kb.h — Header File

Define keys for loadable keyboard driver
#include <sys/kb.h>

The header file **kb.h** defines macros and manifest constants that are used with **nkb**, the user-configurable keyboard driver. It is **include**d with the C programs that the user can modify and compile to remap her keyboard. See the Lexicon entries **nkb** and **keyboard tables** for more information.

nkb is also used with COHERENT system of virtual consoles. **kb.h** sets default definitions for function keys, as follows:

vt0 — **vt15**
Switch to logical session 0 through 15, respectively.

color0 — **color15**
Switch to color session 0 through 15, respectively.

mono0 — **mono15**
Switch to monochrome session 0 through 15, respectively.

vtn Switch to next higher-numbered open session.

vtp Switch to next lower-numbered open session.

vtt Toggle to most recently used open session

See Also

header files, virtual console, vtkb, vtnkb

kernel — Technical Information

Master program of the COHERENT system

The *kernel* is the master program of the COHERENT system. It manages the file systems, processes, devices, and users.

When you boot COHERENT on your system, the COHERENT bootstrap automatically loads and runs the program **/autoboot**. This file usually is linked to the kernel that you build when you installed COHERENT onto your computer.

Your system may have multiple kernels on it. For example, when you update COHERENT, often the old kernel is saved; and you can also build customized versions of the kernel. The COHERENT bootstrap lets you boot other versions of the kernel besides the one that is linked to file **/autoboot**. For details on how to do this, see the Lexicon article **booting**.

For information on the file system that the kernel supports, see the Lexicon entry for **file**.

The Lexicon entry **coff.h** describes the format of programs that the kernel can execute.

The COHERENT system comes with a set of system calls, which you can call from within user application to obtain kernel services. See the Lexicon entry **libc** summarizes the calls that the kernel offers.

The function **ulimit()** returns and sets some limits for the current process. For details, see its entry in the Lexicon.

The Lexicon article **device drivers** describes the suite of drivers that come with the COHERENT system. It gives the major and minor numbers of each, plus information on how to access and manipulate each driver.

Modifying the Kernel

Beginning with release 4.2, COHERENT contains a System V-style mechanism for modifying the kernel and building a new bootable kernel.

File **/etc/conf/mtune** defines the suite of "tunable variables" available within the kernel and its drivers. These variables define many of the kernel's default behaviors. For a complete list of these variables and notes on what each does, see **/etc/conf/mtune**.

File **/etc/conf/stune** sets the values of the variables actually used in your kernel. To modify the values of these variable, you can edit **stune** by hand, or you can use the commands **/etc/conf/bin/cohtune** and **/etc/conf/bin/idtune**. The former command lets you set or modify the values of variables used by device drivers; the latter command lets you set or modify variables used in the kernel itself.

File **/etc/conf/mdevice** names the drivers that are available for inclusion in your kernel. File **/etc/conf/sdevice** names the drivers that actually are included in your kernel. To include or exclude a driver, you can modify **sdevice** by hand; or you can use the command **/etc/conf/bin/idenable**.

Command **/etc/conf/bin/idmkcoh** builds a new bootable kernel that incorporates any changes you have made. For your changes to become effective, you must build a new kernel that incorporates your changes, and then boot it.

Finally, command **/etc/conf/bin/idbld** walks you the configuration of every device drivers in the kernel, then invokes **idmkcoh** to link a new kernel. In effect, this command lets you reconfigure the entire kernel.

Each of the above commands and files is described in its own Lexicon entry.

Two other files are of interest if you wish to modify the kernel.

- Header file **<sys/devices.h>** gives the major-device numbers for every driver in your kernel. It is read when drivers are compiled. If you are adding a new driver, you must add its name and major-device number to this header file.
- Normally, when you build a new kernel, the symbol table is stripped from it and kept in file */kernel_name***.sym**. The symbols in this file are used to decipher kernel tracebacks, and can be read by the debugger **db**. However, if you wish to hot-patch a kernel variable, that variable's symbol (or name) must be kept in the binary itself. File **/etc/conf/install_conf/keeplist** names the variables (or, more properly, the symbols) that are left in the binary after it is linked. You can then use the command **/conf/patch** to hot-patch these variables. We discourage you from doing this unless it is absolutely necessary.

