NAME

boot – start the system kernel or a standalone program

SPARC SYNOPSIS

ok boot [*OBP names*] [*filename*] [*boot-flags*]

>b [*device* [*(c, u, p)*]] [*filename*] [**–a**] *boot-flags*

x86 SYNOPSIS

> b [*filename*] [*boot-args*]
> i

DESCRIPTION

Bootstrapping is the process of loading and executing a standalone program. For the purpose of this discussion, bootstrapping means the process of loading and executing the bootable operating system. Typically, the standalone program is the operating system kernel (see **kernel**(1M)), but any standalone program can be booted instead. As an example, on a SPARC system, the diagnostic monitor for a machine is a good example of a standalone program other than the operating system that can be booted.

SPARC Bootstrap Procedure

On SPARC systems, the bootstrap procedure on most machines consists of the following basic phases.

After the machine is turned on the system firmware (in PROM) executes power-on self test (POST). The form and scope of these tests depends on the version of the firmware in your system.

After the tests have been completed successfully, the firmware attempts to autoboot if the appropriate flag has been set in the non-volatile storage area used by the firmware. The name of the file to load, and the device to load it from can also be manipulated.

These flags and names can be set using the **eeprom**(1M) command from the shell, or by using PROM commands from the **'ok'** prompt after the system has been halted.

The second level program is either **ufsboot**, (when booting from a disk) or **inetboot** (when booting across the network). When booting from a disk, the PROM assumes that the primary bootblock resides in blocks 1 to 15 of the local disk. When booting over the network, the PROM makes a reverse ARP request and when it receives a reply, the PROM makes a TFTP request to the server that responded and fetches inetboot across the network and executes it. Inetboot also makes another reverse ARP request, then uses the **bootparams** protocol to locate its root filesystem. It then fetches the kernel across the network using the NFS protocol and then executes it.

If the pathname to the standalone is relative (does not begin with a slash), the second level boot will look for the standalone in its platform-dependent location, that is, it prepends **/platform/***platform-name* to the name of the standalone. See **filesystem**(5). If the pathname is absolute, **boot** will use the specified path. The **boot** program then loads the standalone at the appropriate address, and then transfers control.

If the standalone is identified as a dynamically-linked executable, boot will load the interpreter (linker/loader) as indicated by the executable format and then transfer control to the interpreter. If the standalone is statically-linked, it will jump directly to the standalone.

Typically, the standalone program is the kernel. Once the program is loaded, it starts the UNIX system, mounts the necessary filesystems (see **vfstab**(4)), and runs **/sbin/init** to bring the system to the "initdefault" state specified in **/etc/inittab** (see **inittab**(4)).

x86 Bootstrap Procedure

On x86 systems, the bootstrapping process consists of two conceptually distinct phases, primary boot and secondary boot. The primary boot is implemented in the BIOS ROM on the system board, in BIOS extensions in ROMs on peripheral boards, and sometimes in an executive program and BIOS extensions on a boot diskette. It is distinguished by its ability to control the installed peripheral devices and to provide I/O services through software interrupts. It begins the booting process by loading the first physical sector from a mass storage device or by loading a program over the LAN. The primary boot is implemented in x86 real-mode code.

The secondary boot is loaded by the primary boot and uses I/O services provided by the primary boot. It is implemented in 32-bit, paged, protected mode code, and it is capable of reading files from a file system. There are two separate versions of the secondary boot:

ufsboot boots from a UFS file system on a hard disk or a CD

inetboot boots over a LAN using the NFS protocol

All versions of the secondary boot contain a built-in shell-like command interpreter. The command language is defined below. The interpreter reads the script in **/etc/bootrc**, which controls the booting process. This file can be modified to change defaults or to adapt to a specific machine.

The standard **/etc/bootrc** script prompts the user to enter a **b** character to boot with specified options, an **i** character to invoke the interpreter interactively, or any other character to boot the default kernel. Once the kernel is loaded, it starts the operating system, loads the necessary modules, mounts the necessary filesystems (see **vfstab**(4)), and runs **/sbin/init** to bring the system to the ''initdefault'' state specified in **/etc/inittab** (see **inittab**(4)).

SPARC OPTIONS

OBP names Specify the open boot prom designations. For example, on Desktop SPARCsystems, the designation **/sbus/esp@0,800000/sd@3,0:a** indicates a SCSI disk (sd) at target 3, lun0 on the SCSI bus, with the esp host adapter plugged into slot 0.

filename Name of a standalone program to boot. The default is to boot **/platform/***platform-name***/kernel/unix** from the root partition, but you can specify another program, such as **/stand/diag** on the command line. Some versions of the firmware allow the default filename to be saved in the non-volatile storage area of the system.

