OpenBoot™ 4.x Command Reference Manual

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054 U.S.A.
650-960-1300

Part No. 816-1177-10
February 2002, Revision A

Send comments about this document to: docfeedback@sun.com

Please
Recycle

Adobe PostScript

Contents

Preface

The *OpenBoot™ 4.x Command Reference Manual* describes how to use Sun™ systems that implement firmware that responds as those described by *IEEE Standard 1275-1994., Standard For Boot Firmware.*

This manual contains information on using the OpenBoot firmware to perform tasks such as:

- Booting the operating system
- Running diagnostics
- Modifying system start-up configuration parameters
- Loading and executing programs
- Troubleshooting

This manual also describes the commands of the OpenBoot Forth Interpreter which you can use to write Forth programs or to use the more advanced features of this firmware (such as its debugging capabilities).

The information in this manual is for a system that uses Version 4.*x* OpenBoot firmware. Other OpenBoot implementations may use different prompts or formatting, and may not support all of the tools and capabilities described in this manual.

Before You Read This Book

This manual is written for all users, including systems designers, systems administrators, and end users, who want to use OpenBoot to configure and debug their SBus and PCI-based systems.

How This Book Is Organized

Chapter 1 describes the user interface and other main features of OpenBoot firmware.

Chapter 2 explains the most common tasks for which OpenBoot firmware is used.

Chapter 3 details how to perform system administration tasks with NVRAM parameters.

Chapter 4 describes functions of the OpenBoot Forth language.

Chapter 5 describes how to load and execute programs from various sources (such as Ethernet, disk, or serial port).

Chapter 6 describes the debugging capabilities of the OpenBoot firmware, including decompiler, Forth source-level debugger, and breakpoints.

Appendix A describes how to set up a TIP connection.

Appendix B describes how to create a bootable floppy diskette from which you can load programs or files.

Appendix C discusses solutions for typical situations when you cannot boot the operating system.

Appendix D contains currently-supported OpenBoot Forth commands.

Typographic Conventions

Typeface or Symbol	Meaning	Examples
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `% You have mail.`
`AaBbCc123`	What you type, when contrasted with on-screen computer output	`% su` `Password:`
AaBbCc123	Book titles, new words or terms, words to be emphasized. Replace command-line variables with real names or values.	Read Chapter 6 in the *User's Guide*. These are called *class* options. To delete a file, type **`rm`** *filename*.

The following typographic conventions are specific to OpenBoot and are included in this manual:

- Keys are indicated by their name. For example:

  Press the Return key.

- When you see two key names separated by a dash, press and hold the first key down, then press the second key. For example:

  To enter Control-C, press and hold Control, then press C, then release both keys.

- When you see two key names separated by a space, press and release the first key and then press and release the second key. For example:

  To enter Escape B, press and release Escape, then press and release B.

- In a command line, square brackets indicate an optional entry and italics indicate an argument that you must replace with the appropriate text. For example:

  `help [`*word*`]`

Code Samples for OpenBoot

In this book, code samples are included in boxes and may display the following:

Typeface or Symbol	Meaning	Example
ok	OpenBoot command prompt	ok
%	UNIX C shell prompt	system%
$	UNIX Bourne and Korn shell prompt	system$
#	Superuser prompt, all shells	system#
ok	OpenBoot command prompt	ok

Related Documentation

A companion document to this manual is:

- *OpenBoot 3.x Quick Reference*

For information on OpenBoot FCode, refer to:

- *Writing FCode 2.x Programs*
- *Writing FCode 3.x Programs*

For information about Open Firmware, refer to the:

IEEE Standard 1275-1994 Standard for Boot (Initialization, Configuration) Firmware, Core Requirements and Practices.

Also see `http://playground.sun.com/1275`.

For more information about Forth and Forth programming, refer to:

- *ANSI X3.215-1994, American National Standard for Information Systems-Programming Languages-FORTH.*
- *Starting FORTH*, Leo Brody. FORTH, Inc., second edition, 1987.
- *Forth: The New Model*, Jack Woehr. M & T Books, 1992.
- Forth Interest Group:

  `http://forth.org/fig.html`

Accessing Sun Documentation Online

A broad selection of Sun system documentation is located at:

`http://www.sun.com/products-n-solutions/hardware/docs`

A complete set of Solaris documentation and many other titles are located at:

`http://docs.sun.com`

Sun Welcomes Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions. You can email your comments to Sun at:

`docfeedback@sun.com`

Please include the part number (816-1177-10) of your document in the subject line of your email.

CHAPTER 1

Overview

This chapter introduces OpenBoot as defined by the *IEEE Standard 1275-1994 Standard for Boot Firmware.* This chapter covers the following topics:

Features of OpenBoot Firmware

OpenBoot firmware is executed immediately after you turn on your system. The primary tasks of OpenBoot firmware are to:

- Test and initialize the system hardware
- Determine the hardware configuration
- Boot the operating system from either a mass storage device or from a network
- Provide interactive debugging facilities for testing hardware and software

The OpenBoot architecture provides a significant increase in functionality and portability when compared to proprietary systems of the past. Although this architecture was first implemented by Sun Microsystems as OpenBoot on SPARC™ systems, its design is processor-independent.

The following paragraphs describe some notable features of OpenBoot firmware.

Plug-in Device Drivers

A plug-in device driver is usually loaded from a plug-in device such as an SBus or a PCI card. The plug-in device driver can be used to boot the operating system from that device or to display text on the device before the operating system has activated its own drivers. This feature enables the input and output devices supported by a particular system to evolve without changing the system PROM.

FCode Interpreter

Plug-in drivers are written in a *machine-independent* interpreted language called *FCode*. Each OpenBoot system PROM contains an FCode interpreter. Thus, the same device and driver can be used on systems with different CPU instruction sets.

Device Tree

The device tree is a data structure describing the devices (permanently installed and plug-in) attached to a system. Both the user and the operating system can determine the hardware configuration of the system by inspecting the device tree.

Programmable User Interface

The OpenBoot *user interface* is based on the interactive programming language *Forth*. By combining sequences of commands to form complete programs, this provides a powerful capability for debugging hardware and software.

User Interface

The user interface is based on an interactive command interpreter that gives you access to an extensive set of functions for hardware and software development, fault isolation, and debugging. Any level of user can use these functions.

The user interface prompt is implementation dependent.

You can enter the OpenBoot environment by:

- Halting the operating system
- Pressing the Stop-A key

- Power cycling the system

If your system is not configured to boot automatically, the system stops at the user interface.

If automatic booting is configured, you can make the system stop at the user interface by pressing the Stop-A keys from the keyboard after the display console banner is displayed but before the system starts booting the operating system.

- When the system hardware detects an error from which it cannot recover.

See “A Caution About Using Some OpenBoot Commands” on page 11 for information on using commands after entering OpenBoot from the operating system.

Device Tree

Devices are attached to a host computer through a hierarchy of interconnected buses. OpenBoot represents the interconnected buses and their attached devices as a tree of nodes, called the *device tree*. A node representing the host computer’s main physical address bus forms the tree’s root node.

Each device node can have:

- Properties, which are data structures describing the node and its associated device
- Methods, which are the software procedures used to access the device
- Data, which are the initial values of the private data used by the methods
- Children, which are other device nodes “attached” to a given node and that lie directly below it in the device tree
- A parent, which is the node that lies directly above a given node in the device tree

Nodes with children usually represent buses and their associated controllers, if any. Each such node defines a physical address space that distinguishes the devices connected to the node from one another. Each child of that node is assigned a physical address in the parent’s address space.

The physical address generally represents a physical characteristic unique to the device (such as the bus address or the slot number where the device is installed). The use of physical addresses to identify devices prevents device addresses from changing when other devices are installed or removed.

Nodes without children are called leaf nodes and generally represent devices. However, some such nodes represent system-supplied firmware services.

Device Path Names, Addresses, and Arguments

OpenBoot deals directly with hardware devices in the system. Each device has a unique name representing the type of device and where that device is located in the system addressing structure. The following example shows a full device path name:

```
/sbus@1f,0/SUNW,fas@e,8800000/sd@3,0:a
```

A full device path name is a series of node names separated by slashes (/). The root of the tree is the machine node, which is not named explicitly but is indicated by a leading slash (/). Each node name has the form:

device-name@*unit-address*:*device-arguments*

TABLE 1-1 describes each of these parameters.

TABLE 1-1 Device Path Name Parameters

Path Name Parameter	Description
device-name	A human-readable string consisting of one to 31 letters, digits and punctuation characters from the set “, . _ + - ” that, ideally, has some mnemonic value. Uppercase and lowercase characters are distinct. In some cases, this name includes the name of the device’s manufacturer and the device’s model name, separated by a comma. Typically, the manufacturer’s upper-case, publicly-listed stock symbol is used as the manufacturer’s name (for example, SUNW,sd). For built-in devices, the manufacturer’s name is usually omitted (for example, sbus).
@	Must precede the address parameter.
unit-address	A text string representing the physical address of the device in its parent’s address space. The format of the text is bus dependent.
:	Must precede the arguments parameter.
device-arguments	A text string whose format depends on the particular device. It can be used to pass additional information to the device’s software.

The full device path name mimics the hardware addressing used by the system to distinguish between different devices. Thus, you can specify a particular device without ambiguity.

In general, the *unit-address* part of a node name represents an address in the physical address space of its parent. The exact meaning of a particular address depends on the bus to which the device is attached. Consider this example:

```
/sbus@1f,0/esp@0,40000/sd@3,0:a
```

Where:

- `1f,0` represents an address on the main system bus, because the SBus is directly attached to the main system bus in this example.
- `0,40000` is an SBus slot number (in this case, `0`) and an offset (in this case, `40000`), because the `esp` device is at offset 40000 on the card in SBus slot 0.
- `3,0` is a SCSI target and logical unit number, because the disk device is attached to a SCSI bus at target 3, logical unit 0.

When specifying a path name, either the *@unit-address* or *device-name* part of a node name is optional, in which case the firmware tries to pick the device that best matches the given name. If there are several matching nodes, the firmware chooses one (but it may not be the one you want).

For example, using `/sbus/esp@0,40000/sd@3,0` assumes that the system in question has exactly one SBus on the main system bus, making `sbus` as unambiguous an address as `sbus@1f,0`. On the same system, however, `/sbus/esp/sd@3,0` might or might not be ambiguous. Since SBus accepts plug-in cards, there could be more than one `esp` device on the same SBus bus. If there were more than one on the system, using `esp` alone would not specify which one, and the firmware might not choose the one you intended.

As another example, `/sbus/@2,1/sd@3,0` would normally be unambiguous, while `/sbus/scsi@2,1/@3,0` usually would not, since both a SCSI disk device driver and a SCSI tape device driver can use the SCSI target, logical unit address `3,0`.

The `:`*device-arguments* part of the node name is also optional. Once again, in the example:

```
/sbus@1f,0/scsi@2,1/sd@3,0:a
```

the argument for the disk device is `a`. The software driver for this device interprets its argument as a disk partition, so the device path name refers to partition `a` on that disk.

Some implementations also enable you to omit path name components. So long as the omission does not create any ambiguity, those implementations select the device that you intended. For example, if our example system had only one `sd` device, `/sd:a` would identify the same device as the much longer preceding expression.

Device Aliases

A device alias, or simply, alias, is a shorthand representation of a *device path*.

For example, the alias `disk` may represent the complete device path name:

```
/sbus@1f,0/esp@0,40000/sd@3,0:a
```

Systems usually have predefined device aliases for the most commonly used devices, so you rarely need to type a full device path name.

TABLE 1-2 describes the `devalias` command, which is used to examine, create, and change aliases.

TABLE 1-2 Examining and Creating Device Aliases

Command	Description
`devalias`	Display all current device aliases.
`devalias` *alias*	Display the device path name corresponding to alias.
`devalias` *alias device-path*	Define an alias representing *device-path*. If an alias with the same name already exists, the new value supersedes the old.

User-defined aliases are lost after a system reset or power cycle. If you want to create permanent aliases, you can either manually store the `devalias` command in a portion of non-volatile RAM (NVRAM) called *nvramrc*, or use the `nvalias` and `nvunalias` commands. (See Chapter 3 for further details.)

Displaying the Device Tree

You can browse the device tree to examine individual device tree nodes. The device tree browsing commands are similar to the Solaris™ commands for changing, displaying and listing the current directory in the Solaris directory tree. Selecting a device node makes it the current node.

The user interface commands for browsing the device tree are shown in TABLE 1-3.

TABLE 1-3 Commands for Browsing the Device Tree

Command	Description
`.properties`	Displays the names and values of the current node's properties.
`dev` *device-path*	Chooses the indicated device node, making it the current node.
`dev` *node-name*	Searches for a node with the given name in the subtree below the current node, and choose the first such node found.
`dev ..`	Chooses the device node that is the parent of the current node.
`dev /`	Chooses the root machine node.
`device-end`	Leaves the device tree.
`"` device-path`" find-device`	Chooses the device node, similar to `dev`.
`ls`	Displays the names of the current node's children.
`pwd`	Displays the device path name that names the current node.
`see` *wordname*	Decompiles the specified word.
`show-devs [`*device-path*`]`	Displays all the devices directly beneath the specified device in the device tree. The `show-devs` command used by itself shows the entire device tree.
`words`	Displays the names of the current node's methods.
`"` *device-path*`" select-dev`	Selects the specified device and make it the active node.
`unselect-dev`	Unselects the previously selected device.

`.properties` displays the names and values of all the properties in the current node:

```
ok dev /zs@1,f0000000
ok .properties
address               ffee9000
port-b-ignore-cd
port-a-ignore-cd
keyboard
device_type           serial
slave                 00000001
intr                  0000000c  00000000
interrupts            0000000c
reg                   00000001  f0000000  00000008
name                  zs
ok
```

`dev` sets the current node to the named node so its contents can be viewed. For example, to make the ACME company's SBus device named "ACME,widget" the current node:

```
ok dev /sbus/ACME,widget
```

`find-device` is similar to `dev`, differing only in the way the input pathname is passed.

```
ok " /sbus/ACME,widget" find-device
```

Note – After choosing a device node with `dev` or `find-device`, you can't execute that node's methods because `dev` does not establish the current instance. For a detailed explanation of this issue, refer to *Writing FCode 3.x Programs*.

`show-devs` lists all the devices in the OpenBoot device tree, as shown in the following example:

```
ok show-devs
/SUNW,UltraSPARC@0,0
/sbus@1f,0
/counter-timer@1f,3c00
/virtual-memory
/memory@0,0
/aliases
/options
/openprom
/chosen
/packages
/sbus@1f,0/cgsix@1,0
/sbus@1f,0/lebuffer@0,40000
/sbus@1f,0/dma@0,81000
/sbus@1f,0/SUNW,bpp@e,c800000
/sbus@1f,0/SUNW,hme@e,8c00000
/sbus@1f,0/SUNW,fas@e,8800000
/sbus@1f,0/sc@f,1300000
/sbus@1f,0/zs@f,1000000
/sbus@1f,0/zs@f,1100000
/sbus@1f,0/eeprom@f,1200000
/sbus@1f,0/SUNW,fdtwo@f,1400000
/sbus@1f,0/flashprom@f,0
/sbus@1f,0/auxio@f,1900000
/sbus@1f,0/SUNW,CS4231@d,c000000
/sbus@1f,0/SUNW,fas@e,8800000/st
/sbus@1f,0/SUNW,fas@e,8800000/sd
/openprom/client-services
/packages/disk-label
/packages/obp-tftp
/packages/deblocker
/packages/terminal-emulator
ok
```

Here is an example of the use of `words`:

```
ok dev /zs
ok words
ring-bell      read           remove-abort? install-abort
close          open           abort?        restore
clear          reset          initkbdmouse   keyboard-addr mouse
1200baud       setbaud        initport       port-addr
```

Getting Help

Whenever you see the ok prompt on the display, you can ask the system for help by typing one of the help commands shown in TABLE 1-4.

TABLE 1-4 Help Commands

Command	Description
`help`	Lists main help categories.
`help` *category*	Shows help for all commands in the category. Use only the first word of the category description.
`help` *command*	Shows help for individual command (where available).

`help`, without any specifier, displays instructions on how to use the help system and lists the available help categories. Because of the large number of commands, help is available only for commands that are used frequently.

If you want to see the help messages for all the commands in a selected category, or, possibly, a list of subcategories, type:

```
ok help category
```

If you want help for a specific command, type:

```
ok help command
```

For example, when you ask for information on the dump command, you might see the following message:

```
ok help dump
Category: Memory access
dump ( addr length -- ) display memory at addr for length bytes
ok
```

The above help message first shows that dump is a command from the Memory access category. The message also shows the format of the command.

Note – In some newer systems, descriptions of additional system-specific commands are available with the `help` command. Help as described may not be available on all systems.

A Caution About Using Some OpenBoot Commands

OpenBoot might not operate correctly after the operating system has begun execution (for example, after Stop-A or halt). This occurs when the operating system can modify the system state in ways that are inconsistent with continued OpenBoot operation. In this case, you might have to power cycle the system to restore normal operation. Not all OpenBoot commands referenced in this document are available on all systems. The behavior of some of the commands may vary from one system to another.

For example, suppose you boot the operating system, exit to OpenBoot, then execute the `probe-scsi` command (described in Chapter 2). You may find that `probe-scsi` fails, and you may not be able to resume the operating system, or you may have to power cycle the systems.

Re-execute an OpenBoot command that failed because the operating system has executed:

1. **Note the value of the** *auto-boot?* **NVRAM configuration variable using** `printenv`**.**
2. **If the value is true, set the value to false using** `setenv`**.**
3. **Reset the system.**
4. **After the system stops at the user interface, type the OpenBoot command that failed.**
5. **Restore the value of the** *auto-boot?* **NVRAM configuration.**
6. **Reset the system.**

CHAPTER 2

Booting and Testing Your System

This chapter describes the most common tasks that you perform using OpenBoot:

Booting Your System

The most important function of OpenBoot firmware is to boot the system. Booting is the process of loading and executing a stand-alone program such as an operating system. Booting can either be initiated automatically or by typing a command at the user interface.

The boot process is controlled by a number of *configuration variables*. Configuration variables are discussed in detail in Chapter 3. The configuration variables that affect the boot process are:

- `auto-boot?`

  This variable controls whether or not the system automatically boots after a system reset or when the power is turned on. This variable is typically `true`.

- `boot-command`

  This variable specifies the command to be executed when `auto-boot?` is `true`. The default value of `boot-command` is `boot` with no command-line arguments.

- `diag-switch?`

  If the value is `true`, run in the Diagnostic mode. This variable is `false` by default.

- `boot-device`

  This variable contains the name of the default boot device that is used when OpenBoot is not in diagnostic mode.

- `boot-file`

  This variable contains the default boot arguments that are used when OpenBoot is not in diagnostic mode.

- `diag-device`

  This variable contains the name of the default diagnostic mode boot device.

- `diag-file`

  This variable contains the default diagnostic mode boot arguments.

Based on the values of the above configuration variables, the boot process can proceed in a number of different ways. For example:

- If `auto-boot?` is `true`, the system will boot from either the default boot device or from the diagnostic boot device depending on whether OpenBoot is in diagnostic mode.
- If `auto-boot?` is `false`, the system may stop at the OpenBoot user interface without booting the system. To boot the system, you can do one of the following:
  - Type the `boot` command without any arguments. The system boots from the default boot device using the default boot arguments.
  - Type the `boot` command with an explicit boot device. The system boots from the specified boot device using the default boot arguments.
  - Type the `boot` command with explicit boot arguments. The system uses the specified arguments to boot from the default boot device.
  - Type the `boot` command with an explicit boot device and with explicit arguments. The system boots from the specified device with the specified arguments.

Booting for the Casual User

Typically, `auto-boot?` is `true`, `boot-command` is `boot`, and OpenBoot is not in diagnostic mode. Consequently, the system automatically loads and executes the program and arguments described by `boot-file` from the device described by `boot-device` when the system is first turned on or following a system reset.

If you want to boot the default program when `auto-boot?` is `false`, simply type `boot` at the `ok` prompt.

Booting for the Expert User

OpenBoot normally loads and executes an operating system or an operating system's loader program, but `boot` can also be used to load and execute other kinds of programs, such as diagnostics. For further details about loading programs other than the operating system, see Chapter 5.

Booting usually happens automatically based on the values contained in the configuration variables described above. However, the user can also initiate booting from the user interface.

OpenBoot performs the following steps during the boot process:

- The firmware may reset the system if a client program has been executed since the last reset. (The execution of a reset is implementation dependent.)
- A device is selected by parsing the `boot` command line to determine the boot device and the boot arguments to use. Depending on the form of the `boot` command, the boot device and argument values may be taken from configuration variables.
- The `bootpath` and `bootargs` properties in the `/chosen` node of the device tree are set with the selected values.
- The selected program is loaded into memory using a protocol that depends on the type of the selected device. For example, a disk boot might read a fixed number of blocks from the beginning of the disk, while a tape boot might read a particular tape file.
- The loaded program is executed. The behavior of the program may be further controlled by any argument string that was either contained in the selected configuration variable or was passed to the `boot` command on the command line.

The program loaded and executed by the boot process can be a *secondary boot program*, whose purpose is to load yet another program. This secondary boot program may use a protocol different from that used by OpenBoot to load the secondary boot program. For example, OpenBoot might use the Trivial File Transfer Protocol (TFTP) to load the secondary boot program while the secondary boot program might then use the Network File System (NFS) protocol to load the operating system.

Typical secondary boot programs accept arguments of the form:

```
filename -flags
```

Where *filename* is the name of the file containing the operating system and where *-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. As shown in the `boot`

command template, OpenBoot treats all such text as a single, opaque *arguments* string that has no special meaning to OpenBoot itself. The arguments string is passed unaltered to the specified program.

The `boot` command has the following format:

```
ok boot [device-specifier] [arguments]
```

The optional parameters for the `boot` command are described in TABLE 2-1.

TABLE 2-1 Optional `boot` Command Parameters

Parameter	Description
[*device-specifier*]	The name (full path name or `devalias`) of the boot device. Typical values include: `cdrom` (CD-ROM drive) `disk` (hard disk) `floppy` (3-1/2" diskette drive) `net` (Ethernet) `tape` (SCSI tape) If *device-specifier* is not specified and if `diagnostic-mode?` returns `false`, `boot` uses the device specified by the `boot-device` configuration variable. If *device-specifier* is not specified and if `diagnostic-mode?` returns `true`, `boot` uses the device specified by the `diag-device` configuration variable.
[*arguments*]	The name of the program to be booted (for example, *stand/diag*) and any program arguments. If *arguments* is not specified and if `diagnostic-mode?` returns `false`, `boot` uses the file specified by the `boot-file` configuration variable. If *arguments* is not specified and if `diagnostic-mode?` returns `true`, `boot` uses the file specified by the `diag-file` configuration variable.

Note – Most commands (such as `boot` and `test`) that require a device name accept either a full device path name or a device alias. In this manual, the term *device-specifier* indicates that either an appropriate device path name or a device alias is acceptable for such commands.

Since a device alias cannot be syntactically distinguished from the *arguments*, OpenBoot resolves this ambiguity as follows:

- If the space-delimited word following `boot` on the command line begins with `/`, the word is a device-path and, thus, a *device-specifier*. Any text to the right of this *device-specifier* is included in *arguments*.
- If the space-delimited word matches an existing device alias, the word is a *device-specifier*. Any text to the right of this *device-specifier* is included in *arguments*.
- Otherwise, the appropriate default boot device is used, and any text to the right of `boot` is included in *arguments*.

Consequently, `boot` command lines have the following possible forms.

```
ok boot
```

With this form, `boot` loads and executes the program specified by the default boot arguments from the default boot device.

```
ok boot device-specifier
```

If `boot` has a single argument that either begins with the character `/` or is the name of a defined `devalias`, `boot` uses the argument as a device specifier. `boot` loads and executes the program specified by the default boot arguments from the specified device.

For example, to explicitly boot from the primary disk, type:

```
ok boot disk
```

To explicitly boot from the primary network device, type:

```
ok boot net
```

If `boot` has a single argument that neither begins with the character `/` nor is the name of a defined `devalias`, `boot` uses all of the remaining text as its arguments.

```
ok boot arguments
```

`boot` loads and executes the program specified by the arguments from the default boot device.

```
ok boot device-specifier arguments
```

If there are at least two space-delimited arguments, and if the first such argument begins with the character / or if it is the name of a defined `devalias`, `boot` uses the first argument as a device specifier and uses all of the remaining text as its arguments. `boot` loads and executes the program specified by the arguments from the specified device.

For all of the above cases, `boot` records the device that it uses in the `bootpath` property of the `/chosen` node. `boot` also records the arguments that it uses in the `bootargs` property of the `/chosen` node.

Device alias definitions vary from system to system. Use the `devalias` command, described in Chapter 1, to obtain the definitions of your system's aliases.

Booting Over the Network

The network boot process involves:

1. Obtaining the IP address of the booting client

   The client knows its Ethernet address and system type, but needs its IP address to transfer the files it needs.

2. Downloading the standalone boot program and executing it

   The client uses TFTP (Trivial File Transfer Protocol) to download the standalone boot program and executes it.

A client booting over a network can use RARP (Reverse Address Resolution Protocol) to obtain its IP address. When booting Solaris software, the loaded standalone program is 'inetboot' which uses the RPC protocol 'bootparams' to obtain boot parameters, and loads the kernel and executes it. To boot with RARP and bootparams, use the command:

```
ok boot full-path-to-network-device
```

Clients that are DHCP (Dynamic Host Configuration Protocol) aware can use DHCP to obtain the IP address, boot parameters, and network configuration information with more efficiency and flexibility than the combination of the RARP and bootparams services. In addition, using DHCP removes the requirement for a boot server on every subnet. To boot with DHCP, use the command:

```
ok boot full-path-to-network-device:dhcp
```

DHCP aware PROM clients support inter operability with BOOTP (Bootstrap Protocol) servers. The client prefers DHCP configurations over BOOTP, but accepts BOOTP configurations if no DHCP configuration is offered.

