherwise the first command
\ may hang, as with some older controllers. We note the time when it
\ is okay to access the bus (now plus some delay), and "execute-command"
\ will delay until that time is reached, if necessary.
\ This allows us to overlap the delay with other work in many cases.

get-msecs scsi-reset-delay + reset-done-time ! resetting on

0 value scsi-time \ Maximum command time in milliseconds
0 value time-limit \ Ending time for command

: set-timeout ( msecs -- ) to scsi-time ;

0 value devaddr
\ Returns true if select failed

: (exec) ( dma-adr,len dir cmd-adr,len -- hwresult )
  reset-wait \ Delay until any prior reset operation is done.
  his-lun h# 80 or regs >message-out rb! \ Set unit number; no disconnect.
  my-id regs >host-selectid rb! \ Set the selection IDs.
  his-id regs >target-selectid rb!

\ Write the command block into the host adapter's command register

dup 0 ?do ( data-adr,len dir cmd-adr,len )
  over i + c@ ( data-adr,len dir cmd-adr,len cmd-byte )
  regs >cmd-adr i ca+ rb! ( data-adr,len dir cmd-adr,len )
loop ( data-adr,len dir cmd-adr,len )
regs >cmd-len rl! drop ( data-adr,len dir )

\ Set the data transfer parameters.

( .. dir ) regs >input? rb! ( data-adr,len ) \ Direction
( .. len ) regs >data-len rl! ( data-adr ) \ Length
( .. adr ) regs >data-adr rl! ( ) \ DMA Address

\ Now we're ready to execute the command.

1 regs >start rb! \ Tell board to start the command.
get-msecs scsi-time + to time-limit \ Set the time limit.
begin regs >start rb@ while \ Wait until command finished.
  scsi-time if
    get-msecs time-limit - 0>= if \ If timeout is enabled, and
    the time-limit has been reached,
reset-scsi-bus true exit \ reset the bus and return error.
then
then
repeat
\ Nonzero phase means that the command didn't finish.
regs >phase rb@
;
\ Returns true if select failed
: execute-command ( data-adr,len dir cmd-adr,len -- hwresult | statbyte false)
\ Temporarily put dir and cmd-adr,len on the return stack to get them
\ out of the way so we can work on the DMA data buffer.
>r >r >r ( data-adr,len )
dup if ( data-adr,len )
\ If the data transfer has a nonzero length, we have to map it in.
2dup false dma-map-in ( data-adr,len dma )
2dup swap r> r> r> ( data-adr,len dma,len dir cmd-adr,len)
(exec) ( data-adr,len phys hwres)
>r swap dma-map-out r> ( hwresult )
else ( data-adr,len )
r> r> r> (exec) ( hwresult )
then ( hwresult )
?dup 0= if ( hwresult | )
\ Command finished; return stat byte and false.
reg >status rb@ false \ Command finished; return status byte and false.
then ( hwresult | statbyte 0 )
;
external
: reset ( -- ) map reset-scsi-bus unmap ;
reset \ Reset the SCSI bus when we are probed.

: open-hardware ( -- okay? )
map
\ Should perform a quick "sanity check" selftest here,
\ returning true if the test succeeds.
true ;
: reopen-hardware ( -- okay? ) true ;

: close-hardware ( -- ) unmap ;
: reclose-hardware ( -- ) ;

: selftest ( -- 0 | error-code )
\ Perform reasonably extensive selftest here, displaying
\ a message and returning an error code if the
\ test fails and returning 0 if the test succeeds.
0 ;
: set-address ( unit target -- )
to his-id to his-lun ;
E.6.3  hacom.fth

\ Common code for SCSI host adapter drivers
\ The following code is intended to be independent of the details of the
\ SCSI hardware implementation. It is loaded after the hardware-dependent
\ file that defines execute-command, set-address, open-hardware, etc.

headers

-1 value inq-buf \ Address of inquiry data buffer
-1 value sense-buf \ Holds extended error information

0 value #retries ( -- n ) \ number of times to retry SCSI transaction

\ Classifies the sense condition as either okay (0), retryable (1),
\ or non-retryable (-1)
: classify-sense ( -- 0 | 1 | -1 )
  sense-buf

  \ Make sure we understand the error class code.
  dup c@ h# 7f and h# 70 <> if drop -1 exit then

  \ Check for filemark, end-of-media, or illegal block length.
  dup 2+ c@ h# e0 and if drop -1 exit then

  2 + c@ h# f and ( sense-key )

  \ no_sense(0) and recoverable(1) are okay.
  dup 1 <= if drop 0 exit then ( sense-key )

  \ not-ready(2) and attention(6) are retryable.
  dup 2 = swap 6 = or if 1 else -1 then

;

0 value open-count

external

\ The SCSI device node defines an address space for its children. That
\ address space is of the form "target#,unit#". target# and unit# are
\ both integers. parse-2int converts a text string (e.g., "3,4") into
\ a pair of binary integers.
: decode-unit ( addr len -- unit# target# ) parse-2int ;

: open ( -- okay? )
  open-count if
    reopen-hardware dup if open-count 1+ to open-count then
    exit
  else
    open-hardware dup if
      1 to open-count
      100 dma-alloc to sense-buf
      100 dma-alloc to inq-buf
    then
    then
  ;

: close ( -- )
  open-count 1- to open-count
  open-count if
    reclose-hardware
else
  close-hardware
  inq-buf 100 dma-free
  sense-buf 100 dma-free
then
;

headers

create sense-cmd 3 c, 0 c, 0 c, 0 c, ff c, 0 c,

: get-sense ( -- ) \ Issue REQUEST SENSE, which is not supposed to fail.
  sense-buf ff true sense-cmd 6 execute-command 0= if drop then
;
\ Give the device a little time to recover before retrying the command.
: delay-retry ( -- ) 1000 0 do loop ;

0 value statbyte \ Local variable used by retry?

\ RETRY? is used by RETRY-COMMAND to determine whether or not to retry the
\ command, considering the following factors:
\  - Success or failure of the command at the hardware level (failure at
\    this level is usually fatal, except in the case of an incoming bus reset)
\  - The value of the status byte returned by the command
\  - The condition indicated by the sense bytes
\  - The number of previous retries
\ \ The input arguments are as returned by "scsi-exec".
\ On output, the top of the stack is true if the command is to be retried,
\ otherwise the top of the stack is false and the results that should be
\ returned by retry-command are underneath it; those results indicate the type
\ of error that occurred.

: retry? ( hw-result | statbyte 0 -- true | [[sensebuf] f-hw] error? false )
  case
    0 of to statbyte endof \ No hardware error; continue checking.
    1 of true exit endof \ Retry after incoming bus reset.
    ( hw-result ) true false exit \ Other hardware errors are fatal.
  endcase

statbyte 0= if false false exit then \ If successful, return "no-error".

statbyte 2 and if \ "Check Condition", so get extended status.
  get-sense classify-sense case
    ( -1|0|1 )
    \ If the sense information says "no sense", return "no-error".
    0 of false false exit endof
    \ If the error is fatal, return "sense-buf,valid,statbyte".
    -1 of sense-buf false statbyte false exit endof
  endcase

\ Otherwise, the error was retryable. However, if we have
\ have already retried the specified number of times, don't
\ retry again; instead return sense buffer and status.
#retries 0= if sense-buf false statbyte false exit then

\ Don't retry if vendor-unique, reserved, intermediate, or
\ "condition met/good" bits are set. Return "no-sense,status".
statbyte h# f5 and if true statbyte false exit then

\ Don't retry if we have already retried the specified number
of times. Return "no-sense,status".

\*retries 0= if true statbyte false exit then

\* Otherwise, it was either a busy or a retryable check condition, 
\* so we retry.

true

\* RETRY-COMMAND executes a SCSI command. If a check condition is indicated,
\* performs a "get-sense" command. If the sense bytes indicate a non-fatal
\* condition (e.g., power-on reset occurred, not ready yet, or recoverable
\* error), the command is retried until the condition either goes away or
\* changes to a fatal error.

\* The command is retried until
\* a) The command succeeds, or
\* b) The select fails, or dma fails, or
\* c) The sense bytes indicate an error that we can't retry at this level, or
\* d) The number of retries is exceeded.

\* #retries is number of times to retry (0: don't retry, -1: retry forever)

\* sensebuf is the address of the sense buffer; it is present only
\* if f-hw is 0 and error? is nonzero. The length of the sense buffer
\* is 8 bytes plus the value in byte 7 of the sense buffer.

\* f-hw is nonzero if there is a hardware error -- dma fails, select fails,
\* etc. -- or if the status byte was neither 0 (okay) nor 2 (check condition).

\* error? is nonzero if there is a transaction error. If error? is 0,
\* f-hw and sensebuf are not returned.

\* If sensebuf is returned, the contents are valid until the next call to
\* retry-command. sensebuf becomes inaccessible when this package is closed.

\* dma-dir is necessary because it is not always possible to infer the DMA
\* direction from the command.

\* Local variables used by retry-command?

0 instance valuedbuf \ Data transfer buffer
0 instance valuedlen \ Expected length of data transfer
0 instance valuedirection-in \ Direction for data transfer

-1 instance valuecbuf \ Command base address
0 instance valuedlen \ Actual length of this command

external

: retry-command ( dma-buf dma-len dma-dir cmdbuf cmdlen #retries -- ... )
    ( ... -- [[sensebuf] f-hw] error? )
to #retries to clen to cbuf to direction-in to dlen to dbuf
begin
dbuf dlen direction-in cbuf clen execute-command ( hwerre stat 0 )
retry?
while
    #retries 1- to #retries
    delay-retry
repeat
;
headers
Collapses the complete error information returned by retry-command into a single error/no-error flag.

: error? ( false | true true | sensebuf false true -- error? )
    dup if swap 0= if nip then then
;

external

Simplified "retry-command" routine for commands with no data transfer phase and simple error checking requirements.

: no-data-command ( cmdbuf -- error? )
    >r 0 0 true r> 6 -1 retry-command error?
;

short-data-command executes a command with the following characteristics:
  a) The data direction is incoming
  b) The data length is less than 256 bytes

The host adapter driver is responsible for supplying the DMA data buffer; if the command succeeds, the buffer address is returned. The buffer contents become invalid when another SCSI command is executed, or when the driver is closed.

: short-data-command ( data-len cmdbuf cmdlen -- true | buffer false )
    >r >r inq-buf swap true r> r> -1 retry-command ( retry-cmd-results )
    error? dup 0= if inq-buf swap then
;

headers

Here begins the implementation of "show-children", a word that is intended to be executed interactively, showing the user the devices that are attached to the SCSI bus.

Tool for storing a big-endian 24-bit number at an unaligned address

: 3c! ( n addr -- ) >r lbsplit drop r@ c! r@ 1+ c! r> 2+ c! ;

Command block template for Inquiry command

create inquiry-cmd h# 12 c, 0 c, 0 c, 0 c, ff c, 0 c,

: inquiry ( -- error? )
    \ 8 retries should be more than enough; inquiry commands aren't supposed to respond with "check condition".
    inq-buf ff true inquiry-cmd 6 8 retry-command error?
;

Returns true if the target number "select-id" responds to the inquiry command.

: probe-target ( select-id -- present? )
    0 swap set-address inquiry 0=
;

Reads the indicated byte from the Inquiry data buffer.

: inq@ ( offset -- value ) inq-buf + c@ ;
: .scsi1-inquiry ( -- ) inq-buf 5 ca+ 4 inq@ fa min type ;
: .scsi2-inquiry ( -- ) inq-buf 8 ca+ d# 28 type ;

Displays the results of an Inquiry command to the indicated device.

: show-lun ( unit target -- )
  over swap set-address                         ( unit )
  inquiry if drop exit then                     ( unit )
  0 inq@ h# 7f = if drop exit then             ( unit )
  '" Unit " ." "                      ( )
  1 inq@ h# 80 and if ." Removable " then ( )
  0 inq@ case
    0 of ." Disk " endof
    1 of ." Tape " endof
    2 of ." Printer " endof
    3 of ." Processor " endof
    4 of ." WORM " endof
    5 of ." Read Only device" endof
    ( default ) ." Device type " dup .h
    endcase
    1 inq@ h# 7f and ?dup if ." Qualifier " .h then
    4 spaces
    3 inq@ 0f and 2 = if .scsi2-inquiry else .scsi1-inquiry then
                   cr
  ;

external

\ Searches for devices on the SCSI bus, displaying the Inquiry information
\ for each device that responds.

: show-children ( -- )
  open 0= if ." Can't open SCSI host adapter" cr exit then
  8 0 do
    i probe-target if
    ." Target " i . cr
    8 0 do i j show-lun loop
  then
  loop
  close
  ;

headers
\ The Diagnose command is useful for generic SCSI devices.
\ It executes both "test-unit-ready" and "send-diagnostic"
\ commands, decoding the error status information they return.

create test-unit-rdy-cmd 0 c, 0 c, 0 c, 0 c, 0 c,
create send-diagnostic-cmd h# 1d c, 4 c, 0 c, 0 c, 0 c, 0 c,

: send-diagnostic ( -- error? ) send-diagnostic-cmd no-data-command ;

external

: diagnose ( -- error? )
  0 0 true test-unit-rdy-cmd 6 -1 ( dma$ dir cmd$ #retries )
  retry-command if
    ( [ sensebuf ] hardware-error? )
    ." Test unit ready failed - " ( [ sensebuf ] hardware-error? )

221
if ." hardware error (no such device?)" cr ( )
else ( sensebuf )
 ." extended status = " cr ( sensebuf )
 base @ >r ( sensebuf )
 8 bounds ?do i 3 u.r loop cr ( )
 then true
else
 send-diagnostic ( fail? )
 then ;

headers

E.6.4 scsicom.fth

\ This file contains some words that are useful for both
\ SCSI disk and SCSI tape device drivers.
\ The SCSI disk and SCSI tape packages need to export dma-alloc and dma-free
\ methods so the deblocker can allocate DMA-capable buffer memory.

external
: dma-alloc ( n -- vaddr ) " dma-alloc" $call-parent ;
: dma-free ( vaddr n -- ) " dma-free" $call-parent ;

headers
: parent-max-transfer ( -- n ) " max-transfer" $call-parent ;

\ Calls the parent device's "retry-command" method. The parent device is
\ assumed to be a driver for a SCSI host adapter (device-type = "scsi").