Example

The Lexicon entry **device drivers** gives an example of how to add a new driver to the kernel. The following example walks you through the process of changing the size of the buffer cache on your system.

The buffer cache is a reserved portion of memory in which the kernel stores data recently read from the disk or to be written to the disk. When you invoke a command from your command line, the kernel checks its buffer cache. If you had invoked the command recently, the kernel should find it within the buffer cache; and it can then call up the command from memory rather than reading it from the disk. This speeds up your system noticeably.

Like everything else in life, the buffer cache involves a tradeoff: the larger the buffer cache, the faster your system will run, but the less memory will be available for running your programs. By default, COHERENT sets aside a portion of memory for the buffer cache; the more memory you have, the more is set aside for the cache. However, you can set the size of the cache by hand. Usually, this is done to limit the size of the cache, which is necessary if your system has limited amounts of memory and you want to run large user programs (e.g., the X Window System).

The following walks you through the process of modifying the kernel to reduce the size of the buffer cache.

1. Log in as the superuser **root**. **cd** to directory **/etc/conf**.
2. Edit file **/etc/conf/stune** and add the following lines:

```
NBUF_SPEC   100
NHASH_SPEC  97
```

   NBUF_SPEC sets the size of the buffer, in blocks. Here, we're setting it to 100 blocks (50 kilobytes), which is very small. **NASH_SPEC** sets the number of hash lists in the kernel; this must be the first prime number smaller than the number of blocks in the cache (in this case, 97). This, too, is very small.

3. Build a new kernel with the following command:

```
/etc/conf/bin/idmkcoh -o /cohtest
```

This builds a new kernel named **cohtest**, which incorporates your changes.

4. Shut down your system and boot the new kernel. For information on how to shut the system down, see the Lexicon entry for **shutdown**. For details on how to boot a kernel other than the default kernel, see the Lexicon entry for **booting**.

That's all there is to it. If you wish to make these variables patchable, so you can change them without going to the bother of building a new kernel, do the following:

1. In the file **/etc/conf/install_conf/keeplist**, change

```
echo '-I SHMMNI:SEMMNI:NMSQID'
```

to

```
echo '-I SHMMNI:SEMMNI:NMSQID:NBUF:NHASH'
```

2. Build a new kernel as described above.

Then, to change limit the size of the buffer cache to 50 kilobytes, use the command:

```
/conf/patch /testcoh NBUF=100 NHASH=97
```

Then, boot the patched kernel. As noted above, you should *not* use **/conf/patch** unless you absolutely must.

Files

/autoboot — The default kernel
/etc/conf — Directory that holds configuration files
/etc/conf/mdevices — Suite of available device drivers
/etc/conf/mtune — Suite of legal patchable variables
/etc/conf/sdevices — Drivers included in kernel
/etc/conf/stune — Patchable variables included in kernel
/etc/conf/install_conf/keeplist — Symbols kept in kernel

See Also

Administering COHERENT, booting, coff.h, COHERENT, device drivers, file, idmkcoh, libc, mtune, stune, ulimit()

Diagnostics

The kernel can produce the following error messages. Most are meaningful only to Mark Williams Company. If you encounter these errors, contact MWC and describe the circumstances during which you saw the error. MWC Support will try to solve this problem for you.

Arena *number* too small *(hardware)*

Bad block *number* (alloc) *(hardware)*
The kernel attempted to allocate a block of memory, only to find that there was something physically wrong with it.

Bad block *number* (free) *(hardware)*
The kernel attempted to free a block of memory, only to find that there was something physically wrong with it.

Bad free *number (hardware)*

Bad freelist *(halt)*
The *freelist* is a list of free blocks on the disk. The COHERENT system maintains this list so it can see where it can write data on the disk. This message indicates that the freelist has been corrupted somehow. To fix this problem, run **/etc/shutdown** to return to single-use mode; use **sync** to flush the buffers; use **umount** to unmount the affected file system; and then run **fsck** to repair the file system.