The default protocol used (that is, RARP or DHCP) when the command `boot net` is executed depends on how the 'net' device alias is specified. If the net devalias specifies only the path to the network device, RARP is used as the default address discovery protocol. If the device alias includes **`dhcp`** as an argument, DHCP is used.

The following examples show how the 'net' device alias must be defined to select RARP or DHCP booting on a Sun Blade 100 (The default is RARP boot):

```
/pci@1f,0/network@c,1                    Boot using RARP
/pci@1f,0/network@c,1:dhcp               Boot using DHCP
```

You can set the desired device alias by using the `nvalias` command.

Arguments Supported by Network Boot

The network boot support package (`obp-tftp`) enables a wide range of system knowledge to be used in the boot process. At one extreme, the client knows nothing except its Ethernet address and system type. At the other end, the client can know the server's IP address, its own IP address, router addresses necessary, and so on.

The following syntax is supported for network boot:

```
boot
<network-device>:[dhcp|bootp,][server-ip],
[boot-filename],[client-ip],[router-ip],[boot-retries],
[tftp-retries],[subnet-mask][boot-arguments]
```

All arguments are optional. Commas are required for missing positional parameters unless they are at the end of the list.

server-ip, *client-ip*, *router-ip*, and *subnet-mask* are specified in Internet standard "dotted-decimal" notation.

If any of *server-ip*, *boot-filename*, *client-ip*, *router-ip*, and *subnet-mask* are specified, the PROM client uses these values instead of any values which are (or may be) obtained by the normal configuration process.

- `dhcp` specifies the use of DHCP as the address discovery protocol to be used. `bootp` is treated as a synonym of DHCP (that is, the client still uses DHCP format messages). The client accepts a BOOTP configuration only if no DHCP configurations are offered.

- *server-ip* is the IP address of the TFTP server from which the standalone boot program is to be downloaded.
- *boot-filename* is the name of the standalone program to be loaded by TFTP from the server. The default file name is constructed from the IP address if the boot protocol is RARP, or from the client class identifier if using DHCP/BOOTP.
- *client-ip* is the IP address of the client (i.e., the system being booted).
- *router-ip* is the IP address of router to be used.
- *boot-retries* is the maximum number of retries attempted before the boot process is determined to have failed.
- *tftp-retries* is the maximum number of retries attempted before the TFTP process is determined to have failed.
- subnet-mask is the *subnet mask* on the local network.

Running Diagnostics

Several diagnostic commands are available from the user interface. These commands let you check devices such as the network controller, the diskette system, memory, installed SBus cards and SCSI devices, and the system clock.

The value returned by `diagnostic-mode?` controls:

- The selection of the device and file that are used by the `boot` and `load` commands (if the device and file are not explicitly specified as arguments to those commands).
- The extent of system self-test during power-on, and the amount of text displayed.

OpenBoot is in diagnostic mode and the `diagnostic-mode?` command returns `true` if at least one of the following conditions is met:

- The configuration variable `diag-switch?` is set to `true`.
- The system's diagnostic switch (if any) is "on".
- Another system-dependent indicator requests extensive diagnostics.

When OpenBoot is in diagnostic mode, the value of `diag-device` is used as the *default boot device* and the value of `diag-file` is used as the *default boot arguments* for the `boot` command.

When OpenBoot is not in diagnostic mode, the value of `boot-device` is used as the *default boot device* and the value of `boot-file` is used as the *default boot arguments* for the `boot` command.

TABLE 2-2 lists diagnostic test commands. Not all of these tests are available in all OpenBoot implementations.

TABLE 2-2 Diagnostic Test Commands

Command	Description
`probe-scsi`	Identifies devices attached to a SCSI bus.
`test` *device-specifier*	Executes the specified device's `selftest` method. For example: `test net` - test the network connection
`watch-clock`	Tests a clock function.
`watch-net`	Monitors a network connection.
`test-all` *device specifier*	Executes the self-test method for all devices at or below device-specifier. If no device-specifier is provided, it executes the self-test methods for all devices in the device tree.
`obdiag`	Invokes an optional interactive menu tool which lists all self-test methods available on a system; provides commands to run selftests.

Note – Running any diagnostic test command may cause the system to reset before booting.

The extent of diagnostic coverage performed by self-test methods may depend on the setting of the `diag-level` configuration variable as well as on the test arguments and on whether or not the system is in diagnostic mode.

Note – To assure maximum coverage of testing make sure your system is in diagnostic mode and `diag-level` is set to `max`.

The extent of diagnostics coverage performed by self-test methods may depend on the setting of the `diag-level` configuration variable as well as on the test arguments and on whether or not the system is in diagnostic mode.

To assure maximum coverage of testing make sure your system is in diagnostic mode and `diag-level` is set to `max`.

Testing the SCSI Bus

To check a SCSI bus for connected devices, type:

```
ok probe-scsi
Target 1
  Unit 0 Disk SEAGATE ST1480 SUN04246266 Copyright (C) 1991 Seagate All rights
reserved
Target 3
  Unit 0 Disk SEAGATE ST1480 SUN04245826 Copyright (C) 1991 Seagate All rights
reserved

ok
```

The actual response depends on the devices on the SCSI bus.

Testing Installed Devices

To test a single installed device, type:

```
ok test device-specifier
```

In general, if no message is displayed, the test succeeded.

Note – Many devices require the system's `diag-switch?` parameter to be `true` in order to run this test.

Testing Removable Media Drives

The removable media drive test determines whether or not the removable media drive being tested is functioning properly. For some implementations, a formatted, high-density (HD) disk, a CD-ROM or a tape must be in the appropriate removable media drive for this test to succeed.

Diskette Drive

To test the diskette drive, type:

```
ok test floppy
Testing floppy disk system. A formatted
disk should be in the drive.
Test succeeded.
ok
```

Note – Not all OpenBoot systems include this test word.

To eject the diskette from the diskette drive of a system capable of software-controlled ejection, type:

```
ok eject floppy
ok
```

CD-ROM Drive

To test the CD-ROM drive, type:

```
ok test cdrom
Testing cdrom drive system. A
cdrom should be in the drive.
Test succeeded.
ok
```

Note – Not all OpenBoot systems include this test word.

To eject the CD-ROM from the CD-ROM drive of a system capable of software-controlled ejection, type:

```
ok eject cdrom
ok
```

Tape Drive

To test the tape drive, type:

```
ok test tape
Testing tape drive system. A
tape should be in the drive.
Test succeeded.
ok
```

Note – Not all OpenBoot systems include this test word.

To eject the tape from the tape drive of a system capable of software-controlled ejection, type:

```
ok eject tape
ok
```

Testing Memory

To test memory, type:

```
ok test /memory
Testing 16 megs of memory at addr 4000000 11
ok
```

In the preceding example, the first number (4000000) is the base address of the testing, and the following number (11) is the number of megabytes to go.

Note – Not all OpenBoot systems include this test word.

Testing the Clock

To test the clock function, type:

```
ok watch-clock
Watching the 'seconds' register of the real time clock chip.
It should be ticking once a second.
Type any key to stop.
1
ok
```

The system responds by incrementing a number once a second. Press any key to stop the test.

Note – Not all OpenBoot systems include this test word.

Testing the Network Controller

To test the primary network controller, type:

```
ok test net
Internal Loopback test - (result)
External Loopback test - (result)
ok
```

The system responds with a message indicating the result of the test.

Note – Depending on the particular network controller and the type of network to which your system is attached, various levels of testing are possible. Some such tests may require that the network interface be connected to the network.

Monitoring the Network

To monitor a network connection, type:

```
ok watch-net
Internal Loopback test - succeeded
External Loopback test - succeeded
Looking for Ethernet packets.
'.' is a good packet. 'X' is a bad packet.
Type any key to stop
.....................X.....X..............
ok
```

The system monitors network traffic, displaying "." each time it receives an error-free packet and "X" each time it receives a packet with an error that can be detected by the network hardware interface.

Note – Not all OpenBoot systems include this test word.

Displaying System Information

The user interface provides one or more commands to display system information. `banner` is provided by all OpenBoot implementations; the remaining commands represent extensions provided by some implementations. These commands, listed in TABLE 2-3, let you display the system banner, the Ethernet address for the Ethernet controller, the contents of the ID PROM, and the version number of OpenBoot. The ID PROM contains information specific to each individual system, including the serial number, date of manufacture, and Ethernet address assigned to the system.

TABLE 2-3 System Information Commands

Command	Description
`banner`	Displays power-on banner.
`.enet-addr`	Displays current Ethernet address.
`.idprom`	Displays ID PROM contents, formatted.
`.traps`	Displays a list of processor-dependent trap types.
`.version`	Displays version and date of the boot PROM.

Also see the device tree browsing commands in TABLE 1-3.

Resetting the System

Occasionally, you may need to reset your system. The `reset-all` command resets the entire system and is similar to performing a power cycle.

To reset the system, type:

```
ok reset-all
```

If your system is set up to run the power-on self-test (POST) and initialization procedures on reset, these procedures begin executing once you initiate this command. (On some systems, POST is only executed after power-on.) Once POST completes, the system either boots automatically or enters the user interface, just as it would have done after a power cycle.

CHAPTER 3

Setting Configuration Variables

This chapter describes how to access and modify nonvolatile RAM (NVRAM) configuration variables. Its sections include:

The procedures described in this chapter assume that the user interface is active. See Chapter 1 for information about starting the user interface.

System Configuration Variables

System configuration variables are stored in the system NVRAM. These variables determine the start-up system configuration and related communication characteristics. You can modify the values of the configuration variables, and any changes you make remain in effect even after a power cycle. Configuration variables should be adjusted cautiously.

TABLE 3-1 lists a typical set of NVRAM configuration variables defined by *IEEE Standard 1275-1994*.

TABLE 3-1 Standard Configuration Variables

Variable	Typical Default	Description
auto-boot?	true	If true, boots automatically after power on or reset.
boot-command	boot	Command that is executed if auto-boot? is true.
boot-device	disk net	Device from which to boot.
boot-file	empty string	Arguments passed to booted program.
diag-device	net	Diagnostic boot source device.
diag-file	empty string	Arguments passed to booted program in diagnostic mode.
diag-switch?	false	If true, run in diagnostic mode.
fcode-debug?	false	If true, includes name fields for plug-in device FCodes.
input-device	keyboard	Console input device (usually keyboard, ttya, or ttyb).
nvramrc	empty	Contents of nvramrc.
oem-banner	empty string	Custom OEM banner (enabled by oem-banner? true).
oem-banner?	false	If true, use custom OEM banner.
oem-logo	no default	Byte array custom OEM logo (enabled by oem-logo? true). Displayed in hexadecimal.
oem-logo?	false	If true, uses custom OEM logo (else, uses Sun logo).
output-device	screen	Console output device (usually screen, ttya, or ttyb).
screen-#columns	80	Number of on-screen columns (characters/line).
screen-#rows	34	Number of on-screen rows (lines).
security-#badlogins	no default	Number of incorrect security password attempts.
security-mode	none	Firmware security level (options: none, command, or full).

TABLE 3-1 Standard Configuration Variables *(Continued)*

Variable	Typical Default	Description
`security-password`	no default	Firmware security password (never displayed).
`use-nvramrc?`	`false`	If `true`, execute commands in `nvramrc` during system start-up.
`local-mac-address?`	`false`	If `true`, network devices use their own MAC addresses.
`error-reset-recovery`	`boot`	Recovery action after an error reset CPU trap (options: `none`, `sync`, or `boot`).

Note – Different OpenBoot implementations may use different defaults and / or different configuration variables.

Displaying and Changing Variable Settings

NVRAM configuration variables can be viewed and changed using the commands listed in TABLE 3-2.

TABLE 3-2 Configuration Variable Commands

Command	Description
`printenv`	Displays current variables and current default values. `printenv` *variable* shows the current value of the named variable.
`setenv` *variable-name value*	Sets *variable* to the given decimal or text *value*. Changes are permanent, but often take effect only after a reset.
`set-default` *variable*	Resets the value of *variable* to the factory default.
`set-defaults`	Resets variable values to the factory defaults.
`password`	Sets `security-password`.

The following pages show how these commands can be used.

Note – Solaris provides the `eeprom` utility for modifying OpenBoot configuration variables.

To display a list of the current variable settings on your system, type:

```
ok printenv

Variable Name           Value                Default Value
oem-logo
oem-logo?               false                false
oem-banner
oem-banner?             false                false
output-device           ttya                 screen
input-device            ttya                 keyboard
diag-file
diag-device             net                  net
boot-file
boot-device             disk                 disk net
auto-boot?              false                true
fcode-debug?            true                 false
use-nvramrc?            false                false
nvramrc
screen-#columns         80                  80
screen-#rows            34                  34
security-mode           none                none
security-password
security-#badlogins     0
diag-switch?            true                 false
ok
```

In the displayed, formatted list of the current settings, numeric variables are often shown in the decimal format.

To change a variable setting, type:

```
ok setenv variable-name value
```

variable-name is the name of the variable. *value* is a numeric value or text string appropriate to the named variable. A numeric value is interpreted as a decimal number, unless preceded by `0x`, which is the qualifier for a hexadecimal number.

For example, to set the `auto-boot?` variable to `false`, type:

```
ok setenv auto-boot? false
ok
```

Note – Many variable changes do not affect the operation of the firmware until the next power cycle or system reset, at which time the firmware uses the variable's new value.

You can reset one or most of the variables to the original defaults using the `set-default` *variable* and `set-defaults` commands.

For example, to reset the `auto-boot?` variable to its default setting (true), type:

```
ok set-default auto-boot?
ok
```

To reset most variables to their default settings, type:

```
ok set-defaults
ok
```

On many SPARC systems, you can reset the NVRAM variables to their default settings by holding down Stop-N while the system is powering up. When issuing this command, hold down Stop-N immediately after turning on the power to the SPARC system, and keep it pressed for a few seconds or until you see the banner (if the display is available). This is a good technique to force a SPARC compatible system's NVRAM variables to a known condition. See also Appendix C.

Setting Security Variables

The NVRAM system security variables are:

- `security-mode`
- `security-password`
- `security-#badlogins`

`security-mode` can restrict the set of operations that users are allowed to perform from the user interface. The three security modes, and their available commands, are listed in the following table in the order of most to least secure.

TABLE 3-3 Commands Available for *security-mode* Settings

Setting	Commands
`full`	All commands except `go` require the password.
`command`	All commands except `boot` and `go` require the password.
`none`	No password required (default).

Command Security

With `security-mode` set to `command`:

- A password is not required if you type the `boot` command by itself. However, if you use the `boot` command with an argument, a password is required.
- The `go` command never asks for a password.
- A password is required to execute any other command.

Examples are shown in the following screen.

```
ok boot     (no password required)
ok go       (no password required)
ok boot filename (password required)
Password:  (password is not echoed as it is typed)
ok reset-all(password required)
Password:  (password is not echoed as it is typed)
```

Caution – It is important to remember your security password and to set the security password before setting the security mode. If you forget this password, you cannot use your system; you must call your vendor's customer support service to make your system bootable again.

To set the security password and `command` security mode, type the following at the `ok` prompt:

```
ok password
ok New password (only first 8 chars are used):
ok Retype new password:
ok setenv security-mode command
ok
```

The security password you assign must be between zero and eight characters. Any characters after the eighth are ignored. You do not have to reset the system; the security feature takes effect as soon as you type the command.

If you enter an incorrect security password, there will be a delay of about 10 seconds before the next boot prompt appears. The number of times that an incorrect security password is typed is stored in the `security-#badlogins` variable.

Full Security

The `full` security mode is the most restrictive. With `security-mode` set to `full`:

- A password is required any time you execute the `boot` command.
- The `go` command never asks for a password.
- A password is required to execute any other command.

Here are some examples.

```
ok go       (no password required)
ok boot     (password required)
Password:   (password is not echoed as it is typed)
ok boot filename (password required)
Password:   (password is not echoed as it is typed)
ok reset-all(password required)
Password:   (password is not echoed as it is typed)
```

Caution – It is important to remember your security password and to set the security password *before* setting the security mode. If you forget this password, you cannot use your system; you must call your vendor's customer support service to make your system bootable again.

To set the security password and full security mode, type the following at the ok prompt:

```
ok password
ok New password (only first 8 chars are used):
ok Retype new password:
ok setenv security-mode full
ok
```

Changing the Power-On Banner

The banner configuration variables are:

- oem-banner
- oem-banner?
- oem-logo
- oem-logo?

To view the power-on banner, type:

```
ok banner
Sun Ultra 1 SBus (UltraSPARC 167 MHz),Keyboard Present
PROM Rev. 3.0, 64MB memory installed, Serial # 289
Ethernet address 8:0:20:d:e2:7b, Host ID: 80000121
ok
```

The banner for your system will be different.

The banner consists of two parts: the text field and the logo (over serial ports, only the text field is displayed). You can replace the existing text field with a custom text message using the oem-banner and oem-banner? configuration variables.

To insert a custom text field in the power-on banner, type:

```
ok setenv oem-banner Hello Mom and Dad
ok setenv oem-banner? true
ok banner
 Hello Mom and Dad
ok
```

The system displays the banner with your new message, as shown in the preceding screen.

The graphic logo is handled differently. `oem-logo` is a 512-byte array, containing a total of 4096 bits arranged in a 64 x 64 array. Each bit controls one pixel. The most significant bit (MSB) of the first byte controls the upper-left corner pixel. The next bit controls the pixel to the right of it, and so on.

To create a new logo, first create a Forth array containing the correct data; then copy this array into `oem-logo`. The array is then installed in `oem-logo` with `$setenv`. The example below fills the top half of `oem-logo` with an ascending pattern.

```
ok create logoarray d# 512 allot
ok logoarray d# 256 0 do i over i + c! loop drop
ok logoarray d# 256 " oem-logo" $setenv
ok setenv oem-logo? true
ok banner
```

To restore the system's original power-on banner, set the `oem-logo?` and `oem-banner?` variables to `false`.

```
ok setenv oem-logo? false
ok setenv oem-banner? false
ok
```

Because the `oem-logo` array is so large, `printenv` displays approximately the first 8 bytes (in hexadecimal). To display the entire array, type `oem-logo dump`. The `oem-logo` array is not erased by `set-defaults`, since it might be difficult to restore the data. However, `oem-logo?` is set to false when `set-defaults` executes, so the custom logo is no longer displayed.

Note – Some systems do not support the `oem-logo` feature.

Input and Output Control

The console is used as the primary means of communication between OpenBoot and the user. The console consists of an input device, used for receiving information supplied by the user, and an output device, used for sending information to the user. Typically, the console is either the combination of a text/graphics display device and a keyboard or an ASCII terminal connected to a serial port.

The configuration variables related to the control of the console are:

- `input-device`
- `output-device`
- `screen-#columns`
- `screen-#rows`

You can use these variables to assign the power-on defaults for the console. These values do not take effect until after the next power cycle or system reset.

Selecting Input and Output Device Options

The `input-device` and `output-device` variables control the firmware's selection of input and output devices after a power-on reset. The default `input-device` value is `keyboard` and the default `output-device` value is `screen`. The values of `input-device` and `output-device` must be device specifiers. The aliases `keyboard` and `screen` are often used as the values of these variables.

When the system is reset, the named device becomes the initial firmware console input or output device. (If you want to temporarily change the input or output device, use the `input` or `output` commands described in Chapter 4).

To set `ttya` as the initial console input device, type:

```
ok setenv input-device ttya
ok
```

If you select `keyboard` for `input-device`, and the device is not plugged in, input is accepted from a fallback device (typically `ttya`) after the next power cycle or system reset. If you select `screen` for `output-device`, but no frame buffer is available, output is sent to the fall-back device after the next power cycle or system reset.

To specify an SBus frame buffer as the default output device (especially if there are multiple frame buffers in the system), type:

```
ok setenv output-device /sbus/SUNW,leo
ok
```

Serial Port Characteristics

The following values represent the typical range of communications characteristics for serial ports:

- *baud* = 110, 300, 1200, 2400, 4800, 9600, 19200, or 38400 bits/second
- *#bits* = 5, 6, 7, or 8 (data bits)
- *parity* = n (none), e (even), or o (odd), parity bit
- *#stop* = 1 (1), (1.5), or 2 (2) stop bits

Note – rts/cts and xon/xoff handshaking are not implemented on some systems. When a selected protocol is not implemented, the handshake variable is accepted but ignored; no messages are displayed.

Selecting Boot Options

You can use the `auto-boot?` configuration variable to determine whether or not the system boots automatically after a power cycle or system reset.

If `auto-boot?` is `true` and if OpenBoot is not in diagnostic mode, the system boots automatically after a power-cycle or system reset using the `boot-device` and `boot-file` values.

If `auto-boot?` is `true` and if OpenBoot is in diagnostic mode, the system boots automatically after a power-cycle or system reset using the `diag-device` and `diag-file` values.

These variables can also be used during manual booting to select the boot device and the program to be booted. For example, to specify default booting from the network server, type:

```
ok setenv boot-device net
ok
```

Changes to `boot-file`, `boot-device`, `diag-file`, and `diag-device` take effect the next time that `boot` is executed.

Controlling Power-On Self-Test (POST)

The Power-on Testing variables are:

- `diag-switch?`
- `diag-level`

Setting `diag-switch?` to `true` causes the function `diagnostic-mode?` to return `true`. When `diagnostic-mode?` returns `true`, the system:

- Performs more thorough self tests during any subsequent power-on or system reset process. The coverage of self tests executed depends on the value of `diag-level`. For maximum coverage, set the diag-level value to `max`.
- May display additional status messages (the details are implementation dependent).
- Uses different configuration variables for booting. For more details on the effects on the boot process, see Chapter 2.

Most systems have a factory default of `false` for the `diag-switch?` variable. To set `diag-switch?` to `true`, type:

```
ok setenv diag-switch? true
ok
```

Note – Some systems have a hardware diagnostic switch that also cause `diagnostic-mode?` to return `true`. Such systems run the full tests at power-on and system reset if either the hardware switch is set or `diag-switch?` is `true`.

Note – Some implementations enable you to force `diag-switch?` to `true` by using an implementation-dependent key sequence during power-on. Check your system's documentation for details, or see Appendix C.

To set `diag-switch?` to `false`, type:

```
ok setenv diag-switch? false
ok
```

When not in diagnostic mode, the system does not announce the diagnostic tests as they are performed (unless a test fails) and may perform fewer tests.

Using nvramrc

You can use the nvramrc configuration variable store user-defined commands executed during start-up. The contents of the variable are called the *script*.

Typically, nvramrc is used by a device driver to save start-up configuration variables, to patch device driver code, or to define installation-specific device configuration and device aliases. It can also be used for bug patches or for user-installed extensions. Commands are stored in ASCII, just as the user would type them at the console.

If the use-nvramrc? configuration variable is true, the script is evaluated during the OpenBoot start-up sequence as shown:

1. Perform power-on self-test (POST)
2. Perform system initialization
3. Evaluate the script (if use-nvramrc? is true)
4. Execute probe-all (evaluate FCode)
5. Execute install-console
6. Execute banner
7. Execute secondary diagnostics
8. Perform default boot (if auto-boot? is true)

It is sometimes desirable to modify the sequence probe-all install-console banner. For example, commands that modify the characteristics of plug-in display devices may need to be executed after the plug-in devices have been probed, but before the console device has been selected. Such commands would need to be executed between probe-all and install-console. Commands that display output on the console would need to be placed after install-console or banner.

This is accomplished by creating a custom script which contains either banner or suppress-banner since 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.

Most user interface commands can be used in the script. The following cannot:

- boot
- go

- nvedit
- password
- reset-all
- setenv security-mode

Editing the Contents of the Script

The script editor, nvedit, lets you create and modify the script using the commands listed in TABLE 3-4.

TABLE 3-4 Commands Affecting nvramrc

Command	Description
nvalias *alias device-path*	Stores the command "devalias *alias device-path*" in the script. The alias persists until either nvunalias or set-defaults is executed.
$nvalias	Performs the same function as nvalias except that it takes its arguments, *name-string* and *device-string*, from the stack.
nvedit	Enters the script editor. If data remains in the temporary buffer from a previous nvedit session, resumes editing those previous contents. If not, reads the contents of nvramrc into the temporary buffer and begins editing it.
nvquit	Discards the contents of the temporary buffer, without writing it to nvramrc. Prompts for confirmation.
nvrecover	Recovers the contents of nvramrc if they have been lost as a result of the execution of set-defaults; then enters the editor as with nvedit. nvrecover fails if nvedit is executed between the time that the nvramrc contents were lost and the time that nvrecover is executed.
nvrun	Executes the contents of the temporary buffer.
nvstore	Copies the contents of the temporary buffer to nvramrc; discards the contents of the temporary buffer.
nvunalias *alias*	Deletes the specified alias from nvramrc.
$nvunalias	Performs the same function as nvunalias except that it takes its argument, *name-string*, from the stack.

Use the editing commands shown in TABLE 3-5 in the script editor.

TABLE 3-5 Script Editor Keystroke Commands

Keystroke	Description
Control-B	Moves backward one character.
Escape B	Moves backward one word.
Control-F	Moves forward one character.
Escape F	Moves forward one word.
Control-A	Moves backward to beginning of the line.
Control-E	Moves forward to end of the line.
Control-N	Moves to the next line of the script editing buffer.
Control-P	Moves to the previous line of the script editing buffer.
Return (Enter)	Inserts a new line at the cursor position and advances to the next line.
Control-O	Inserts a new line at the cursor position and stays on the current line.
Control-K	Erases from the cursor position to the end of the line, storing the erased characters in a save buffer. If at the end of a line, joins the next line to the current line (that is, deletes the new line).
Delete	Erases the previous character.
Backspace	Erases the previous character.
Control-H	Erases the previous character.
Escape H	Erases from beginning of word to just before the cursor, storing erased characters in a save buffer.
Control-W	Erases from beginning of word to just before the cursor, storing erased characters in a save buffer.
Control-D	Erases the next character.
Escape D	Erases from the cursor to the end of the word, storing the erased characters in a save buffer.
Control-U	Erases the entire line, storing the erased characters in a save buffer.
Control-Y	Inserts the contents of the save buffer before the cursor.
Control-Q	Quotes the next character (that is, allows you to insert control characters).
Control-R	Retypes the line.
Control-L	Displays the entire contents of the editing buffer.
Control-C	Exits the script editor, returning to the OpenBoot command interpreter. The temporary buffer is preserved, but is not written back to the script. Use `nvstore` afterwards to write it back.