: retry-command ( dma-addr dma-len dma-dir cmd-addr cmd-len #retries -- ... )
 ( ... -- false ) \ No error
 ( ... -- true true ) \ Hardware error
 ( ... -- sensebuf false true ) \ Fatal error with extended status
 " retry-command" $call-parent
 ;

\ Simplified command execution routines for common simple command forms
: short-data-command ( data-len cmdbuf cmdlen -- true | buffer false )
 " short-data-command" $call-parent ;

\ Some tools for reading and writing 2-, 3-, and 4-byte numbers to and from
\ SCSI command and data buffers. The ones defined below are used both in
\ the SCSI disk and the SCSI tape packages. Other variations that are
\ used only by one of the packages are defined in the package where they
\ are used.

: +c! ( n addr -- addr' ) tuck c! 1+ ;
: 3c! ( n addr -- ) >r lbsplit drop r> +c! +c! c! ;

: -c@ ( addr -- n addr' ) dup c@ swap l- ;
: 3c@ ( addr -- n ) 2 + -c@ -c@ c@ 0 bljoin ;
: 4c@ ( addr -- n ) 3 + -c@ -c@ -c@ c@ bljoin ;
"Scratch" command buffer useful for construction of read and write commands

create cmdbuf 0 c, 0 c, 0 c, 0 c, 0 c, 0 c, 0 c, 0 c, 0 c, 0 c, 0 c, 0 c,
: cb! ( byte index -- ) cmdbuf + c! ; \ Write byte to command buffer

The deblocker converts a block/record-oriented interface to a byte-oriented
interface, using internal buffering. Disk and tape devices are usually
block- or record-oriented, but the OBP external interface is byte-oriented,
in order to be independent of particular device block sizes.

0 instance value deblocker
: init-deblocker ( -- okay? )
  " " deblocker" $open-package to deblocker
  deblocker if
    true
  else
    ." Can't open deblocker package" cr false
  then
;
headerless
: selftest ( -- error? )
fcode-revision h# 3.0000 >= if
  my-unit " set-address" $call-parent
  " diagnose" $call-parent
else
  0
then
;
headers

E.6.5 scsidisk.fth

SCSI disk package implementing a "block" device-type interface

" sd" encode-string " name" property
" block"    device-type

fload scsicom.fth \ Utility routines for SCSI commands

hex

0 means no timeout
: set-timeout ( msecs -- ) " set-timeout" $call-parent ;

0 instance value offset-low \ Offset to start of partition
0 instance value offset-high

0 instance value label-package

Sets offset-low and offset-high, reflecting the starting location of the
partition specified by the "my-args" string.

: init-label-package ( -- okay? )
  0 to offset-high 0 to offset-low
  my-args " disk-label" $open-package to label-package
  label-package if
    0 0 " offset" label-package $call-method to offset-high to offset-low
  true
else
  ." Can't open disk label package" cr false
then

\ Ensures that the disk is spinning, but doesn't wait forever.
create sstart-cmd h# 1b c, 1 c, 0 c, 0 c, 1 c, 0 c,
: timed-spin ( -- error? )
  d# 15000 set-timeout
  sstart-cmd no-data-command
  0 set-timeout

0 instance value /block \ Device native block size
create mode-sense-cmd h# 1a c, 0 c, 0 c, 0 c, d# 12 c, 0 c,
create read-capacity-cmd h# 25 c, 0 c, 0 c, 0 c, d# 12 c, 0 c,
  0 c, 0 c, 0 c, 0 c,
: read-block-size ( -- n ) \ Ask device about its block size.
  \ First try "mode sense" - data returned in bytes 9,10,11.
  d# 12 mode-sense-cmd 6 short-data-command if 0 else 9 + 3c@ then
  ?dup if exit then
  \ Failing that, try "read capacity" - data returned in bytes 4,5,6,7.
  8 read-capacity-cmd 0a short-data-command if 0 else 4 + 4c@ then
  ?dup if exit then
  d# 512 \ Default to 512 if the device won't tell us.
;
external
\ Return device block size; cache it the first time we find the information.
\ This method is called by the deblocker.
: block-size ( -- n )
  /block if /block exit then \ Don't ask if we already know.

  read-block-size dup to /block
;
headers
\ Read or write "#blks" blocks starting at "block#" into memory at "addr"
\ Input? is true for reading or false for writing.
\ Command is 8 for reading or h# a for writing.
\ We use the 6-byte forms of the disk read and write commands.
: 2c! ( n addr -- ) >r lbsplit 2drop r> +c! c! ;
: 4c! ( n addr -- ) >r lbsplit r> +c! +c! +c! c! ;
: r/w-blocks ( addr block# #blks input? command -- actual# )
  3 pick h# 100000 u== if \ Use 10-byte form ( addr block# #blks dir cmd )
  h# 20 or 0 cb! \ 28 (read) or 2a (write) ( addr block# #blks dir )
  -rot swap ( addr dir #blks block# )
  cmdbuf 2 + 4c! ( addr dir #blks )
  dup cmdbuf 7 + 2c! ;
d# 10                                       ( addr dir #blks cmd-len )
else                        \
  Use 6-byte form ( addr block# #blks dir cmd )
  0 cb!                              ( addr block# #blks dir )
-rot swap                          ( addr dir #blks block# )
cmdbuf 1+ 3c!                     ( addr dir #blks )
dup 4 cb!                         ( addr dir #blks )
6                                   ( addr dir #blks cmd-len )
then
  tuck >r >r                     ( addr input? #blks ) ( R: #blks cmd-len )
/block * swap cmdbuf r> -1       ( addr #bytes input? cmd cmd-len #retries )
retry-command if                ( [ sensebuf ] hw? )
  0= if drop then r> drop 0
else                             ( )
  r>
then                             ( actual# )
;
external
\ These three methods are called by the deblocker.

: max-transfer ( -- n ) parent-max-transfer ;
: read-blocks ( addr block# #blocks -- #read )
  true  d# 8  r/w-blocks ;
: write-blocks ( addr block# #blocks -- #written )
  false d# 10 r/w-blocks ;
\ Methods used by external clients

: open ( -- flag )
  my-unit " set-address" $call-parent

\ It might be a good idea to do an inquiry here to determine the
\ device configuration, checking the result to see if the device
\ really is a disk.

\ Make sure the disk is spinning.

timed-spin if false exit then

block-size to /block

init-deblocker 0= if false exit then

init-label-package 0= if
  deblocker close-package false exit
then
true
;

: close ( -- )
  label-package close-package
  deblocker close-package
;

: seek ( offset.low offset.high -- okay? )
  offset-low offset-high x+ " seek" deblocker $call-method
;
: read ( addr len -- actual-len )
  " read" deblocker $call-method ;
: write ( addr len -- actual-len )
  " write" deblocker $call-method ;
: load ( addr -- size )
  " load" label-package $call-method ;

headers
E.6.6 scsitape.fth

\ SCSI tape package implementing a "byte" device-type interface.
\ Supports both fixed-length-record and variable-length-record tape devices.

" st" encode-string " name" property
" byte" device-type

fload scsicom.fth \ Utility routines for SCSI commands

hex

external

false instance value at-eof? \ Turned on when read-blocks hits file mark.
headers

false instance value fixed-len? \ True if the device has fixed-length blocks.
false instance value written? \ True if the tape has been written.
0 instance value /tapeblock \ Max length for variable-length records;
\ actual length for fixed-length records.

create write-eof-cmd h# 10 c, 1 c, 0 c, 0 c, 1 c, 0 c,

external

\ Writes a file mark.

: write-eof ( -- error? ) write-eof-cmd no-data-command ;
headers

\ Writes a file mark if the tape has been written since the last seek
\ or rewind or write-eof.

: ?write-eof ( -- )
    written? if
        false to written?
        write-eof if ." Can't write file mark." cr then then
    then
;
create rewind-cmd 1 c, 1 c, 0 c, 0 c, 0 c, 0 c,

: rewind ( -- error? ) \ Rewinds the tape.
    ?write-eof
    false to at-eof?
    rewind-cmd no-data-command
;
create skip-files-cmd h# 11 c, 1 c, 0 c, 0 c, 0 c, 0 c,

: skip-files ( n -- error? ) \ Skips n file marks.
    ?write-eof
    false to at-eof? ( n )
    skip-files-cmd 2 + 3c! ( )
    skip-files-cmd no-data-command ( error? )
;
\ Asks the device its record length.
\ Also determines fixed or variable length.

create block-limit-cmd 5 c, 0 c, 0 c, 0 c, 0 c, 0 c,
: 2c@ ( addr -- n ) 1 + -c@ c@ bwjoin ;

: get-record-length ( -- )
6 block-limit-cmd 6 short-data-command if
  d# 512 true ( blocksize fixed-len )
else
  dup 1 + 3c@ swap 4 + 2c@ ( max-len min-len )
  over = ( blocksize fixed-len? )
  then
  fixed-len? ( blocksize )
  to /tapeblock
  ( )
  then
true instance value first-install? \ Used for rewind-on-first-open.

\ Words to decode various interesting fields in the extended status buffer.
\ Used by actual-#blocks.

\ Incorrect length
: ili? ( statbuf -- flag ) 2 + c@ h# 20 and 0<> ;

\ End of Media, End of File, or Blank Check
: eof? ( statbuf -- flag )
  dup 2 + c@ h# c0 and 0<> swap 3 + c@ h# f and 8 = or ;

\ Difference between requested count and actual count
: residue ( statbuf -- residue ) 3 + 4c@ ;

0 instance value #requested \ Local variable for r/w-some and actual-#blocks

\ Decodes the status information returned by the SCSI command to
\ determine the number of blocks actually tranferred.

  if \ Error
    ( true | xstatbuf false )
    if
      \ Hardware error; none tranferred
      ( )
    else
      \ Decode status buffer
      ( xstatbuf )
      >r #requested ( #requested ) ( r: xstatbuf )
      r@ ili? r@ eof? or if
      ( #requested ) ( r: xstatbuf )
      r@ residue -
      ( #xfered ) ( r: xstatbuf )
      then
      ( #xfered ) ( r: xstatbuf )
      r> eof?
      ( #xfered flag )
      then
      else \ no error, #request = #xfered ( )
      #requested false ( #xfered flag )
then
to at-eof?
;

\ Reads or writes at most "#blks" blocks, returning the actual number
\ of blocks transferred, and an error indicator that is true if either a
\ fatal error occurs or the end of a tape file is reached.

: r/w-some ( addr #blks input? cmd -- actual# error? )
  0 cb! swap ( addr dir #blks )
  fixed-len? if ( addr dir #blks )
    \ If the tape has fixed-length records, multiply the
    \ requested number of blocks by the record size.
    dup to #requested ( addr dir #blks )
    dup /tapeblock * swap 1 ( addr dir #bytes cmd-cnt 1=fixed-len )
  else \ variable length ( addr dir #bytes )
    \ If the tape has variable length records, transfer one record.
    drop /tapeblock ( addr dir #bytes )
    dup to #requested ( addr dir #bytes )
    dup 0 ( addr dir #bytes cmd-cnt 0=variable-len )
  then
  1 cb! cmdbuf 2 + 3c! ( addr dir #bytes )
  swap cmdbuf 6 -1 ( dma-addr,len dir cmd-addr,len #retries)
  retry-command actual-#blocks ( actual# )
;
\ Discard (for read) or flush (for write) any bytes that are buffered by
\ the deblocker.

: flush-deblocker ( -- )
  deblocker close-package init-deblocker drop
;

The deblocker package calls max-transfer to determine an appropriate
\ internal buffer size.

: max-transfer ( -- n )
  fixed-len? if
    \ Use the largest multiple of /tapeblock that is <= parent-max-transfer.
    parent-max-transfer /tapeblock / /tapeblock *
  else
    /tapeblock
  then
;
\ The deblocker package calls block-size to determine an appropriate
\ granularity for accesses.

: block-size ( -- n )
  fixed-len? if /tapeblock else 1 then
;
\ The deblocker uses read-blocks and write-blocks to access tape records.

228
The assumption of sequential access is guaranteed because this code is only called from the deblocker. Since the SCSI tape package implements its own "seek" method, the deblocker seek method is never called, and the deblocker's internal position only changes sequentially.

: read-blocks ( addr block# #blocks -- #read )
  nip ( addr #blocks ) \ Sequential access
  \ Don't read past a file mark
  at-eof? if 2drop 0 exit then ( addr #blocks )
  true 8 r/w-some ( #read ) ;

: write-blocks ( addr block# #blocks -- #read )
  nip ( addr #blocks ) \ Sequential access
  true to written? ( addr #blocks )
  false h# a r/w-some ( #written ) ;

\ Methods used by external clients

: read ( addr len -- actual-len ) " read" deblocker $call-method ;

: write ( addr len -- actual-len )
  " write" deblocker $call-method ( actual-len )
  flush-deblocker \ Make the tape structure reflect the write pattern ;

: open ( -- okay? )
  my-unit " set-address" $call-parent
  \ It might be a good idea to do an inquiry here to determine the
  \ device configuration, checking the result to see if the device
  \ really is a tape.
  first-install? if
    rewind if
      " Can't rewind tape" cr
      false exit
    then
    false to first-install?
  then
  get-record-length
  init-deblocker ( okay? ) ;

: close ( -- )
  deblocker close-package
  ?write-eof ;

0 value buf
h# 200 constant /buf

\ It would be better to keep track of the current file number and
\ just seek forward if the requested file number/position is greater
\ than the current file number/position. Taking care of end-of-file
\ conditions would be tricky though.
: seek  ( byte# file# -- error? )
  flush-deblocker  
  rewind   if  2drop true exit then  
  ?dup if  
     skip-files if  drop true exit then  
     then  
  ?dup if  
     /buf alloc-mem to buf  
     begin  dup 0> while  
        buf over /buf min read  
        dup 0= if  2drop true exit then  
        -  
        repeat  
        drop  
        buf /buf free-mem  
     then  
  false  
; 
: load  ( loadaddr -- size )
  my-args  dup if  
    $number if  
      ." Invalid tape file number" cr  
      drop 0 exit  
      then  
    else  
      nip  
    then  
  0 swap seek if  
    ." Can't select the requested tape file" cr  
  0 exit  
  then  
\ Try to read the entire tape file. We ask for a huge size  \
  (almost 2 Gbytes), and let the deblocker take care of  \
  breaking it up into manageable chunks. The operation  \
  will cease when a file mark is reached.
  h# 70000000 read  
; 
headers
Annex F
Answers to common questions
(informative)

F.1 What is the expected (not required) overall flow of the use of Open Firmware?