Bad segment count *(hardware)*
Bus error at *number (hardware)*
Cannot allocate stack *(hardware)*

Cannot create process *(hardware)*
Corrupt arena *(hardware)*
Illegal instruction at *number (hardware)*
Inode *number* busy *(hardware)*
Inode table overflow *(hardware)*
Not a separate I/D machine *(hardware)*

Out of i-nodes *(halt)*
A COHERENT file system has one i-node for each file it maintains. The number of i-nodes is set when the file system is created. If you have numerous small files on a file system, it is possible to exhaust that file system's resources even though the command **df** shows that space remains on the file system. To get around this problem, you must delete files, one file for each i-node needed; or you must use **ar** to archive a mass of files. To do this, first use **/etc/shutdown** to return the system to single-user mode, as described above. Delete files, or use **ar** as described above. Then use **sync** to flush all buffers, and use the command **umount** to unmount the affected file system. Then run **fsck** on the affected file system before rebooting. **fsck** checks COHERENT file systems and fixes them if necessary. Consult the Lexicon entry on **fsck** before you use this program for the first time.

Out of space (*m*,*n*) *(halt)*
When this error message appears, your file system still has i-nodes but the alloted disk space has been exhausted; perhaps you have a few large files that are eating up disk space. To get around this problem, you must delete or compress files to clear up disk space. First, use **/etc/shutdown** to return to single-user mode, as described above; then delete files or compress them as described above until enough space has been cleared to allow you to continue your work. Use **sync** to flush buffer, use **umount** to unmount the affected file system, and run **fsck** on the affected file system. Then reboot.

Random trap *(hardware)*
Raw I/O from non user *(hardware)*
System too large *(hardware)*

keyboard — Technical Information

How COHERENT handles the console keyboard

COHERENT comes with two device drivers for the keyboard, as follows:

vtkb	Non-configurable driver
vtnkb	Configurable driver

To change the keyboard driver you are using, or to modify the behavior of the driver **vtnkb**, log in as the superuser **root** and type:

```
cd /etc/conf
console/mkdev
```

The script **/etc/conf/console/mkdev** walks you through the process of configuring your console: you can switch keyboard drivers from load to non-loadable (or vice-versa), change the number of virtual consoles you use, or change the keyboard-translation table you use by default.

Once you have configured the console, type the following command:

```
bin/idmkcoh -o /kernel_name
```

where *kernel_name* is the what you wish to name the newly built kernel. This builds a new kernel that incorporates the changes you requested. To invoke these changes, simply reboot and invoke kernel *kernel_name* in the usual manner.

See Also

Administering COHERENT, vtkb, vtnkb

kill — Command

Signal a process
kill [- *signal* **]** *pid ...*

COHERENT assigns each active process a unique process id, or *pid*, and uses the pid to identify the process. **kill** sends *signal* to each *pid*. *signal* must be one of the numbers described in the header file **<signal.h>**. The signal can be given by number or by name, as **define**d in these header files. By default, *signal* is **SIGTERM**, which terminates a given process.

If *pid* is zero, **kill** signals each process in the same process group (that is, every process started by the same user from the same tty).

If *pid* is negative (but not -1), **kill** signals every process in the process group whose ID equals the absolute value of *pid*.

If *pid* is -1, **kill** signals each process that you own. If you are logged in as the superuser **root**, this signals every process except processes 0 (the kernel) and 1 (**init**).

The shell prints the process id of a process if the command is detached. The command **ps** prints a list of all active processes, with process ids and command-line arguments.

A user can kill only the processes he owns; the superuser, however, can kill anything. A process cannot ignore or catch **SIGKILL**.

See the Lexicon article for **signal()** for a table of the signals and what each means.

Files

<signal.h> — Signals

See Also

commands, getpid(), init, kill(), ksh, ps, sh, signal()

kill() — System Call (libc)

Kill a system process
#include <signal.h>
kill(*pid, sig***)**
int *pid*, *sig*;

kill() is the COHERENT system call that sends a signal to a process. *pid* is the process identifier of the process to be signalled, and *sig* identifies the signal to be sent, as set in the header file **signal.h**. This system call is most often used to kill processes, hence its name.

See the Lexicon article for **signal()** for a table of the signals and what each means. If *sig* is zero, **kill()** performs error checking, but sends no signal. You can use this feature to check the validity of *pid*.

Example

For an example of using this system call in a C program, see **signal()**.