Activating the Script

Use the following steps to create and activate the script:

1. **At the** `ok` **prompt, type** `nvedit`.

   Edit the script using editor commands.

2. **Press Control-C to get out of the editor and back to the** `ok` **prompt.**

   If you have not yet typed `nvstore` to save your changes, you may type `nvrun` to execute the contents of the temporary edit buffer.

3. **Type** `nvstore` **to save your changes.**

4. **Enable the interpretation of the script by typing:**

```
setenv use-nvramrc? true
```

5. **Type** `reset-all` **to reset the system and execute the script, or execute the contents directly by typing:**

```
nvramrc evaluate
```

Example Script

The following example shows you how to create a simple colon definition in the script.

```
ok nvedit
0: : hello ( -- )
1: ." Hello, world. " cr
2: ;
3: ^C
ok nvstore
ok setenv use-nvramrc? true
ok reset-all
…
ok hello
Hello, world.
ok
```

Notice the `nvedit` line number prompts (`0:`, `1:`, `2:`, `3:`) in the above example. These prompts are system-dependent.

CHAPTER 4

Using Forth Tools

This chapter introduces the Forth programming language as it is implemented in OpenBoot. Even if you are familiar with Forth, work through the examples shown in this chapter. The examples provide specific, OpenBoot-related information.

The version of Forth contained in OpenBoot is based on ANS Forth. Appendix D lists the complete set of available commands.

Note – This chapter assumes that you know how to start and leave the user interface. At the `ok` prompt, if you type commands that hang the system and you cannot recover using a key sequence, you may need to perform a power cycle to return the system to normal operation.

Forth Commands

Forth has a very simple command structure. Forth commands, also called Forth *words*, consist of any combination of characters that can be printed. For example, letters, digits, or punctuation marks. Examples of legitimate words are shown below:

- `@`
- `dump`
- `.`
- `0<`
- `+`
- `probe-scsi`

Forth words must be separated from one another by one or more spaces (blanks). Characters that are normally treated as “punctuation” in some other programming languages do not separate Forth words. In fact, many of those “punctuation” characters are Forth words.

Pressing Return at the end of any command line executes the typed commands. In all the examples shown, a Return at the end of the line is assumed.

You can type more than one word at a command line. Multiple words on a line are executed one at a time, from left to right, in the order in which they were typed. For example:

```
ok testa testb testc
ok
```

is equivalent to:

```
ok testa
ok testb
ok testc
ok
```

In OpenBoot, Forth word names are case-insensitive. Therefore, `testa`, `TESTA`, and `TesTa` all invoke the same command. However, words are conventionally written in lowercase.

Some commands generate large amounts of output (for example, `dump` or `words`). You can interrupt such a command by pressing any key except `q`. If you press `q`, the output is aborted, not suspended. Once a command is interrupted, output is suspended and the following message appears:

```
More [<space>,<cr>,q] ?
```

Press the space bar (`<space>`) to continue, press Return (`<cr>`) to output one more line and pause again, or type `q` to abort the command. When a command generates more than one page of output, the system automatically displays this prompt at the end of each page.

Data Types

The terms shown in TABLE 4-1 describe the data types used by Forth.

TABLE 4-1 Forth Data Type Definitions

Notation	Description
byte	An 8-bit value.
cell	The implementation-defined fixed size of a cell is specified in address units and the corresponding number of bits. Data-stack elements, return-stack elements, addresses, execution tokens, flags, and integers are one cell wide. On OpenBoot systems, a cell consists of at least 32-bits, and is sufficiently large to contain a virtual address. The cell size may vary between implementations. A 32-bit implementation has a cell size of 4. A 64-bit implementation has a cell size of 8. OpenBoot 4.*x* is a 64-bit implementation.
doublet	A 16-bit value.
octlet	A 64-bit value; only defined on 64-bit implementations,
quadlet	A 32-bit value.

Using Numbers

Enter a number by typing its value, for example, 55 or -123. Forth accepts only integers (whole numbers); it does not understand fractional values (for example, 2/3). A period at the end of a number signifies a double number. Periods or commas embedded in a number are ignored, so 5.77 is understood as 577. By convention, such punctuation usually appears every four digits. Use one or more spaces to separate a number from a word or from another number.

Unless otherwise specified, OpenBoot performs integer arithmetic on data items that are one cell in size, and creates results that are one cell in size.

Although OpenBoot implementations are encouraged to use base 16 (hexadecimal) by default, they are not required to do so. Consequently, you must establish a specific number base if your code depends on a given base for proper operation. You can change the number base with the commands `decimal` and `hex` to cause all subsequent numeric input and output to be performed in base 10 or 16, respectively.

For example, to operate in decimal, type:

```
ok decimal
ok
```

To change to hexadecimal, type:

```
ok hex
ok
```

To identify the current number base, you can use:

```
ok 10 .d
16
ok
```

The `16` on the display shows that you are operating in hexadecimal. If `10` appeared on the display, it would mean that you are in decimal base. The `.d` command displays a number in base 10, regardless of the current number base.

Forth Stack

The Forth *stack* is a last-in, first-out buffer used for temporarily holding numeric information. Think of it as a stack of books: the last one you put on the top of the stack is the first one you take off. Understanding the stack is essential to using Forth.

To put a number on the stack, simply type its value.

```
ok 44  (The value 44 is now on top of the stack)
ok 7 (The value 7 is now on top, with 44 just underneath)
ok
```

Displaying Stack Contents

The contents of the stack are normally invisible. However, properly visualizing the current stack contents is important for achieving the desired result. To show the stack contents with every ok prompt, type:

```
ok showstack
44 7 ok 8
44 7 8 ok noshowstack
ok
```

The topmost stack item is always shown as the last item in the list, immediately before the ok prompt. In the above example, the topmost stack item is 8.

If showstack has been previously executed, noshowstack removes the stack display prior to each prompt.

Note – In some of the examples in this chapter, showstack is enabled. In those examples, each ok prompt is immediately preceded by a display of the current contents of the stack. The examples work the same if showstack is not enabled, except that the stack contents are not displayed.

Nearly all words that require numeric parameters fetch those parameters from the top of the stack. Any values returned are generally left on top of the stack, where they can be viewed or consumed by another command. For example, the Forth word + removes two numbers from the stack, adds them together, and leaves the result on the stack. In the example below, all arithmetic is in hexadecimal.

```
44 7 8 ok +
44 f ok +
53 ok
```

Once the two values are added together, the result is put onto the top of the stack. The Forth word . removes the top stack item and displays that value on the screen. For example:

```
53 ok 12
53 12 ok .
12
53 ok .
53
ok   (The stack is now empty)
ok 3 5 + .
8
ok   (The stack is now empty)
ok .
Stack Underflow
ok
```

Stack Diagram

To aid understanding, conventional coding style requires that a *stack diagram* of the form (--) appear on the first line of every definition of a Forth word. The stack diagram specifies what the execution of the word does to the stack.

Entries to the left of -- represent those stack items that the word removes from the stack and uses during its operation. The right-most of these items is on top of the stack, with any preceding items beneath it. In other words, arguments are pushed onto the stack in left to right order, leaving the most recent one (the right-most one in the diagram) on the top.

Entries to the right of -- represent those stack items that the word leaves on the stack after it finishes execution. Again, the right-most item is on top of the stack, with any preceding items beneath it.

For example, a stack diagram for the word + is:

```
( nu1 nu2 -- sum )
```

Therefore, + removes two numbers (nu1 and nu2) from the stack and leaves their sum (sum) on the stack. As a second example, a stack diagram for the word . is:

```
( nu -- )
```

The word . removes the number on the top of the stack (nu) and displays the number.

Words that have no effect on the contents of the stack (such as showstack or decimal), have a (--) stack diagram.

Occasionally, a word requires another word or other text immediately following it on the command line. The word `see`, used in the following form, is such an example:

```
see thisword
```

Stack items are generally written using descriptive names to help clarify correct usage. See TABLE 4-2 for stack item abbreviations used in this manual.

TABLE 4-2 Stack Item Notation

Notation	Description
|	Alternate stack results shown with space, for example, (`input -- addr len false | result true`).
|	Alternate stack items shown without space, for example, (`input -- addr len|0 result`).
`???`	Unknown stack item(s).
…	Unknown stack item(s). If used on both sides of a stack comment, means the same stack items are present on both sides.
`< > <space>`	Space delimiter. Leading spaces are ignored.
`a-addr`	Variable-aligned address.
`addr`	Memory address (generally a virtual address).
`addr len`	Address and length for memory region.
`byte b`*xxx*	8-bit value (low order byte in a cell).
`char`	7-bit value (low order byte in a cell, high bit of low order byte unspecified).
`cnt`	Count.
`len`	Length.
`size`	Count or length.
`d`*xxx*	Double (extended-precision) numbers. Two cells, most significant cell on top of stack.
`<eol>`	End-of-line delimiter.
`false`	0 (false flag).
`n n1 n2 n3`	Normal signed, one-cell values.
`nu nu1`	Signed or unsigned one-cell values.
`<nothing>`	Zero stack items.
`o o1 o2 oct1 oct2`	Octlet (64-bit signed value).

TABLE 4-2 Stack Item Notation *(Continued)*

Notation	Description
`oaddr`	Octlet (64-bit) aligned address.
`octlet`	An 8-byte quantity.
`phys`	Physical address (actual hardware address).
`phys.lo phys.hi`	Lower / upper cell of physical address.
`pstr`	Packed string.
`quad q`*xxx*	Quadlet (32-bit value, low order four bytes in a cell).
`qaddr`	Quadlet (32-bit) aligned address.
`true`	-1 (true flag).
u*xxx*	Unsigned positive, one-cell values.
`virt`	Virtual address (address used by software).
`waddr`	Doublet (16-bit) aligned address.
`word w`*xxx*	Doublet (16-bit value, low order two bytes in a cell).
`x x1`	Arbitrary, one cell stack item.
`x.lo x.hi`	Low / high significant bits of a data item.
`xt`	Execution token.
`xxx?`	Flag. Name indicates usage, e.g. `done? ok? error?`
`xyz-str xyz-len`	Address and length for unpacked string.
`xyz-sys`	Control-flow stack items, implementation-dependent.
`( C: -- )`	Compilation stack diagram.
`( -- )( E: -- )`	Execution stack diagram.
`( R: -- )`	Return stack diagram.

Manipulating the Stack

Stack manipulation commands (described in TABLE 4-1) allow you to add, delete, and reorder items on the stack.

TABLE 4-3 Stack Manipulation Commands

Command	Stack Diagram	Description
`clear`	`( ??? -- )`	Empties the stack.
`depth`	`( ... -- ... u )`	Returns the number of items on the stack.
`drop`	`( x -- )`	Removes top item from the stack.
`2drop`	`( x1 x2 -- )`	Removes 2 items from the stack.
`3drop`	`( x1 x2 x3 -- )`	Removes 3 items from the stack.
`dup`	`( x -- x x )`	Duplicates the top stack item.
`2dup`	`( x1 x2 -- x1 x2 x1 x2 )`	Duplicates 2 stack items.
`3dup`	`( x1 x2 x3 -- x1 x2 x3 x1 x2 x3 )`	Duplicates 3 stack items.
`?dup`	`( x -- x x | 0 )`	Duplicates the top stack item if it is non-zero.
`nip`	`( x1 x2 -- x2 )`	Discards the second stack item.
`over`	`( x1 x2 -- x1 x2 x1 )`	Copies second stack item to top of stack.
`2over`	`( x1 x2 x3 x4 -- x1 x2 x3 x4 x1 x2 )`	Copies second 2 stack items.
`pick`	`( xu … x1 x0 u -- xu … x1 x0 xu )`	Copies *u*-th stack item (`1 pick` = `over`).
`>r`	`( x -- ) (R: -- x )`	Moves a stack item to the return stack.
`r>`	`( -- x ) ( R: x -- )`	Moves a return stack item to the stack.
`r@`	`( -- x ) ( R: x -- x )`	Copies the top of the return stack to the stack.
`roll`	`( xu … x1 x0 u -- xu-1 … x1 x0 xu )`	Rotates `u` stack items (`2 roll` = `rot`).
`rot`	`( x1 x2 x3 -- x2 x3 x1 )`	Rotates 3 stack items.

TABLE 4-3 Stack Manipulation Commands *(Continued)*

Command	Stack Diagram	Description
`-rot`	`( x1 x2 x3 -- x3 x1 x2 )`	Inversely rotates 3 stack items.
`2rot`	`( x1 x2 x3 x4 x5 x6 -- x3 x4 x5 x6 x1 x2 )`	Rotates 3 pairs of stack items.
`swap`	(x1 x2 -- x2 x1)	Exchanges the top 2 stack items.
`2swap`	(x1 x2 x3 x4 -- x3 x4 x1 x2)	Exchanges 2 pairs of stack items.
`tuck`	(x1 x2 -- x2 x1 x2)	Copies top stack item below second item.

A typical use of stack manipulation is to display the top stack item while preserving all stack items, as shown in this example:

```
5 77 ok dup  (Duplicates the top item on the stack)
5 77 77 ok .   (Removes and displays the top stack item)
77
5 77 ok
```

Creating Custom Definitions

Forth provides an easy way to create new command words from sequences of existing words. TABLE 4-4 shows the Forth words used to create such new words.

TABLE 4-4 Colon Definition Words

Command	Stack Diagram	Description
: *new-name*	(--)	Starts a new colon definition of the word *new-name*.
;	(--)	Ends a colon definition.

This kind of word is called a *colon definition*, named after the word that is used to create them. For example, suppose you want to create a new word, add4, that adds any four numbers together and displays the result. You could create the definition as follows:

```
ok : add4  + + + .  ;
ok
```

The ; (semicolon) marks the end of the definition that defines add4 to have the behavior (+ + + .). The three addition operators (+) reduce the four stack items to a single sum on the stack; then . removes and displays that result. An example follows.

```
ok 1 2 3 3 + + + .
9
ok 1 2 3 3 add4
9
ok
```

Definitions are forgotten if a system reset takes place. To keep useful definitions, put them into the script or save them in a text file on a host system. This text file can then be loaded as needed. (See Chapter 5 for more information on loading files.)

When you type a definition from the user interface, the ok prompt becomes a] (right square bracket) prompt after you type the : (colon) and before you type the ; (semicolon). For example, you could type the definition for add4 like this:

```
ok : add4
]  + + +
]  .
]  ;
ok
```

The above use of] while inside a multi-line definition is a characteristic of Sun's implementation.

- The stack diagram shows proper use of a word, so include a stack diagram with every definition you create, even if the stack effect is nil (--). Use generous stack comments in complex definitions to trace the flow of execution. For example, when creating add4, you could define it as:

```
ok : add4  ( n1 n2 n3 n4 -- )  + + + .  ;
```

Or you could define it as follows:

```
ok : add4  ( n1 n2 n3 n4 -- )
]  + + + ( sum )
]  .
]  ;
```

Note – The ”(“ is a Forth word meaning ignore the text up to the “)”. Like any other Forth word, the “(“ must have one or more spaces after it.

Using Arithmetic Functions

Single-Precision Integer Arithmetic

The commands listed in TABLE 4-5 perform single-precision arithmetic.

TABLE 4-5 Single-Precision Arithmetic Functions

Command	Stack Diagram	Description
+	(nu1 nu2 -- sum)	Adds *nu1* + *nu2*.
-	(nu1 nu2 -- diff)	Subtracts *nu1* - *nu2*.
*	(nu1 nu2 -- prod)	Multiplies *nu1* times *nu2*.
*/	(n1 n2 n3 -- quot)	Calculates *nu1* * *nu2* / *n3*. Inputs, outputs and intermediate products are all one cell.
/	(n1 n2 -- quot)	Divides *n1* by *n2*; remainder is discarded.
1+	(nu1 -- nu2)	Adds one.
1-	(nu1 -- nu2)	Subtracts one.
2+	(nu1 -- nu2)	Adds two.
2-	(nu1 -- nu2)	Subtracts two.
abs	(n -- u)	Absolute value.

TABLE 4-5 Single-Precision Arithmetic Functions *(Continued)*

Command	Stack Diagram	Description
bounds	(start len -- len+start start)	Converts start,len to end,start for do or ?do loop.
even	(n -- n | n+1)	Rounds to nearest even integer >= *n*.
max	(n1 n2 -- n3)	*n3* is maximum of *n1* and *n2*.
min	(n1 n2 -- n3)	*n3* is minimum of *n1* and *n2*.
mod	(n1 n2 -- rem)	Remainder of *n1* / *n2*.
*/mod	(n1 n2 n3 -- rem quot)	Remainder, quotient of *n1* * *n2* / *n3*.
/mod	(n1 n2 -- rem quot)	Remainder, quotient of *n1* / *n2*.
negate	(n1 -- n2)	Changes the sign of *n1*.
u*	(u1 u2 -- uprod)	Multiplies 2 unsigned numbers, yielding an unsigned product.
u/mod	(u1 u2 -- urem uquot)	Divides unsigned one-cell number by an unsigned one-cell number; yields one-cell remainder and quotient.
<<	(x1 u -- x2)	Synonym for lshift.
>>	(x1 u -- x2)	Synonym for rshift.
2*	(x1 -- x2)	Multiplies by 2.
2/	(x1 -- x2)	Divides by 2.
>>a	(x1 u -- x2)	Arithmetic right-shift *x1* by *u* bits.
and	(x1 x2 -- x3)	Bitwise logical AND.
invert	(x1 -- x2)	Inverts all bits of *x1*.
lshift	(x1 u -- x2)	Left-shifts *x1* by *u* bits. Zero-fill low bits.
not	(x1 -- x2)	Synonym for invert.
or	(x1 x2 -- x3)	Bitwise logical OR.
rshift	(x1 u -- x2)	Right-shifts *x1* by *u* bits. Zero-fill high bits.
u2/	(x1 -- x2)	Logical right shift 1 bit; zero shifted into high bit.
xor	(x1 x2 -- x3)	Bitwise exclusive OR.

Double Number Arithmetic

The commands listed in TABLE 4-6 perform double number arithmetic.

TABLE 4-6 Double Number Arithmetic Functions

Command	Stack Diagram	Description
`d+`	`( d1 d2 -- d.sum )`	Adds *d1* to *d2*, yielding double number *d.sum*.
`d-`	`( d1 d2 --d.diff )`	Subtracts *d2* from *d1*, yielding double number *d.diff*.
`fm/mod`	`( d n -- rem quot )`	Divides *d* by *n*.
`m*`	`( n1 n2 -- d )`	Signed multiply with double-number product.
`s>d`	`( n1 -- d1 )`	Converts a number to a double number.
`sm/rem`	`( d n -- rem quot )`	Divides *d* by *n*, symmetric division.
`um*`	`( u1 u2 -- ud )`	Unsigned multiply yielding unsigned double number product.
`um/mod`	`( ud u -- urem uprod )`	Divides *ud* by *u*.

Data Type Conversion

The commands listed in TABLE 4-7 perform data type conversion.

TABLE 4-7 32-Bit Data Type Conversion Functions

Command	Stack Diagram	Description
`bljoin`	`( b.low b2 b3 b.hi -- quad )`	Joins four bytes to form a quadlet
`bwjoin`	`( b.low b.hi -- word )`	Joins two bytes to form a doublet.
`lbflip`	`( quad1 -- quad2 )`	Reverses the bytes in a quadlet.
`lbsplit`	`( quad -- b.low b2 b3 b.hi )`	Splits a quadlet into four bytes.
`lwflip`	`( quad1 -- quad2 )`	Swaps the doublets in a quadlet.
`lwsplit`	`( quad -- w.low w.hi )`	Splits a quadlet into two doublets.

TABLE 4-7 32-Bit Data Type Conversion Functions *(Continued)*

Command	Stack Diagram	Description
`wbflip`	`( word1 -- word2 )`	Swaps the bytes in a doublet.
`wbsplit`	`( word -- b.low b.hi )`	Splits a doublet into two bytes.
`wljoin`	`( w.low w.hi -- quad )`	Joins two doublets to form a quadlet.

The data type conversion commands listed in TABLE 4-8 are available only on 64-bit OpenBoot implementations.

TABLE 4-8 64-Bit Data Type Conversion Functions

Command	Stack Diagram	Description
`bxjoin`	`( b.lo b.2 b.3 b.4 b.5 b.6 b.7 b.hi -- o )`	Joins eight bytes to form an octlet.
`lxjoin`	`( quad.lo quad.hi -- o )`	Joins two quadlets to form an octlet.
`wxjoin`	(w.lo w.2 w.3 w.hi -- o)	Joins four doublets to form an octlet.
`xbflip`	(oct1 -- oct2)	Reverses the bytes in an octlet.
`xbsplit`	(o -- b.lo b.2 b.3 b.4 b.5 b.6 b.7 b.hi)	Splits an octlet into eight bytes.
`xlflip`	(oct1 -- oct2)	Reverses the quadlets in an octlet. The bytes in each quadlet are not reversed.
`xlsplit`	(o -- quad.lo quad.hi)	Splits an octlet into two quadlets.
`xwflip`	(oct1 -- oct2)	Reverses the doublets in an octlet. The bytes in each doublet are not reversed.
`xwsplit`	(o -- w.lo w.2 w.3 w.hi)	Splits an octlet into four doublets.

Address Arithmetic

The commands listed in TABLE 4-9 perform address arithmetic.

TABLE 4-9 Address Arithmetic Functions

Command	Stack Diagram	Description
`aligned`	`( n1 -- n1 | a-addr)`	Increases *n1* if necessary to yield a variable aligned address.
`/c`	`( -- n )`	The number of address units to a byte: 1.
`/c*`	`( nu1 -- nu2 )`	Synonym for `chars`.
`ca+`	`( addr1 index -- addr2 )`	Increments *addr1* by *index* times the value of `/c`.
`ca1+`	`( addr1 -- addr2 )`	Synonym for `char+`.
`cell+`	`( addr1 -- addr2 )`	Increments *addr1* by the value of `/n`.
`cells`	`( nu1 -- nu2 )`	Multiplies *nu1* by the value of `/n`.
`char+`	`( addr1 -- addr2 )`	Increments *addr1* by the value of `/c`.
`chars`	`( nu1 -- nu2 )`	Multiplies *nu1* by the value of `/c`.
`/l`	`( -- n )`	Number of address units to a quadlet; typically 4.
`/l*`	`( nu1 -- nu2 )`	Multiplies *nu1* by the value of `/l`.
`la+`	`( addr1 index -- addr2 )`	Increments *addr1* by *index* times the value of `/l`.
`la1+`	`( addr1 -- addr2 )`	Increments *addr1* by the value of `/l`.
`/n`	`( -- n )`	Number of address units in a cell.
`/n*`	`( nu1 -- nu2 )`	Synonym for `cells`.
`na+`	`( addr1 index -- addr2 )`	Increments *addr1* by *index* times the value of `/n`.
`na1+`	`( addr1 -- addr2 )`	Synonym for `cell+`.
`/w`	`( -- n )`	Number of address units to a doublet; typically 2.
`/w*`	`( nu1 -- nu2 )`	Multiplies *nu1* by the value of `/w`.
`wa+`	`( addr1 index -- addr2 )`	Increments *addr1* by *index* times the value of `/w`.
`wa1+`	`( addr1 -- addr2 )`	Increments *addr1* by the value of `/w`.

The address arithmetic commands listed in TABLE 4-10 are available only on 64-bit OpenBoot implementations.

TABLE 4-10 64-Bit Address Arithmetic Functions

Command	Stack Diagram	Description
`/x`	`( -- n )`	Number of address units in an octlet, typically 8.
`/x*`	`( nu1 -- nu2 )`	Multiplies *nu1* by the value of `/x`.
`xa+`	`( addr1 index -- addr2 )`	Increments *addr1* by *index* times the value of `/x`.
`xa1+`	`( addr1 -- addr2 )`	Increments *addr1* by the value of `/x`.

Accessing Memory

Virtual Memory

The user interface provides interactive commands for examining and setting memory. With the user interface, you can:

- Read and write to any virtual address.
- Map virtual addresses to physical addresses.

Memory operators let you read from and write to any memory location. All memory addresses shown in the examples that follow are virtual addresses.

A variety of 8-bit, 16-bit, and 32-bit (and in some systems, 64-bit) operations are provided. In general:

- a `c` (character) prefix indicates an 8-bit (one byte) operation
- a `w` (word) prefix indicates a 16-bit (doublet) operation
- an `l` (longword) prefix indicates a 32-bit (quadlet) operation
- an `x` prefix indicates a 64-bit (octlet) operation

waddr, qaddr, and oaddr indicate addresses with alignment restrictions. For example, qaddr indicates 32-bit (4 byte) alignment; on many systems such an address must be a multiple of 4, as shown in the following example:

```
ok 4028 l@
ok 4029 l@
Memory address not aligned
ok
```

Forth, as implemented in OpenBoot, adheres closely to the ANS Forth Standard. If you explicitly want a 16-bit fetch, a 32-bit fetch or (on some systems) a 64-bit fetch, use w@, l@ or x@ instead of @. Other memory and device register access commands also follow this convention.

TABLE 4-11 lists commands used to access memory.

TABLE 4-11 Memory Access Commands

Command	Stack Diagram	Description
!	(x a-addr --)	Stores a number at *a-addr*.
+!	(nu a-addr --)	Adds *nu* to the number stored at *a-addr*.
@	(a-addr -- x)	Fetches a number from *a-addr*.
2!	(x1 x2 a-addr --)	Stores 2 numbers at *a-addr*, *x2* at lower address.
2@	(a-addr -- x1 x2)	Fetches 2 numbers from *a-addr*, *x2* from lower address.
blank	(addr len --)	Sets *len* bytes of memory beginning at *addr* to the space character (decimal 32).
c!	(byte addr --)	Stores *byte* at *addr*.
c@	(addr -- byte)	Fetches a *byte* from *addr*.
cpeek	(addr -- false | byte true)	Attempts to fetch the byte at *addr*. Returns the data and true if the access was successful. Returns false if a read access error occurred.
cpoke	(byte addr -- okay?)	Attempts to store the *byte* to *addr*. Returns true if the access was successful. Returns false if a write access error occurred.