At power-on time the platform and all plug-in options do any necessary internal self-test and initialization. The scope of Open Firmware begins when the platform starts to determine its I/O topology, i.e., determine what devices are reachable and by what physical paths through the bus system. The Open Firmware in the platform builds a device tree data structure to represent this topology by asking each bus node to probe itself, thus finding what devices are installed and creating nodes containing their device names and properties. If there are bus bridges subordinate to those first-level buses, the process is repeated recursively, thus probing deeper into the tree.

“Probing” involves reading bytes at designated (but bus-dependent) addresses where devices might reside, then for populated addresses, checking that an FCode ROM has been located. If so, successive bytes are read from the FCode ROM on the card and interpreted as FCode functions by the platform firmware. As it runs, this FCode program from the card creates properties and methods in the current (originally blank) node in the device tree. At any point, this program can cause succeeding bytes to be incrementally compiled, rather than interpreted. A device that might be “interesting” at boot time would nominally fill in the full set of properties (via property, 5.3.5.3) and methods (via external-token, 5.3.3.1) specified by Open Firmware for its device type (see 3.7). Other devices that are less “interesting” at boot time, like a fax modem, are only expected to fill in their “name” and possibly their “reg” and “interrupts” properties. Note that the node on the device tree is not filled in by platform-based code based on data read from the card ROM, but rather an FCode program is retrieved from the card and executed, thus filling in the node.

Once the platform’s boot firmware has constructed the device tree and possibly conducted some further level of system self-testing, it selects boot devices based on some combination of information in the platform’s boot ROM, nonvolatile RAM, and, once selected, the system boot console. Boot devices consist mainly of the console display and keyboard devices, and the device that provides the code image being booted. The platform’s Open Firmware polls the console input device for characters, prints output characters to the console display, and calls the load method of the selected source device to read a client program into memory. If the load method completes successfully, the firmware transfers control to the client program.

The client program could immediately take over the system, wiping away all Open Firmware structures, or it could be a secondary loader program that uses the facilities of the Open Firmware client interface (Clause 6), and indirectly the Open Firmware methods of the devices, to perform various functions and load successive images for portions of the run-time OS.

Ultimately, when the run-time OS assumes control of the system, it may choose to inherit some or all of the device tree information from Open Firmware, or to keep the entire device tree around for fatal system errors or system resets. Alternatively, the OS could wipe away all traces of Open Firmware from system memory and conduct its own system probe, etc.

For a more detailed description on any of these steps, see 4.2, other related areas of the body of the Open Firmware standard, or the responses to the following questions.

F.2 How does the overall firmware operation work, considering the interactions between the CPU firmware, the FCode drivers, and the operating system?

Here is an overview of the operation of a hypothetical computer system that uses Open Firmware. It describes the interactions among the hardware, the firmware, and the operating system. We assume that the system in question
uses the complete set of Open Firmware interfaces. In practice, some systems will choose to implement only one or two of the three Open Firmware interfaces. For such systems, the behaviors of any non-Open Firmware interfaces may, of course, differ from the behavior described herein.

After the system is powered on, power-on self-test (POST) code (whose details are outside the scope of this standard) executes, determining that the core hardware is operational, then passes control to the Open Firmware. The Open Firmware initializes itself and any core hardware that is needed for the basic operation of the firmware. This initialization step includes building the portion of the device tree that represents the system’s built-in devices. The mechanism for accomplishing this is not specified herein; although a firmware system might choose to use FCode to represent the built-in part of the device tree, that built-in portion could equally well be “hard-coded” as an internal data structure.

The next step is to augment the device tree to include plug-in devices. It is here that the Open Firmware device interface comes into play. The firmware scans the slots of its expansion bus or buses. For each card that it finds, it interprets the FCode program stored on that card, resulting in the creation of one or more device nodes describing that card. In the case of a card that is a bridge to another bus, an entire hierarchy of device nodes might be created, perhaps as a result of a recursive process in which FCode programs on cards subordinate to that bus bridge are interpreted. (Device nodes for plug-in devices could also be created by some mechanism other than interpreting FCode programs, for example, by reading bus-dependent standard configuration registers.)

The advantage of FCode is that its design is neither bus-specific, processor-specific, nor operating system-specific, thus avoiding the need to invent and support a new configuration mechanism for each new bus.

After all of the plug-in slots have been probed, the device tree contains a complete representation of the hardware configuration. Each device node has a set of properties that describe the static characteristics of the associated device and (optionally) a set of methods that the firmware can use to drive the device. The firmware selects devices for its console input and output functions, using the device node methods to drive the devices, and displays a banner identifying the system and its configuration.

The firmware selects a boot device and uses its device node methods to load a program, perhaps from a well-known group of disk sectors. Typically, the program is an intermediate boot program whose job is to load the operating system, which might be stored in a disk file. The intermediate boot program would be responsible for understanding the file system layout and file format used by that operating system. The intermediate boot program could use firmware services provided by the Open Firmware client interface to do things like allocating memory and performing low-level disk reads. Thus, the intermediate boot program would be concerned only with knowing the specifics of the file system format, and would not need to know processor-specific or system-configuration-specific information. This separation of function and responsibility simplifies the design of such programs, makes them easier to maintain, and allows the same boot program to be used in different system configurations.

After the boot program has loaded the operating system, the boot program can exit if it is no longer needed, or it can remain resident to provide additional services to assist the operating system in its configuration process. For example, the operating system might wish to take advantage of the booter’s knowledge of the file-system format, using the booter to load configuration-dependent extension modules before the OS is fully operational.

As the OS is configuring itself, it uses firmware services via the Open Firmware client interface for such purposes as determining the hardware configuration, displaying progress messages, and (perhaps with the help of the booter program) performing disk or network I/O.

At some point, the operating system completes its configuration process to the extent that it no longer needs firmware services and can then reclaim any resources used by the firmware, assuming complete responsibility for managing all system resources. Once it has done so, the OS must ensure that the firmware is not invoked (i.e., by taking control of traps or interrupts that could enter the firmware and by not calling firmware client interface services). Alternatively, the OS could allow the firmware to remain resident and operational by not reclaiming the resources (primarily memory) that the firmware is using. Doing so would allow the OS to continue to invoke
firmware services as needed. The Open Firmware design assumes that the operating system will not be making heavy use of firmware services once the OS is fully operational; however, it is reasonable for an OS to make occasional use of firmware services. Examples of such occasional uses include the following:

— Sending error messages to a diagnostic console
— Browsing the firmware device tree when a user wishes to inspect the hardware configuration
— Displaying and setting firmware configuration variables in nonvolatile memory
— Debugging the operating system using Open Firmware’s software debugging features
— Rebooting

Since the Open Firmware execution model assumes a single thread of control, it is the operating system’s responsibility to ensure that firmware client interface services are called in a manner consistent with this execution model. Typically, on a multiprocessor system, the firmware initialization, booting, and operating system configuration steps all execute on a single processor, and the operating system enables multiprocessor mode only after those steps are complete.

If the client program desires to take over the management of resources and wishes to continue to use certain non-memory-allocating services of Open Firmware, it must use the device tree client interface functions at some well-defined point in time to find out about all “available” and “existing” resources and respect Open Firmware’s use of allocated resources not appearing in the “available” properties.

Upon reentering Open Firmware via the “enter” or “exit” client interface function, the states of the resources are as they were left by Open Firmware and the client program modifications to them via use of the client interface.

The handling of “fatal errors” is implementation-dependent because the specification of these errors is implementation-dependent. For the most part, the ISA or platform implementation of Open Firmware should specify which errors it is prepared to handle and which it is not, and which errors cause an internal reset and which do not. For example, on SPARC implementations, one might expect a watchdog reset to be handled by Open Firmware with very little state remaining, but a divide-by-zero error or unaligned-access error might be handled without resetting all state information.

F.3 Which Open Firmware interfaces do I need to be compliant? Can I mix Open Firmware-compliant interfaces with other existing firmware interfaces?

The three interfaces defined by this specification are individually optional. A compliant firmware implementation can have any combination of the three interfaces, depending on the capabilities and the requirements of the computer system. Some examples follow.

A general-purpose computer system with an expansion bus, intended to run a general-purpose operating system (or perhaps several different operating systems) would benefit from all three of the Open Firmware interfaces. The device interface would allow the firmware to identify and use plug-in devices added to the system via the expansion bus. The client interface would allow the operating system to configure itself to use those devices automatically. The user interface would allow a user or system administrator to manage the system when the operating system is not running.

A computer system with no expansion bus would not need the device interface, but it could still benefit from the client interface and the user interface. Although the configuration of a nonexpandable system is presumably fixed, an operating system might still need to determine that configuration at run-time, thus letting a single “shrink-wrapped” operating system work on a variety of different systems within the same “family.” Some of the systems in that family might be expandable, and others nonexpandable. The single operating system would not need to know any of the hardware system configuration details in advance. The user interface would allow the user to test the hardware and to control, for example, which disk boots the operating system.
An expandable computer system with a built-in operating system (perhaps an OS in ROM) might implement the Open Firmware device interface directly within the OS, giving that OS the ability to identify and initialize plug-in devices on standard buses.

A non-expandable system that is intended to run a single operating system that uses a proprietary interface to the firmware might choose to have only the Open Firmware user interface, perhaps for its debugging capabilities.

Many combinations of Open Firmware-compliant interfaces mixed with other proprietary firmware interfaces are plausible. Furthermore, since each of the Open Firmware interfaces is extensible, an Open Firmware compliant interface can be augmented to support system-specific requirements, such as “legacy” buses with proprietary identification mechanisms, other OS-to-firmware interface protocols, and additional user interface paradigms, such as backwards compatibility with a vendor’s pre-existing firmware user interface or graphical user interface shells.

Another possibility would be to implement another client interface “on top of” the Open Firmware client interface.

F.4 What is the distinction between a probe address and a unit address?

The probe address is the address of a plug-in card, or equivalently, the address of the slot into which it plugs. The process of creating one or more device nodes for a plug-in card involves executing an FCode program. Before that FCode program is executed, the firmware system does not know where the devices on that card are located within the address space available to it. The system firmware knows only the address of the card itself: the probe address, which is established by set-args and returned by my-address and my-space. It is the responsibility of the card’s FCode program to know the addresses of the devices on the card, publishing those addresses in a “reg” property for each device. Within a given “reg” property, the first such address is the device node’s unit address, the address that identifies a device node. The unit address of a device is the address that matches when resolving a pathname (to select a particular device node).

In most cases, the FCode program for a device (or group of devices) does not know the device’s unit address directly. Instead, it knows how to calculate the unit address from the probe address. In general, that calculation depends both on the design of the particular bus and on the details of the individual card.

As a particularly simple example, consider an SBus card [see B2] with two logically independent devices, device A and device B, on it. According to the SBus specification, each SBus slot has its own 28-bit address space, separate from the address spaces of other slots. The complete unit address of a device on an SBus consists of a slot number and an offset within that slot’s 28-bit space. The probe address of an SBus slot is the slot number and an offset of zero. (Coincidentally, that probe address also happens to be the address of an SBus card’s FCode program; that is not necessarily the case for other buses.) In our example, let us assume that device A begins at offset (hex) 20000 within the slot’s address space, and device B begins at offset (hex) 30000. The FCode program would calculate device A’s unit address by adding 20000 to the probe address offset (which happens to be zero for SBuses) returned by my-address, and using the slot number returned by my-space. For SBus, in which the addressing is strictly slot-based, the slot number of the probe address is always the same as the slot number of the devices in that slot. That observation may appear to be trivial, but it is not necessarily so for other buses. Some buses have separate configuration, memory, and I/O address spaces. The configuration space is usually addressed with something like a slot number, while the other address spaces, in which individual devices reside, might have no fixed relationship with slots.

In general, an arbitrarily complex calculation, possibly involving dynamic address allocation, might be necessary in order to determine the unit address of a particular device on an arbitrary bus. However, when selecting the unit address representation for a new bus, it is generally a good idea to choose a form that does not change when other cards are installed or removed, thus ensuring that the pathname of a particular device does not depend on the presence or absence of other unrelated devices. This argues against the use of dynamic allocation for the unit address portion of a device’s address (however, on some buses there may be no choice). Note that the unit address is not the only device address that can be published in a “reg” property; the unit address is the first such address,
but there can be additional ones, and the recommendation against dynamic allocation does not apply to those others, which do not appear in pathnames.

One additional consideration results from device nodes that have no “\texttt{reg}” properties, and thus have no fixed unit address. A pathname selecting such a wildcard node can still have a unit address component; a wildcard node matches any unit address. For an instance created from such a wildcard match, \texttt{my-unit} returns the unit address from the pathname, rather than the unit address from the “\texttt{reg}” property.

\textbf{F.5}Do calls get passed from driver to driver down the device tree, or does the platform’s boot firmware “talk” directly to the leaf driver?

When the platform’s boot firmware or a client program opens a device driver, it gets an instance handle for the leaf driver itself (e.g., the “\texttt{enet}” device driver in figure F.1). Open instances are also created for each of the drivers in the path to the leaf node, giving each node an opportunity to prepare the communication path to its child and a handle to invoke the services of its parent. The high-level code talks directly to the leaf node, and the leaf node asks its parent for help (via the parent’s methods) when needed.