−a The boot program interprets this flag to mean **ask me**, and so it prompts for the name of the standalone. The **'−a'** flag is then handed onto the standalone.

boot-flags The boot program passes all *boot-flags* to *filename*. They are not interpreted by **boot**. See the **kernel**(1M) and **kadb**(1M) manual pages for information on the options available with the default standalone program, **kernel/unix**.

x86 OPTIONS

filename Name of a standalone program to boot. The default is to boot **/platform/***platform-name***/kernel/unix** from the root partition, but you can specify another program on the command line.

boot-args The boot program passes all *boot-args* to *filename*. They are not interpreted by **boot**. See the **kernel**(1M) and **kadb**(1M) manual pages for information on the options available with the kernel.

x86 BOOT SEQUENCE DETAILS

After a PC-compatible machine is turned on, the system firmware in the BIOS ROM executes a power-on self test (POST), runs BIOS extensions in peripheral board ROMs, and invokes software interrupt INT 19h, Bootstrap. The INT 19h handler typically performs the standard PC-compatible boot, which consists of trying to read the first physical sector from the first diskette drive, or, if that fails, from the first hard disk. The processor then jumps to the first byte of the sector image in memory.

x86 Hard Disk Boot

The first sector on a hard disk contains the master boot block, which contains the master boot program and the FDISK table, named for the PC program that maintains it. The master boot finds the active partition in the FDISK table, loads its first sector, and jumps to its first byte in memory. This completes the standard PC-compatible hard disk boot sequence.

The x86 FDISK partition begins with a one-cylinder boot slice, which contains the partition boot program (**pboot**) in the first sector, the standard Solaris disk label and volume table of contents (VTOC) in the second and third sectors, and the **bootblk** program in the fourth and subsequent sectors. When the Solaris FDISK partition is the active partition, the master boot program (**mboot**) reads the partition boot program in the first sector into memory and jumps to it. It in turn reads the **bootblk** program into memory and jumps to it. If the drive contains multiple FDISK partitions, the user is given the opportunity to reboot another partition. **bootblk** reads **ufsboot** from the file system in the root slice and jumps to its first byte in memory.

Like all versions of the secondary boot, **ufsboot** switches the processor to 32-bit, paged, protected mode and performs some limited machine initialization. It then invokes its command interpreter, which interprets **/etc/bootrc**.

x86 CD-ROM Boot

The standard BIOS ROM and BIOS extensions on SCSI Host Bus Adapters (HBAs) do not support booting from CD-ROM. A special Solaris boot diskette is provided for booting from CD-ROM or other devices not supported by the BIOS. It is typically used to install the Solaris operating system from CD-ROM, although it can also be used for system maintenance or network booting.

This diskette is in the form of a standard DOS diskette, but is not a DOS system disk. It contains an executive program and BIOS extensions that support booting over the network or reading CD-ROMs through various SCSI HBAs. When the diskette is booted, the executive program loads and runs each of the BIOS extension files. If an extension detects the presence of the device it supports, it installs itself in memory and chains its service routine to the appropriate software interrupt.

After the extensions have all been loaded, the executive displays a menu of bootable devices. The user chooses the device to be booted, typically the CD.

Once the bootable device has been selected, the boot process proceeds in the same manner as the hard disk boot, i.e., the executive reads the first physical sector from the device and jumps to its first byte.

x86 Network Booting

When booting is to occur over the network, a boot request multicast packet using the Remote Program Load (RPL) protocol is generated by the PC. Somewhere on the same network segment, an RPL server is listening to these boot requests. The information contained in the request is validated and a reply is sent back to the PC. The PC then requests bootfiles to be downloaded to its memory. These bootfiles have been previously configured for this PC. Once downloading has finished, the PC begins executing these programs, which initiate running of the secondary boot program **inetboot**. Like other versions of the secondary boot, **inetboot** switches the processor to 32-bit, paged, protected mode and performs some limited machine initialization. It then issues Reverse Address Resolution Protocol (RARP) requests to find out its own IP address and a **bootparams** RPC to find out which server to mount as its root file system using NFS. When the root has been mounted, it then invokes its command interpreter, which interprets **/etc/bootrc**.

Secondary Boot Programming Language for x86

The wide range of hardware that must be supported on x86 demands great flexibility in the booting process. This flexibility is achieved in part by making the secondary boot programmable. The secondary boot contains an interpreter that accepts a simple command language similar to those of **sh** and **csh**. The primary differences are that pipelines, loops, standard output, and output redirection are not supported.

x86 Lexical Structure

The boot interpreter splits input lines into words separated by blanks and tabs. The metacharacters are dollar sign (**$**), single-quote (**´**), double-quote (**"**), number sign (**#**), new-line, and backslash (****). The special meaning of metacharacters can be avoided by preceding them with a backslash. A new-line preceded by a backslash is treated as a blank. A number sign introduces a comment, which continues to the next new-line.