TABLE 4-11 Memory Access Commands *(Continued)*

Command	Stack Diagram	Description
`comp`	`( addr1 addr2 len -- diff? )`	Compares two byte arrays. *diff?* is 0 if the arrays are identical, *diff?* is -1 if the first byte that is different is lesser in the string at *addr1*, *diff?* is 1 otherwise.
`dump`	`( addr len -- )`	Displays *len* bytes of memory starting at *addr*.
`erase`	`( addr len -- )`	Sets *len* bytes of memory beginning at *addr* to `0`.
`fill`	`( addr len byte -- )`	Sets *len* bytes of memory beginning at *addr* to the value *byte*.
`l!`	`( q qaddr -- )`	Stores a quadlet *q* at *qaddr*.
`l@`	`( qaddr -- q )`	Fetches a quadlet *q* from *qaddr*.
`lbflips`	`( qaddr len -- )`	Reverses the bytes in each quadlet in the specified region.
`lwflips`	`( qaddr len -- )`	Swaps the doublets in each quadlet in specified region.
`lpeek`	`( qaddr -- false | quad true )`	Attempts to fetch the quadlet at *qaddr*. Returns the data and `true` if the access was successful. Returns `false` if a read access error occurred.
`lpoke`	`( q qaddr -- okay? )`	Attempts to store the quadlet *q* at *qaddr*. Returns `true` if the access was successful. Returns `false` if a write access error occurred.
`move`	`( src-addr dest-addr len -- )`	Copies *len* bytes from *src-addr* to *dest-addr*.
`off`	`( a-addr -- )`	Stores `false` at *a-addr*.
`on`	`( a-addr -- )`	Stores `true` at *a-addr*.
`unaligned-l!`	`( q addr -- )`	Stores a quadlet *q*, any alignment
`unaligned-l@`	`( addr -- q )`	Fetches a quadlet *q*, any alignment.
`unaligned-w!`	`( w addr -- )`	Stores a doublet *w*, any alignment.
`unaligned-w@`	`( addr -- w )`	Fetches a doublet *w*, any alignment.
`w!`	`( w waddr -- )`	Stores a doublet *w* at *waddr*.
`w@`	`( waddr -- w)`	Fetches a doublet *w* from *waddr*.
`<w@`	`( waddr -- n )`	Fetches a doublet *n* from *waddr*, sign-extended.

TABLE 4-11 Memory Access Commands *(Continued)*

Command	Stack Diagram	Description
wbflips	(waddr len --)	Swaps the bytes in each doublet in the specified region.
wpeek	(waddr -- false | w true)	Attempts to fetch the doublet *w* at *waddr*. Returns the data and `true` if the access was successful. Returns `false` if a read access error occurred.
wpoke	(w waddr -- okay?)	Attempts to store the doublet *w* to *waddr*. Returns `true` if the access was successful. Returns `false` if a write access error occurred.

The memory access commands listed in TABLE 4-12 are available only on 64-bit OpenBoot implementations.

TABLE 4-12 64-Bit Memory Access Functions

Command	Stack Diagram	Description
<l@	(qaddr -- n)	Fetches quadlet from *qaddr*, sign-extended.
x@	(oaddr -- o)	Fetches octlet from an octlet aligned address.
x!	(o oaddr --)	Stores octlet to an octlet aligned address.
xbflips	(oaddr len --)	Reverses the bytes in each octlet in the given region.The behavior is undefined if *len* is not a multiple of /x.
xlflips	(oaddr len --)	Reverses the quadlets in each octlet in the given region. The bytes in each quadlet are not reversed. The behavior is undefined if *len* is not a multiple of /x.
xwflips	(oaddr len --)	Reverses the doublets in each octlet in the given region. The bytes in each doublet are not reversed. The behavior is undefined if *len* is not a multiple of /x.

The dump command is particularly useful. It displays a region of memory as both bytes and ASCII values. The example below displays the contents of 20 bytes of memory starting at virtual address 10000.

```
ok 10000 20 dump(Display 20 bytes of memory starting at virtual address 10000)
       \/ 1  2  3  4  5  6  7   8  9  a  b  c  d  e  f  v123456789abcdef
 10000 05 75 6e 74 69 6c 00 40  4e d4 00 00 da 18 00 00 .until.@NT..Z...
 10010 ce da 00 00 f4 f4 00 00  fe dc 00 00 d3 0c 00 00 NZ..tt..~\..S...
ok
```

Some implementations support variants of `dump` that display memory as 16-, 32- and 64-bit values. You can use `sifting dump` to find out if your system has such variants. See “Searching the Dictionary” on page 69.

If you try to access an invalid memory location (with `@`, for example), the operation may abort and display an error message, such as `Data Access Exception` or `Bus Error`.

TABLE 4-13 lists memory mapping commands.

TABLE 4-13 Memory Mapping Commands

Command	Stack Diagram	Description
`alloc-mem`	`( len -- a-addr )`	Allocates *len* bytes of memory; return the virtual address.
`free-mem`	`( a-addr len -- )`	Frees memory allocated by `alloc-mem`.

The following example shows the use of `alloc-mem` and `free-mem`.

- `alloc-mem` allocates 4000 bytes of memory, and displays the starting address (`ef7a48`) of the reserved area.
- `dump` displays the contents of 20 bytes of memory starting at `ef7a48`.
- This region of memory is then filled with the value `55`.
- Finally, `free-mem` returns the 4000 allocated bytes of memory starting at `ef7a48`.

```
ok
ok 4000 alloc-mem .
ef7a48
ok
ok ef7a48 constant temp
ok temp 20 dump
        0  1  2  3  4  5  6  7  \/  9  a  b  c  d  e  f   01234567v9abcdef
ef7a40  00 00 f5 5f 00 00 40 08  ff ef c4 40 ff ef 03 c8  ..u_..@..oD@.o.H
ef7a50  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
ef7a60  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
ok temp 20 55 fill
ok temp 20 dump
        0  1  2  3  4  5  6  7  \/  9  a  b  c  d  e  f   01234567v9abcdef
ef7a40  00 00 f5 5f 00 00 40 08  55 55 55 55 55 55 55 55  ..u_..@.UUUUUUUU
ef7a50  55 55 55 55 55 55 55 55  55 55 55 55 55 55 55 55  UUUUUUUUUUUUUUUU
ef7a60  55 55 55 55 55 55 55 55  00 00 00 00 00 00 00 00  UUUUUUUU........
ok
ok temp 4000 free-mem
ok
```

Device Registers

Device registers cannot be reliably accessed using the virtual memory access operators discussed in the last section. There are special operators for accessing device registers, and these operators require that the system be properly set up prior to their use. For a detailed explanation of this topic, see *Writing FCode 3.x Programs.*

Using Defining Words

The dictionary contains all the available Forth words. Forth *defining words* create new Forth words.

Defining words require two stack diagrams. The first diagram shows the stack effect when the new word is created. The second (or "Execution:") diagram shows the stack effect when that word is later executed.

TABLE 4-14 lists the defining words that you can use to create new Forth words.

If you create a Forth command with the same name as an existing command, the new command is created normally. Depending on the implementation, a warning message `new-name isn't unique` may be displayed. Previous uses of that command name remain unaffected. Subsequent uses of that command name use the latest definition of that command name. To correct the original definition such that *all* uses of the command name get the corrected behavior, make the correction with `patch`. For further details, see "Using `patch` and `(patch)`" on page 107.

TABLE 4-14 Defining Words

Command	Stack Diagram	Description
`:` *name*	(--)(E: … -- ???)	Begins creation of a colon definition.
`;`	(--)	Ends creation of a colon definition.
`alias` *new-name old-name*	(--) (E: … -- ???)	Creates *new-name* with the same behavior as *old-name*.
`buffer:` *name*	(size --)(E: -- a-addr)	Creates a named data buffer. *name* returns *a-addr*.
`constant` *name*	(x --)(E: -- x)	Creates a constant (for example, `3 constant bar`).
`2constant` *name*	(x1 x2 --)(E: -- x1 x2)	Creates a 2-number constant.

TABLE 4-14 Defining Words *(Continued)*

Command	Stack Diagram	Description
`create` *name*	(--)(E: -- a-addr)	Creates a new command whose behavior will be set by further commands.
`$create`	(name-str name-len --)	`create` using the name specified by *name-str name-len*.
`defer` *name*	(--)(E: … -- ???)	Creates a command with alterable behavior. Alters with `to`.
`does>`	(… -- … a-addr)(E: … -- ???)	Specifies the run-time behavior of a `created` word.
`field` *name*	(offset size -- offset+size)(E: addr -- addr+offset)	Creates a field offset pointer named *name*.
`struct`	(-- 0)	Starts a `struct`…`field` definition.
`value` *name*	(x --)(E: -- x)	Creates a named variable. Changes with `to`.
`variable` *name*	(--)(E: -- a-addr)	Creates a named variable. *name* returns *a-addr*.

`value` lets you create a name for a numerical value that can be changed. Later execution of that name leaves the assigned value on the stack. The following example creates a word named `foo` with an initial value of 22, and then calls `foo` to use its value in an arithmetic operation.

```
ok 22 value foo
ok foo 3 + .
25
ok
```

The value can be changed with the word `to`. For example:

```
ok 43 value thisval
ok thisval .
43
ok 10 to thisval
ok thisval .
10
ok
```

Words created with `value` are convenient, because you do not have to use `@` to retrieve their values.

The defining word `variable` creates a name with an associated one-cell memory location. Later execution of that name leaves the address of the memory on the stack. `@` and `!` are used to read or write to that address. For example:

```
ok variable bar
ok 33 bar !
ok bar @ 2 + .
35
ok
```

The defining word `defer` creates a word whose behavior can be changed later, by creating a slot that can be loaded with different behaviors at different times. For example:

```
ok hex
ok defer printit
ok ['] .d  to  printit
ok ff printit
255
ok : myprint ( n -- ) ." It is " .h
] ." in hex " ;
ok ['] myprint to printit
ok ff printit
It is ff in hex
ok
```

Searching the Dictionary

The dictionary contains all the available Forth words. TABLE 4-15 lists some useful tools for searching the dictionary. Please note that some of these tools work only with methods or commands while others work with all types of words (including, for example, variables and values).

TABLE 4-15 Dictionary Searching Commands

Command	Stack Diagram	Description
`'` *name*	(-- xt)	Finds the named word in the dictionary. Returns the execution token. Uses outside definitions.
`[']` *name*	(-- xt)	Similar to `'` but is used either inside or outside definitions.
`.calls`	(xt --)	Displays a list of all commands which use the execution token *xt*.
`$find`	(str len -- xt true | str len false)	Searches for word named by *str,len*. If found, leaves *xt* and *true* on stack. If not found, leaves name string and *false* on stack.
`find`	(pstr -- xt n | pstr false)	Searches for word named by *pstr*. If found, leaves *xt* and *true* on stack. If not found, leaves name string and *false* on stack. (We recommend using `$find` to avoid using packed strings.)
`see` *thisword*	(--)	Decompiles the specified word.
`(see)`	(xt --)	Decompiles the word whose execution token is *xt*.
`$sift`	(text-addr text-len --)	Displays all command names containing text-string.
`sifting` *text*	(--)	Displays all command names containing *text*. *text* contains no spaces.
`words`	(--)	Displays the names of words in the dictionary as described below.

Before you can understand the operation of the dictionary searching tools, you need to understand how words become *visible*. If there is an *active package* at the time a word is defined, the new word becomes a method of the active package, and is visible only when that package is the active package. The commands `dev` and `find-device` can be used to select or change the active package. The command `device-end` deselects the currently active package leaving no active package.

If there is no active package at the time a word is defined, the word is *globally visible* (that is, not specific to a particular package and always available).

The dictionary searching commands first search through the words of the active package, if there is one, and then through the globally visible words.

Note – The Forth commands `only` and `also` affect which words are visible.

You can use `.calls` to locate all of the Forth commands that use a specified word in their definition. `.calls` takes an execution token from the stack and searches the entire dictionary to produce a listing of the names and addresses of every Forth command which uses that execution token. For example:

```
ok ' input .calls
    Called from input at 1e248d8
    Called from io at 1e24ac0
    Called from install-console at 1e33598
    Called from install-console at 1e33678
ok
```

`see`, used in the form:

`see` *thisword*

displays a "pretty-printed" listing of the source for *thisword* (without the comments, of course). For example:

```
ok see see
: see
    ' ['] (see) catch if
    drop
    then
;
ok
```

For more details on the use of `see`, refer to "Using `patch` and `(patch)`" on page 107.

`sifting` takes a string from the input stream and searches vocabularies in the dictionary search order to find every command name that contains the specified string as shown in the following screen.

```
ok sifting input

   In vocabulary options
(1e333f8) input-device
   In vocabulary forth
(1e2476c) input (1e0a9b4) set-input (1e0a978) restore-input
(1e0a940) save-input (1e0a7f0) more-input? (1e086cc) input-file
ok
```

`words` displays all the visible word names in the dictionary, starting with the most recent definition. If a node is currently selected (as with `dev`), the list produced by `words` is limited to the words in that selected node.

Compiling Data Into the Dictionary

The commands listed in TABLE 4-16 control the compilation of data into the dictionary.

TABLE 4-16 Dictionary Compilation Commands

Command	Stack Diagram	Description
,	(n --)	Places a number in the dictionary.
c,	(byte --)	Places a byte in the dictionary.
w,	(word --)	Place a 16-bit number in the dictionary.
l,	(quad --)	Places a 32-bit number in the dictionary.
[	(--)	Begins interpreting.
]	(--)	Ends interpreting, resumes compilation.
allot	(n --)	Allocates *n* bytes in the dictionary.
>body	(xt -- a-addr)	Finds the data field address from the execution token.
body>	(a-addr -- xt)	Finds the execution token from the data field address.

TABLE 4-16 Dictionary Compilation Commands *(Continued)*

Command	Stack Diagram	Description
`compile`	(--)	Compiles the next word at run time. (Recommend using `postpone` instead.)
`[compile] name`	(--)	Compiles the next (immediate) word. (Recommend using `postpone` instead.)
`here`	(-- addr)	Address of top of dictionary.
`immediate`	(--)	Marks the last definition as immediate.
`to` *name*	(n --)	Installs a new action in a `defer` word or `value`.
`literal`	(n --)	Compiles a number.
`origin`	(-- addr)	Returns the address of the start of the Forth system.
`patch` *new-word old-word word-to-patch*	(--)	Replaces *old-word* with *new-word* in *word-to-patch*.
`(patch)`	(new-n old-n xt --)	Replaces *old-n* with *new-n* in word indicated by *xt*.
`postpone` *name*	(--)	Delays the execution of the word name.
`recurse`	(... -- ???)	Compiles a recursive call to the word being compiled.
`recursive`	(--)	Makes the name of the colon definition being compiled visible in the dictionary, and thus allows the name of the word to be used recursively in its own definition.
`state`	(-- addr)	Variable that is non-zero in compile state.

The dictionary compilation commands listed in TABLE 4-17 are available only on 64-bit OpenBoot implementations.

TABLE 4-17 64-Bit Dictionary Compilation Commands

Command	Stack Diagram	Description
`x,`	(o --)	Compiles an octlet, `o`, into the dictionary (doublet-aligned).

Displaying Numbers

Basic commands to display stack values are shown in TABLE 4-18.

TABLE 4-18 Basic Number Display

Command	Stack Diagram	Description
`.`	(n --)	Displays a number in the current base.
`.r`	(n size --)	Displays a number in a fixed width field.
`.s`	(--)	Displays contents of data stack.
`showstack`	(??? -- ???)	Executes `.s` automatically before each `ok` prompt.
`noshowstack`	(??? -- ???)	Turns off automatic display of the stack before each `ok` prompt
`u.`	(u --)	Displays an unsigned number.
`u.r`	(u size --)	Displays an unsigned number in a fixed width field.

The `.s` command displays the entire stack contents without disturbing them. It can usually be used safely for debugging purposes. (This is the function that `showstack` performs automatically.)

Changing the Number Base

You can print numbers in a specific number base or change the operating number base using the commands in TABLE 4-19.

TABLE 4-19 Changing the Number Base

Command	Stack Diagram	Description
`.d`	(n --)	Displays *n* in decimal without changing base.
`.h`	(n --)	Displays *n* in hex without changing base.
`base`	(-- addr)	Variable containing number base.
`decimal`	(--)	Sets the number base to 10.

TABLE 4-19 Changing the Number Base *(Continued)*

Command	Stack Diagram	Description
d# *number*	(-- n)	Interprets *number* in decimal; base is unchanged.
hex	(--)	Set the number base to 16.
h# *number*	(-- n)	Interprets *number* in hex; base is unchanged.

The d# and h# commands are useful when you want to input a number in a specific base without explicitly changing the current base. For example:

```
ok decimal       (Changes base to decimal)
ok 4 h# ff 17 2
4 255 17 2 ok
```

The .d and .h commands act like "." but display the value in decimal or hexadecimal, respectively, regardless of the current base setting. For example:

```
ok hex
ok ff .  ff .d
ff 255
```

Controlling Text Input and Output

This section describes text and character input and output commands.

TABLE 4-20 lists commands to control text input.

TABLE 4-20 Controlling Text Input

Command	Stack Diagram	Description
(*ccc*)	(--)	Creates a comment. Conventionally used for stack diagrams.
\ *rest-of-line*	(--)	Treats the rest of the line as a comment.
ascii *ccc*	(-- char)	Gets numerical value of first ASCII character of next word.

TABLE 4-20 Controlling Text Input *(Continued)*

Command	Stack Diagram	Description
`accept`	(addr len1 -- len2)	Gets a line of edited input from the console input device; stores at *addr*. *len1* is the maximum allowed length. *len2* is the actual length received.
`expect`	(addr len --)	Gets and displays a line of input from the console; stores at *addr*. (Recommend using `accept` instead.)
`key`	(-- char)	Read a character from the console input device.
`key?`	(-- flag)	True if a key has been typed on the console input device.
`parse`	(char -- str len)	Parses text from the input buffer delimited by *char*.
`parse-word`	(-- str len)	Skips leading spaces and parses text from the input buffer delimited by white space.
`word`	(char -- pstr)	Collects a string delimited by *char* from the input buffer and places it as a packed string in memory at *pstr*. (Recommend using `parse` instead.)

Comments are used with Forth source code (generally in a text file) to describe the function of the code. The `(` (open parenthesis) is the Forth word that begins a comment. Any character up to the closing parenthesis `)` is ignored by the Forth interpreter. Stack diagrams are one example of comments using `(`.

Note – Remember to follow the (with a space, so that it is recognized as a Forth word.

`\` (backslash) indicates a comment terminated by the end of the line of text.

`key` waits for a key to be pressed, then returns the ASCII value of that key on the stack.

`ascii`, used in the form `ascii` *x*, returns on the stack the numerical code of the character *x*.

`key?` looks at the keyboard to see whether the user has recently typed any key. It returns a flag on the stack: `true` if a key has been pressed and `false` otherwise. See “Conditional Flags” on page 82 for a discussion on the use of flags.

TABLE 4-21 lists general-purpose text display commands.

TABLE 4-21 Displaying Text Output

Command	Stack Diagram	Description
`."` *ccc"*	(--)	Compiles a string for later display.
`(cr`	(--)	Moves the output cursor back to the beginning of the current line.
`cr`	(--)	Terminates a line on the display and goes to the next line.
`emit`	(char --)	Displays the character.
`exit?`	(-- flag)	Enables the scrolling control prompt: `More [<space>,<cr>,q] ?` The return flag is `true` if the user wants the output to be terminated.
`space`	(--)	Displays a `space` character.
`spaces`	(+n --)	Displays +*n* spaces.
`type`	(addr +n --)	Displays the +*n* characters beginning at *addr*.

`cr` sends a carriage-return or linefeed sequence to the console output device. For example:

```
ok 3 . 44 . cr 5 .
3 44
5
ok
```

`emit` displays the letter whose ASCII value is on the stack.

```
ok ascii a
61 ok 42
61 42 ok emit emit
Ba
ok
```

TABLE 4-22 shows commands used to manipulate text strings.

TABLE 4-22 Manipulating Text Strings

Command	Stack Diagram	Description
",	(addr len --)	Compiles an array of bytes from *addr* of length *len*, at the top of the dictionary as a packed string.
" ccc"	(-- addr len)	Collects an input stream string, either interpreted or compiled.
." *ccc*"		Displays the string *ccc*.
.(ccc)	(--)	Displays the string *ccc* immediately.
`-trailing`	(addr +n1 -- addr +n2)	Removes trailing spaces.
`bl`	(-- char)	ASCII code for the space character; decimal 32.
`count`	(pstr -- addr +n)	Unpacks a packed string.
`lcc`	(char -- lowercase-char)	Converts a character to lowercase.
`left-parse-string`	(addr len char -- addrR lenR addrL lenL)	Splits a string at *char* (which is discarded).
`pack`	(addr len pstr -- pstr)	Stores the string *addr*,*len* as a packed string at *pstr*.
`upc`	(char -- uppercase-char)	Converts a character to uppercase.

Some string commands specify an address (the location in memory where the characters reside) and a length (the number of characters in the string). Other commands use a packed string or `pstr`, which is a location in memory containing a byte for the length, immediately followed by the characters. The stack diagram for the command indicates which form is used. For example, `count` converts a packed string to an address-length string.

The command `."` is used in the form: `."` *string*". It outputs text immediately when it is encountered by the interpreter. A " (double quotation mark) marks the end of the text string. For example:

```
ok  : testing 34 .  ." This is a test"  55 . ;
ok
ok testing
34 This is a test55
ok
```

When " is used outside a colon definition, only two interpreted strings of up to 80 characters each can be assembled concurrently. This limitation does not apply in colon definitions.

Redirecting Input and Output

Normally, OpenBoot uses a keyboard for command input, and a frame buffer with a connected display screen for display output. (Server systems may use an ASCII terminal connected to a serial port. For more information on how to connect a terminal to your system, see your system's installation manual.) You can redirect the input, the output, or both, to a serial port. This may be useful, for example, when debugging a frame buffer.

TABLE 4-23 lists commands you can use to redirect input and output.

TABLE 4-23 I/O Redirection Commands

Command	Stack Diagram	Description
`input`	(device --)	Selects device, for example `ttya`, `keyboard`, or *device-specifier*, for subsequent input.
`io`	(device --)	Selects device for subsequent input and output.
`output`	(device --)	Selects device, for example `ttya`, `keyboard`, or *device-specifier*, for subsequent output.

The commands `input` and `output` temporarily change the current devices for input and output. The change takes place as soon as you enter a command; you do not have to reset your system. A system reset or power cycle causes the input and output devices to revert to the default settings specified in the NVRAM configuration variables `input-device` and `output-device`. These variables can be modified, if needed (see Chapter 3).

`input` must be preceded by one of the following: `keyboard`, `ttya`, `ttyb`, or *device-specifier* text string. For example, if input is currently accepted from the keyboard, and you want to make a change so that input is accepted from a terminal connected to the serial port `ttya`, type:

```
ok ttya input
ok
```

At this point, the keyboard becomes nonfunctional (except perhaps for Stop-A), but any text entered from the terminal connected to ttya is processed as input. All commands are executed as usual.

To resume using the keyboard as the input device, *use the terminal keyboard* to type:

```
ok keyboard input
ok
```

Similarly, output must be preceded by one of the following: screen, ttya, or ttyb or *device-specifier*. For example, if you want to send output to a serial port instead of the normal display screen, type:

```
ok ttya output
ok
```

The screen does *not* show the answering ok prompt, but the terminal connected to the serial port shows the ok prompt and all further output as well.

io is used in the same way, except that it changes both the input and output to the specified place. For example:

```
ok ttya io
ok
```

Generally, the argument to input, output, and io is a *device-specifier*, which can be either a device path name or a device alias. *The device must be specified as a Forth string, using double quotation marks* ("), as shown in the two examples below:

```
ok " /sbus/cgsix" output
```

or:

```
ok screen output
```

In the preceding examples, keyboard, screen, ttya, and ttyb are predefined Forth words that put their corresponding device alias string on the stack.

Command-Line Editor

OpenBoot implements a command line editor (similar to EMACS, a common text editor), some optional extensions and an optional history mechanism for the user interface. You use these tools to re-execute previous commands without retyping them, to edit the current command line to fix typing errors, or to recall and change previous commands.

TABLE 4-24 lists line-editing commands available at the ok prompt.

TABLE 4-24 Command-Line Editor Keystroke Commands

Keystroke	Description
Return (Enter)	Finishes editing of the line and submits the entire visible line to the interpreter regardless of the current cursor position.
Control-B	Moves backward one character.
Escape B	Moves backward one word.
Control-F	Moves forward one character.
Escape F	Moves forward one word.
Control-A	Moves backward to beginning of line.
Control-E	Moves forward to end of line.
Delete	Erases previous character.
Backspace	Erases previous character.
Control-H	Erases previous character.
Escape H	Erases from beginning of word to just before the cursor, storing erased characters in a save buffer.
Control-W	Erases from beginning of word to just before the cursor, storing erased characters in a save buffer.
Control-D	Erases next character.
Escape D	Erases from cursor to end of the word, storing erased characters in a save buffer.
Control-K	Erases from cursor to end of line, storing erased characters in a save buffer.
Control-U	Erases entire line, storing erased characters in a save buffer.

TABLE 4-24 Command-Line Editor Keystroke Commands *(Continued)*

Keystroke	Description
Control-R	Retypes the line.
Control-Q	Quotes next character (allows you to insert control characters).
Control-Y	Inserts the contents of the save buffer before the cursor.