\textbf{F.6}Where is the driver’s state information kept? How does it remember what it learned or set up at probe time and between calls to its methods?

There are generally three places where a driver stores state information: the device tree, static variables, and the open instance context.

The device tree should be used to store properties and methods—things that are of general, long-term interest to other entities and to the driver itself, related to what the driver can do, how to link with it, etc. If a driver must coordinate its activities with other nodes for the same type of device (e.g., identical LAN cards in separate slots), it must rendezvous by searching the device tree, since this is the only mechanism for sharing information among tree nodes. Also, whenever a driver is called, it always has access to its own node, to a handle back to its parent’s node on the device tree, and to the device tree as a whole, giving it access to the parent’s services and knowledge of its current location in the platform’s topology. Since some systems may choose to keep the device tree around when the run-time OS is active, the amount of memory that a driver attaches to the device tree should be small.

Static variables are basically device tree node-global variables. An instance of each variable is set up when the node is probed and it stays around as long as Open Firmware is active. Since it is only node-global and not platform-global, it cannot be used to share information among instances of the same driver for multiple, identical devices. Its main use is for long-term private data structures. Again, since some systems may choose to keep the Open Firmware around while the run-time OS is running, the amount of memory devoted to static variables should also be minimized.

Open-instance context memory is provided to the driver for each currently open instance handle. It is created at open-time and destroyed at close-time. This is where the bulk of the driver’s information should normally be stored. Its main use is for variables, queues, buffers, etc., that are used during and between invocations of the driver’s methods related to the current open instance.

\textbf{F.7}What is a support package?

Open Firmware assumes relatively mature device drivers: Keyboards are expected to provide ISO 8859-1 (Latin) characters (see ISO 8859-1 : 1987 in 2.1), console displays are expected to take these codes and display them, and boot image source devices are expected to load the related image. But many of these devices have several large operations in common. Many LAN devices can use the same boot protocols. Many console devices use simple pixel frame-buffers for which software must “paint” characters. Rather than require the entire functionality for each device to reside in its ROM, Open Firmware provides for support packages that are located in the platform’s ROM and shareable by all interested devices, such as library routines.
To use a support package, a driver locates the package off the root and opens it (with open-package). This effectively makes the open instance of the support package a child of the driver itself. For example, the TFTP boot support package can be opened by the "enet" driver, as shown in figure F.1. The "enet" driver passes the load command on to the load method for the "obp-tftp" support package. The support package performs the load operation, invoking the read and write methods of its parent, the "enet" driver, to send protocol packets and receive protocol and load data over the LAN.

The top three instances in the chain were opened from the pathname, e.g., /sbus/enet", as with open-dev. Client programs or other "applications" invoke the driver via the ihandle returned by open-dev.

The bottom instance in the chain was opened from the package "obp-tftp", as with open-package. The instance that opened the package is responsible for storing its ihandle, allowing calls in the downward direction. Upward calls are done with the ihandle returned by my-parent.

Figure F.1—Support package relationships

F.8 I'm confused by the display drivers: Are they simple frame-buffers or do they contain their own fonts, etc.?

They can be either. Again, as mentioned above, Open Firmware assumes fairly mature device drivers that can directly display characters. Therefore, a display with its own fonts and rendering capability would handle the characters directly. However, there are several support packages supplied for the simpler frame-buffer display devices. In figure F.1, the display driver can write a buffer of characters by invoking the draw-character method of the display low-level interface support package which in turn uses the services of the 8-bit frame-buffer support routines to write the character pixels back into the parent's (i.e., the simple display driver's) frame-buffer.

F.9 What are the relationships among address types?

In general, each bus in a system has its own, possibly orthogonal address space. The number of bits and the assignment of values for each may vary. Some buses have separate memory and I/O register address spaces. Commonly, the system CPU is served by a memory management unit (MMU) that creates a huge, logically contiguous virtual memory space by mapping regions of virtual memory onto dynamically reassigned physical RAM pages. (Notice that FCode programs never directly use any of these address formats. Instead addresses are used as parameters to Open Firmware interface routines that may translate their interpretation as appropriate for the specific hardware platform.)

In this document, addresses directly used by the driver are referred to as virtual addresses. Typical access to system memory is via a virtual address, which may or may not involve mapping to the physical RAM pages by an MMU, depending on how the given platform operates during boot time.
Virtual addresses are also used by the driver to access card registers. Consider the example in figure F.2. In order for the driver to write a control register on the Network I/O card, the virtual address used by the driver must be mapped onto the I/O bus address value that will select the targeted device register when it is asserted on the directly connected expansion I/O bus. Setting up this mapping generally involves calling the device driver for each bus bridge, starting at the bottom of the tree and working up, mapping from one bus’s address space to the next, and possibly setting up mapping registers in each bus bridge to support the driver’s access. The driver for the top bus bridge, in conjunction with the platform’s firmware, finally establishes the mapping into the driver’s virtual address space.

Physical addresses are generally used by an I/O card to access system memory. In order for the Network I/O card in figure F.2 to traverse DMA data structures in system memory, the driver must pass an initial pointer to the I/O card and link together the data structures using address values that are card-relative: i.e., memory address values that will map to the proper system memory locations when the card asserts them on its directly connected expansion I/O bus. The driver would set up these I/O card-relative addresses as follows. First it would use `dma-alloc` to allocate the one or more major blocks in system memory that will contain the data structures. `dma-alloc` returns a virtual address, V0. Next, the driver would call `dma-map-in` in order to translate its virtual address for each overall block into the corresponding card-relative, I/O bus base address, P0. Finally, the driver can translate any of its pointers to a data object in the memory region, into an I/O address value that is usable by the card via simple byte math: P1 = P0 + (V1 – V0).

Incidentally, processor caches, if any, for the DMA memory region are flushed by `dma-map-in`. This is too early in the above scenarios, since the driver is still writing into the DMA data structures. Properly speaking, when the driver is done writing into the DMA structures and before it sends a command to the I/O card related to these structures, it should do a `dma-sync` to flush the processor’s caches out to system memory, where the card can “see” the most recently written values.
F.10 Does Open Firmware support only Ethernet/IEEE 802.3 [B3]?

Historically, OpenBoot systems have mainly worked over Ethernet networks. Therefore, the “network” device type for these devices is more developed and there are more support packages for the related protocols. However, any network link type can be accommodated in one of the following three ways:

— Many link types are similar enough protocol-wise that they can adapt themselves to the current Ethernet/IEEE 802.3 interface, providing link-related translations as appropriate. These links can use the existing support packages (e.g., “obp-tftp”).

— Any link-type can be used for booting using the Open Firmware interfaces, as long as they don’t require the use of the existing support packages for Ethernet/IEEE 802.3. The load command comes directly to the driver anyway.

— Future “network-XXX” device types are expected to be defined. An attractive work item would be a more “generic” network link interface. This would support a suite of load protocol support packages that would work with almost any link.

F.11 Does Open Firmware expect that a system has a platform-global network MAC address?

No. The “local-mac-address” property is used initially to inform the platform of the LAN card’s factory-default MAC address. The mac-address method is used by the LAN card’s driver to determine whether or not to override the factory preset with an alternative value. It is up to the platform to determine what MAC address values should be used by specific LAN cards. These could be left at their factory settings, overridden from configuration tables, or set to a machine-global MAC value, if appropriate.

F.12 Does the ROM in a plug-in card contain ASCII Forth code?

Not really. Only the human user interface uses Forth directly in ASCII form. The ROM on a plug-in card contains FCode. FCode is semantically similar to Forth source code, but the representation is different. Whereas Forth source code is written as a series of human-readable text strings, FCode is a series of binary byte codes. The set of predefined FCode byte codes encompasses most of the core words of ANSI X3.215-1994 (omitting only those Forth words that are meaningful only for Forth source code in its text form) and adds additional functions specific to the Open Firmware environment. A developer typically writes an FCode program in Forth source form (ASCII text), which is then translated into the FCode bytes by a tokenizer program, thus producing the image that is stored in the card ROM.

During the development and debugging phase, an FCode program does not necessarily need not be converted from source form to binary form. If an Open Firmware user interface with the Firmware Debugging command group is available, the FCode program can be tested directly in source form. The tokenization step could then be delayed until the program is fully debugged and ready for installation on production units.

F.13 How does Open Firmware pass interrupts to the device driver?

It doesn’t! Open Firmware is based on a very simple, boot-time I/O model. I/O devices are polled for completion. The interrupt specifications in Open Firmware are mainly used to pass system information to the run-time OS. Also the process model is single-threaded; there is only one, non-preempted program running at a time. For multi-CPU machines, only one processor is assumed to be actively running Open Firmware at boot-time.
F.14 Can an Open Firmware firmware design be undermined by a provision added by a new Instruction Set Architecture (ISA) or bus annex?

Drivers for plug-in I/O cards should use only the FCode numbers documented in this Open Firmware document, all of which are required in conforming platforms. New ISA specifications should not add to this list, since drivers written using new FCode numbers would not port to other ISAs. Similarly, I/O cards and their drivers should only have to comply with the bus annex that presents the requirements for the bus to which the I/O card is directly attached.

However, in platforms with multiple-bus hierarchies, as in figure F.2, the hardware and driver design of the bus bridges and their drivers must ensure that all Open Firmware and bus annex facilities are accessible when bus transactions must traverse intermediate buses. In other words, the expansion bus bridge in figure F.2 must ensure that all Open Firmware facilities can be successfully passed through the primary I/O bus to the expansion I/O bus.
Annex G
Summary lists
(informative)

The following summary lists are provided for the reader’s convenience. These lists are not definitive; in the event of discrepancies between them and the information contained elsewhere in this document, the other information has precedence.

G.1 Configuration variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>auto-boot?</td>
<td>If true, boot automatically after power-on or reset-all.</td>
</tr>
<tr>
<td>boot-device</td>
<td>Default device-name for boot, if diagnostic-mode? is false.</td>
</tr>
<tr>
<td>boot-file</td>
<td>Default arguments for boot, if diagnostic-mode? is false.</td>
</tr>
<tr>
<td>diag-device</td>
<td>Default device-name for boot, if diagnostic-mode? is true.</td>
</tr>
<tr>
<td>diag-file</td>
<td>Default arguments for boot, if diagnostic-mode? is true.</td>
</tr>
<tr>
<td>diag-switch?</td>
<td>If true, diagnostic-mode? returns true.</td>
</tr>
<tr>
<td>fcode-debug?</td>
<td>If true, save names for FCodes with headers.</td>
</tr>
<tr>
<td>input-device</td>
<td>Default console input device.</td>
</tr>
<tr>
<td>nvramrc</td>
<td>Contents of the script.</td>
</tr>
<tr>
<td>oem-baner</td>
<td>Contain custom banner text, enabled by oem-baner?.</td>
</tr>
<tr>
<td>oem-baner?</td>
<td>If true, banner displays custom message in oem-baner.</td>
</tr>
<tr>
<td>oem-logo</td>
<td>Contain custom logo for banner, enabled by oem-logo?.</td>
</tr>
<tr>
<td>oem-logo?</td>
<td>If true, banner displays custom logo in oem-logo.</td>
</tr>
<tr>
<td>output-device</td>
<td>Default console output device.</td>
</tr>
<tr>
<td>screen-#columns</td>
<td>Maximum number of columns on console output device.</td>
</tr>
<tr>
<td>screen-#rows</td>
<td>Maximum number of rows on console output device.</td>
</tr>
<tr>
<td>security-#badlogins</td>
<td>Contain total count of invalid security access attempts.</td>
</tr>
<tr>
<td>security-mode</td>
<td>Contain level of security access protection.</td>
</tr>
<tr>
<td>security-password</td>
<td>Contain security password text string.</td>
</tr>
<tr>
<td>selftest-#megas</td>
<td>Number of megabytes of memory to test.</td>
</tr>
<tr>
<td>use-nvramrc?</td>
<td>If true, the script is evaluated at system start-up.</td>
</tr>
</tbody>
</table>

G.2 Assigned FCode numbers

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>end0</td>
</tr>
<tr>
<td>0x01-</td>
<td>Beginning codes of 2-byte FCode sequences</td>
</tr>
<tr>
<td>0x0F</td>
<td>b(lit)</td>
</tr>
<tr>
<td>0x11</td>
<td>b(')</td>
</tr>
<tr>
<td>0x12</td>
<td>b(')</td>
</tr>
<tr>
<td>0x13</td>
<td>bbranch</td>
</tr>
<tr>
<td>0x14</td>
<td>b?branch</td>
</tr>
<tr>
<td>0x15</td>
<td>b(loop)</td>
</tr>
<tr>
<td>0x16</td>
<td>b(+loop)</td>
</tr>
<tr>
<td>0x17</td>
<td>b(do)</td>
</tr>
<tr>
<td>0x18</td>
<td>b(?do)</td>
</tr>
<tr>
<td>0x19</td>
<td>i</td>
</tr>
<tr>
<td>0x1A</td>
<td>j</td>
</tr>
<tr>
<td>0x1B</td>
<td>b(leave)</td>
</tr>
<tr>
<td>0x1C</td>
<td>b(of)</td>
</tr>
<tr>
<td>0x1D</td>
<td>execute</td>
</tr>
<tr>
<td>0x1E</td>
<td>+</td>
</tr>
<tr>
<td>0x1F</td>
<td>-</td>
</tr>
<tr>
<td>0x20</td>
<td>*</td>
</tr>
<tr>
<td>0x21</td>
<td>/</td>
</tr>
<tr>
<td>0x22</td>
<td>mod</td>
</tr>
<tr>
<td>0x23</td>
<td>and</td>
</tr>
<tr>
<td>0x24</td>
<td>or</td>
</tr>
<tr>
<td>0x25</td>
<td>xor</td>
</tr>
<tr>
<td>0x26</td>
<td>invert</td>
</tr>
<tr>
<td>0x27</td>
<td>lshift</td>
</tr>
<tr>
<td>0x28</td>
<td>rshift</td>
</tr>
<tr>
<td>0x29</td>
<td>&gt;&gt;a</td>
</tr>
<tr>
<td>0x2A</td>
<td>/mod</td>
</tr>
<tr>
<td>0x2B</td>
<td>u/mod</td>
</tr>
<tr>
<td>0x2C</td>
<td>negate</td>
</tr>
<tr>
<td>0x2D</td>
<td>abs</td>
</tr>
<tr>
<td>0x2E</td>
<td>min</td>
</tr>
<tr>
<td>0x2F</td>
<td>max</td>
</tr>
<tr>
<td>0x30</td>
<td>&gt;&gt;r</td>
</tr>
<tr>
<td>0x31</td>
<td>r&gt;</td>
</tr>
<tr>
<td>0x32</td>
<td>z@</td>
</tr>
<tr>
<td>0x33</td>
<td>exit</td>
</tr>
</tbody>
</table>
IEEE Standard for Boot (Initialization Configuration) Firmware:

0x34 0= 0x72 !
0x35 0< 0x73 1!
0x36 0< 0x74 w!
0x37 0< 0x75 c!
0x38 0< 0x76 2@
0x39 0< 0x77 2!
0x3A 0> 0x78 move
0x3B 0> 0x79 fill
0x3C 0= 0x7A comp
0x3D 0<> 0x7B noop
0x3E u> 0x7C lwsplit
0x3F u<= 0x7D wjsplit
0x40 0<> 0x7E b(<mark)
0x41 0< 0x7F b(<)
0x42 >= 0x80 b(resolve)
0x43 <= 0x81 set-token-table
0x44 = 0x82 set-table
0x45 <=> 0x83 new-token
0x46 <= 0x84 b(;
0x47 <= 0x85 b(>
0x48 <= 0x86 b(create)
0x49 <= 0x87 b(defer)
0x4A <= 0x88 b(buffer:)
0x4B <= 0x89 b(field)
0x4C <= 0x8A b(value)
0x4D <= 0x8B b(constant)
0x4E <= 0x8C b(constant)
0x4F <= 0x8D b(constant)
0x50 <= 0x8E b(constant)
0x51 <= 0x8F b(constant)
0x52 <= 0x90 b(constant)
0x53 <= 0x91 b(constant)
0x54 <= 0x92 b(constant)
0x55 <= 0x93 b(constant)
0x56 <= 0x94 b(constant)
0x57 <= 0x95 b(constant)
0x58 <= 0x96 b(constant)
0x59 <= 0x97 b(constant)
0x5A <= 0x98 b(constant)
0x5B <= 0x99 b(constant)
0x5C <= 0x9A b(constant)
0x5D <= 0x9B b(constant)
0x5E <= 0x9C b(constant)
0x5F <= 0x9D b(constant)
0x60 <= 0x9E b(constant)
0x61 <= 0x9F b(constant)
0x62 <= 0xA0 b(constant)
0x63 <= 0xA1 b(constant)
0x64 <= 0xA2 b(constant)
0x65 <= 0xA3 b(constant)
0x66 <= 0xA4 b(constant)
0x67 <= 0xA5 b(constant)
0x68 <= 0xA6 b(constant)
0x69 <= 0xA7 b(constant)
0x6A <= 0xA8 b(constant)
0x6B <= 0xA9 b(constant)
0x6C <= 0xAA b(constant)
0x6D <= 0xAB b(constant)
0x6E <= 0xAC b(constant)
0x6F <= 0xAD b(constant)
0x70 <= 0xAE b(constant)
0x71 <= 0xAF b(constant)

IEEE Std 1275-1994
<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x240</td>
<td>left-parse-string</td>
</tr>
<tr>
<td>0x241</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x241</td>
<td>0x5FF</td>
</tr>
<tr>
<td>0x600</td>
<td>Vendor FCodes</td>
</tr>
<tr>
<td>0x7FF</td>
<td></td>
</tr>
<tr>
<td>0x800</td>
<td>Local FCodes</td>
</tr>
<tr>
<td>0xFFF</td>
<td></td>
</tr>
</tbody>
</table>

* These are historical FCodes.
† These are obsolete FCodes.
Annex H
Historical notes

(informative)

H.1 Overview and references

This Open Firmware standard is based on the following documents:
— OpenBoot Command Reference [B5]
— Writing FCode Programs [B7]

H.2 Obsolete FCodes

H.2.1 Previously implemented FCodes

Pre-Open Firmware versions of SBus [B2] firmware have used the following FCodes. None of these are required for a Open Firmware implementation; however, some implementations may choose to support these for the purpose of backwards compatibility.

H.2.1.1 Generic 1-bit frame-buffer support

The “fb1” generic frame-buffer support package implements the display device low-level interfaces for framebuffers with one memory bit per pixel. It applies only to frame buffers organized as a series of doublets with big-endian addressing, with the most significant bit within a doublet corresponding to the leftmost pixel within the group of sixteen pixels controlled by that doublet. In normal (not inverse) video mode, background pixels are drawn with zero-bits, and foreground pixels with one-bits.

The working group committee feels that this class of devices is too restricted to justify requiring the presence of this set of support routines in all implementations of Open Firmware. Furthermore, the working group believes that the number of new devices that fit into this category is dwindling rapidly. However, there are a number of existing SBus devices that use these support routines. An implementation that intends to support those existing devices is advised to implement the following FCode functions.

Execution of fb1-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.

fb1-blank-screen ( -- ) F,O 0x174
    Implement the “fb1” blank-screen function.

Typically implemented as: fb1-invert-screen fb1-invert-screen

NOTE—Typical generic implementations of this function are likely to be quite slow, since they probably will access each pixel on the screen four times. For most devices, there is a device-specific implementation for the blank-screen function that is much faster, for example disabling video output for about 20 ms. It is recommended that such device-specific implementations be used instead of the generic fb1-blank-screen function.

fb1-delete-characters ( n -- ) F,O 0x177
    Implement the “fb1” delete-characters function.

fb1-delete-lines ( n -- ) F,O 0x179
    Implement the “fb1” delete-lines function.
IEEE Std 1275-1994
IEEE STANDARD FOR BOOT (INITIALIZATION CONFIGURATION) FIRMWARE:

**fb1-draw-character**  
( char -- ) F,O 0x170  
Implement the “fb1” draw-character function.

**fb1-draw-logo**  
( line# addr width height -- ) F,O 0x17A  
Implement the “fb1” draw-logo function.

**fb1-erase-screen**  
( -- ) F,O 0x173  
Implement the “fb1” erase-screen function.

**fb1-insert-characters**  
( n -- ) F,O 0x176  
Implement the “fb1” insert-characters function.

**fb1-insert-lines**  
( n -- ) F,O 0x178  
Implement the “fb1” insert-lines function.

**fb1-install**  
( width height #columns #lines -- ) F,O 0x17B  
Install all built-in generic 1-bit frame-buffer routines.

Install the “fb1” generic 1-bit frame-buffer routines into the display device interface defer words, configuring the “fb1” routines for a frame-buffer height pixels high, with successive scan lines width pixels apart. #columns and #lines indicate the maximum number of text columns and lines that the device is capable of supporting(#columns and #lines usually depend upon the width and height of the font to be used, among other things.) width is the difference between the starting memory addresses of two consecutive scan lines in the frame-buffer, multiplied by eight (the number of pixels per byte). For frame-buffers where all memory locations correspond to displayable pixels, this is the same as the width of the screen in pixels. height is the height of the display in scan lines.

Set screen-width to the width argument, screen-height to the height argument, #columns to the minimum of #columns and screen-#columns, and #lines to the minimum of #lines and screen-#rows.

Set window-top and window-left to center the text region on the screen (the calculation typically involves #columns, #lines, char-width, char-height, screen-width, and screen-height). The calculation assumes that width pixels per scan line are displayable. If some are not (for example, some number of pixels at the right of the display), it is the responsibility of the display driver to adjust window-left to locate the text region in an appropriate place after fb1-install returns.

Usage restriction: char-width and char-height must be set before fb1-install is executed; otherwise, the centering is likely to be incorrect.

See also: set-font

**fb1-invert-screen**  
( -- ) F,O 0x175  
Implement the “fb1” invert-screen function.

**fb1-reset-screen**  
( -- ) F,O 0x171  
Implement the “fb1” reset-screen function.

This routine is usually implemented as a no-op.

**fb1-slide-up**  
( n -- ) F,O 0x17C  
Like fb1-delete-lines, but do not erase lines.

Delete n lines at and below the cursor line, as with delete-lines, except do not erase the n lines at the bottom of the screen. The typical use for this command is to scroll the enable plane for frame-buffers with separate overlay and enable planes.

**fb1-toggle-cursor**  
( -- ) F,O 0x172  
Implement the “fb1” toggle-cursor function.
### H.2.1.2 Other previously implemented FCodes

**dma-alloc**  \((\text{#bytes} \rightarrow \text{virtual})\)  F,O  0x101  

Used to allocate some memory for DMA or other purposes.

Some existing FCode programs use \texttt{dma-alloc} to allocate memory for general purposes not intended for DMA. Programs using this technique are not guaranteed to work.

**Equivalent to**: "\texttt{dma-alloc} $\text{call-parent}$"  

**See**: \texttt{alloc-mem}.

**driver**  \((\text{addr len} \rightarrow)\)  F,O  0x118  

Creates the “\texttt{name}” property.

Removes the manufacturer name prefix from the string \texttt{addr len}, then creates the “\texttt{name}” property from the remainder of the string. Previous versions of SBus firmware have implemented the process of removing the manufacturer name prefix in inconsistent ways; thus, there is no single definition of \texttt{driver} that will ensure backwards compatibility in all cases.

**NOTE—**SBus [B2] developers were advised to avoid the use of this FCode function when the inconsistency was discovered, and the committee believes that its use has largely been eliminated.

**fcode-version**  \((-\rightarrow n)\)  F,O  0x212  

Return revision level of device interface (obsolete).

This obsolete FCode has a behavior similar to \texttt{fcode-revision}.

**firmware-version**  \((-\rightarrow n)\)  F,O  0x211  

Return revision level of the OpenBoot Firmware.

Encode the value as two doublets, holding the major/minor release number. For example, if the release number was 2.12, return the value \texttt{0x0002.000C}.

The allocation of version numbers is determined by the implementor of the Open Firmware and is not specified in this document.

This FCode is obsolete.

**group-code**  \((-\rightarrow a\text{-addr})\)  F,O  0x123  

Group offset for memory-test-suite (obsolete).

A \texttt{variable} containing a group offset for distinguishing various self-tests in early versions of \texttt{memory-test-suite}.

The value in \texttt{group-code} is added to the code# for individual tests, and the sum is displayed with \texttt{display-status}.

**intr**  \((\text{sbus-interrupt# vector} \rightarrow)\)  F,O  0x117  

Creates the “\texttt{intr}” property.

See the description of the “\texttt{intr}” property for more details.

**memmap**  \((\text{physoffset space size} \rightarrow \text{virtual})\)  F,O  0x104  

Creates a memory mapping for some locations (obsolete.)

**my-params**  \((-\rightarrow addr len)\)  F,O  0x10F  

Contents of custom parameters (obsolete).

\textit{addr} is the address and \textit{len} the length of the value of the “\texttt{params}” property of the \texttt{active package}, or a zero-length string if the \texttt{active package} has no “\texttt{params}” property.

**“params”**  S,O  

Standard \textit{property name} to set device-dependent modes.

This property, if present, holds the value to be passed by the obsolete FCode \texttt{my-params}.
>physical ( virtual -- physoffset space ) F,O 0x106
Return physical address for virtual address (obsolete).

Given a virtual address, return the mapped physical address as a (physoffset space) pair.

probe ( arg-addr arg-len reg-addr reg-len fcode-addr fcode-len -- ) F,O 0x238
Execute FCode at given location (obsolete).

Execute FCode at location given by fcode-addr fcode-len, passing arguments arg-addr arg-len and with registers reg-addr reg-len. fcode-addr, fcode-len and reg-addr, reg-len are the text representations of physical addresses within the address space of the active package.

probe-virtual ( arg-addr arg-len reg-addr reg-len fcode-addr -- ) F,O 0x239
Like probe, but FCode is located at virtual address fcode-addr.

processor-type ( -- processor-type ) F,O 0x210
Returns type of CPU (obsolete).

Returns the type of processor (instruction set architecture). 0x5 indicates SPARC, other values are not used.

H.2.2 Non-implemented FCodes

Pre-Open Firmware systems assigned the following FCode numbers, but the functions were not supported. To avoid any possible confusion, however, these FCode numbers are reserved and should not be reassigned.

<table>
<thead>
<tr>
<th>Function</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>adr-mask</td>
<td>0x229</td>
</tr>
<tr>
<td>b(code)</td>
<td>0xBF</td>
</tr>
<tr>
<td>4-byte-id</td>
<td>0xFE</td>
</tr>
<tr>
<td>convert</td>
<td>0xA1</td>
</tr>
<tr>
<td>frame-buffer-busy?</td>
<td>0x156</td>
</tr>
<tr>
<td>poll-packet</td>
<td>0x1A2</td>
</tr>
<tr>
<td>return-buffer</td>
<td>0x1A0</td>
</tr>
<tr>
<td>set-token-table</td>
<td>0xB3</td>
</tr>
<tr>
<td>set-table</td>
<td>0xB4</td>
</tr>
<tr>
<td>VME support words</td>
<td>0x190-0x196</td>
</tr>
<tr>
<td>xmit-packet</td>
<td>0x1A1</td>
</tr>
</tbody>
</table>

H.3 Obsolete properties

Pre-Open Firmware versions of SBus [B2] firmware used the following properties. None of these are required for an Open Firmware implementation; however, some implementations may choose to support these for the purpose of backwards compatibility.

“params” S,O
Standard property name to contain my-params data (obsolete).
Prop-encoded-array:
Data array, encoded with encode-bytes.
This property, if present, holds the value to be passed by the obsolete FCode my-params.