See Also

libc, signal(), signal.h
POSIX Standard, §3.3.2

ksh — Command

The Korn shell
ksh *token ...*

The COHERENT system offers two command interpreters: **sh**, the Bourne shell; and **ksh**, the Korn shell. **sh** is the default COHERENT command interpreter. The shell tutorial included in this manual describes the Bourne shell in detail.

This article describes **ksh**, the Korn shell. **ksh** is a superset of the Bourne shell, and contains many features that you may well find useful. These include MicroEMACS-style editing of command lines; command hashing; a full-featured aliasing feature; and a job-control facility.

Invoking ksh

To invoke **ksh** from within the Bourne shell, simply type **ksh** at the command-line prompt. To use **ksh** as your default shell, instead of **sh**, append the command **/usr/bin/ksh** to the end of your entry in the file **/etc/passwd**. (See the Lexicon entry for **passwd** for more information on this file.)

You can invoke **ksh** with one or more built-in options; these are described below.

Commands

A *command* consists of one or more *tokens*. A *token* is a string of text characters (i.e., one or more alphabetic characters, punctuation marks, and numerals) delineated by spaces, tabs, or newlines.

A *simple command* consists of the command's name, followed by zero or more tokens that represent arguments to the command, names of files, or shell operators. A *complex command* will use shell constructs to execute one or more commands conditionally. In effect, a complex command is a mini-program that is written in the shell's programming language and interpreted by **ksh**.

Shell Operators

The shell includes a number of operators that form pipes, redirect input and output to commands, and let you define conditions under which commands are executed.

command **|** *command*
The *pipe* operator: let the output of one command serve as the input to a second. You can combine commands with '|' to form *pipelines*. A pipeline passes the standard output of the first (leftmost) command to the standard input of the second command. For example, in the pipeline

```
sort customers | uniq | more
```

ksh invokes **sort** to sort the contents of file **customers**. It pipes the output of **sort** to the command **uniq**, which outputs one unique copy of the text that is input into it. **ksh** then pipes the output of **uniq** to the command **more**, which displays it on your terminal one screenful at a time. Note that under COHERENT, unlike MS-DOS, pipes are executed concurrently: that is, **sort** does not have to finish its work before **uniq** and **more** can begin to receive input and get to work.

command **;** *command*
Execute commands on a command line sequentially. The command to the left of the ';' executes to completion; then the command to the right of it executes. For example, in the command line

```
a | b ; c | d
```

first execute the pipeline **a | b** then, when **a** and **b** complete, execute the pipeline **c | d**.

command **&**
Execute a command in the background. This operator must follow the command, not precede it. It prints the process identifier of the command on the standard output, so you can use the **kill** command to kill that process should something go wrong. This operator lets you execute more than one command simultaneously. For example, the command

```
/etc/fdformat -v /dev/fha0 &
```

formats a high-density, 5.25-inch floppy disk in drive 0 (that is, drive A); but while the disk is being formatted, **ksh** returns the command line prompt so you can immediately enter another command and begin to work. If you did not use the '&' in this command, you would have to wait until formatting was finished before you could enter another command.

ksh also prints a message on your terminal when a command that you are running in the background finishes processing. It does not check these "child" processes very often, however, so a command may have finished some time before **ksh** informs you of the fact. See the Lexicon article for the command **ps** for information on all processes; also see the description of the built-in command **jobs**, below.

command **&&** *command*
Execute a command upon success. **ksh** executes the command that follows the token '&&' only if the command that precedes it returns a zero exit status, which signifies success. For example, the command

```
cd /etc
fdformat -v /dev/fha0 && badscan -o proto /dev/fha0 2400
```

formats a floppy disk, as described above. If the format was successful, it then invokes the command **badscan** to scan the disk for bad blocks; if it was not successful, however, it does nothing.

command **||** *command*
Execute a command upon failure. This is identical to operator '&&', except that the second command is executed if the first returns a non-zero status, which signifies failure. For example, the command

```
/etc/fdformat -v /dev/fha0 || echo "Format failed!"
```

formats a floppy disk. If formatting failed, it echoes the message **Format failed!** on your terminal; however, if formatting succeeds, it does nothing.

Note that the tokens newline, ';' and '&' bind less tightly than '&&' and '||'. **ksh** parses command lines

from left to right if separators bind equally.