The command-line history extension saves previously-typed commands in an EMACS-like command history ring that contains at least 8 entries. Commands may be recalled by moving either forward or backward around the ring. Once recalled, a command may be edited and/or resubmitted (by typing the Return key). The command-line history extension keys are:

TABLE 4-25 Command-Line History Keystroke Commands

Keystroke	Description
Control-P	Selects and displays the previous command in the command history ring.
Control-N	Selects and displays the next command in the command history ring.
Control-L	Displays the entire command history ring.

The command completion extension enables the system to complete long Forth word names by searching the dictionary for one or more matches based on the already-typed portion of a word. When you type a portion of a word followed by the command completion keystroke, Control-Space, the system behaves as follows:

- If the system finds exactly one matching word, the remainder of the word is automatically displayed.
- If the system finds several possible matches, the system displays all of the characters that are common to all of the possibilities.
- If the system cannot find a match for the already-typed characters, the system deletes characters from the right until there is at least one match for the remaining characters.
- The system beeps if it cannot determine an unambiguous match.

The command completion extension keys are:

TABLE 4-26 Command Completion Keystroke Commands

Keystroke	Description
Control-Space	Completes the name of the current word.
Control-?	Displays all possible matches for the current word.
Control-/	Displays all possible matches for the current word.

Conditional Flags

Forth conditionals use flags to indicate true/false values. A flag can be generated in several ways, based on testing criteria. The flag can then be displayed from the stack with the word ".", or it can be used as input to a conditional control command. Control commands can cause one behavior if a flag is true and another behavior if the flag is false. Thus, execution can be altered based on the result of a test.

A `0` value indicates that the flag value is `false`. A `-1` or any other nonzero number indicates that the flag value is `true`.

TABLE 4-27 lists commands that perform relational tests, and leave a `true` or `false` flag result on the stack.

TABLE 4-27 Comparison Commands

Command	Stack Diagram	Description
<	(n1 n2 -- flag)	True if *n1* < *n2*.
<=	(n1 n2 -- flag)	True if *n1* <= *n2*.
<>	(n1 n2 -- flag)	True if *n1* is not equal to *n2*.
=	(n1 n2 -- flag)	True if *n1* = *n2*.
>	(n1 n2 -- flag)	True if *n1* > *n2*.
>=	(n1 n2 -- flag)	True if *n1* >= *n2*.
0<	(n -- flag)	True if *n* < 0.
0<=	(n -- flag)	True if *n* <= 0.
0<>	(n -- flag)	True if *n* <> 0.
0=	(n -- flag)	True if *n* = 0 (also inverts any flag).

TABLE 4-27 Comparison Commands *(Continued)*

Command	Stack Diagram	Description
`0>`	(n -- flag)	True if *n* > 0.
`0>=`	(n -- flag)	True if *n* >= 0.
`between`	(n min max -- flag)	True if *min* <= *n* <= *max*.
`false`	(-- 0)	The value `FALSE`, which is 0.
`true`	(-- -1)	The value `TRUE`, which is -1.
`u<`	(u1 u2 -- flag)	True if *u1* < *u2*, unsigned.
`u<=`	(u1 u2 -- flag)	True if *u1* <= *u2*, unsigned.
`u>`	(u1 u2 -- flag)	True if *u1* > *u2*, unsigned.
`u>=`	(u1 u2 -- flag)	True if *u1* >= *u2*, unsigned.
`within`	(n min max -- flag)	True if *min* <= *n* < *max*.

`>` takes two numbers from the stack, and returns `true` (`-1`) on the stack if the first number was greater than the second number, or returns `false` (`0`) otherwise. An example follows:

```
ok 3 6 > .
0                (3 is not greater than 6)
ok
```

`0=` takes one item from the stack, and returns `true` if that item was 0 or returns `false` otherwise. This word inverts any flag to its opposite value.

Control Commands

The following sections describe words used in a Forth program to control the flow of execution.

The `if-else-then` Structure

The commands `if`, `else`, and `then` provide a simple control structure.

The commands listed in TABLE 4-28 control the flow of conditional execution.

TABLE 4-28 `if…else…then` Commands

Command	Stack Diagram	Description
`if`	(flag --)	Executes the following code when flag is `true`.
`else`	(--)	Executes the following code when flag is `false`.
`then`	(--)	Terminates `if…else…then`.

The format for using these commands is:

flag **`if`**
    (*do this if true*)
`then`
(*continue normally*)

or

flag **`if`**
    (*do this if true*)
`else`
    (*do this if false*)
`then`
(*continue normally*)

The `if` command consumes a flag from the stack. If the flag is `true` (nonzero), the commands following the `if` are performed. Otherwise, the commands (if any) following the `else` are performed.

```
ok : testit  ( n -- )
] 5 >  if  ." good enough "
] else  ." too small "
] then
] ." Done. "  ;
ok
ok 8 testit
good enough Done.
ok 2 testit
too small Done.
ok
```

Note – The] prompt reminds you that you are part way through creating a new colon definition. It reverts to ok after you finish the definition with a semicolon.

The case Statement

A high-level case command is provided for selecting alternatives with multiple possibilities. This command is easier to read than deeply-nested if…then commands.

TABLE 4-29 lists the conditional case commands.

TABLE 4-29 case Statement Commands

Command	Stack Diagram	Description
case	(selector -- selector)	Begins a case…endcase conditional.
endcase	(selector --)	Terminates a case…endcase conditional.
endof	(--)	Terminates an of…endof clause in a case…endcase
of	(selector test-value -- selector | {empty})	Begins an of…endof clause in a case conditional.

Here is a simple example of a case command:

```
ok : testit  ( testvalue -- )
]   case
]   0  of  ." It was zero " endof
]   1  of  ." It was one "  endof
]   ff of  ." Correct "     endof
]   -2 of  ." It was minus-two " endof
]   ( default )  ." It was this value: "  dup .
]   endcase   ." All done."  ;
ok
ok 1 testit
It was one All done.
ok ff testit
Correct All done.
ok 4 testit
It was this value: 4 All done.
ok
```

Note – The (optional) `default` clause can use the test value which is still on the stack, but should *not* remove it (use the phrase "`dup .`" instead of "`.`"). A successful `of` clause automatically removes the test value from the stack.

The `begin` Loop

A `begin` loop executes the same commands repeatedly until a certain condition is satisfied. Such a loop is also called a conditional loop.

TABLE 4-30 lists commands to control the execution of conditional loops.

TABLE 4-30 `begin` (Conditional) Loop Commands

Command	Stack Diagram	Description
`again`	(--)	Ends a `begin`…`again` infinite loop.
`begin`	(--)	Begins a `begin`…`while`…`repeat`, `begin`…`until`, or `begin`…`again` loop.
`repeat`	(--)	Ends a `begin`…`while`…`repeat` loop.
`until`	(flag --)	Continues executing a `begin`…`until` loop until *flag* is true.
`while`	(flag --)	Continues executing a `begin`…`while`…`repeat` loop while *flag* is true.

There are two general forms:

```
begin any commands…flag until
```

and

```
begin   any commands…flag while
        more commands      repeat
```

In both cases, the commands in the loop are executed repeatedly until the proper flag value causes the loop to be terminated. Then execution continues normally with the command following the closing command word (`until` or `repeat`).

In the begin…until case, until removes a flag from the top of the stack and inspects it. If the flag is false, execution continues just after the begin, and the loop repeats. If the flag is true, the loop is exited.

In the begin…while…repeat case, while removes a flag from the top of the stack and inspects it. If the flag is true, the loop continues by executing the commands just after the while. The repeat command automatically sends control back to begin to continue the loop. If the flag is false when while is encountered, the loop is exited immediately; control goes to the first command after the closing repeat.

An easy mnemonic for either of these loops is: If true, fall through.

A simple example follows.

```
ok begin 4000 c@ . key? until   (repeat until any key is pressed)
43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43
ok
```

The loop starts by fetching a byte from location 4000 and displaying the value. Then, the key? command is called, which leaves a true on the stack if the user has pressed any key, and false otherwise. This flag is consumed by until and, if the value is false, then the loop continues. Once a key is pressed, the next call to key? returns true, and the loop terminates.

Unlike many versions of Forth, the user interface allows the interactive use of loops and conditionals — that is, without first creating a definition.

The do Loop

A do loop (also called a counted loop) is used when the number of iterations of the loop can be calculated in advance. A do loop normally exits just *before* the specified ending value is reached.

TABLE 4-31 lists commands to control the execution of counted loops.

TABLE 4-31 do (Counted) Loop Commands

Command	Stack Diagram	Description
+loop	(n --)	Ends a do…+loop construct; adds n to loop index and returns to do (if *n* < 0, index goes from *start* to *end* inclusive).
?do	(end start --)	Begins ?do…loop to be executed 0 or more times. Index goes from *start* to *end*-1 inclusive. If *end* = *start*, loop is not executed.
?leave	(flag --)	Exits from a do…loop if *flag* is non-zero.
do	(end start --)	Begins a do…loop. Index goes from *start* to *end*-1 inclusive. Example: `10 0 do i . loop` (prints 0 1 2…d e f).
i	(-- n)	Leaves the loop index on the stack.
j	(-- n)	Leaves the loop index of the next outer enclosing loop on the stack.
leave	(--)	Exits from do…loop.
loop	(--)	End of do…loop.

The following screen shows several examples of how loops are used.

```
ok 10 5 do i .  loop
5 6 7 8 9 a b c d e f
ok
ok 2000 1000 do i .  i c@ . cr   i c@ ff = if leave then  4 +loop
1000 23
1004 0
1008 fe
100c 0
1010 78
1014 ff
ok : scan ( byte -- )
]    6000 5000      ( Scan memory 5000 - 6000 for bytes not equal to the specified byte )
]    do dup i c@ <> ( byte error? )
]      if i . then  ( byte )
]    loop
]    drop ( the original byte was still on the stack, discard it )
]  ;
ok 55 scan
5005 5224 5f99
ok 6000 5000 do i i c! loop      ( Fill a region of memory with a stepped pattern )
ok
ok 500 value testloc
ok : test16 ( -- ) 1.0000 0 ( do 0-ffff )      ( Write different 16-bit values to a location )
]     do i testloc w! testloc w@ i <> ( error? )      ( Also check the location )
]       if ." Error - wrote " i . ." read " testloc w@ . cr
]        leave ( exit after first error found )      ( This line is optional )
]       then
]     loop
]  ;
ok test16
ok 6000 to testloc
ok test16
Error - wrote 200 read 300
ok
```

Additional Control Commands

TABLE 4-32 contains descriptions of additional program execution control commands.

TABLE 4-32 Program Execution Control Commands

Command	Stack Diagram	Description
`abort`	(--)	Aborts current execution and interprets keyboard commands.
`abort"` *ccc*"	(abort? --)	If *abort?* is true, displays message and aborts.
`eval`	(addr len --)	Interprets Forth source from *addr len*.
`execute`	(xt --)	Executes the word whose execution token is on the stack.
`exit`	(--)	Returns from the current word. (Cannot be used in counted loops.)
`quit`	(--)	Same as `abort`, but leaves stack intact.

`abort` causes immediate termination and returns control to the keyboard. `abort"` is similar to `abort` but is different in two respects. `abort"` removes a flag from the stack and only aborts if the flag is `true`. Also, `abort"` prints any desired message when the abort takes place.

`eval` takes a string from the stack (specified as an address and a length). The characters in that string are then interpreted as if they were entered from the keyboard. If a Forth text file has been loaded into memory (see Chapter 5), then `eval` can be used to compile the definitions contained in the file.

CHAPTER 5

Loading and Executing Programs

The user interface provides several methods for loading and executing a program on the system. These methods load a file into memory from Ethernet, a hard disk, a floppy disk, or a serial port, and support the execution of Forth, FCode and binary executable programs.

Most of these methods require the file to have a Client program header; see IEEE 1275.1-1994 Standard for Boot (Initialization Configuration) Firmware for a description. This header is similar to the `a.out` header used by many UNIX systems. Sun's FCode tokenizer can generate files with the Client program header.

OpenBoot commands for loading files from various sources are listed in TABLE 5-1.

TABLE 5-1 File Loading Commands and Extensions

Command	Stack Diagram	Description
`boot` *[device-specifier] [arguments]*	(--)	Depending on the values of various configuration variables and the optional arguments, determines the file and device to be used. Resets the system, loads the identified program from the identified device, and executes the program.
`byte-load`	(addr xt --)	Interprets FCode beginning at *addr*. If *xt* is 1 (the usual case), uses `rb@` to read the FCode. Otherwise, uses the access routine whose execution token is *xt*.
`dl`	(--)	Loads a Forth source text file over a serial line until Control-D is detected and then interpret. Using `tip` as an example, type: **`~C cat`** filename **`Control-D`**

TABLE 5-1 File Loading Commands and Extensions *(Continued)*

Command	Stack Diagram	Description
`dlbin`	(--)	Loads a binary file over a serial line. Using `tip` as an example, type: **`~C cat`** filename
`dload` *filename*	(addr --)	Loads the specified file over the Ethernet at the given address.
`eval`	(… str len -- ???)	Synonym for `evaluate`.
`evaluate`	(… str len -- ???)	Interprets Forth source text from the specified string.
`go`	(--)	Begins executing a previously-loaded binary program, or resumes executing an interrupted program.
`init-program`	(--)	Prepares system to execute a binary file.
`load` [*device-specifier*] [*arguments*]	(--)	Depending on the values of various configuration variables and the optional arguments, determines the file and device to be used, and loads the identified program from the identified device.
`load-base`	(-- addr)	Address at which `load` places the data it reads from a device.
`?go`	(--)	Executes Forth, FCode or binary programs.

Using `boot`

Although `boot` is normally used to boot the operating system, it can be used to load and execute any client program. Although booting usually happens automatically, the user can also initiate booting from the user interface.

The `boot` process is as follows:

- The system may be reset if a client program has been executed since the last reset. (The execution of a reset is implementation dependent.)
- A device is selected by parsing the `boot` command line to determine the boot device and the boot arguments to use. Depending on the form of the `boot` command, the boot device and/or argument values may be obtained from configuration variables.

- The `bootpath` and `bootargs` properties in the `/chosen` node of the device tree are set with the selected values.
- The selected program is loaded into memory using a protocol that depends on the type of the selected device. For example, a disk boot might read a fixed number of blocks from the beginning of the disk, while a tape boot might read a particular tape file.
- The loaded program is executed. The behavior of the program may be further controlled by the argument string (if any) that was either contained within the selected configuration variable or was passed to the `boot` command on the command line.

`boot` has the following general format:

```
boot [device-specifier] [arguments]
```

where *device-specifier* and *arguments* are optional. For a complete discussion of the use of the `boot` command, see “Booting for the Expert User” on page 15.

Using `dl` to Load Forth Text Files Over Serial Port A

Forth programs loaded with `dl` must be ASCII files.

To load a file over the serial line, connect the test system's serial port A to a system that is able to transfer a file on request (in other words, a *server*). Start a terminal emulator on the server, and use that terminal emulator to download the file using `dl`.

The following example assumes the use of the Solaris terminal emulator `tip`. (See Appendix A, for information on this procedure.)

1. **At the `ok` prompt of the test system, type:**

```
ok dl
```

2. **In the `tip` window of the server, type:**

   `~C`

   to obtain a command line with which to issue a Solaris command on the server.

Note – The `C` is case-sensitive and must be capitalized.

Note – `tip` will only recognize the ~ as a `tip` command if it is the first character on the command line. If `tip` fails to recognize the `~C`, type Enter in the `tip` window and repeat `~C`.

3. **At the `local command` prompt, use `cat` to send the file.**

```
~C (local command) cat filename
(Away two seconds)
Control-D
```

4. **When `tip` displays a message of the form (`Away` *n* `seconds`), type:**

   `Control-D`

   in the `tip` window to signal `dl` that the end of the file has been reached.

   `dl` then automatically interprets the file, and the `ok` prompt reappears on the screen of the test system.

Using `load`

The syntax and behavior of `load` are similar to `boot`, except the program is only loaded and not executed. The `load` command also does not do a system reset prior to loading, as might `boot`.

The general form of the `load` command is:

```
load [device-specifier] [arguments]
```

The parsing of the `load` command's parameters is affected by the same configuration variables as `boot`, and `load`'s *device-specifier* and *arguments* are identified by the same process. (See "Booting for the Expert User" on page 15 for the details.)

Once the *device-specifier* and *arguments* are identified, loading proceeds as follows:

1. The *device-specifier* and *arguments* are saved in the `bootpath` and `bootargs` properties, respectively, of the `/chosen` node.

2. If the *device-specifier* was obtained from a configuration variable, its value may be a list of devices. If the list contains only a single entry, that entry is used by `load` as the *device-specifier*.

Note – If the list contains more than one entry, an attempt is made to open each listed device, beginning with the first entry, and continuing until the next to last entry. If the system successfully opens a device, that device is closed and is used by `load` as the *device-specifier*. If none of these devices can be opened, the last device in the list is used by `load` as the *device-specifier*.

---

3. `load` attempts to open the device specified by *device-specifier*. If the device cannot be opened, loading is terminated.
4. If the device is successfully opened, the device's `load` method is invoked to load the specified program from the specified device at the system's default load address.
5. If `load` is successful, and if the beginning of the loaded image is a valid client program header for the system:
   a. Memory is allocated at the address and of the size specified in that header.
   b. The loaded image is moved from the default load address to the newly allocated memory.
   c. The system is initialized such that a subsequent `go` command will begin the execution of the loaded program.

Using `dlbin` to Load FCode or Binary Executables Over Serial Port A

FCode or binary programs loaded with `dlbin` must be `Client program header` files. `dlbin` loads the files at the entry point indicated in the `Client program header`. Link binary files for 4000 (hex). Recent versions of the FCode Tokenizer create a `Client program header` file with entry point 4000.

To load a file over the serial line, connect the test system's serial port A to a system that is able to transfer a file on request (i.e. a *server*). Start a terminal emulator on the server, and use that terminal emulator to download the file using `dlbin`.

The following example assumes the use of the Solaris terminal emulator `tip`. (See Appendix A, for information on this procedure.)

1. At the test system's `ok` prompt, type:

```
ok dlbin
```

In the `tip` window of the server, type:

```
~C
```

to obtain a command line with which to issue a Solaris command on the server.

Note – The `C` is case-sensitive and must be capitalized.

Note – `tip` will only recognize the ~ as a `tip` command if it is the first character on the command line. If `tip` fails to recognize the `~C`, press Return in the `tip` window and repeat `~C again`.

---

1. At the "local command" prompt, use `cat` to send the file.

```
~C (local command) cat filename
(Away two seconds)
```

When `tip` completes the download, it displays a message of the form `(Away` *n* `seconds)`, and the `ok` prompt reappears on the screen of the test system.

To execute an FCode program, type:

```
ok 4000 1 byte-load
```

To execute the downloaded program, type:

```
ok go
```

Using dload to Load From Ethernet

dload loads files over Ethernet at a specified address, as shown below.

```
ok 4000 dload filename
```

In the above example, *filename* must be relative to the server's root. Use **4000** (hex) as the address for dload input. dload uses the trivial file transfer protocol (TFTP), so the server may need to have its permissions adjusted for this to work.

Forth Programs

Forth programs loaded with dload must be ASCII files beginning with the two characters "\ " (backslash and space). To execute the loaded Forth program, type:

```
ok 4000 file-size @ eval
```

In the above example, *file-size* contains the size of the loaded image.

FCode Programs

FCode programs loaded with dload must be Client program header files. To execute the loaded FCode program, type:

```
ok 4000 1 byte-load
```

byte-load is used by OpenBoot to interpret FCode programs on expansion boards such as SBus. The 1 in the example is a specific value of a parameter that specifies the separation between FCode bytes in the general case. Since dload loads into system memory, 1 is the correct spacing.

Binary Executables

`dload` requires binary programs to be in Client program header. Executable binary programs loaded must be either linked to `dload`'s input address (e.g., 4000) or be position independent. To execute the binary program, type:

```
ok go
```

To run the program again, type:

```
ok init-program go
```

`dload` does not use intermediate booters (unlike the boot command). Thus, any symbol information included in the `Client program header` file is available to the user interface's symbolic debugging capability. (See Chapter 6 for more information on symbolic debugging.)

Using ?go

Once a program has been loaded into the system, `?go` can be used to execute that program regardless of the type of the program.

`?go` examines the start of the loaded image. If the image begins with the string "\ " (backslash and space), the image is assumed to be Forth text. The Forth interpreter is invoked to interpret the image.

CHAPTER 6

Debugging

OpenBoot provides debugging tools that include a Forth language decompiler, a machine language disassembler, register display commands, a symbolic debugger, breakpoint commands, a Forth source-level debugger, a high-level language patching facility, and exception tracing. This chapter describes the capabilities specified by *IEEE Standard 1275-1994*.

Using the Forth Language Decompiler

The built-in Forth language decompiler can be used to recreate the source code for any previously-defined Forth word. The command:

```
ok see old-name
```

displays a listing of the source for `old-name` (without the source comments, of course).

A companion to `see` is `(see)` which is used to decompile the Forth word whose execution token is taken from the stack. For example:

```
ok ' old-name (see)
```

`(see)` produces a listing in a format identical to `see`.

```
ok see see
: see
   ' ['] (see) catch if
      drop
   then
;
ok see (see)
defer (see) is
: (f0018a44)
   40 rmargin ! dup dup (f00189c4) dup (f0018944) (f0018980) (f0018658)
   ??cr
;
ok f0018a44 (see)
: (f0018a44)
   40 rmargin ! dup dup (f00189c4) dup (f0018944) (f0018980) (f0018658)
   ??cr
;
```

The preceding listing shows that:

- `see` itself is composed only of Forth source words that were compiled as `external` or as `headers` with `fcode-debug?` set to true.
- `(see)` is a defer word. `(see)` also contains words that were compiled as `headerless` and are, consequently, displayed as hex addresses surrounded by parentheses.
- Decompiling a word with `(see)` produces a listing identical to that produced by `see`.

For words implemented in Forth assembler language, `see` displays a Forth assembler listing. For example, decompiling `dup` displays:

```
ok see dup
code dup
f0008c98     sub         %g7, 8, %g7
f0008c9c     stx         %g4, [%g0 + %g7]
f0008ca0     ld          [%g5], %l0
f0008ca4     jmp         %l0, %g2, %g0
f0008ca8     add         %g5, 4, %g5
```

Using the Disassembler

The built-in disassembler translates the contents of memory into equivalent assembly language.

TABLE 6-1 lists commands that disassemble memory into equivalent opcodes.

TABLE 6-1 Disassembler Commands

Command	Stack Diagram	Description
`+dis`	(--)	Continues disassembling where the last disassembly left off.
`dis`	(addr --)	Begins disassembling at the specified address.

`dis` begins to disassemble the data content of any desired location. The system pauses when:

- Any key is pressed while disassembly is taking place.
- The disassembler output fills the display screen.
- A `call` or `jump` opcode is encountered.

Disassembly can then be stopped or the `+dis` command can be used to continue disassembling at the location where the last disassembly stopped.

Memory addresses are normally shown in hexadecimal. However, if a symbol table is present, memory addresses are displayed symbolically whenever possible.

Displaying Registers

You can enter the user interface from the middle of an executing program as a result of a program crash, a user abort, or an encountered breakpoint. (Breakpoints are discussed in “Breakpoints” on page 103.) In all these cases, the user interface automatically saves all the CPU data register values in a buffer area. These values can then be inspected or altered for debugging purposes.

SPARC Registers

TABLE 6-2 lists the SPARC register commands.

TABLE 6-2 SPARC Register Commands

Command	Stack Diagram	Description
%g0 through %g7	(-- value)	Returns the value in the specified global register.
%i0 through %i7	(-- value)	Returns the value in the specified input register.
%l0 through %l7	(-- value)	Returns the value in the specified local register.
%o0 through %o7	(-- value)	Returns the value in the specified output register.
%pc %npc %y	(-- value)	Returns the value in the specified register.
%f0 through %f31	(-- value)	Returns the value in the specified floating point register.
.fregisters	(--)	Displays the values in %f0 through %f31.
.locals	(--)	Displays the values in the i, l and o registers.
.registers	(--)	Displays values in processor registers.
.window	(window# --)	Same as w .locals; displays the desired window.
ctrace	(--)	Displays the return stack showing C subroutines.
set-pc	(new-value --)	Sets %pc to *new-value*, and sets %npc to (*new-value*+4).
to regname	(new-value --)	Changes the value stored in any of the above registers. Use in the form: *new-value* to regname.
w	(window# --)	Sets the current window for displaying %ix, %lx, or %ox.

TABLE 6-3 SPARC V9 Register Commands

Command	Stack Diagram	Description
%fprs	(-- value)	Returns the value in the specified register.
%asi		
%pstate		
%tl-c		
%pil		
%tstate		
%tt		
%tba		
%cwp		
%cansave		
%canrestore		
%otherwin		
%wstate		
%cleanwin		
.pstate	(--)	Formatted display of the processor state register.
.ver	(--)	Formatted display of the version register.
.ccr	(--)	Formatted display of the %ccr register.
.trap-registers	(--)	Displays trap-related registers.

The values of all of these registers are saved and can be altered with to. After the values have been inspected and/or modified, program execution can be continued with the go command. The saved (and possibly modified) register values are copied back into the CPU, and execution resumes at the location specified by the saved program counter.

If you change %pc with to, you should also change %npc. (It is easier to use set-pc, which changes both registers automatically.)

On SPARC V9 systems, if N is the current window, N-1 specifies the window for the caller, N-2 specifies the callers's caller, etc.

Breakpoints

The user interface provides a breakpoint capability to assist in the development and debugging of stand-alone programs. (Programs that run over the operating system generally do not use this OpenBoot feature, but use other debuggers designed to run with the operating system.) The breakpoint feature lets you stop the program under

test at desired points. After program execution has stopped, registers or memory can be inspected or changed, and new breakpoints can be set or cleared. You can resume program execution with the `go` command.

TABLE 6-4 lists the breakpoint commands that control and monitor program execution.