A string enclosed in a pair of single-quote or double-quote characters forms all or part of a single word. White space and new-line characters within a quoted string become part of the word. Characters within a quoted string can be quoted by preceding them with a backslash character; thus a single-quote character can appear in a single-quoted string by preceding it with a backslash. Two backslashes produce a single backslash, and a new-line preceded by a backslash produces a new-line in the string.

x86 Variables

The boot maintains a set of variables, each of which has a string value. The first character of a variable name must be a letter, and subsequent characters can be letters, digits, or underscores. The **set** command creates a variable and/or assigns a value to it, or displays the values of variables. The **unset** command deletes a variable.

Variable substitution is performed when the interpreter encounters a dollar-sign that is not preceded by a backslash. The variable name following the dollar sign is replaced by the value of the variable, and parsing continues at the beginning of the value. Variable substitution is performed in double-quoted strings, but not in single-quoted strings. A variable name can be enclosed in braces to separate it from following characters.

x86 Commands

A command is a sequence of words terminated by a new-line character. The first word is the name of the command and subsequent words are arguments to the command. All commands are built-in commands. Standalone programs are executed with the **run** command.

x86 Conditional Execution of Commands

Commands can be conditionally executed by surrounding them with the **if**, **elseif**, **else**, and **endif** commands:

```
if <expr1>
   ...
elseif <expr2>
   ...
elseif <expr3>
   ...
else
   ...
endif
```

An **if** block may be embedded in other **if** blocks.

x86 Expressions

The **set**, **if**, and **elseif** commands evaluate arithmetic expressions with the syntax and semantics of the C programming language. The **||**, **&&**, **|**, **^**, **&**, **==**, **!=**, **<**, **>**, **<=**, **>=**, **>>**, **<<**, **+**, **−**, **∗**, **/**, **%**, **~**, and **!** operators are accepted, as are **(**, **)**, and comma. Signed 32-bit integer arithmetic is performed.

Expressions are parsed after the full command line has been formed. Each token in an expression must be a separate argument word, so blanks must separate all tokens on the command line.

Before an arithmetic operation is performed on an operand word, it is converted from a string to a signed 32-bit integer value. After an optional leading sign, a leading 0 produces octal conversion and a leading 0x or 0X produces hexadecimal conversion. Otherwise, decimal conversion is performed. A string that is not a legal integer is converted to zero.

Several built-in functions for string manipulation are provided. Built-in function names begin with a dot. String arguments to these functions are not converted to integers. To cause an operator, e.g., **-**, to be treated as a string, it must be preceded by a backslash, and that backslash must be quoted with another backslash. Also be aware that a null string can produce a blank argument, and thus an expression syntax error. For example:

```
if .strneq ( ${usrarg}X , \− , 1 )
```

is the safe way to test whether the variable *usrarg* starts with a −, even if it could be null.

x86 I/O

The boot interpreter takes its input from the system console or from one or more files. The source command causes the interpreter to read a file into memory and begin parsing it. The console command causes the interpreter to take its input from the system console. Reaching EOF causes the interpreter to resume parsing the previous input source. CTRL-D entered at the beginning of console line is treated as EOF.

The **echo** command writes its arguments to the display. The **read** command reads the system console and assigns word values to its argument variables.

x86 Debugging

The **verbose** command turns verbose mode on and off. In verbose mode, the interpreter displays lines from the current source file and displays the command as actually executed after variable substitution.

The **singlestep** command turns singlestep mode on and off. In singlestep mode, the interpreter displays **step ?** before processing the next command, and waits for keyboard input, which is discarded. Processing proceeds when ENTER is pressed. This allows slow execution in verbose mode.

x86 Initialization

When the interpreter is first invoked by the boot, it begins execution of a compiled-in initialization string. This string typically consists of "source /etc/bootrc\n" to run the boot script in the root file system.

x86 Communication With Standalone Programs

The boot passes information to standalone programs through arguments to the **run** command. A standalone program can pass information back to the boot by setting a boot interpreter variable using the **var_ops()** boot service function. It can also pass information to the kernel using the **setprop()** boot service function. The **whoami** property is set to the name of the standalone program.

x86 Built-in Commands

console
: Interpret input from the console until CTRL-D.

echo arg1 . . .
: Display the arguments separated by blanks and terminate with a new-line.

echo -n arg1 . . .
: Display the arguments separated by blanks, but do not terminate with a new-line.

else
elseif
endif See if.

getprop *propname varname*
: Assign the value of property *propname* to the variable *varname*. A property value of length zero produces a null string. If the property does not exist, the variable is not set.