>*file* Redirect standard output. The *standard input*, *standard output*, and *standard error* streams are normally connected to the terminal. A pipeline attaches the output of one command to the input of another command. In addition, **ksh** includes a set of operators that redirect input and output into files rather than other commands.

The operator '>' redirects output into a file. For example, the command

```
sort customers >customers.sort
```

sorts file **customers** and writes the sorted output into file **customers.sort**. It creates **customers.sort** if it does not exist, and destroys its previous contents if it does exist.

>>*file* Redirect output into a file, and append. If the file does not exist, this operator creates it; however, if the file already exists, this operator appends the output to that file's contents rather than destroying those contents. For example, the command

```
sort customers.now | uniq >>customers.all
```

sorts file **customers.now**, pipes its output to command **uniq**, which throws away duplicate lines of input, and appends the results to file **customers.all**.

<*file* Redirect standard input. Here, **ksh** reads the contents of a file and processes them as if you had typed them from your keyboard. For example, the command

```
ed textfile <edit.script
```

invokes the line-editor **ed** to edit **textfile**; however, instead of reading editing commands from your keyboard, the shell passes **ed** the contents of **edit.script**. This command would let you prepare an editing script that you could execute repeatedly upon files rather than having to type the same commands over and over.

<< *token*
Prepare a "here document". This operator tells **ksh** to accept standard input from the shell input until it reads a line that contains only *token.* For example, the command

```
cat >FOO <<\!
      Here is some text.
!
```

redirects all text between '<<\!' and '!' to the **cat** command. The '>' in turn redirects the output of **cat** into file **FOO**. **ksh** performs parameter substitution on the here document unless the leading *token* is quoted; parameter substitution and quoting are described below.

command **2>** *file*
Redirect the standard error stream into a file. For example, the command

```
nroff -ms textfile >textfile.p 2>textfile.err
```

invokes the command **nroff** to format the contents of **textfile**. It redirects the output of **nroff** (i.e., the standard output) into **textfile.p**; it also redirects any error messages that **nroff** may generate into file **textfile.err**.

Note in passing that a command may use up to 20 streams. By default, stream 0 is the standard input; stream 1 is the standard output; and stream 2 is the standard error. **ksh** lets you redirect any of these streams individually into files, or combine streams into each other.

<&*n* **ksh** can redirect the standard input and output to duplicate other file descriptors. (See the Lexicon article **file descriptor** for details on what these are.) This operator duplicates the standard input from file descriptor *n*.

>&*n* Duplicate the standard output from file descriptor *n*. For example,

```
2>&1
```

redirects file descriptor 2 (the standard error) to file descriptor 1 (the standard output).

Note that each command executed as a foreground process inherits the file descriptors and signal traps (described below) of the invoking shell, modified by any specified redirection. Background processes take input from the null device **/dev/null** (unless redirected), and ignore interrupt and quit signals.

File-Name Patterns

The shell interprets an input token that contain any of the special characters '?', '*', or '[' as a file name *pattern*.

? Match any single character except newline. For example, the command

```
ls name?
```

will print the name of any file that consists of the string **name** plus any one character. If **name** is followed by no characters, or is followed by two or more characters, it will not be printed.

* Match a string of non-newline characters of any length (including zero).

```
ls name*
```

prints the name of any file that begins with the string **name**, regardless of whether it is followed by any other characters. Likewise, the command

```
ls name?*
```

prints the name of any file that consists of the string **name** followed by at least one character. Unlike **name***, the token **name?*** must be followed by at least one character before it will be printed.

~*name*
Replace the name of user *name* with his **$HOME** directory. For example, the command

```
ls -l ~norm/src
```

lists the contents of the *src* subdirectory located under the **$HOME** directory for user **norm**. This spares you from having to know where a given user's HOME directory is located.

The character '~' on its own is a synonym for the home directory of whoever is running the command. For example, the command

```
/usr/lib/uucp FOO mwcbbs:~
```

copies file **FOO** into directory **/usr/spool/uucppublic** on system **mwcbbs**. In this instance, '~' expands into **/usr/spool/uucppublic** because the command **uucico** invokes **setuid()** to change the ownership of the process to user **uucp**, whose home directory is **/usr/spool/uucppublic**.