TABLE 6-4 Breakpoint Commands

Command	Stack Diagram	Description
`+bp`	(addr --)	Adds a breakpoint at the specified address.
`-bp`	(addr --)	Removes the breakpoint at the specified address.
`--bp`	(--)	Removes the most-recently-set breakpoint.
`.bp`	(--)	Displays all currently set breakpoints.
`.breakpoint`	(--)	Performs a specified action when a breakpoint occurs. This word can be altered to perform any desired action. For example, to display registers at every breakpoint, type: `['] .registers to .breakpoint`. The default behavior is `.instruction`. To perform multiple behaviors, create a single definition which calls all desired behaviors, then load that word into `.breakpoint`.
`.instruction`	(--)	Displays the address, opcode for the last-encountered breakpoint.
`.step`	(--)	Performs a specified action when a single step occurs (see `.breakpoint`).
`bpoff`	(--)	Removes all breakpoints.
`finish-loop`	(--)	Executes until the end of this loop.
`go`	(--)	Continues from a breakpoint. This can be used to go to an arbitrary address by setting up the processor's program counter before issuing `go`.
`gos`	(n --)	Executes `go` n times.
`hop`	(--)	(Like the `step` command.) Treats a subroutine call as a single instruction.
`hops`	(n --)	Executes `hop` n times.
`return`	(--)	Executes until the end of this subroutine.
`returnl`	(--)	Executes until the end of this leaf subroutine.
`skip`	(--)	Skips (does not execute) the current instruction.

TABLE 6-4 Breakpoint Commands *(Continued)*

Command	Stack Diagram	Description
`step`	(--)	Single-steps one instruction.
`steps`	(n --)	Executes `step` *n* times.
`till`	(addr --)	Executes until the given address is encountered. Equivalent to `+bp go`.

To debug a program using breakpoints, use the following procedure.

1. **Load the test program into memory.**
2. **See Chapter 5 for more information. The register values are initialized automatically.**
3. **(Optional) Disassemble the downloaded program to verify a properly-loaded file.**
4. **Begin single-stepping the test program using the `step` command.**
5. **You can also set a breakpoint, then execute (for example, using the commands *addr* `+bp` and `go`) or perform other variations.**

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.

The debugger commands are shown in TABLE 6-5.

TABLE 6-5 Forth Source-level Debugger Commands

Command	Description
`c`	“Continue”. Switches from stepping to tracing, thus tracing the remainder of the execution of the word being debugged.
`d`	“Down a level”. Marks for debugging the word whose name was just displayed, then executes it.
`u`	“Up a level”. Un-marks the word being debugged, marks its caller for debugging, and finishes executing the word that was previously being debugged.

TABLE 6-5 Forth Source-level Debugger Commands *(Continued)*

Command	Description
`f`	Starts a subordinate Forth interpreter with which Forth commands can be executed normally. When that interpreter is terminated (with `resume`), control returns to the debugger at the place where the `f` command was executed.
g	"Go." Turns off the debugger and continues execution.
`q`	"Quit". Aborts the execution of the word being debugged and all its callers and returns to the command interpreter.
s	"see". Decompiles the word being debugged.
$	Displays the address,len on top of the stack as a text string.
h	"Help". Displays symbolic debugger documentation.
?	"Short Help". Displays brief symbolic debugger documentation.
`debug` *name*	Marks the specified Forth word for debugging. Enters the Forth Source-level Debugger on all subsequent attempts to execute *name*. After executing `debug`, the execution speed of the system might decrease until debugging is turned off with `debug-off`. (Do not debug basic Forth words such as ".".)
`(debug`	Like `debug` except that `(debug` takes an execution token from the stack instead of a name from the input stream.
`debug-off`	Turns off the Forth Source-level Debugger so that no word is being debugged.
`resume`	Exits from a subordinate interpreter, and goes back to the stepper (See the `f` command in this table).
`stepping`	Sets "step mode" for the Forth Source-level Debugger, allowing the interactive, step-by-step execution of the word being debugged. Step mode is the default.
`tracing`	Sets "trace mode" for the Forth Source-level Debugger. Tracing enables the execution of the word being debugged, while showing the name and stack contents for each word called by that word.
`<space-bar>`	Executes the word just displayed and proceeds to the next word.

Every Forth word is defined as a series of one or more words that could be called "component" words. While debugging a specified word, the debugger displays information about the contents of the stack while executing each of the word's "component" words. Immediately before executing each component word, the debugger displays the contents of the stack and the name of the component word that is about to be executed.

In trace mode, that component word is then executed, and the process continues with the next component word.

In step mode (the default), the user controls the debugger's execution behavior. Before the execution of each component word, the user is prompted for one of the keystrokes specified in TABLE 6-5.

Using `patch` and `(patch)`

OpenBoot provides the ability to change the definition of a previously compiled Forth word using high-level Forth language. While the changes will typically be made in the appropriate source code, the `patch` facility provides a means of quickly correcting errors uncovered during debugging.

`patch` reads the input stream for the following information:

- The name of the new code to be inserted.
- The name of the old code to be replaced.
- The name of the word containing the old code.

For example, consider the following example in which the word `test` is replaced with the number `555`:

```
ok : patch-me  test 0 do  i . cr  loop ;
ok patch 555 test patch-me
ok see patch-me
: patch-me
    h# 555 0 do
    i . cr
    loop
;
```

When using patch, some care must be taken to select the right word to replace. This is especially true if the word you are replacing is used several times within the target word and the occurrence of the word that you want to replace is not the first occurrence within the target word. In such a case, some subterfuge is required.

```
ok : patch-me2  dup dup dup ( This third dup should be drop) ;
ok : xx dup ;
ok patch xx dup patch-me2
ok patch xx dup patch-me2
ok patch drop dup patch-me2
ok see patch-me2
: patch-me2
    xx xx drop
;
```

Another use for patch is the case where the word to be patched contains some functionality that needs to be completely discarded. In this case, the word exit should be patched over the first word whose functionality is to be eliminated. For example, consider a word whose definition is:

```
ok : foo good bad unneeded ;
```

In this example, the functionality of bad is incorrect and the functionality of unneeded should be discarded. A first attempt to patch foo might be:

```
ok : right this that exit ;
ok patch right bad foo
```

on the expectation that the use of exit in the word right would prevent the execution of unneeded. Unfortunately, exit terminates the execution of the word which contains it, in this case right. The correct way to patch foo is:

```
ok : right this that ;
ok patch right bad foo
ok patch exit unneeded foo
```

(patch) is similar to patch except that (patch) obtains its arguments from the stack. The stack diagram for (patch) is:

```
( new-n1 num1? old-n2 num2? xt -- )
```

where:

- `new-n1` and `old-n2` can be either execution tokens or literal numbers.
- `num1?` and `num2?` are flags indicating whether `new-n1` or `old-n2`, respectively, are numbers.
- `xt` is the execution token of the word to be patched.

For example, consider the following example in which we reverse the affect of our first `patch` example by replacing the number 555 with `test`:

```
ok see patch-me
: patch-me
    h# 555 0 do
    i . cr
    loop
;
ok ['] test false 555 true ['] patch-me (patch)
ok see patch-me
: patch-me
    test 0 do
    i . cr
    loop
;
```

Using ftrace

The ftrace command shows the sequence of Forth words that were being executed at the time of the last exception. An example of ftrace follows.

```
ok : test1 1 ! ;
ok : test2 1 test1 ;
ok test2
Memory address not aligned
ok ftrace
!    Called from test1 at ffeacc5c
test1  Called from test2 at ffeacc6a
(ffe8b574) Called from (interpret at ffe8b6f8
execute Called from catch at ffe8a8ba
    ffefeff0
    0
    ffefebdc
catch    Called from (fload) at ffe8ced8
    0
(fload) Called from interact at ffe8cf74
execute Called from catch at ffe8a8ba
    ffefefd4
    0
    ffefebdc
catch Called from (quit at ffe8cf98
```

In this example, test2 calls test1, which tries to store a value to an unaligned address. This results in the exception: Memory address not aligned.

The first line of ftrace output shows the last command that caused the exception to occur. The next lines show locations from which the subsequent commands were being called.

The last few lines are usually the same in any ftrace output, because that is the calling sequence in effect when the Forth interpreter interprets a word from the input stream.

APPENDIX **A**

Setting Up a TIP Connection

You can use the TTYA or TTYB ports on your SPARC system to connect to a second Sun workstation. By connecting two systems in this way, you can use a shell window on the Sun workstation as a terminal to your SPARC system. (See the `tip` man page for detailed information about terminal connection to a remote host.)

The TIP method is preferable to simply connecting to a dumb terminal, since it lets you use windowing and operating system features when working with the boot PROM. A communications program or another non-Sun computer can be used in the same way, if the program can match the output baud rate used by the PROM TTY port.

Note – In the following pages, “SPARC system” refers to your system, and “Sun workstation” refers to the system you are connecting to your system.

Use the following procedure to set up the TIP connection.

1. **Connect the Sun workstation TTYB serial port to your SPARC system TTYA serial port using a serial connection cable. Use a 3-wire Null Modem Cable, and connect wires 3-2, 2-3, and 7-7. (Refer to your system installation manual for specifications on null modem cables.)**

2. **At the Sun workstation, add the following lines to the** `/etc/remote` **file.**

   If you are running a pre-Solaris 2.0 version of the operating environment, type:

```
hardwire:\
    :dv=/dev/ttyb:br#9600:el=^C^S^Q^U^D:ie=%$:oe=^D:
```

   If you are running version 2.x of the Solaris operating environment, type:

```
hardwire:\
    :dv=/dev/term/b:br#9600:el=^C^S^Q^U^D:ie=%$:oe=^D:
```

3. **In a Shell Tool window on the Sun workstation, type:**

```
hostname% tip hardwire
connected
```

   The Shell Tool window is now a TIP window directed to the Sun workstation TTYB.

Note – Use a Shell Tool, not a Command Tool; some TIP commands may not work properly in a Command Tool window.

---

4. **At your SPARC system, enter the Forth Monitor so that the** `ok` **prompt is displayed.**

Note – If you do not have a video monitor attached to your SPARC system, connect the SPARC system TTYA to the Sun workstation TTYB and turn on the power to your SPARC system. Wait for a few seconds, and press Stop-Auto interrupt the power-on sequence and start the Forth Monitor. Unless the system is completely inoperable, the Forth Monitor is enabled, and you can continue with the next step in this procedure.

---

5. **If you need to redirect the standard input and output to TTYA, type:**

```
ok ttya io
```

   There will be no echoed response.

6. **Press Return on the Sun workstation keyboard. The `ok` prompt shows in the TIP window.**

   Typing ~# in the TIP window is equivalent to pressing Stop-A at the SPARC system.

Note – *Do not* type `Stop-A` from a Sun workstation being used as a TIP window to your SPARC system. Doing so will abort the operating system on the workstation. (If you accidentally type `Stop-A`, you can recover by immediately typing `go` at the `ok` prompt.)

---

7. **Redirect the input and output to the screen and keyboard, if needed, by typing:**

```
ok screen output keyboard input
```

8. **When you are finished using the TIP window, end your TIP session and exit the window.**

Note – When entering ~ (tilde character) commands in the TIP window, ~ must be the first character entered on the line. To ensure that you are at the start of a new line, press Return first.

Common Problems With TIP

This section describes solutions for TIP problems occurring in pre-Solaris 2.0 operating environments.

Problems with TIP may occur if:

- The lock directory is missing or incorrect.

There should be a directory named `/usr/spool/uucp`. The owner should be `uucp` and the mode should be `drwxr-sr-x`.

- TTYB is enabled for logins.

The status field for TTYB (or the serial port you are using) must be set to `off` in `/etc/ttytab`. Be sure to execute `kill -HUP 1` (see `init(8)`) as root if you have to change this entry.

- `/dev/ttyb` is inaccessible.

Sometimes, a program will have changed the protection of `/dev/ttyb` (or the serial port you are using) so that it is no longer accessible. Make sure that `/dev/ttyb` has the mode set to crw-rw-rw-.

- The serial line is in tandem mode.

If the TIP connection is in tandem mode, the operating system sometimes sends XON (^S) characters (particularly when programs in other windows are generating lots of output). The XON characters are detected by the Forth word `key?`, and can cause confusion. The solution is to turn off tandem mode with the `~s !tandem` TIP command.

- The `.cshrc` file generates text.

TIP opens a sub-shell to run `cat`, thus causing text to be attached to the beginning of your loaded file. If you use `dl` and see any unexpected output, check your `.cshrc` file.

APPENDIX **B**

Building a Bootable Floppy Disk

This appendix outlines the steps necessary to create a bootable floppy disk. Information about the OS commands can be found in the `man` pages. Refer to the specific OS release for information about particular files and their locations within the file system.

1. **Format the diskette.**

   The `fdformat` command is an example of a utility for formatting floppy disks.

2. **Create the diskette's file systems.**

   If available, use the `newfs` command.

3. **Mount the diskette to a temporary partition.**

   If available, use the `mount` command.

4. **Copy the second-level disk booter to the diskette, using the `cp` command.**

   `boot` and `ufsboot` are examples of second-level booters.

5. **Install a boot block on the floppy.**

   If available, use the `installboot` command.

6. **Copy the file that you want to boot to the mounted diskette, using the `cp` command.**

7. **Unmount the diskette, using `umount`, if available.**

8. **You can now remove the diskette from the drive.**

   Use `eject floppy`, if available.

APPENDIX **C**

Troubleshooting Guide

What do you do if your system fails to boot properly? This appendix discusses some common failures and ways to alleviate them.

Power-On Initialization Sequence

Familiarize yourself with the system power-on initialization messages. You can then identify problems more accurately because these messages show you the types of functions the system performs at various stages of system startup. They also show the transfer of control from POST to OpenBoot to the Booter to the kernel.

The example that follows shows the OpenBoot initialization sequence in a Sun Ultra™ 1 system. The messages before the banner appear on TTYA only if the `diag-switch?` parameter is true.

Note – The actual OpenBoot initialization sequence is system dependent. The messages on your system may be different.

CODE EXAMPLE C-1 OpenBoot Initialization Sequence

```
...ttya initialized (At this point, POST has finished
execution and has transferred control to OpenBoot)
Probing Memory Bank #0 16 + 16 : 32 Megabytes (Probe memory)
Probing Memory Bank #1 0 + 0 : 0 Megabytes
Probing Memory Bank #2 0 + 0 : 0 Megabytes
Probing Memory Bank #3 0 + 0 : 0 Megabytes
   (If use-nvramrc? is true, the firmware
 executes NVRAMRC commands. The firmware then checks for Stop-x commands, and
 probes the devices. The Keyboard LEDs are then flashed.)
Probing UPA Slot at le,0 Nothing there (Probe devices)
Probing /sbus@lf,0 at 0,0 cgsix
Probing /sbus@lf,0 at 1,0 Nothing there
Probing /sbus@lf,0 at 2,0 Nothing there
Sun Ultra 1 UPA/SBus (UltraSPARC 167 MHz), Keyboard Present (Display the banner)
OpenBoot 3.0, 32 MB memory installed, Serial #7570016
Ethernet address 8:0:20:73:82:60, Host ID: 80738260.
ok boot disk3
Boot device: /sbus/espdma@e,8400000/esp@e,8800000/sd@3,0
(The firmware is TFTP-ing the boot program)
sd@3,0 File and args: (Control is transferred to the booter after this
message is displayed)
FCode UFS Reader 1.8 01 Feb 1995 17:07:00,IEEE 1275 Client Interface.(Booter
starts executing)
Loading: /platform/sun4u/ufsboot
cpu0: SUNW,UltraSPARC (upaid 0 impl 0x0 ver 0x0 clock 143 MHz)
SunOS Release 5.5 Version quick_gate_build:04/13/95 (UNIX(R) System V Release
 4.0)
   (Control is passed to the kernel after this message is displayed)
Copyright (c) 1983-1995, Sun Microsystems, Inc.
   (The kernel starts to execute)
DEBUG enabled (More kernel messages)
```

Emergency Procedures

Some OpenBoot systems provide the capability of commanding OpenBoot by means of depressing a combination of keys on the system's keyboard (i.e. a "keyboard chord").

Emergency Procedures for Systems with Standard (non-USB) Keyboards

When issuing any of these commands, press the keys immediately after turning on the power to your system, and hold the keys down for a few seconds until the keyboard LEDs flash.

TABLE C-1 SPARC-Compatible System Keyboard Chords

Command	Description
Stop	Bypasses POST. This command does not depend on security-mode. (Note: some systems bypass POST as a default; in such cases, use `Stop-D` to start POST.)
Stop-A	Aborts.
Stop-D	Enters diagnostic mode (set `diag-switch?` to `true`).
Stop-F	Enters Forth on TTYA instead of probing. Uses `fexit` to continue with the initialization sequence. Useful if hardware is broken.
Stop-N	Resets NVRAM contents to default values.

Note – These commands are disabled if the PROM security is on. Also, if your system has `full` security enabled, you can not apply any of the suggested commands unless you have the password to get to the `ok` prompt.

Emergency Procedures for Systems with USB Keyboards

The following paragraphs describe how to perform the functions of the Stop commands on systems that have USB keyboards.

Stop-A

Stop-A (Abort) works the same as it does on systems with standard keyboards, except that it does not work during the first few seconds after the system is reset.

Stop-N Equivalent

1. **After turning on the power to your system, wait until the front panel power button LED begins to blink.**
2. **Quickly press the front panel power button twice (similar to the way you would double-click a mouse).**

   A screen similar to the following is displayed to indicate that you have successfully reset the NVRAM contents to the default values:

```
Sun Blade 1000 (2 X UltraSPARC-III) , Keyboard Present
OpenBoot 4.0, 256 MB memory installed, Serial #12134241.
Ethernet address 8:0:20:b9:27:61, Host ID: 80b92761.

Safe NVRAM mode, the following nvram configuration variables have
been overridden:

'diag-switch?' is true
'use-nvramrc?' is false
'input-device', 'output-device' are defaulted
'ttya-mode', 'ttyb-mode' are defaulted

These changes are temporary and the original values will be
restored after the next hardware or software reset.
```

Note that some NVRAM configuration parameters are reset to their defaults. They include parameters that are more likely to cause problems, such as TTYA settings. These NVRAM settings are only reset to the defaults for this power cycle. If you do nothing other than reset the system at this point, the values are not permanently changed. Only settings that you change manually at this point become permanent. All other customized NVRAM settings are retained.

Typing **`set-defaults`** discards any customized NVRAM values and permanently restores the default settings for all NVRAM configuration parameters.

Note – Once the power button LED stops blinking and stays lit, pressing the power button again will power off the system.

Stop-F Functionality

The Stop-F functionality is not available in systems with USB keyboards.

Stop-D Functionality

The Stop-D (diags) key sequence is not supported on systems with USB keyboards, however, the Stop-D functionality can be closely emulated by using the power button double-tap (see Stop-N Functionality), since this temporarily sets `diag-switch?` to `true`. If you want the diagnostic mode turned on permanently, type:

```
ok setenv diag-switch? true
```

Preserving Data After a System Crash

The `sync` command forces any information on its way to the hard disk to be written out immediately. This is useful if the operating system has crashed, or has been interrupted without preserving all data first.

`sync` actually returns control to the operating system, which then performs the data saving operations. After the disk data has been synchronized, the operating system begins to save a core image of itself. If you do not need this core dump, you can interrupt the operation with the `Stop-A` key sequence.

Common Failures

This section describes some common failures and how you can fix them.

Blank Screen —No Output

Problem: Your system screen is blank and does not show any output.

Here are possible causes for this problem:

- Hardware has failed.

Refer to your system documentation.

- Keyboard is not attached.

If the keyboard is not plugged in, the output goes to TTYA instead. To fix this problem, power down the system, plug in the keyboard, and power on again.

- Monitor is not turned on or is not plugged in.

Check the power cable on the monitor. Make sure the monitor cable is plugged into the system frame buffer; then turn the monitor on.

- `output-device` is set to TTYA or TTYB.

This means the NVRAM parameter `output-device` is set to `ttya` or `ttyb` instead of being set to `screen`. Connect a terminal to TTYA and reset the system. After getting to the `ok` prompt on the terminal, type: `screen output` to send output to the frame buffer. Use `setenv` to change the default display device, if needed.

- System has multiple frame buffers.

If your system has several plugged-in frame buffers, or it has a built-in frame buffer and one or more plugged-in frame buffers, then it is possible that the wrong frame buffer is being used as the console device. See "Setting the Console to a Specific Monitor" on page 124.

System Boots From the Wrong Device

Problem: Your system is supposed to boot from the disk; instead, it boots from the net.

There are two possible causes for this:

- The `diag-switch?` NVRAM parameter is set to `true`.

Interrupt the booting process with `Stop-A`. Type the following commands at the `ok` prompt:

```
ok setenv diag-switch? false
ok boot
```

The system should now start booting from the disk.

- The `boot-device` NVRAM parameter is set to `net` instead of `disk`.

Interrupt the booting process with `Stop-A`. Type the following commands at the `ok` prompt:

```
ok setenv boot-device disk
ok boot
```

Note that the preceding commands cause the system to boot from the disk defined as `disk` in the device aliases list. If you want to boot from another service, set `boot-device` accordingly.

Problem: Your system is booting from a disk instead of from the net.

- `boot-device` is not set to `net`.

Interrupt the booting process with `Stop-A`. Type the following commands at the `ok` prompt:

```
ok setenv boot-device net
ok boot
```

Problem: Your system is booting from the wrong disk. (For example, you have more than one disk in your system. You want the system to boot from `disk2`, but the system is booting from `disk1` instead.)

- `boot-device` is not set to the correct disk.

Interrupt the booting process with `Stop-A`. Type the following commands at the `ok` prompt:

```
ok setenv boot-device disk2
ok boot
```

System Will Not Boot From Ethernet

Problem: Your system fails to boot from the net.

The problem could be one of the following:

- NIS maps are out-of-date.

Report the problem to your system administrator.

- Ethernet cable is not plugged in.

Plug in the ethernet cable. The system should continue with the booting process.

- Server is not responding: `no carrier` messages.

Report the problem to your system administrator.

- `tpe-link-test` is disabled.

Refer to the troubleshooting information in your system documentation. (Note: systems that do not have Twisted Pair Ethernet will not have the `tpe-link-test` parameter.)

System Will Not Boot From Disk

Problem: You are booting from a disk and the system fails with the message: `The file just loaded does not appear to be executable.`

- The boot block is missing or corrupted.

Install a new boot block.

Problem: You are booting from a disk and the system fails with the message: `Can't open boot device.`

- The disk may be powered down (especially if it is an external disk).

Turn on power to the disk, and make sure the SCSI cable is connected to the disk and the system.

SCSI Problems

Problem: Your system has more than one disk installed, and you get SCSI-related errors.

- Your system might have duplicate SCSI target number settings.

Try the following procedure:

1. Unplug all but one of the disks.
2. At the `ok` prompt, type:

```
ok probe-scsi
```

   Note the target number and its corresponding unit number.

3. Plug in another disk and perform Step 2 again.
4. If you get an error, change the target number of this disk to be one of the unused target numbers.
5. Repeat Steps 2, 3, and 4 until all the disks are plugged back in.

Setting the Console to a Specific Monitor

Problem: You have more than one monitor attached to the system, and the console is not set to the intended monitor.

- If you have more than one monitor attached to the system, OpenBoot always assigns the console to the frame buffer specified by the `output-device` NVRAM parameter. The default value of `output-device` is `screen`, which is an alias for one of the frame buffers found by the firmware.

A common way to change this default is to change `output-device` to the appropriate frame buffer:

```
ok nvalias myscreen /sbus/cgsix
ok setenv output-device myscreen
ok reset-all
```

Another way of setting the console to a specific monitor is to change the `sbus-probe-list` NVRAM parameter.

```
ok printenv sbus-probe-list (Display the current and default
 values)
```

If the frame buffer that you are choosing as the console is in slot 2, change `sbus-probe-list` to probe slot 2 first:

```
ok setenv sbus-probe-list 2013
ok reset-all
```

If a non-SBus frame buffer is installed, this second method may not work.

APPENDIX **D**

Forth Word Reference

This appendix contains the Forth commands supported by OpenBoot.

The commands are usually listed in the order in which they were introduced in the chapters. Some of the tables in this appendix show commands that are not listed elsewhere in this manual. These additional commands (such as memory mapping or output display primitives, or machine-specific register commands) are also part of the set of words in the OpenBoot implementation of Forth; they are included with relevant groups of commands.

The chapter topics include:

Stack Item Notation

TABLE D-1 Stack Item Notation

Notation	Description
|	Alternate stack results shown with space, e.g. (`input -- addr len false | result true`)
|	Alternate stack items shown without space, e.g. (`input -- addr len|0 result`).
`???`	Unknown stack item(s).
...	Unknown stack item(s). If used on both sides of a stack comment, means the same stack items are present on both sides.
`< > <space>`	Space delimiter. Leading spaces are ignored.
`a-addr`	Variable-aligned address.
`addr`	Memory address (generally a virtual address).
`addr len`	Address and length for memory region
`byte b`*xxx*	8-bit value (low order byte in a 32-bit word).
`char`	7-bit value (low order byte), high bit unspecified.
`cnt len size`	Count or length.
`dxxx`	Double (extended-precision) numbers. 2 stack items, most significant cell on top of stack.
`<eol>`	End-of-line delimiter.
`false`	0 (false flag).
`n n1 n2 n3`	Normal signed values (32-bit).
`nu nu1`	Signed or unsigned values (32-bit).
`<nothing>`	Zero stack items.
`phys`	Physical address (actual hardware address).
`phys.lo phys.hi`	Lower/upper cell of physical address
`pstr`	Packed string.
`quad qxxx`	Quadlet (32-bit value).
`qaddr`	Quadlet (32-bit) aligned address
`{text}`	Optional text. Causes default behavior if omitted.

TABLE D-1 Stack Item Notation *(Continued)*

Notation	Description
"text<delim>"	Input buffer text, parsed when command is executed. Text delimiter is enclosed in <>.
[text<delim>]	Text immediately following on the same line as the command, parsed immediately. Text delimiter is enclosed in <>.
true	-1 (true flag).
uxxx	Unsigned value, positive values (32-bit).
virt	Virtual address (address used by software).
waddr	Doublet (16-bit) aligned address
word wxxx	Doublet (16-bit value, low order two bytes in a 32-bit word).
x x1	Arbitrary stack item.
x.lo x.hi	Low / high significant bits of a data item
xt	Execution token.
xxx?	Flag. Name indicates usage (e.g. done? ok? error?).
xyz-str xyz-len	Address and length for unpacked string.
xyz-sys	Control-flow stack items, implementation-dependent.
(C: --)	Compilation stack diagram.
(--) (E: --)	Execution stack diagram.
(R: --)	Return stack diagram.