getproplen *propname varname*
Assign the length in hexadecimal of the value of property *propname* to the variable *varname*. Property value lengths include the terminating null. If the property does not exist, the variable is set to 0xFFFFFFFF (-1).

if **<***expr***>**
If the expression *expr* is true, execute instructions to the next **elseif**, **else**, or **endif**. If **expr** is false, do not execute the instructions.

elseif **<***expr***>**
If the preceding **if** and **elseif** commands all failed, and *expr* is true, execute instructions to the next **elseif**, **else**, or **endif**. Otherwise, do not execute the instructions.

else If the preceding **if** and **elseif** commands all failed, execute instructions to the next **elseif**, **else**, or **endif**. Otherwise, do not execute the instructions.

endif Revert to the execution mode of the surrounding block.

help Display a help screen that contains summaries of all available boot shell commands.

read *name1 . . .*
Read a line from the console, break it into words, and assign them as values to the variables *name1*, etc.

readt *time . . .*
Same as **read**, but timeout after *time* seconds.

run *name arg1 . . .*
Load and transfer control to the standalone program *name*, passing it *arg1* and further arguments.

set Display all the current variables and their values.

set *name*
Set the value of the variable name to the null string.

set *name word*
Set the value of the variable *name* to *word*.

set *name* **<***expr***>**
Set the value of the variable name to the value of *expr*. *expr* must consist of more than one word. The value is encoded in unsigned hexadecimal, so that -1 is represented by 0xFFFFFFFF.

setcolor
Set the text mode display attributes. Allowable colors are *black*, *blue*, *green*, *cyan*, *red*, *magenta*, *brown*, *white*, *gray*, *lt_blue*, *lt_green*, *lt_cyan*, *lt_red*, *lt_magenta*, *yellow*, and *hi_white*.

setprop *propname word*
Set the value of the property *propname* to *word*.

singlestep
singlestep on
Turn on singlestep mode, in which the interpreter displays **step ?** before each command is processed, and waits for keyboard input. Press ENTER to execute the next command.

singlestep off
Turn off singlestep mode.

source *name*
Read the file *name* into memory and begin to interpret it. At EOF, return to the previous source of input.

unset *name*
Delete the variable name.

verbose
verbose on
Turn on verbose mode, which displays lines from source files and commands to be executed.

verbose off
Turn off verbose mode.

x86 Built-in Functions

The following built-in functions are accepted within expressions:

.strcmp(*string1*, *string2*)
Returns an integer value that is less than, equal to, or greater than zero, as *string1* is lexicographically less than, equal to, or greater than *string2*.

.strncmp(*string1*, *string2*, *n*)
Returns an integer value that is less than, equal to, or greater than zero, as *string1* is lexicographically less than, equal to, or greater than *string2*. At most, *n* characters are compared.

.streq (*string1*, *string2*)
Returns true if *string1* is equal to *string2*, and false otherwise.

.strneq (*string1*, *string2*, *n*)
Returns true if *string1* is equal to *string2*, and false otherwise. At most, *n* characters are compared.

.strfind (*string*, *addr*, *n*)
Scans *n* locations in memory starting at *addr*, looking for the beginning of *string*. The *string* in memory need not be null-terminated. Returns true if *string* is found, and false otherwise. **.strfind** can be used to search for strings in the ROM BIOS and BIOS extensions that identify different machines and peripheral boards.

FILES

/platform/*platform-name***/ufsboot**
second level program to boot from a disk or CD.
/platform/*platform-name***/kernel/unix**
default program to boot system.
/etc/inittab table in which the "initdefault" state is specified.

	/sbin/init	program that brings the system to the "initdefault" state.
x86 Only	**/etc/bootrc**	script that controls the booting process.

SEE ALSO

uname(1), **eeprom**(1M), **init**(1M), **installboot**(1M), **kadb**(1M), **kernel**(1M), **shutdown**(1M), **inittab**(4), **vfstab**(4)

System Administration Guide, Volume I

x86 Only

rpld(1M)

WARNINGS

The boot program is not smart enough to know which files can be used as bootable programs. If the booting of a file that is not bootable is requested, the boot program loads it and branches to it. What happens after that is unpredictable.

NOTES

Because the ‘‘-’’ key on national language keyboards has been moved, an alternate key must be used to supply arguments to the **boot** command on an x86 system using these keyboards. Use the ‘‘-’’ on the numeric keypad. The specific language keyboard and the alternate key to be used in place of the ‘‘-’’ during bootup is shown below.

Keyboard	**Substitute Key**
Italy	´
Spain	´
Sweden	+
France	?
Germany	?

For example, **b -r** would be typed as **b +r** on Swedish keyboards, although the screen display will show as **b -r**.