“scsi-initiator-id” S,O
Standard property name to contain SCSI host address (obsolete).
prop-encoded-array:
Integer, encoded with encode-int.
This property, if present, contains an integer 0–15 indicating the address of the main SCSI host adapter of the system.
H.4 New FCodes and methods

Most pre-Open Firmware systems do not implement the following FCodes and methods:

<table>
<thead>
<tr>
<th>FCode#</th>
<th>Name</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xC7</td>
<td>#</td>
<td>Not the same as old # (now called u#).</td>
</tr>
<tr>
<td>0xC9</td>
<td>#&gt;</td>
<td>Not the same as old #&gt; (now called u#&gt;).</td>
</tr>
<tr>
<td>0xC8</td>
<td>#s</td>
<td>Not the same as old #s (now called u#s).</td>
</tr>
<tr>
<td>0xDE</td>
<td>behavior</td>
<td></td>
</tr>
<tr>
<td>0x23E</td>
<td>byte-load</td>
<td>On pre-Open Firmware, &quot;byte-load&quot; $find could be used.</td>
</tr>
<tr>
<td>0xDD</td>
<td>compile,</td>
<td>On pre-Open Firmware, &quot;(compile)&quot; $find could be used.</td>
</tr>
<tr>
<td>0x128</td>
<td>decode-phys</td>
<td></td>
</tr>
<tr>
<td>0xDA</td>
<td>get-token</td>
<td></td>
</tr>
<tr>
<td>method</td>
<td>encode-unit</td>
<td></td>
</tr>
<tr>
<td>0x227</td>
<td>lbflip</td>
<td></td>
</tr>
<tr>
<td>0x228</td>
<td>lbflips</td>
<td></td>
</tr>
<tr>
<td>0x229</td>
<td>lwflip</td>
<td>On pre-Open Firmware, the “wflip” tokenizer macro was used.</td>
</tr>
<tr>
<td>0x23D</td>
<td>next-property</td>
<td></td>
</tr>
<tr>
<td>0x23F</td>
<td>set-args</td>
<td>On pre-Open Firmware, &quot;set-args&quot; $find could be used.</td>
</tr>
<tr>
<td>0xDB</td>
<td>set-token</td>
<td></td>
</tr>
<tr>
<td>0xDC</td>
<td>state</td>
<td>On pre-Open Firmware, &quot;state&quot; $find could be used.</td>
</tr>
<tr>
<td>0x89</td>
<td>unloop</td>
<td></td>
</tr>
</tbody>
</table>

H.5 New properties

Standard meanings for most of the following properties were introduced by this standard:

```
"#address-cells" Standard property to define the package’s address format.
"address-bits" Standard property to indicate number of network address bits.
"bootargs" Standard property containing the chosen boot command arguments.
"bootpath" Standard property containing the chosen boot device-path.
"character-set" Standard property to specify the character set for this device.
"compatible" Standard property to define alternate `name` property values.
"max-frame-size" Standard property to indicate maximum allowable packet size.
"#size-cells" Standard property to define the package’s address size format.
"status" Standard property to indicate the operational status of this device.
"stdin" Standard property containing the ihandle of the console input device.
"stdout" Standard property containing the ihandle of the console output device.
```

H.6 New user interface commands

Most pre-Open Firmware systems do not implement the following user interface commands.
apply ( ... "method-name< >device-specifier< >" -- ??? ) Execute named method in the specified package.
char ( "text< >" -- char ) Generate numeric code for next character from input buffer.
[char] (C: [text< >] -- ) Generate numeric code for next character from input buffer.
close-dev ( ihandle -- ) Close device and all of its parents.
$/create (E: -- a-addr ) Call create; new name specified by name string.
environment? ( str len -- false | value true ) Return system information based on input keyword.
f/m/mod ( d n -- rem quot ) Divide d by n.
noshowstack ( -- ) Turn off showstack (automatic stack display).
parse ( delim "text<delim>" -- str len ) Parse text from the input buffer, delimited by delim.
parse-word ( "text< >" -- str len ) Parse text from the input buffer, delimited by white space.
(patch) ( new-n1 num1? old-n2 num2? xt -- ) Change contents of command indicated by xt.
postpone (C: [old-name< >] -- ) Delay execution of the immediately following command.
recurse ( ... -- ??? ) Compile recursive call to the command being compiled.
s" ( [text<"] -- text-str text-len ) Gather the immediately following string.
s>d ( n1 -- d1 ) Convert a number to a double number.
sm/rem ( d n -- rem quot ) Divide d by n, symmetric division.
status ( -- ) defer word that can be used to modify the user interface prompt.

H.7 FCode name changes

The following FCodes names have changed from their pre-Open Firmware versions for clarity and consistency. While this can affect the tokenizer and/or user interface behavior, the actual behavior of the function associated with that FCode number has not changed. Existing (already-tokenized) FCode programs that use these FCodes will be unaffected.

Items marked with a * have retained the old name, as a synonym.

<table>
<thead>
<tr>
<th>Old Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>u#</td>
</tr>
<tr>
<td>#&gt;</td>
<td>u#&gt;</td>
</tr>
<tr>
<td>#s</td>
<td>u#s</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>lshift *</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>rshift *</td>
</tr>
<tr>
<td>attribute</td>
<td>property</td>
</tr>
<tr>
<td>/c*</td>
<td>chars *</td>
</tr>
<tr>
<td>ca1+</td>
<td>char+ *</td>
</tr>
<tr>
<td>decode-2int</td>
<td>parse-2int</td>
</tr>
<tr>
<td>delete-attribute</td>
<td>delete-property</td>
</tr>
<tr>
<td>eval</td>
<td>evaluate *</td>
</tr>
<tr>
<td>flip</td>
<td>wbflip</td>
</tr>
<tr>
<td>get-inherited-attribute</td>
<td>get-inherited-property</td>
</tr>
<tr>
<td>get-my-attribute</td>
<td>get-my-property</td>
</tr>
<tr>
<td>get-package-attribute</td>
<td>get-package-property</td>
</tr>
<tr>
<td>is</td>
<td>to</td>
</tr>
<tr>
<td>lflips</td>
<td>lwflips</td>
</tr>
<tr>
<td>map-sbus</td>
<td>map-low</td>
</tr>
<tr>
<td>nal+</td>
<td>cells+ *</td>
</tr>
<tr>
<td>/n*</td>
<td>cells *</td>
</tr>
<tr>
<td>not</td>
<td>invert *</td>
</tr>
<tr>
<td>u*x</td>
<td>um*</td>
</tr>
<tr>
<td>version</td>
<td>fcode-revision</td>
</tr>
<tr>
<td>wflips</td>
<td>wbflips</td>
</tr>
</tbody>
</table>
H.8 User interface name changes

The following user interface command names have changed from their pre-Open Firmware versions, with no change in behavior.

<table>
<thead>
<tr>
<th>Old name</th>
<th>New name</th>
</tr>
</thead>
<tbody>
<tr>
<td>.attributes</td>
<td>.properties</td>
</tr>
<tr>
<td>cd</td>
<td>dev</td>
</tr>
<tr>
<td>reset</td>
<td>reset-all</td>
</tr>
<tr>
<td>select-dev</td>
<td>open-dev</td>
</tr>
<tr>
<td>unselect-dev</td>
<td>device-end</td>
</tr>
</tbody>
</table>

H.9 Other variances from the Open Firmware standard

The following items describe additional areas where existing pre-Open Firmware implementations may not comply with provisions of this specification. This list is not exhaustive.

H.9.1 dl command

In some existing pre-Open Firmware implementations, the dl command receives text from a specific serial line device regardless of the device that is the current input source.

H.9.2 Client interface

Most pre-Open Firmware implementations have a different client interface.

H.9.3 byte-load command

In some existing pre-Open Firmware implementations, byte-load does not save and restore the tables that map program-defined FCode functions to their assigned FCode numbers. On such implementations, an FCode program that executes byte-load cannot depend on being able to interpret any of its program-defined FCode functions after byte-load returns. However, it can continue to execute code that was previously compiled into a definition that called byte-load, and it can interpret system-defined FCode functions.

byte-load is not an FCode on many systems. Use:

```
" byte-load" $find drop execute
```

to achieve the equivalent effect.

Previous usage of byte-load does not support the execution token semantics of the “xt” parameter. Instead, the value of that parameter must always be 1.
H.9.4 *fcode-revision* command

This FCode returns the revision level of the FCode *device interface* (i.e., which FCodes are supported). Systems which support Open Firmware will return a value of (hex) 0003.0000 (i.e., 3.0), or possibly greater as may be required by future editions of this specification.

OpenBoot version 2.x systems return a similar encoding, i.e., (hex) 0002.00xx. OpenBoot version 1.x systems return a value of (hex) 0000.xxxx.

H.9.5 “hierarchical” devices

Some existing pre-Open Firmware implementations of SBus [B2] declare a “device_type” *property* value of “hierarchical” instead of the required “sbus”.

Some existing pre-Open Firmware implementations of SCSI devices declare a “device_type” *property* value of “hierarchical” instead of the suggested “scsi”.

H.9.6 *within* command

The definition of *within* has been changed slightly to conform with ANS Forth. The change affects only the behavior for arguments spanning the barrier between positive and negative numbers, i.e., 0x8000.0000. Ordinary usage is not affected.

H.9.7 "hex strings"

Some pre-Open Firmware user interfaces and *tokenizers* do not support embedded hex values within a "string. However, an *FCode program* that used a newer tokenizer to create such a string will operate *property*, even in an older system that does not recognize such a construct from the user interface.

H.9.8 *noshowstack* command

*noshowstack* is not supported in most pre-Open Firmware systems. However, the *status* command, while not documented, may still be used on these systems to vary the behavior at the *ok* prompt. Alternatively, a power-cycle may be used to turn off a *showstack*. Or, on some systems, *showstack* has a toggled behavior.

H.9.9 Path resolution

The path resolution algorithm has been refined to allow embedded alias names and to correct other problems. Typical usage should be unaffected.

H.9.10 “name” property

The “name” *property* now recommends a 6-digit Organizationally Unique Identifier (OUI), as well as the stock symbol identifier.

H.9.11 New standard system nodes

The /chosen standard system node is new.
H.9.12  Version 1.x

The first OpenBoot systems shipped were numbered as version “1.x”. While similar in many respects to Open Firmware, a number of features were not supported, in addition to the other differences previously listed. For detailed information on FCode version 1.x, consult Writing FCode Programs [B7].
# Annex I
## Index of Open Firmware glossary terms

(informative)

<table>
<thead>
<tr>
<th>Term</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>49, 73, 103</td>
</tr>
<tr>
<td>--bp</td>
<td>94, 119</td>
</tr>
<tr>
<td>-1</td>
<td>52, 106</td>
</tr>
<tr>
<td>-bp</td>
<td>94, 119</td>
</tr>
<tr>
<td>-rot</td>
<td>49, 72, 178</td>
</tr>
<tr>
<td>-trailing</td>
<td>77, 189</td>
</tr>
<tr>
<td>map</td>
<td>158</td>
</tr>
<tr>
<td>!</td>
<td>50, 58, 75, 101, 240</td>
</tr>
<tr>
<td>&quot;</td>
<td>77, 102, 111</td>
</tr>
<tr>
<td>&quot;#address-cells&quot;</td>
<td>23, 108, 175, 247</td>
</tr>
<tr>
<td>&quot;#address-cells&quot;</td>
<td>175</td>
</tr>
<tr>
<td>&quot;#size-cells&quot;</td>
<td>159, 173, 175</td>
</tr>
<tr>
<td>&quot;size-cells&quot;</td>
<td>23, 159, 175, 184</td>
</tr>
<tr>
<td>&quot;/&quot;</td>
<td>103</td>
</tr>
<tr>
<td>&quot;/aliases&quot;</td>
<td>109</td>
</tr>
<tr>
<td>&quot;/chosen&quot;</td>
<td>125</td>
</tr>
<tr>
<td>&quot;/openprom&quot;</td>
<td>169</td>
</tr>
<tr>
<td>&quot;/options&quot;</td>
<td>169</td>
</tr>
<tr>
<td>&quot;address-bits&quot;</td>
<td>107</td>
</tr>
<tr>
<td>&quot;address&quot;</td>
<td>20, 55, 107, 145</td>
</tr>
<tr>
<td>&quot;address-bits&quot;</td>
<td>27</td>
</tr>
<tr>
<td>&quot;available&quot;</td>
<td>24, 27, 110, 125, 140, 159, 176</td>
</tr>
<tr>
<td>&quot;block&quot;</td>
<td>25, 26, 27, 117, 132</td>
</tr>
<tr>
<td>&quot;bootargs&quot;</td>
<td>85, 118</td>
</tr>
<tr>
<td>&quot;bootpath&quot;</td>
<td>84, 119</td>
</tr>
<tr>
<td>&quot;byte&quot;</td>
<td>25, 26, 121, 132</td>
</tr>
<tr>
<td>&quot;character-set&quot;</td>
<td>124</td>
</tr>
<tr>
<td>&quot;compatible&quot;</td>
<td>20, 127, 247</td>
</tr>
<tr>
<td>&quot;deblocker&quot;</td>
<td>24, 25, 26, 28, 117, 121, 129</td>
</tr>
<tr>
<td>&quot;decode-unit&quot;</td>
<td>108, 182</td>
</tr>
<tr>
<td>&quot;device_type&quot;</td>
<td>6, 20, 24, 25, 56, 117, 121, 132, 134, 159, 163, 182, 187, 200, 208, 25, 27, 117, 133, 170</td>
</tr>
<tr>
<td>&quot;disk-label&quot;</td>
<td>25, 27, 117, 133, 170</td>
</tr>
<tr>
<td>&quot;display&quot;</td>
<td>25, 29, 56, 112, 132, 134, 136</td>
</tr>
<tr>
<td>&quot;draw-logo&quot;</td>
<td>136</td>
</tr>
<tr>
<td>&quot;existing&quot;</td>
<td>24, 110, 140</td>
</tr>
<tr>
<td>&quot;fb1&quot;</td>
<td>31, 48, 125, 126, 134, 145, 151, 155, 180, 183, 193, 243</td>
</tr>
<tr>
<td>&quot;fb8&quot;</td>
<td>25, 29, 31, 32, 58, 125, 126, 134, 142, 145, 151, 155, 180, 183, 193</td>
</tr>
<tr>
<td>&quot;font&quot;</td>
<td>25</td>
</tr>
</tbody>
</table>