Commands for Browsing the Device Tree

TABLE D-2 Commands for Browsing the Device Tree

Command	Description
`.properties`	Displays the names and values of the current node's properties.
`dev` *device-path*	Chooses the specified device node, making it the current node.
`dev` *node-name*	Searches for a node with the specified name in the subtree below the current node, and chooses the first such node found.
`dev ..`	Choose the device node that is the parent of the current node.
`dev /`	Chooses the root machine node.
`device-end`	Leaves the device tree.
" *device-path*" `find-device`	Chooses the specified device node, similar to `dev`.
`ls`	Displays the names of the current node's children.
`pwd`	Displays the device path name that names the current node.
`see` *wordname*	Decompiles the specified word.
`show-devs` *[device-path]*	Displays all the devices known to the system directly beneath a given device in the device hierarchy. `show-devs` used by itself shows the entire device tree.
`words`	Displays the names of the current node's methods.
" *device-path*" `select-dev`	Selects the specified device and makes it the active node.

Common Options for the boot Command

TABLE D-3 Common Options for the boot Command

Parameter	Description
boot [*device-specifier*] [*filename*] [*options*]	
[*device-specifier*]	The name (full path name or alias) of the boot device. Typical values include: `cdrom` (CD-ROM drive) `disk` (hard disk) `floppy` (3-1/2" diskette drive) `net` (Ethernet) `tape` (SCSI tape)
[*filename*]	The name of the program to be booted (for example, `stand/diag`). *filename* is relative to the root of the selected device and partition (if specified). If *filename* is not specified, the boot program uses the value of the `boot-file` NVRAM parameter (see Chapter 3).
[*options*]	(These options are specific to the operating system, and may differ from system to system.)

System Information Display Commands

TABLE D-4 System Information Display Commands

Command	Description
`banner`	Displays power-on banner.
`.enet-addr`	Displays current Ethernet address.
`.idprom`	Displays ID PROM contents, formatted.
`.traps`	Displays a list of SPARC trap types.
`.version`	Displays version and date of the boot PROM.
`show-devs`	Displays all installed and probed devices.

Configuration Variables

TABLE D-5 Configuration Variables

Command	Description
`printenv`	Displays all current parameters and current default values. (Numbers are usually shown as decimal values.) `printenv` *parameter* shows the current value of the named parameter.
`setenv` *parameter value*	Sets parameter to the specified decimal or text value. (Changes are permanent, but usually only take effect after a reset.)
`set-default` *parameter*	Resets the value of the named parameter to the factory default.
`set-defaults`	Resets parameter values to the factory defaults.
`password`	Sets security-password.

`nvramrc` Editor Commands

TABLE D-6 `nvramrc` Editor Commands

Command	Description
`nvalias` *alias device-path*	Stores the command "`devalias` *alias device-path*" in `nvramrc`. The alias persists until the `nvunalias` or `set-defaults` commands are executed.
`nvedit`	Enters the `nvramrc` editor. If data remains in the temporary buffer from a previous `nvedit` session, resumes editing those previous contents. If not, reads the contents of `nvramrc` into the temporary buffer and begins editing it.
`nvquit`	Discards the contents of the temporary buffer, without writing it to `nvramrc`. Prompts for confirmation.

TABLE D-6 `nvramrc` Editor Commands *(Continued)*

Command	Description
`nvrecover`	Recovers the contents of `nvramrc` if they have been lost as a result of the execution of `set-defaults`; then enters the editor as with `nvedit`. `nvrecover` fails if `nvedit` is executed between the time that the `nvramrc` contents were lost and the time that `nvrecover` is executed.
`nvrun`	Executes the contents of the temporary buffer.
`nvstore`	Copies the contents of the temporary buffer to `nvramrc`; discards the contents of the temporary buffer.
`nvunalias` *alias*	Deletes the corresponding alias from `nvramrc`.

NVRAM Script Editor Keystroke Commands

TABLE D-7 NVRAM Script Editor Keystroke Commands

Keystroke	Description
Control-B	Moves backward one character.
Escape B	Moves backward one word.
Control-F	Moves forward one character.
Escape F	Moves forward one word.
Control-A	Moves backward to beginning of the line.
Control-E	Moves forward to end of the line.
Control-N	Moves to the next line of the script editing buffer.
Control-P	Moves to the previous line of the script editing buffer.
Return (Enter)	Inserts a new line at the cursor position and advances to the next line.
Control-O	Inserts a new line at the cursor position and stays on the current line.
Control-K	Erases from the cursor position to the end of the line, storing the erased characters in a save buffer. If at the end of a line, joins the next line to the current line (i.e. deletes the new line).
Delete	Erases the previous character.
Backspace	Erases the previous character.
Control-H	Erases the previous character.
Escape H	Erases from beginning of word to just before the cursor, storing erased characters in a save buffer.
Control-W	Erases from beginning of word to just before the cursor, storing erased characters in a save buffer.
Control-D	Erases the next character.
Escape D	Erases from the cursor to the end of the word, storing the erased characters in a save buffer.
Control-U	Erases the entire line, storing the erased characters in a save buffer.
Control-Y	Inserts the contents of the save buffer before the cursor.
Control-Q	Quotes the next character (i.e. allows you to insert control characters).

TABLE D-7 NVRAM Script Editor Keystroke Commands *(Continued)*

Keystroke	Description
Control-R	Retypes the line.
Control-L	Displays the entire contents of the editing buffer.
Control-C	Exits the script editor, returning to the OpenBoot command interpreter. The temporary buffer is preserved, but is not written back to the script. (Use `nvstore` afterwards to write it back.)

Stack Manipulation Commands

TABLE D-8 Stack Manipulation Commands

Command	Stack Diagram	Description
`clear`	(??? --)	Empties the stack.
`depth`	(-- u)	Returns the number of items on the stack.
`drop`	(x --)	Removes top item from the stack.
`2drop`	(x1 x2 --)	Removes 2 items from the stack.
`3drop`	(x1 x2 x3 --)	Removes 3 items from the stack.
`dup`	(x -- x x)	Duplicates the top stack item.
`2dup`	(x1 x2 -- x1 x2 x1 x2)	Duplicates 2 stack items.
`3dup`	(x1 x2 x3 -- x1 x2 x3 x1 x2 x3)	Duplicates 3 stack items.
`?dup`	(x -- x x | 0)	Duplicates the top stack item if it is non-zero.
`nip`	(x1 x2 -- x2)	Discards the second stack item.
`over`	(x1 x2 -- x1 x2 x1)	Copies second stack item to top of stack.
`2over`	(x1 x2 x3 x4 -- x1 x2 x3 x4 x1 x2)	Copies second 2 stack items.
`pick`	(xu … x1 x0 u -- xu … x1 x0 xu)	Copies *u*-th stack item (`1 pick` = `over`).
`>r`	(x --) (R: -- x)	Moves a stack item to the return stack.
`r>`	(-- x) (R: x --)	Moves a return stack item to the stack.
`r@`	(-- x) (R: x -- x)	Copies the top of the return stack to the stack.
`roll`	(xu … x1 x0 u -- xu-1 … x1 x0 xu)	Rotates *u* stack items (`2 roll` = `rot`).
`rot`	(x1 x2 x3 -- x2 x3 x1)	Rotates 3 stack items.
`-rot`	(x1 x2 x3 -- x3 x1 x2)	Inversely rotates 3 stack items.
`2rot`	(x1 x2 x3 x4 x5 x6 -- x3 x4 x5 x6 x1 x2)	Rotates 3 pairs of stack items.

TABLE D-8 Stack Manipulation Commands *(Continued)*

Command	Stack Diagram	Description
`swap`	(x1 x2 -- x2 x1)	Exchanges the top 2 stack items.
`2swap`	(x1 x2 x3 x4 -- x3 x4 x1 x2)	Exchanges 2 pairs of stack items.
`tuck`	(x1 x2 -- x2 x1 x2)	Copies top stack item below second item.

Single-Precision Arithmetic Functions

TABLE D-9 Single-Precision Arithmetic Functions

Command	Stack Diagram	Description
+	(nu1 nu2 -- sum)	Adds *nu1* + *nu2*.
-	(nu1 nu2 -- diff)	Subtracts *nu1* - *nu2*.
*	(nu1 nu2 -- prod)	Multiplies *nu1* * *nu2*.
*/	(nu1 nu2 nu3 -- quot)	Calculates *nu1* * *nu2* / *n3*.
/	(n1 n2 -- quot)	Divides *n1* by *n2*; remainder is discarded.
1+	(nu1 -- nu2)	Add 1.
1-	(nu1 -- nu2)	Subtracts 1.
2+	(nu1 -- nu2)	Adds 2.
2-	(nu1 -- nu2)	Subtracts 2.
abs	(n -- u)	Absolute value.
bounds	(n count -- n+count n)	Prepares arguments for do or ?do loop.
even	(n -- n | n+1)	Rounds to nearest even integer >= *n*.
max	(n1 n2 -- n1|n2)	Returns the maximum of *n1* and *n2*.
min	(n1 n2 -- n1|n2)	Returns the minimum of *n1* and *n2*.
mod	(n1 n2 -- rem)	Remainder of *n1* / *n2*.
*/mod	(n1 n2 n3 -- rem quot)	Remainder, quotient of *n1* * *n2* / *n3*.
/mod	(n1 n2 -- rem quot)	Remainder, quotient of *n1* / *n2*.
negate	(n1 -- n2)	Changes the sign of *n1*.
u*	(u1 u2 -- uprod)	Multiplies 2 unsigned numbers, yielding an unsigned product.
u/mod	(u1 u2 -- urem uquot)	Divides unsigned number by an unsigned number, yielding remainder and quotient.

Bit-wise Logical Operators

TABLE D-10 Bit-wise Logical Operators

Command	Stack Diagram	Description
`2*`	(x1 -- x2)	Multiplies by 2.
`2/`	(x1 -- x2)	Divides by 2.
`>>a`	(x1 u -- x2)	Arithmetic right-shift *x1* by *u* bits.
`and`	(x1 x2 -- x3)	Bitwise logical AND.
`invert`	(x1 -- x2)	Inverts all bits of *x1*.
`lshift`	(x1 u -- x2)	Left-shifts *x1* by *u* bits. Zero-fill low bits.
`or`	(x1 x2 -- x3)	Bitwise logical OR.
`rshift`	(x1 u -- x2)	Right-shifts *x1* by *u* bits. Zero-fill high bits.
`u2/`	(x1 -- x2)	Logical right shift 1 bit; zero shifted into high bit.
`xor`	(x1 x2 -- x3)	Bitwise exclusive OR.

Double Number Arithmetic Functions

TABLE D-11 Double Number Arithmetic Functions

Command	Stack Diagram	Description
`d+`	(d1 d2 -- d.sum)	Adds *d1* to *d2*, yielding double number *d.sum*.
`d-`	(d1 d2 -- d.diff)	Subtracts *d2* from *d1*, yielding double number *d.diff*.
`fm/mod`	(d n -- rem quot)	Divides *d* by *n*.
`m*`	(n1 n2 -- d)	Signed multiply with double-number product.
`s>d`	(n1 -- d1)	Converts a number to a double number.

TABLE D-11 Double Number Arithmetic Functions *(Continued)*

Command	Stack Diagram	Description
`sm/rem`	(d n -- rem quot)	Divides *d* by *n*, symmetric division.
`um*`	(u1 u2 -- ud)	Unsigned multiply yielding unsigned double number product.
`um/mod`	(ud u -- urem uprod)	Divides *ud* by *u*.

32-Bit Data Type Conversion Functions

TABLE D-12 32-Bit Data Type Conversion Functions

Command	Stack Diagram	Description
`bljoin`	(b.low b2 b3 b.hi -- quad)	Joins four bytes to form a quadlet
`bwjoin`	(b.low b.hi -- word)	Joins two bytes to form a doublet.
`lbflip`	(quad1 -- quad2)	Reverses the bytes within a quadlet
`lbsplit`	(quad -- b.low b2 b3 b.hi)	Splits a quadlet into four bytes.
`lwflip`	(quad1 -- quad2)	Swaps the doublets within a quadlet.
`lwsplit`	(quad -- w.low w.hi)	Splits a quadlet into two doublets.
`wbflip`	(word1 -- word2)	Swaps the bytes within a doublet.
`wbsplit`	(word -- b.low b.hi)	Splits a doublet into two bytes.
`wljoin`	(w.low w.hi -- quad)	Joins two doublets to form a quadlet.

64-Bit Data Type Conversion Functions

TABLE D-13 64-Bit Data Type Conversion Functions

Command	Stack Diagram	Description
`bxjoin`	(b.lo b.2 b.3 b.4 b.5 b.6 b.7 b.hi -- o)	Joins 8 bytes to form an octlet.
`lxjoin`	(quad.lo quad.hi -- o)	Joins 2 quadlets to form an octlet.
`wxjoin`	(w.lo w.2 w.3 w.hi -- o)	Joins four doublets to form an octlet.
`xbflip`	(oct1 -- oct2)	Reverses the bytes within an octlet.
`xbflips`	(oaddr len --)	Reverses the bytes within each octlet in the given region.The behavior is undefined if *len* is not a multiple of `/x`.
`xbsplit`	(o -- b.lo b.2 b.3 b.4 b.5 b.6 b.7 b.hi)	Splits an octlet into 8 bytes.
`xlflip`	(oct1 -- oct2)	Reverses the quadlets within an octlet. The bytes within each quadlet are not reversed.
`xlflips`	(oaddr len --)	Reverses the quadlets within each octlet in the given region. The bytes within each quadlet are not reversed. The behavior is undefined if *len* is not a multiple of `/x`.
`xlsplit`	(o -- quad.lo quad.hi)	Splits one octlet into 2 quadlets.
`xwflip`	(oct1 -- oct2)	Reverses the doublets within an octlet. The bytes within each doublet are not reversed.
`xwflips`	(oaddr len --)	Reverses the doublets within each octlet in the given region. The bytes within each doublet are not reversed. The behavior is undefined if *len* is not a multiple of `/x`.
`xwsplit`	(o -- w.lo w.2 w.3 w.hi)	Splits an octlet into 4 doublets.

Address Arithmetic Functions

TABLE D-14 Address Arithmetic Functions

Command	Stack Diagram	Description
`aligned`	(n1 -- n1 | a-addr)	Increases *n1* if necessary to yield a variable aligned address.
`/c`	(-- n)	The number of address units to a byte: 1.
`/c*`	(nu1 -- nu2)	Synonym for `chars`.
`ca+`	(addr1 index -- addr2)	Increments *addr1* by *index* times the value of `/c`.
`ca1+`	(addr1 -- addr2)	Synonym for `char+`.
`char+`	(addr1 -- addr2)	Increments *addr1* by the value of `/c`.
`cell+`	(addr1 -- addr2)	Increments *addr1* by the value of `/n`.
`chars`	(nu1 -- nu2)	Multiplies *nu1* by the value of `/c`.
`cells`	(nu1 -- nu2)	Multiplies *nu1* by the value of `/n`.
`/l`	(-- n)	Number of address units to a quadlet; typically 4.
`/l*`	(nu1 -- nu2)	Multiplies *nu1* by the value of `/l`.
`la+`	(addr1 index -- addr2)	Increments *addr1* by *index* times the value of `/l`.
`la1+`	(addr1 -- addr2)	Increments *addr1* by the value of `/l`.
`/n`	(-- n)	Number of address units in a cell.
`/n*`	(nu1 -- nu2)	Synonym for `cells`.
`na+`	(addr1 index -- addr2)	Increments *addr1* by *index* times the value of `/n`.
`na1+`	(addr1 -- addr2)	Synonym for `cell+`.
`/w`	(-- n)	Number of address units to a doublet; typically 2.
`/w*`	(nu1 -- nu2)	Multiplies *nu1* by the value of `/w`.
`wa+`	(addr1 index -- addr2)	Increments *addr1* by *index* times the value of `/w`.
`wa1+`	(addr1 -- addr2)	Increments *addr1* by the value of `/w`.

64-Bit Address Arithmetic Functions

TABLE D-15 64-Bit Address Arithmetic Functions

Command	Stack Diagram	Description
`/x`	(-- n)	Number of address units in an octlet, typically 8.
`/x*`	(nu1 -- nu2)	Multiplies *nu1* by the value of `/x`.
`xa+`	(addr1 index -- addr2)	Increments *addr1* by *index* times the value of `/x`.
`xa1+`	(addr1 -- addr2)	Increments *addr1* by the value of `/x`.

Memory Access Commands

TABLE D-16 Memory Access Commands

Command	Stack Diagram	Description
`!`	(x a-addr --)	Stores a number at *a-addr*.
`+!`	(nu a-addr --)	Adds *nu* to the number stored at *a-addr*.
`@`	(a-addr -- x)	Fetches a number from *a-addr*.
`2!`	(x1 x2 a-addr --)	Stores 2 numbers at *a-addr*, *x2* at lower address.
`2@`	(a-addr -- x1 x2)	Fetches 2 numbers from *a-addr*, *x2* from lower address.
`blank`	(addr len --)	Sets *len* bytes of memory beginning at *addr* to the space character (decimal 32).
`c!`	(byte addr --)	Stores *byte* at *addr*.
`c@`	(addr -- byte)	Fetches a *byte* from *addr*.
`cpeek`	(addr -- false | byte true)	Attempts to fetch the byte at *addr*. Returns the data and `true` if the access was successful. Returns `false` if a read access error occurred.
`cpoke`	(byte addr -- okay?)	Attempts to store the *byte* to *addr*. Returns `true` if the access was successful. Returns `false` if a write access error occurred.

TABLE D-16 Memory Access Commands *(Continued)*

Command	Stack Diagram	Description
`comp`	(addr1 addr2 len -- diff?)	Compares two byte arrays. *diff?* is 0 if the arrays are identical, *diff?* is -1 if the first byte that is different is lesser in the string at *addr1*, *diff?* is 1 otherwise.
`dump`	(addr len --)	Displays *len* bytes of memory starting at *addr*.
`erase`	(addr len --)	Sets *len* bytes of memory beginning at *addr* to `0`.
`fill`	(addr len byte --)	Sets *len* bytes of memory beginning at *addr* to the value *byte*.
`l!`	(q qaddr --)	Stores a quadlet *q* at *qaddr*.
`l@`	(qaddr -- q)	Fetches a quadlet *q* from *qaddr*.
`lbflips`	(qaddr len --)	Reverses the bytes within each quadlet in the specified region.
`lwflips`	(qaddr len --)	Swaps the doublets within each quadlet in specified region.
`lpeek`	(qaddr -- false | quad true)	Attempts to fetch the 32-bit quantity at *qaddr*. Returns the data and `true` if the access was successful. Returns `false` if a read access error occurred.
`lpoke`	(quad qaddr -- okay?)	Attempts to store the 32-bit quantity at *qaddr*. Returns `true` if the access was successful. Returns `false` if a a write access error occurred.
`move`	(src-addr dest-addr len --)	Copies *len* bytes from *src-addr* to *dest-addr*.
`off`	(a-addr --)	Stores `false` at *a-addr*.
`on`	(a-addr --)	Stores `true` at *a-addr*.
`unaligned-l!`	(q addr --)	Stores a quadlet *q*, any alignment
`unaligned-l@`	(addr -- q)	Fetches a quadlet *q*, any alignment.
`unaligned-w!`	(w addr --)	Stores a doublet *w*, any alignment.
`unaligned-w@`	(addr -- w)	Fetches a doublet *w*, any alignment.
`w!`	(w waddr --)	Stores a doublet *w* at *waddr*.
`w@`	(waddr -- w)	Fetches a doublet *w* from *waddr*.
`<w@`	(waddr -- n)	Fetches a doublet *w* from *waddr*, sign-extended.

TABLE D-16 Memory Access Commands *(Continued)*

Command	Stack Diagram	Description
`wbflips`	(waddr len --)	Swaps the bytes within each doublet in the specified region.
`wpeek`	(waddr -- false | w true)	Attempts to fetch the 16-bit quantity at *waddr*. Returns the data and `true` if the access was successful. Returns `false` if a read access error occurred.
`wpoke`	(w waddr -- okay?)	Attempts to store the 16-bit quantity to *waddr*. Returns `true` if the access was successful. Returns `false` if a write access error occurred.

64-Bit Memory Access Functions

TABLE D-17 64-Bit Memory Access Functions

Command	Stack Diagram	Description
`<l@`	(qaddr -- n)	Fetches a quadlet from qaddr, sign-extended.
`x,`	(o --)	Compiles an octlet into the dictionary (doublet-aligned).
`x@`	(oaddr -- o)	Fetches an octlet from an octlet aligned address.
`x!`	(o oaddr --)	Stores an octlet to an octlet aligned address.
`xbflips`	(oaddr len --)	Reverses the bytes within each octlet in the given region.The behavior is undefined if *len* is not a multiple of `/x`.
`xlflips`	(oaddr len --)	Reverses the quadlets within each octlet in the given region. The bytes within each quadlet are not reversed. The behavior is undefined if *len* is not a multiple of `/x`.
`xwflips`	(oaddr len --)	Reverses the doublets within each octlet in the given region. The bytes within each doublet are not reversed. The behavior is undefined if *len* is not a multiple of `/x`.

Memory Mapping Commands

TABLE D-18 Memory Mapping Commands

Command	Stack Diagram	Description
`alloc-mem`	(size -- virt)	Allocates and maps *size* bytes of available memory; returns the virtual address. Unmap with `free-mem`.
`free-mem`	(virt size --)	Frees memory allocated by `alloc-mem`.
`map?`	(virt --)	Displays memory map information for the virtual address.

Defining Words

TABLE D-19 Defining Words

Command	Stack Diagram	Description
`:` *new-name*	(--) (E: … -- ???)	Starts a new colon definition of the word *new-name*.
`;`	(--)	Ends a colon definition.
`alias` *new-name old-name*	(--) (E: … -- ???)	Creates *new-name* with the same behavior as *old-name*.
`buffer:` *name*	(size --) (E: -- a-addr)	Creates a named array in temporary storage.
`constant` *name*	(n --) (E: -- n)	Defines a constant (for example, `3 constant bar`).
`2constant` *name*	(n1 n2 --) (E: -- n1 n2)	Defines a 2-number constant.
`create` *name*	(--) (E: -- a-addr)	Generic defining word.
`defer` *name*	(--) (E: … -- ???)	Defines a word for forward references or execution vectors using execution token.

TABLE D-19 Defining Words *(Continued)*

Command	Stack Diagram	Description
`does>`	(… -- … a-addr) (E: … -- ???)	Starts the run-time clause for defining words.
`field` *name*	(offset size -- offset+size) (E: addr -- addr+offset)	Creates a named offset pointer.
`struct`	(-- 0)	Initializes for `field` creation.
`value` *name*	(n --) (E: -- n)	Creates a changeable, named quantity.
`variable` *name*	(--) (E: -- a-addr)	Defines a variable.

Dictionary Searching Commands

TABLE D-20 Dictionary Searching Commands

Command	Stack Diagram	Description
' *name*	(-- xt)	Finds the named word in the dictionary. Returns the execution token. Uses outside definitions.
['] *name*	(-- xt)	Similar to ' but is used either inside or outside definitions.
.calls	(xt --)	Displays a list of all words that call the word whose execution token is *xt*.
$find	(str len -- str len false | xt true)	Searches for word named by *str,len*. If found, leaves *xt* and true on stack. If not found, leaves name string and false on stack.
find	(pstr -- pstr false | xt n)	Searches for word named by *pstr*. If found, leaves *xt* and true on stack. If not found, leaves name string and false on stack. (Recommend using $find to avoid use of packed string.)
see *thisword*	(--)	Decompiles the named command.
(see)	(xt --)	Decompiles the word indicated by the execution token.
sift	(pstr --)	Displays names of all dictionary entries containing the string pointed to by *pstr*.
sifting *ccc*	(--)	Displays names of all dictionary entries containing the sequence of characters. *ccc* contains no spaces.
words	(--)	Displays all visible words in the dictionary.

Dictionary Compilation Commands

TABLE D-21 Dictionary Compilation Commands

Command	Stack Diagram	Description
,	(n --)	Places a number in the dictionary.
c,	(byte --)	Places a byte in the dictionary.
w,	(word --)	Places a 16-bit number in the dictionary.
l,	(quad --)	Places a 32-bit number in the dictionary.
[	(--)	Enters interpretation state.
]	(--)	Ends interpreting, enters compilation state.
allot	(n --)	Allocates *n* bytes in the dictionary.
>body	(xt -- a-addr)	Finds the data field address from the execution token.
body>	(a-addr -- xt)	Finds the execution token from the data field address.
compile	(--)	Compiles the next word at run time. (Recommend using postpone instead.)
[compile] *name*	(--)	Compiles the next (immediate) word. (Recommend using postpone instead.)
forget *name*	(--)	Removes word from dictionary and all subsequent words.
here	(-- addr)	Address of top of dictionary.
immediate	(--)	Marks the last definition as immediate.
to *name*	(n --)	Installs a new action in a defer word or value.
literal	(n --)	Compiles a number.
origin	(-- addr)	Returns the address of the start of the Forth system.
patch *new-word old-word word-to-patch*	(--)	Replaces *old-word* with *new-word* in *word-to-patch.*
(patch)	(new-n old-n xt --)	Replaces *old-n* with *new-n* in word indicated by *xt*.

TABLE D-21 Dictionary Compilation Commands *(Continued)*

Command	Stack Diagram	Description
`postpone` *name*	(--)	Delays the execution of the word name.
`recursive`	(--)	Makes the name of the colon definition being compiled visible in the dictionary, and thus allows the name of the word to be used recursively in its own definition.
`state`	(-- addr)	Variable that is non-zero in compile state.