[!*xyz***]** Exclude characters *xyz* from the string search. For example, the command

```
ls [!abc]*
```

prints all files in the current directory except those that begin with **a**, **b**, or **c**.

[*C-c***]** Enclose alternatives to match a single character. A hyphen '-' indicates a range of characters. For example, the command

```
ls name[ABC]
```

will print the names of files **nameA**, **nameB**, and **nameC** (assuming, of course, that those files exist in the current directory). The command

```
ls name[A-K]
```

prints the names of files **nameA** through **nameK** (again, assuming that they exist in the current directory).

When **ksh** reads a token that contains one of the above characters, it replaces the token in the command line with an alphabetized list of file names that match the pattern. If it finds no matches, it passes the token unchanged to the command. For example, when you enter the command

```
ls name[ABC]
```

ksh replaces the token **name[ABC]** with **nameA**, **nameB**, and **nameC** (again, if they exist in the current directory), so the command now reads:

```
ls nameA nameB nameC
```

It then passes this second, transformed version of the command line to the command **ls**.

Note that the slash '/' and leading period '.' must be matched explicitly in a pattern. The slash, of course, separates the elements of a path name; while a period at the begin of a file name usually (but not always) indicates that that file has special significance.

LEXICON

Pattern Matching in Prefixes and Suffices

Special constructs let you match patterns in the prefixes and suffices of a string:

{#*parameter***}**
This operator gives the number of characters in *parameter*. For example:

```
foo=BAZZ
echo ${#foo} -> 4
```

{*parameter***%***word***}**
This returns the shortest string in which the suffix of *parameter* matches *word*. For example, given that **xyzzy=usr/bin/cpio**, then the command

```
echo ${xyzzy%/*}
```

echoes the string **usr/bin**.

{*parameter***%%***word***}**
This returns the longest string in which the suffix of *parameter* matches *word*. For example, given that **xyzzy=usr/bin/cpio**, then the command

```
echo ${xyzzy%/*}
```

echoes the string **usr**.

{*parameter***#***word***}**
This returns the shortest string in which the prefix of *parameter* matches *word*. For example, given that **plugh=usr/bin/cpio**, the command

```
echo ${plugh#*/}
```

echoes **bin/cpio**.

{*parameter***##***word***}**
This returns the longest string in which the prefix of *parameter* matches *word*. For example, given that **plugh=usr/bin/cpio**, the command

```
echo ${plugh##*/}
```

echoes **cpio**.

The following shows how to use these expressions to implement the command **basename**:

```
basename () {
      set $(echo ${1##*/}) $2
      echo ${1%$2}
}
```

Quoting Text

From time to time, you will want to "turn off" the special meaning of characters. For example, you may wish to pass a token that contains a literal asterisk to a command; to do so, you need a way to tell **ksh** not to expand the token into a list of file names. Therefore, **ksh** includes the **quotation operators** '\', '"', and '"; these "turn off" (or *quote*) the special meaning of operators.

The backslash '\' quotes the following character. For example, the command

```
ls name\*
```

lists a file named **name***, and no other.

The shell ignores a backslash immediately followed by a newline, called a *concealed newline*. This lets you give more arguments to a command than will fit on one line. For example, the command

```
cc -o output file1.c file2.c file3.c \
      file4.c file5.c file19.c
```

invokes the C compiler **cc** to compile a set of C source files, the names of which extend over more than one line of input. You will find this to be extremely helpful, especially when you write scripts and **makefile**s, to help you write neat, easily read commands.

A pair of apostrophes ' ' prevents interpretation of any enclosed special characters. For example, the command

```
find . -name '*.c' -print
```

finds and prints the name of any C-source file in the current directory and any subdirectory. The command **find** interprets the '*' internally; therefore, you want to suppress the shell's expansion of that operator, which is accomplished by enclosing that token between apostrophes.

A pair of quotation marks "" has the same effect. Unlike apostrophes, however, **ksh** will perform parameter substitution and command-output substitution (described below) within quotation marks. Note that everything between quotation marks will be a single argument, even if there are spaces between the tokens. For example, the command

```
grep "x y" *.c
```

calls the string-search command **grep** to look for the string **x<space>y**.