- "Hierarchical" 249
- "Interrupts" 20, 150, 180, 202, 229
- "intr" 150, 151, 180, 245
- "load" 26, 163
- "local-mac-address" 27, 156, 157, 163, 236
- "mac-address" 27, 156, 158, 163
- "max-frame-size" 159
- "max-frame-size" 27
- "memory" 27, 159, 182, 189
- "mmu" 23
- "model" 20, 56, 160
- "network" 25, 26, 28, 132, 156, 158, 159, 163, 174, 236
- "obp-tftp" 26, 28, 163, 167, 233, 236
- "open" 126
- "packages" 144, 170
- "params" 245
- "ranges" 23, 173, 200
- "reg" 20, 24, 27, 40, 41, 56, 110, 140, 159, 161, 175, 184, 187, 200, 202, 229, 232
- "relative-addressing" 176
- "restore" 177
- "ring-bell" 116
- "screen" 89
- "scsi-2" 208
- "selftest" 59
- "serial" 25, 27, 132, 182
- "status" 20, 186
- "stdin" 87, 186
- "stdout" 87, 186

# 50, 78, 102, 190, 240, 248
#> 50, 78, 102, 240, 248
#address-cells 130, 138, 158, 161
#columns 29, 32, 56, 126, 142, 195, 241, 243, 244
#line 52, 77, 141, 155, 240
#lines 29, 32, 56, 142, 155, 241, 243, 244
#out 52, 77, 169, 240
#s 50, 78, 179, 240, 248
$call-method 52, 55, 64, 123, 141, 155, 163, 164
IEEEStd 1275-1994
IEEE STANDARD FOR BOOT (INITIALIZATION CONFIGURATION) FIRMWARE:

$call-parent 55, 123 .instruction 94, 119, 150, 186
$call-method 241 .properties 90, 171, 249
$call-parent 241 .r 50, 78, 172, 240
$callback 67, 89, 123 .registers 92, 175
$create 81, 128, 247 .s 50, 78, 179, 184, 240
$find 52, 144, 240, 249 .step 94, 186
$number 52, 77, 165, 240
$nvalias 89, 165 / 20, 49, 73, 103, 239
$nvunalias 89, 166 /aliases 20
$open-package 27, 54, 169 /c 50, 74, 122, 124, 125,
$open-package 241 240
$setenv 20, 86, 164, 166, 167, 183 /c* 74, 122, 248
$sift 91, 184 /chosen 21, 23, 61, 84, 85, 87, 182,

' 82, 102, 109, 111 /L 240
( 76, 102 /l 51, 74, 153, 154, 157
(.) 78, 103, 104, 190 /l* 51, 74, 153, 240
(cr 52, 128, 240 /mod 49, 73, 160, 239
(debug 92, 130 240
(is-user-word) 53, 152 /n 50, 74, 124, 161, 162,
(is-user-word) 241 240
(patch) 91, 170, 248 /openprom 20, 176
(see) 91, 181 /options 20, 86
(u.) 78, 104, 190 /packages 9, 17, 21, 22, 27, 44

* 49, 73, 103, 239 /w 51, 74, 192, 193, 240
*/ 73, 103 /w* 51, 74, 193, 240
*/mod 73, 160

+ 49, 73, 103, 239 /n 81, 104, 111
+! 50, 75, 103, 240 : 81, 104, 111
+bp 94, 119 < 50, 78, 104, 240
+dis 93, 133 <# 50, 78, 104, 240
+loop 54, 80, 117, 157 << 73, 104, 248
, 51, 81, 103, 126, 128, 154,

<p>| | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>49, 73, 103, 239</td>
<td>50, 78, 103, 240</td>
<td>50, 78, 104, 240</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50, 75, 103, 240</td>
<td>52, 73, 106, 239</td>
<td>51, 82, 118, 240</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>51, 81, 103, 126, 128, 154,</td>
<td>51, 75, 193, 240</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 239 = 50, 78, 104, 240
-1 240
-rot 240

<p>| | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>50, 78, 103, 240</td>
<td>76, 103, 111</td>
<td>76, 99, 103, 111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>76, 103, 111</td>
<td>51, 82, 118, 240</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>94, 108, 192</td>
<td>31, 57, 145, 183, 241</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>249</td>
<td>76, 148</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>94, 119</td>
<td>77, 165</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>94, 119</td>
<td>48, 241, 246</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>91, 123</td>
<td>49, 72, 99, 172, 239</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>78, 129</td>
<td>78, 104</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>92, 145</td>
<td>78, 104</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>78, 147</td>
<td>78, 104</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Command</td>
<td>Description</td>
<td>Page Numbers</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>---------------------</td>
<td>-----------------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>?do</td>
<td>49, 54, 73, 80, 114, 119, 135, 154, 156</td>
<td>abs 49, 73, 107, 239</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>?dup</td>
<td>49, 72, 137, 240</td>
<td>accept 76, 107</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>?leave</td>
<td>80, 154</td>
<td>adr-mask 48</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>@</td>
<td>50, 58, 75, 104, 240</td>
<td>aerr@ 204</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[</td>
<td>82, 105</td>
<td>again 79, 108, 112</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[']</td>
<td>19, 76, 82, 105, 111</td>
<td>alarm 59, 109, 192, 241</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[char]</td>
<td>76, 110, 124, 247</td>
<td>alias 81, 109</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[compile]</td>
<td>82, 127</td>
<td>align 81, 109</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>\</td>
<td>76, 105</td>
<td>alloc-mem 27, 52, 75, 109, 120, 125, 145, 176, 245</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>]</td>
<td>82, 105</td>
<td>allot 81, 110, 153, 192</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tokenizer</td>
<td>198</td>
<td>and 49, 73, 110, 239</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>52, 100, 105, 240</td>
<td>ascii 76, 110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0&lt;</td>
<td>50, 78, 105, 239</td>
<td>attribute 248</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0&lt;=</td>
<td>52, 78, 105, 239</td>
<td>auto-boot? 35, 83, 84, 85, 110, 118, 239</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0&lt;&gt;</td>
<td>50, 78, 105, 239</td>
<td>aux! 204</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0=</td>
<td>50, 78, 105, 239</td>
<td>aux@ 204</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0&gt;</td>
<td>50, 78, 105, 239</td>
<td>averr! 204</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0&gt;=</td>
<td>52, 78, 105, 239</td>
<td>averr@ 204</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>52, 100, 105, 240</td>
<td>b(&quot;(”) 53, 111, 239</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1-</td>
<td>73, 106</td>
<td>b(‘) 53, 60, 111, 143, 239</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1+</td>
<td>73, 106</td>
<td>b(+loop) 54, 114, 117, 239</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>52, 100, 106, 240</td>
<td>b(;) 53, 111, 112, 113, 117, 119, 240</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2-</td>
<td>73, 106</td>
<td>b(;) 53, 111, 112, 113, 117, 119, 240</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2!</td>
<td>50, 75, 106, 240</td>
<td>b(&lt;mark) 54, 112, 113, 118, 240</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2*</td>
<td>49, 73, 106, 240</td>
<td>b(&gt;resolve) 54, 112, 113, 119, 240</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2+</td>
<td>73, 106</td>
<td>b(?) 54, 114, 117, 239</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2/</td>
<td>49, 73, 106, 240</td>
<td>b(buffer:) 52, 53, 113, 114, 120, 141, 163, 164, 240</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2@</td>
<td>50, 75, 106, 240</td>
<td>b(create) 52, 53, 114, 141, 163, 164, 240</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2constant</td>
<td>80, 127</td>
<td>b(defer) 52, 53, 113, 114, 120, 141, 163, 164, 240</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2drop</td>
<td>49, 72, 136, 240</td>
<td>b(case) 54, 113, 240</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2dup</td>
<td>49, 72, 137, 240</td>
<td>b(code) 48, 240, 246</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2over</td>
<td>49, 72, 169, 240</td>
<td>b(constant) 52, 53, 114, 141, 163, 164, 240</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2rot</td>
<td>49, 72, 178, 240</td>
<td>b(create) 52, 53, 114, 141, 163, 164, 240</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2swap</td>
<td>49, 72, 187, 240</td>
<td>b(defer) 52, 53, 113, 114, 120, 141, 163, 164, 240</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>52, 100, 106, 240</td>
<td>b(defer) 52, 53, 113, 114, 120, 141, 163, 164, 240</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3drop</td>
<td>72, 137</td>
<td>b(do) 54, 114, 117, 239</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3dup</td>
<td>72, 137</td>
<td>b(endcase) 54, 115, 116, 240</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4-byte-id</td>
<td>48</td>
<td>b(endof) 54, 116, 118, 240</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4-byte-id</td>
<td>240</td>
<td>b(field) 52, 53, 116, 141, 163, 164, 240</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>abort</td>
<td>51, 80, 106, 107, 140, 144, 149, 155, 187, 192, 241</td>
<td>b(leave) 54, 116, 239</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>abort&quot;</td>
<td>80, 107, 109, 110, 115, 134, 135, 158, 164</td>
<td>b(lit) 53, 117, 239</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