Assembly Language Programming

TABLE D-22 Assembly Language Programming

Command	Stack Diagram	Description
`code` *name*	(-- code-sys) (E: ... -- ???)	Begins the creation of an assembly language routine called *name*. Commands that follow are interpreted as assembler mnemonics. Note that if the assembler is not installed, `code` is still present, except that machine code must be entered numerically (for example, in hex) with ",".
`c;`	(code-sys --)	Ends the creation of an assembly language routine. Automatically assembles the Forth interpreter "next" function so that the created assembly-code word, when executed, returns control to the calling routine as usual.
`label` *name*	(-- code-sys) (E: -- a-addr)	Begins the creation of an assembly language routine called *name*. Words created with `label` leave the address of the code on the stack when executed. The commands that follow are interpreted as assembler mnemonics. As with `code`, `label` is present even if the assembler is not installed.
`end-code`	(code-sys --)	Ends the assembly language patch started with `label`.

Basic Number Display

TABLE D-23 Basic Number Display

Command	Stack Diagram	Description
`.`	(n --)	Displays a number in the current base.
`.r`	(n size --)	Displays a number in a fixed width field.
`.s`	(--)	Displays contents of data stack.
`showstack`	(--)	Executes `.s` automatically before each `ok` prompt.
`noshowstack`	(--)	Turns off automatic display of the stack before each `ok` prompt.
`u.`	(u --)	Displays an unsigned number.
`u.r`	(u size --)	Displays an unsigned number in a fixed width field.

Changing the Number Base

TABLE D-24 Changing the Number Base

Command	Stack Diagram	Description
`.d`	(n --)	Displays *n* in decimal without changing base.
`.h`	(n --)	Displays *n* in hex without changing base.
`base`	(-- addr)	Variable containing number base.
`decimal`	(--)	Sets the number base to 10.
`d#` *number*	(-- n)	Interprets *number* in decimal; base is unchanged.
`hex`	(--)	Sets the number base to 16.
`h#` *number*	(-- n)	Interprets *number* in hex; base is unchanged.

Numeric Output Word Primitives

TABLE D-25 Numeric Output Word Primitives

Command	Stack Diagram	Description
#	(+l1 -- +l2)	Converts a digit in pictured numeric output.
#>	(l -- addr +n)	Ends pictured numeric output.
<#	(--)	Initializes pictured numeric output.
(.)	(n -- addr len)	Converts a number to a string.
(u.)	(u -- addr len)	Converts unsigned to string.
digit	(char base -- digit true | char false)	Converts a character to a digit.
hold	(char --)	Inserts the char in the pictured numeric output string.
$number	(addr len -- true | n false)	Converts a string to a number.
#s	(l -- 0)	Converts the rest of the digits in pictured numeric output.
sign	(n --)	Sets sign of pictured output.

Controlling Text Input

TABLE D-26 Controlling Text Input

Command	Stack Diagram	Description
(*ccc*)	(--)	Begins a comment.
\ *rest-of-line*	(--)	Skips the rest of the line.
ascii *ccc*	(-- char)	Gets numerical value of first ASCII character of next word.
accept	(addr len1 -- len2)	Gets a line of edited input from the console input device; stores at *addr*. *len1* is the maximum allowed length. *len2* is the actual length received.

TABLE D-26 Controlling Text Input *(Continued)*

Command	Stack Diagram	Description
`expect`	(addr len --)	Gets and displays a line of input from the console; stores at *addr*. (Recommend using `accept` instead.)
`key`	(-- char)	Reads a character from the console input device.
`key?`	(-- flag)	True if a key has been typed on the console input device.
`parse`	(char -- str len)	Parses text from the input buffer delimited by *char*.
`parse-word`	(-- str len)	Skips leading spaces and parses text from the input buffer delimited by white space.
`safe-parse-word`	(-- str len)	Similar to `parse-word` but intended for use in cases where the null string as input is indicative of an error.
`word`	(char -- pstr)	Collects a string delimited by *char* from the input buffer and places it as a packed string in memory at *pstr*. (Recommend using `parse` instead.)

Displaying Text Output

TABLE D-27 Displaying Text Output

Command	Stack Diagram	Description
`." ccc"`	(--)	Compiles a string for later display.
`(cr`	(--)	Moves the output cursor back to the beginning of the current line.
`cr`	(--)	Terminates a line on the display and goes to the next line.
`emit`	(char --)	Displays the character.
`exit?`	(-- flag)	Enables the scrolling control prompt: `More [<space>,<cr>,q] ?` The return flag is `true` if the user wants the output to be terminated.
`space`	(--)	Displays a `space` character.
`spaces`	(+n --)	Displays +*n* spaces.
`type`	(addr +n --)	Displays *n* characters.

Formatted Output

TABLE D-28 Formatted Output

Command	Stack Diagram	Description
`#lines`	(-- rows)	Value holding the number of lines on the output device.
`#out`	(-- a-addr)	Variable holding the column number on the output device.

Manipulating Text Strings

TABLE D-29 Manipulating Text Strings

Command	Stack Diagram	Description
`",`	(addr len --)	Compiles an array of bytes from *addr* of length *len*, at the top of the dictionary as a packed string.
`" ccc"`	(-- addr len)	Collects an input stream string, either interpreted or compiled. Within the string, `"(00,ff…)` can be used to include arbitrary byte values.
`.( ccc)`	(--)	Displays a string immediately.
`-trailing`	(addr +n1 -- addr +n2)	Removes trailing spaces.
`bl`	(-- char)	ASCII code for the space character; decimal 32.
`count`	(pstr -- addr +n)	Unpacks a packed string.
`lcc`	(char -- lowercase-char)	Converts a character to lowercase.
`left-parse-string`	(addr len char -- addrR lenR addrL lenL)	Splits a string at *char* (which is discarded).
`pack`	(addr len pstr -- pstr)	Makes a packed string from *addr len*; places it at *pstr*.
`p" ccc"`	(-- pstr)	Collects a string from the input stream; stores as a packed string.
`upc`	(char -- uppercase-char)	Converts a character to uppercase.

I/O Redirection Commands

TABLE D-30 I/O Redirection Commands

Command	Stack Diagram	Description
`input`	(device --)	Selects device (`keyboard`, or *device-specifier*) for subsequent input.
`io`	(device --)	Selects device for subsequent input and output.
`output`	(device --)	Selects device (`screen`, or *device-specifier*) for subsequent output.

ASCII Constants

TABLE D-31 ASCII Constants

Command	Stack Diagram	Description
`bell`	(-- n)	ASCII code for the bell character; decimal 7.
`bs`	(-- n)	ASCII code for the backspace character; decimal 8.

Command Line Editor Keystroke Commands

TABLE D-32 Command Line Editor Keystroke Commands

Keystroke	Description
Control-B	Moves backward one character.
Escape B	Moves backward one word.
Control-F	Moves forward one character.
Escape F	Moves forward one word.

TABLE D-32 Command Line Editor Keystroke Commands *(Continued)*

Keystroke	Description
Control-A	Moves backward to beginning of line.
Control-E	Moves forward to end of line.
Delete	Erases previous character.
Backspace	Erases previous character.
Control-H	Erases previous character.
Escape H	Erases from beginning of word to just before the cursor, storing erased characters in a save buffer.
Control-W	Erases from beginning of word to just before the cursor, storing erased characters in a save buffer.
Control-D	Erases next character.
Escape D	Erases from cursor to end of the word, storing erased characters in a save buffer.
Control-K	Erases from cursor to end of line, storing erased characters in a save buffer.
Control-U	Erases entire line, storing erased characters in a save buffer.
Control-R	Retypes the line.
Control-Q	Quotes next character (allows you to insert control characters).
Control-Y	Inserts the contents of the save buffer before the cursor.
Control-P	Selects and displays the previous line for subsequent editing.
Control-N	Selects and displays the next line for subsequent editing.
Control-L	Displays the entire contents of the editing buffer.

Command Completion Keystroke Commands

TABLE D-33 Command Completion Keystroke Commands

Keystroke	Description
Control-Space	Completes the name of the current word.
Control-/	Displays all possible matches for the current word.

Comparison Commands

TABLE D-34 Comparison Commands

Command	Stack Diagram	Description
<	(n1 n2 -- flag)	True if *n1* < *n2*.
<=	(n1 n2 -- flag)	True if *n1* <= *n2*.
<>	(n1 n2 -- flag)	True if *n1* is not equal to *n2*.
=	(n1 n2 -- flag)	True if *n1* = n2.
>	(n1 n2 -- flag)	True if *n1* > n2.
>=	(n1 n2 -- flag)	True if *n1* >= n2.
0<	(n -- flag)	True if *n* < 0.
0<=	(n -- flag)	True if *n* <= 0.
0<>	(n -- flag)	True if *n* <> 0.
0=	(n -- flag)	True if *n* = 0 (also inverts any flag).
0>	(n -- flag)	True if *n* > 0.
0>=	(n -- flag)	True if *n* >= 0.
between	(n min max -- flag)	True if *min* <= *n* <= *max*.
false	(-- 0)	The value FALSE, which is 0.
true	(-- -1)	The value TRUE, which is -1.
u<	(u1 u2 -- flag)	True if *u1* < *u2*, unsigned.

TABLE D-34 Comparison Commands *(Continued)*

Command	Stack Diagram	Description
`u<=`	(u1 u2 -- flag)	True if *u1* <= *u2*, unsigned.
`u>`	(u1 u2 -- flag)	True if *u1* > *u2*, unsigned.
`u>=`	(u1 u2 -- flag)	True if *u1* >= *u2*, unsigned.
`within`	(n min max -- flag)	True if *min* <= *n* < *max*.

if-else-then Commands

TABLE D-35 if-else-then Commands

Command	Stack Diagram	Description
if	(flag --)	Executes the following code when *flag* is true.
else	(--)	Executes the following code when *flag* is false.
then	(--)	Terminates if…then…else.

case Statement Commands

TABLE D-36 case Statement Commands

Command	Stack Diagram	Description
case	(selector -- selector)	Begins a case…endcase conditional.
endcase	(selector | {empty} --)	Terminates a case…endcase conditional.
endof	(--)	Terminates an of…endof clause within a case…endcase.
of	(selector test-value -- selector | {empty})	Begins an of…endof clause within a case conditional.

begin (Conditional) Loop Commands

TABLE D-37 begin (Conditional) Loop Commands

Command	Stack Diagram	Description
again	(--)	Ends a begin…again infinite loop.
begin	(--)	Begins a begin…while…repeat, begin…until, or begin…again loop.
repeat	(--)	Ends a begin…while…repeat loop.
until	(flag --)	Continues executing a begin…until loop until *flag* is true.
while	(flag --)	Continues executing a begin…while…repeat loop while *flag* is true.

do (Counted) Loop Commands

TABLE D-38 do (Counted) Loop Commands

Command	Stack Diagram	Description
+loop	(n --)	Ends a do…+loop construct; adds *n* to loop index and returns to do (if *n* < 0, index goes from *start* to *end*, inclusive).
?do	(end start --)	Begins ?do…loop to be executed 0 or more times. Index goes from *start* to *end*-1, inclusive. If *end* = *start*, loop is not executed.
?leave	(flag --)	Exits from a do…loop if *flag* is non-zero.
do	(end start --)	Begins a do…loop. Index goes from *start* to *end*-1, inclusive. Example: 10 0 do i . loop (prints 0 1 2…d e f).
i	(-- n)	Leaves the loop index on the stack.
j	(-- n)	Leaves the loop index for next outer enclosing loop.
leave	(--)	Exits from do…loop.
loop	(--)	End of do…loop.

Program Execution Control Commands

TABLE D-39 Program Execution Control Commands

Command	Stack Diagram	Description
`abort`	(--)	Aborts current execution and interprets keyboard commands.
`abort"` *ccc*"	(abort? --)	If *abort?* is true, aborts and displays message.
`eval`	(... str len -- ???)	Synonym for `evaluate`.
`evaluate`	(... str len -- ???)	Interprets Forth source text from the specified string.
`execute`	(xt --)	Executes the word whose execution token is on the stack.
`exit`	(--)	Returns from the current word. (Cannot be used in counted loops.)
`quit`	(--)	Same as `abort`, but leaves stack intact.

File Loading Commands

TABLE D-40 File Loading Commands

Command	Stack Diagram	Description
`?go`	(--)	Executes Forth, FCode, or binary programs.
`boot` *[specifiers]* `-h`	(--)	Loads file from specified source.
`byte-load`	(addr span --)	Interprets loaded FCode binary file. *span* is usually 1.
`dl`	(--)	Loads a Forth file over a serial line with `tip` and interpret. Type: **`~C cat`** *filename* **`^-D`**
`dlbin`	(--)	Loads a binary file over a serial line with `tip`. Type: **`~C cat`** *filename*

TABLE D-40 File Loading Commands *(Continued)*

Command	Stack Diagram	Description
`dload` *filename*	(addr --)	Loads the specified file over Ethernet at the given address.
`eval`	(addr len --)	Interprets loaded Forth text file.
`go`	(--)	Begins executing a previously-loaded binary program, or resumes executing an interrupted program.
`init-program`	(--)	Initializes to execute a binary file.
`load` *device-specifier argument*	(--)	Loads data from specified device into memory at the address given by `load-base`.
`load-base`	(-- addr)	Address at which `load` places the data it reads from a device.

Disassembler Commands

TABLE D-41 Disassembler Commands

Command	Stack Diagram	Description
`+dis`	(--)	Continues disassembling where the last disassembly left off.
`dis`	(addr --)	Begins disassembling at the specified address.

Breakpoint Commands

TABLE D-42 Breakpoint Commands

Command	Stack Diagram	Description
+bp	(addr --)	Adds a breakpoint at the given address.
-bp	(addr --)	Removes the breakpoint at the given address.
--bp	(--)	Removes the most-recently-set breakpoint.
.bp	(--)	Displays all currently set breakpoints.
.breakpoint	(--)	Performs a specified action when a breakpoint occurs. This word can be altered to perform any desired action. For example, to display registers at every breakpoint, type: `['] .registers is .breakpoint`. The default behavior is `.instruction`. To perform multiple behaviors, create a single definition which calls all desired behaviors, then load that word into `.breakpoint`.
.instruction	(--)	Displays the address, opcode for the last-encountered breakpoint.
.step	(--)	Performs a specified action when a single step occurs. (See `.breakpoint`).
bpoff	(--)	Removes all breakpoints.
finish-loop	(--)	Executes until the end of this loop.
go	(--)	Continues from a breakpoint. This can be used to go to an arbitrary address by setting up the processor's program counter before issuing `go`.
gos	(n --)	Executes `go` *n* times.
hop	(--)	(Like the `step` command.) Treats a subroutine call as a single instruction.
hops	(n --)	Executes `hop` *n* times.
return	(--)	Executes until the end of this subroutine.
returnl	(--)	Executes until the end of this leaf subroutine.
skip	(--)	Skips (does not execute) the current instruction.

TABLE D-42 Breakpoint Commands *(Continued)*

Command	Stack Diagram	Description
`step`	(--)	Single-steps one instruction.
`steps`	(n --)	Executes `step` *n* times.
`till`	(addr --)	Executes until the given address is encountered. Equivalent to `+bp go`.

Forth Source-level Debugger Commands

TABLE D-43 Forth Source-level Debugger Commands

Command	Description
`c`	“Continue”. Switches from stepping to tracing, thus tracing the remainder of the execution of the word being debugged.
`d`	“Down a level”. Marks for debugging the word whose name was just displayed, then executes it.
`u`	“Up a level”. Un-marks the word being debugged, marks its caller for debugging, and finishes executing the word that was previously being debugged.
`f`	Starts a subordinate Forth interpreter. When that interpreter exits (with `resume`), control returns to the debugger at the place where the `f` command was executed.
`g`	“Go.” Turns off the debugger and continues execution.
`q`	“Quit”. Aborts the execution of the word being debugged and all its callers and returns to the command interpreter.
`s`	“see”. Decompiles the word being debugged.
`$`	Displays the address,len on top of the stack as a text string.
`h`	“Help”. Displays symbolic debugger documentation.
`?`	“Short Help”. Displays brief symbolic debugger documentation.
`debug` *name*	Marks the specified Forth word for debugging. Enters the Forth Source-level Debugger on all subsequent attempts to execute *name*. After executing `debug`, the execution speed of the system may decrease until debugging is turned off with `debug-off`. (Do not debug basic Forth words such as “`dup`”.)
`(debug`	Like `debug`, except that `(debug` takes an execution token from the stack instead of a name from the input stream.
`debug-off`	Turns off the Forth Source-level Debugger so that no word is being debugged.
`resume`	Exits from a subordinate interpreter, and goes back to the stepper. (See the `f` command in this table.)

TABLE D-43 Forth Source-level Debugger Commands *(Continued)*

Command	Description
`stepping`	Sets step mode for the Forth Source-level Debugger, allowing the interactive, step-by-step execution of the word being debugged. Step mode is the default.
`tracing`	Sets trace mode for the Forth Source-level Debugger. Tracing enables the execution of the word being debugged, while showing the name and stack contents for each word called by that word.
`<space-bar>`	Executes the word just displayed and proceeds to the next word.

Time Utilities

TABLE D-44 Time Utilities

Command	Stack Diagram	Description
`get-msecs`	(-- ms)	Returns the approximate current time in milliseconds.
`ms`	(n --)	Delays for *n* milliseconds. Resolution is 1 millisecond.

Miscellaneous Operations

TABLE D-45 Miscellaneous Operations

Command	Stack Diagram	Description
`callback` *string*	(value --)	Calls Solaris with the given value and string.
`catch`	(... xt -- ??? error-code | ??? false)	Executes *xt*; returns `throw` error code or 0 if `throw` is not called.
`eject floppy`	(--)	Ejects the diskette from the floppy drive.
`firmware-version`	(-- n)	Returns major/minor CPU firmware version (that is, 0x00030001 = firmware version 3.1).
`forth`	(--)	Restores main Forth vocabulary to top of search order.
`ftrace`	(--)	Shows calling sequence when exception occurred.
`noop`	(--)	Does nothing.
`reset-all`	(--)	Resets the entire system (similar to a power-cycle).
`sync`	(--)	Calls the operating system to write any pending information to the hard disk. Also boots after syncing file systems.
`throw`	(error-code --)	Returns given error code to `catch`.

Multiprocessor Commands

TABLE D-46 Multiprocessor Commands

Command	Stack Diagram	Description
`switch-cpu`	(cpu# --)	Switches to indicated CPU.

Memory Mapping Commands

TABLE D-47 Memory Mapping Commands

Command	Stack Diagram	Description
`map?`	(virt --)	Displays memory map information for the virtual address.
`memmap`	(phys space size -- virt)	Maps a region of physical addresses; returns the allocated virtual address. Unmaps with `free-virtual`.
`obio`	(-- space)	Specifies the device address space for mapping.
`obmem`	(-- space)	Specifies the onboard memory address space for mapping.
`sbus`	(-- space)	Specifies the SBus address space for mapping.

Memory Mapping Primitives

TABLE D-48 Memory Mapping Primitives

Command	Stack Diagram	Description
`iomap?`	(virt --)	Displays IOMMU page map entry for the virtual address.
`iomap-page`	(phys space virt --)	Maps physical page given by *phys* and *space* to the virtual address.
`iomap-pages`	(phys space virt size --)	Performs consecutive `iomap-page`s to map a region of memory given by *size*.
`iopgmap@`	(virt -- pte | 0)	Returns IOMMU page map entry for the virtual address.
`iopgmap!`	(pte virt --)	Stores a new page map entry for the virtual address.
`map-page`	(phys space virt --)	Maps one page of memory starting at address *phys* on to virtual address *virt* in the specified address *space*. All addresses are truncated to lie on a page boundary.
`map-pages`	(phys space virt size --)	Performs consecutive `map-page`s to map a region of memory to the specified *size*.
`map-region`	(region# virt --)	Maps a region.
`map-segments`	(smentry virt len --)	Performs consecutive `smap!` operations to map a region of memory.
`pgmap!`	(pmentry virt --)	Stores a new page map entry for the virtual address.
`pgmap?`	(virt --)	Displays the page map entry (decoded and in English) corresponding to the virtual address.
`pgmap@`	(virt -- pmentry)	Returns the page map entry for the virtual address.
`pagesize`	(-- size)	Returns the *size* of a page.

TABLE D-48 Memory Mapping Primitives *(Continued)*

Command	Stack Diagram	Description
`rmap!`	(rmentry virt --)	Stores a new region map entry for the virtual address.
`rmap@`	(virt -- rmentry)	Returns the region map entry for the virtual address.
`segmentsize`	(-- size)	Returns the *size* of a segment.
`smap!`	(smentry virt --)	Stores a new segment map entry for the virtual address.
`smap?`	(virt --)	Formatted display of the segment map entry for the virtual address.
`smap@`	(virt -- smentry)	Returns the segment map entry for the virtual address.

Cache Manipulation Commands

TABLE D-49 Cache Manipulation Commands

Command	Stack Diagram	Description
`clear-cache`	(--)	Invalidates all cache entries.
`cache-off`	(--)	Disables the cache.
`cache-on`	(--)	Enables the cache.
`ecdata!`	(data offset --)	Stores the *data* at the cache *offset*.
`ecdata@`	(offset -- data)	Fetches (returns) *data* from the cache *offset*.
`ectag!`	(value offset --)	Stores the tag *value* at the cache *offset*.
`ectag@`	(offset -- value)	Return the tag *value* at the cache *offset*.
`flush-cache`	(--)	Writes back any pending data from the cache.

Reading / Writing Machine Registers in Sun-4u Machines

TABLE D-50 Reading / Writing Machine Registers in Sun-4u Machines

Command	Stack Diagram	Description
`aux!`	(data --)	Writes auxiliary register.
`aux@`	(-- data)	Reads auxiliary register.

Alternate Address Space Access Commands

TABLE D-51 Alternate Address Space Access Commands

Command	Stack Diagram	Description
`spacec!`	(byte addr asi --)	Stores the *byte* in *asi* at *addr*.
`spacec?`	(addr asi --)	Displays the *byte* in *asi* at *addr*.
`spacec@`	(addr asi -- byte)	Fetches the *byte* from *asi* at *addr*.
`spaced!`	(quad1 quad2 addr asi --)	Stores the two quadlets in *asi* at *addr*. Order is implementation-dependent.
`spaced?`	(addr asi --)	Displays the two quadlets in *asi* at *addr*. Order is implementation-dependent.
`spaced@`	(addr asi -- quad1 quad2)	Fetches the two quadlets from *asi* at *addr*. Order is implementation-dependent.
`spacel!`	(quad addr asi --)	Stores the quadlet in *asi* at *addr*.
`spacel?`	(addr asi --)	Displays the quadlet in *asi* at *addr*.
`spacel@`	(addr asi -- quad)	Fetches the quadlet from *asi* at *addr*.
`spacew!`	(w addr asi --)	Stores the doublet in *asi* at *addr*.
`spacew?`	(addr asi --)	Displays the doublet in *asi* at *addr*.
`spacew@`	(addr asi -- w)	Fetches the doublet from *asi* at *addr*.
`spacex!`	(x addr asi --)	Stores the number in *asi* at *addr*.
`spacex?`	(addr asi --)	Displays the word in *asi* at *addr*.
`spacex@`	(addr asi -- x)	Fetches the word from *asi* at *addr*.

SPARC Register Commands

TABLE D-52 SPARC Register Commands

Command	Stack Diagram	Description
%g0 through %g7	(-- value)	Returns the value in the specified global register.
%i0 through %i7	(-- value)	Returns the value in the specified input register.
%l0 through %l7	(-- value)	Returns the value in the specified local register.
%o0 through %o7	(-- value)	Returns the value in the specified output register.
%pc %npc %y	(-- value)	Returns the value in the specified register.
%f0 through %f31	(-- value)	Returns the value in the specified floating point register.
.fregisters	(--)	Displays the values in %f0 through %f31.
.locals	(--)	Displays the values in the i, l and o registers.
.registers	(--)	Displays values in processor registers.
.window	(window# --)	Same as w .locals; displays the desired window.
ctrace	(--)	Displays the return stack showing C subroutines.
set-pc	(new-value --)	Sets %pc to *new-value*, and sets %npc to (*new-value*+4).
to *regname*	(new-value --)	Changes the value stored in any of the above registers. Use in the form: *new-value* to *regname*.
w	(window# --)	Sets the current window for displaying %ix, %lx, or %ox.

SPARC V9 Register Commands

TABLE D-53 SPARC V9 Register Commands

Command	Stack Diagram	Description
%fprs %asi %pstate %tl-c %pil %tstate %tt %tba %cwp %cansave %canrestore %otherwin %wstate %cleanwin	(-- value)	Returns the value in the specified register.
.pstate	(--)	Formatted display of the processor state register.
.ver	(--)	Formatted display of the version register.
.ccr	(--)	Formatted display of the ccr register.
.trap-registers	(--)	Displays trap-related registers.

Emergency Keyboard Commands

TABLE D-54 Emergency Keyboard Commands

Command	Description
Stop	Bypasses POST. This command does not depend on security-mode. (Note: some systems bypass POST as a default; in such cases, use Stop-D to start POST.)
Stop-A	Aborts.

TABLE D-54 Emergency Keyboard Commands

Command	Description
`Stop-D`	Enters diagnostic mode (sets `diag-switch?` to `true`).
`Stop-F`	Enters Forth on TTYA instead of probing. Uses `fexit` to continue with the initialization sequence. Useful if hardware is broken.
`Stop-N`	Resets NVRAM contents to default values.

Diagnostic Test Commands

TABLE D-55 Diagnostic Test Commands

Command	Description
`probe-scsi`	Identifies devices attached to a SCSI bus.
`test` *device-specifier*	Executes the specified device's `selftest` method. For example: `test net` - test the network connection
`watch-clock`	Tests a clock function.
`watch-net`	Monitors a network connection.
`test-all` *device specifier*	Executes the selftest method for all devices at or below *device-specifier*. If no device specifier is provided, executes the selftest methods for all devices in the device tree.
`obdiag`	Invokes an optional interactive menu tool which lists all selftest methods available on a system; provides commands to run selftests.

Index

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X