257
end0  59, 138, 139, 239
end1  59, 139, 240
endcase  79, 115, 139
endof  54, 79, 116, 139, 167
environment?  82, 139, 247
erase  75, 140
erase-screen  30, 32, 57, 58, 140, 142, 151, 177, 196, 244
erase-screen  241
eval  80, 134, 140, 248
evaluate  51, 80, 118, 140, 240, 248
even  73, 140
execute  19, 51, 80, 140, 239
execute-command  209
execute-device-method  38, 39, 40, 95, 110, 140, 181, 188
exit  51, 80, 140, 239
exit?  77, 141, 155
expect  50, 76, 141, 153, 184, 240
external  95, 141, 147
external-token  52, 53, 141
external-token  229, 240
f  177
fb1-blink-screen  243
fb1-delete-characters  243
fb1-delete-lines  243, 244
fb1-draw-character  243
fb1-draw-logo  244
fb1-erase-screen  244
fb1-insert-characters  244
fb1-insert-lines  244
fb1-install  125, 126, 145, 180, 193, 243, 244
fb1-invert-screen  244
fb1-reset-screen  244
fb1-slide-up  244
fb1-toggle-cursor  244
fb1-install  244
fb8-blink-screen  32, 58, 142
fb8-delete-characters  32, 58, 142
fb8-delete-lines  32, 58, 142
fb8-draw-character  32, 58, 142
fb8-draw-logo  32, 58, 142
fb8-erase-screen  32, 58, 142
fb8-insert-characters  32, 58, 142
fb8-insert-lines  32, 58, 142
fb8-invert-screen  32, 58, 142
fb8-reset-screen  32, 58, 143
fb8-toggle-cursor  32, 58, 143
fb8-blink-screen  241
fb8-delete-characters  241
fb8-delete-lines  241
fb8-draw-character  241
fb8-draw-logo  241
fb8-erase-screen  241
fb8-insert-characters  241
fb8-insert-lines  241
fb8-invert-screen  241
fb8-reset-screen  241
fb8-toggle-cursor  241
fcode-debug?  95, 143, 147, 163, 239
fcode-revision  59, 143
fcode-revision  240, 248, 249
fc ode-version  241, 245
ferror  49, 59, 143, 240
field  53, 81, 116, 143, 187
fill  50, 75, 144, 240
find  82, 144
find-device  39, 40, 64, 90, 144, 149, 169, 183
find-method  19, 55, 123, 144
find-package  54, 144, 169
find-package  241
find-method  241
find-package  241
finish-device  59, 121, 139, 145, 171
finish-device  241
firmware-version  245
firmware-version  241
fload  198
fm/mod  73, 145, 247
font  31
fontbytes  31, 57, 145, 183, 241
forget  81, 145
forth  82, 145
frame-buffer-adr  30, 31, 57, 145
frame-buffer-busy?  48
frame-buffer-adr  241
frame-buffer-busy?  241
free-mem  27, 52, 75, 109, 145, 176
free-mem  125
free-mem  55, 145, 158
free-mem  240
free-virtual  241
full  86, 180
<table>
<thead>
<tr>
<th>Keyword</th>
<th>Value</th>
<th>Keyword</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>m*</td>
<td>73, 157</td>
<td>net</td>
<td>132</td>
</tr>
<tr>
<td>mac-address</td>
<td>59, 156, 157, 163, 236</td>
<td>network</td>
<td>157</td>
</tr>
<tr>
<td>mac-address</td>
<td>241</td>
<td>new-device</td>
<td>59, 115, 121, 145, 163, 171, 182</td>
</tr>
<tr>
<td>map</td>
<td>24, 140, 160</td>
<td>new-name</td>
<td>131</td>
</tr>
<tr>
<td>map-in</td>
<td>23, 108, 158, 202</td>
<td>new-token</td>
<td>52, 53, 147, 164</td>
</tr>
<tr>
<td>map-low</td>
<td>55, 108, 158</td>
<td>new-device</td>
<td>241</td>
</tr>
<tr>
<td>map-out</td>
<td>23, 158, 202</td>
<td>new-token</td>
<td>240</td>
</tr>
<tr>
<td>map-page</td>
<td>205</td>
<td>next-property</td>
<td>54, 164</td>
</tr>
<tr>
<td>map-pages</td>
<td>205</td>
<td>next-property</td>
<td>241</td>
</tr>
<tr>
<td>map-segments</td>
<td>205</td>
<td>nip</td>
<td>49, 72, 164, 240</td>
</tr>
<tr>
<td>map-low</td>
<td>241, 248</td>
<td>no-data-command</td>
<td>209</td>
</tr>
<tr>
<td>map-out</td>
<td>145</td>
<td>nodelight-byte</td>
<td>86, 164</td>
</tr>
<tr>
<td>map-sbus</td>
<td>248</td>
<td>none</td>
<td>86</td>
</tr>
<tr>
<td>map?</td>
<td>203</td>
<td>noop</td>
<td>52, 82, 164, 185, 240</td>
</tr>
<tr>
<td>mask</td>
<td>59, 158, 159, 241</td>
<td>noshowstack</td>
<td>90, 164, 184, 247, 250</td>
</tr>
<tr>
<td>max</td>
<td>49, 73, 159, 239</td>
<td>not</td>
<td>73, 165, 248</td>
</tr>
<tr>
<td>max-transfer</td>
<td>28, 117, 159, 208</td>
<td>nvalias</td>
<td>15, 89, 165</td>
</tr>
<tr>
<td>memmap</td>
<td>48, 241, 245</td>
<td>nvedit</td>
<td>87, 165, 166</td>
</tr>
<tr>
<td>memory</td>
<td>21, 61</td>
<td>nvquit</td>
<td>87, 165, 166</td>
</tr>
<tr>
<td>memory-test-suite</td>
<td>59, 158, 159, 245</td>
<td>nvramrc</td>
<td>87, 166, 239</td>
</tr>
<tr>
<td>min</td>
<td>49, 73, 159, 239</td>
<td>nvrecover</td>
<td>87, 166</td>
</tr>
<tr>
<td>mips-off</td>
<td>200</td>
<td>nvrun</td>
<td>87, 166</td>
</tr>
<tr>
<td>mips-on</td>
<td>200</td>
<td>nvstore</td>
<td>87, 165, 166</td>
</tr>
<tr>
<td>mmu</td>
<td>21</td>
<td>nvunalias</td>
<td>89, 166</td>
</tr>
<tr>
<td>mmu-nctx</td>
<td>200</td>
<td>o#</td>
<td>77, 166</td>
</tr>
<tr>
<td>mmu-npmg</td>
<td>201</td>
<td>obio</td>
<td>203</td>
</tr>
<tr>
<td>mod</td>
<td>49, 73, 160, 239</td>
<td>obnmem</td>
<td>203</td>
</tr>
<tr>
<td>model</td>
<td>20, 56, 160, 241</td>
<td>octal</td>
<td>77, 167</td>
</tr>
<tr>
<td>modify</td>
<td>24, 140, 158, 160</td>
<td>oem-baner</td>
<td>89, 112, 167, 239</td>
</tr>
<tr>
<td>ms</td>
<td>59, 160, 241</td>
<td>oem-logo</td>
<td>89, 112, 167, 239</td>
</tr>
<tr>
<td>my-args</td>
<td>40, 55, 115, 121, 156, 161, 171, 182</td>
<td>of</td>
<td>54, 79, 118, 167</td>
</tr>
<tr>
<td>my-params</td>
<td>48</td>
<td>off</td>
<td>51, 75, 167, 240</td>
</tr>
<tr>
<td>my-parent</td>
<td>40, 54, 161</td>
<td>offset</td>
<td>28, 168</td>
</tr>
<tr>
<td>my-self</td>
<td>54, 115, 123, 124, 161, 188</td>
<td>offset16</td>
<td>46, 54, 168, 240</td>
</tr>
<tr>
<td>my-space</td>
<td>55, 108, 115, 161, 175, 182</td>
<td>on</td>
<td>51, 75, 168, 240</td>
</tr>
<tr>
<td>my-unit</td>
<td>40, 55, 108, 161, 175</td>
<td>open</td>
<td>6, 14, 22, 25, 26, 27, 28, 29, 37, 38, 39, 57, 117, 121, 125, 134, 149, 152, 158, 163, 168, 181, 182, 202, 208</td>
</tr>
<tr>
<td>my-address</td>
<td>232, 241</td>
<td>open-dev</td>
<td>22, 38, 39, 44, 65, 95, 115, 149, 155, 156, 168, 169</td>
</tr>
<tr>
<td>my-args</td>
<td>241</td>
<td>open-package</td>
<td>22, 27, 54, 168, 169</td>
</tr>
<tr>
<td>my-params</td>
<td>241, 245</td>
<td>open-dev</td>
<td>249</td>
</tr>
<tr>
<td>my-parent</td>
<td>241</td>
<td>open-package</td>
<td>233, 241</td>
</tr>
<tr>
<td>my-self</td>
<td>241</td>
<td>or</td>
<td>49, 73, 169, 239</td>
</tr>
<tr>
<td>my-space</td>
<td>232, 241</td>
<td>output</td>
<td>88, 150, 152, 169</td>
</tr>
<tr>
<td>my-unit</td>
<td>232, 241</td>
<td>output-device</td>
<td>87, 88, 150, 169, 239</td>
</tr>
<tr>
<td>negate</td>
<td>49, 73, 163, 239</td>
<td>over</td>
<td>49, 72, 169, 171, 240</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pack</td>
<td>52, 77, 169, 240</td>
</tr>
<tr>
<td>Command</td>
<td>Reference</td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------------</td>
<td>-------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pagesize</td>
<td>205</td>
<td></td>
<td></td>
</tr>
<tr>
<td>parse</td>
<td>76, 170, 247</td>
<td></td>
<td></td>
</tr>
<tr>
<td>parse-2int</td>
<td>55, 170</td>
<td></td>
<td></td>
</tr>
<tr>
<td>parse-word</td>
<td>76, 170, 247</td>
<td></td>
<td></td>
</tr>
<tr>
<td>parse-2int</td>
<td>241, 248</td>
<td></td>
<td></td>
</tr>
<tr>
<td>password</td>
<td>88, 170, 181</td>
<td></td>
<td></td>
</tr>
<tr>
<td>patch</td>
<td>91, 170</td>
<td></td>
<td></td>
</tr>
<tr>
<td>peer</td>
<td>54, 171, 241</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pgmap!</td>
<td>203</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pgmap?</td>
<td>203</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pgmap@</td>
<td>203</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pick</td>
<td>49, 72, 171, 240</td>
<td></td>
<td></td>
</tr>
<tr>
<td>poll-packet</td>
<td>48</td>
<td></td>
<td></td>
</tr>
<tr>
<td>postpone</td>
<td>82, 171, 248</td>
<td></td>
<td></td>
</tr>
<tr>
<td>printenv</td>
<td>20, 86, 164, 171, 181</td>
<td></td>
<td></td>
</tr>
<tr>
<td>probe</td>
<td>48, 241, 246</td>
<td></td>
<td></td>
</tr>
<tr>
<td>probe-all</td>
<td>35, 83, 84, 90, 171</td>
<td></td>
<td></td>
</tr>
<tr>
<td>probe-self</td>
<td>23, 171</td>
<td></td>
<td></td>
</tr>
<tr>
<td>probe-virtual</td>
<td>48</td>
<td></td>
<td></td>
</tr>
<tr>
<td>probe-virtual</td>
<td>241, 246</td>
<td></td>
<td></td>
</tr>
<tr>
<td>processor-type</td>
<td>48</td>
<td></td>
<td></td>
</tr>
<tr>
<td>processor-type</td>
<td>241, 246</td>
<td></td>
<td></td>
</tr>
<tr>
<td>property</td>
<td>56, 132, 160, 162, 172, 175, 229, 241, 248</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pwd</td>
<td>90, 172</td>
<td></td>
<td></td>
</tr>
<tr>
<td>quit</td>
<td>80, 172</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r&gt;</td>
<td>49, 72, 172, 239</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r@</td>
<td>49, 72, 172, 239</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rb!</td>
<td>58, 121, 173, 179, 241</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rb@</td>
<td>58, 122, 174, 241</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read</td>
<td>23, 24, 26, 27, 28, 35, 65, 117, 121, 149, 163, 168, 174, 181, 182, 233</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read-blocks</td>
<td>28, 174</td>
<td></td>
<td></td>
</tr>
<tr>
<td>recurse</td>
<td>82, 174, 248</td>
<td></td>
<td></td>
</tr>
<tr>
<td>recursive</td>
<td>82, 174</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reg</td>
<td>56, 161, 175, 241</td>
<td></td>
<td></td>
</tr>
<tr>
<td>relative-addressing</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>release</td>
<td>24, 27, 110, 125, 158, 159, 160, 176, 191</td>
<td></td>
<td></td>
</tr>
<tr>
<td>remove-abort</td>
<td>27, 149, 176, 182</td>
<td></td>
<td></td>
</tr>
<tr>
<td>repeat</td>
<td>79, 112, 119, 176, 193</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reset</td>
<td>23, 176, 249</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reset-all</td>
<td>66, 85, 88, 110, 176, 239</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reset-screen</td>
<td>30, 32, 57, 58, 143, 152, 177, 196, 244</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reset-all</td>
<td>249</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reset-screen</td>
<td>177, 241</td>
<td></td>
<td></td>
</tr>
<tr>
<td>restore</td>
<td>25, 27, 134, 152, 177, 182</td>
<td></td>
<td></td>
</tr>
<tr>
<td>resume</td>
<td>91, 92, 177</td>
<td></td>
<td></td>
</tr>
<tr>
<td>retry-command</td>
<td>209</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
set-token 48, 53, 173, 174, 177, 178, 179, 183
set-args 232, 241
till 94, 188
set-callback 122 to 9, 81, 82, 92, 116, 119, 120,
set-font 241
set-table 240
set-token 240
toggle-cursor 57, 189, 244
set-token-table 240 toggle-cursor 241
tokenizer[ 198
setenv 20, 86, 99, 164, 166, 167, 181, 183
setprop 20, 86 tracing 92, 129, 189
short-data-command 209 translate 24, 140, 158, 160, 189
show-children 208 true 22, 40, 50, 51, 52, 59, 75,
show-devs 89, 183
showstack 90, 164, 184, 247, 250
sifting 91, 184
sign 50, 78, 184, 240
sm/rem 73, 184, 248
smap! 203, 205
smap? 203
smap@ 203
source 76, 184
tuck 49, 72, 189, 240
spaces 77, 184
type 50, 76, 189, 190, 240
span 50, 76, 184, 240
start0 46, 47, 59, 168, 185, 240
u# 51, 78, 190, 240, 248
start1 46, 47, 59, 168, 185, 240
u#> 51, 78, 190, 240, 248
start2 46, 47, 59, 168, 185, 240
u#s 51, 78, 190, 240, 248
start4 46, 47, 59, 168, 185, 240
u* 73, 190
state 51, 82, 185, 240
u*x 248
state-valid 93, 146, 185
u. 50, 78, 190, 240
status 185, 248, 250
stdin 21, 88, 149, 186
u/mod 49, 73, 191, 239
stdout 21, 88, 169, 186
u< 50, 78, 190, 240
step 93, 94, 148, 186, 187
u<= 52, 78, 190, 240
stepping 92, 129, 187, 189
u> 50, 78, 190, 240
steps 94, 187
u>= 52, 78, 190, 240
structure 81, 187
u2/ 51, 73, 190, 240
suppress-banner 84, 89, 90, 112, 171, 187
suspend-fcode 59, 187
um* 50, 73, 190, 240, 248
suspend-fcode 241
um/mod 50, 73, 190, 240, 248
sverr! 205
unaligned-l! 75, 191
sverr@ 205
unaligned-l@ 75, 191
swap 49, 72, 187, 240
unaligned-w! 75, 191
sym 94, 187, 188
unloop 50, 80, 191, 240
sym>value 67, 94, 187, 188
unmap 24, 140, 160, 191
sync 67, 89, 188
unselect-dev 249
untill 79, 113, 191
test 22, 88, 181, 188
upc 52, 77, 191, 240
test-all 88, 181, 188
use-nvramrc? 35, 83, 87, 165, 166, 191, 239
then 79, 119, 188
user-abort 59, 192
throw 39, 40, 51, 80, 109, 110, 121,
user-abort 241
122, 123, 124, 125, 134, 135,
IEEE
CORE REQUIREMENTS AND PRACTICES
Std 1275-1994

vac-hwflush 201
vac-linesize 201
value 9, 17, 29, 31, 53, 56, 57, 81,
82, 92, 120, 125, 126, 145,
150, 151, 155, 163, 180, 189,
192, 193
value>sym 67, 94, 108, 192
variable 17, 50, 51, 52, 53, 59, 76,
77, 81, 82, 88, 93, 112, 120,
148, 150, 155, 158, 163, 169,
184, 185, 186, 192, 245
version 248
version1 46, 47, 59, 168, 192, 240
w! 51, 75, 192, 240
w, 51, 81, 126, 128, 154, 192,
240
w@ 51, 75, 192, 240
wa+ 51, 74, 193, 240
w1+ 51, 74, 193, 240
wbflip 51, 74, 193, 240, 248
wbflips 51, 75, 193, 241, 248
wbsplit 51, 74, 193, 240
wflips 248
while 79, 113, 176, 193
window-left 30, 31, 32, 57, 126, 142, 193,
243, 244
window-top 30, 31, 32, 57, 142, 155, 193,
243, 244
window-left 241
window-top 241
within 50, 78, 193, 240, 249
wljoin 51, 74, 194, 240
word 76, 194
words 91, 194
wpeek 58, 194, 241
wpoke 58, 194, 241
write 23, 25, 26, 27, 28, 35, 65,
117, 121, 134, 152, 163, 168,
169, 181, 182, 194, 233, 234
write-blocks 28, 194
x+ 248
x- 248
xdr+ 248
xdrbytes 248
xdrint 248
xdrphys 248
xdrstring 248
xdrtoint 248
xdrtostream 248
xmit-packet 48
xmit-packet 241
xor 49, 73, 194, 239
Annex J
Bibliography

(informative)


\[^{10}\] IEEE Std 1014-1987 has been withdrawn and is out of print; however, copies can be obtained from the IEEE Standards Department, IEEE, Inc., 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA.

\[^{11}\] This document is in progress. When it is approved and published, it will supersede the current standard, ISO/IEC 9316 : 1989, Information processing systems—Small Computer System Interface (SCSI).