Scripts

Shell commands can be stored in a file, or *script.* The command

```
ksh script [ parameter ... ]
```

executes the commands in *script* with a new subshell **ksh**. Each *parameter* is a value for a positional parameter, as described below.

If you have used the command **chmod** to make *script* executable, then it is executed under the Bourne shell **sh**, without requiring the **ksh** command. Because all executable scripts are executed by the Bourne shell by default, not the Korn shell, you should avoid constructions that are unique to the Korn shell.

To ensure that a script is executed by **ksh**, begin the script with the line:

```
#!/usr/bin/ksh
```

Parameters of the form '$*n*' represent command-line arguments within a script. *n* can range from zero through nine; **$0** always gives the name of the script. These parameters are also called *positional parameters*.

If no corresponding parameter is given on the command line, the shell substitutes the null string for that parameter. For example, if the script **format** contains the following line:

```
nroff -ms $1 >$1.out
```

then invoking **format** with the command line:

```
format mytext
```

invokes the command **nroff** to format the contents of **mytext**, and writes the output into file **mytext.out**. If, however, you invoke this command with the command line

```
format mytext yourtext
```

the script will format **mytext** but ignore **yourtext** altogether.

Reference **$*** represents all command-line arguments. If, for example, we change the contents of script **format** to read

```
nroff -ms $* >$1.out
```

then the command

```
format mytext yourtext
```

will invoke **nroff** to format the contents of **mytext** and **yourtext**, and write the output into file **mytext.out**.

Commands in a script can also be executed with the . (dot) command. It resembles the **ksh** command, but the current shell executes the script commands without creating a new subshell or a new environment; therefore, you cannot use command-line arguments.

Variables

Shell variables are names that can be assigned string values on a command line, in the form

name=value

The name must begin with a letter, and can contain letters, digits, and underscores '_'. Note that no white space can appear around the '=', or the assignment will not work.

LEXICON

In shell input, '$*name*' or '${*name*}' represents the value of the variable. For example:

```
TEXT=mytext
nroff -ms $TEXT >$TEXT.out
```

Here, **ksh** expands **$TEXT** before it executes the **nroff** command. This technique is very useful in large, complex scripts: by using variables, you can change the behavior of the script by editing one line, rather than having to edit numerous variables throughout the script.

Note that if an assignment precedes a command on the same command line, the effect of the assignment is local to that command; otherwise, the effect is permanent. For example,

```
kp=one testproc
```

assigns variable **kp** the value **one** only for the execution of the script **testproc**.

ksh sets the following variables by default:

The number of actual positional parameters given to the current command.

@ The list of positional parameters "$1 $2 ...".

* The list of positional parameters "$1" "$2" ... (the same as '$@' unless some parameters are quoted).

- Options set in the invocation of the shell or by the **set** command.

? The exit status returned by the last command.

! The process number of the last command invoked with '&'.

$ The process number of the current shell.

Environmental Variables

ksh references the following environmental variables:

ENV If this variable is set at start-up, after all **.profile** files have been executed, the expanded value is used as the shell's start-up file. It typically defines environmental variables and aliases.

FCEDIT
This sets the editor used by the command **fc**.

HOME Initial working directory; usually specified in the password file **/etc/passwd**.

IFS Delimiters for tokens; by default space, tab, and newline.

KSH_VERSION
The current version of the Korn shell that you are using.

MAIL **ksh** check the file this names, at intervals specified by environmental variable **MAILCHECK**. If the file specified by this variable is new since last checked, the shell prints "You have mail." on the your terminal. If the file has increased in size since the last check, **ksh** prints "You have new mail." on your terminal.

Note that by default, **ksh** does not check **MAIL** when you log in. If you want it to do so, add the following lines to file **/etc/.kshrc**:

```
# The following lines emulate the mail notification of the Bourne Shell.
if [ -s $MAIL ]
then
      echo "You have mail."
fi
```

MAILCHECK
Specifies the number of seconds between checking for new mail. If not specified, **MAILCHECK** defaults to 600 seconds (ten minutes).

PATH Colon-separated list of directories searched for commands.

PS1 First prompt string, usually '$'. Note that in this variable and **PS2**, **ksh** expands the symbol **!** into the current number of the command line. For example, the prompt **ksh !>** prints the prompt **ksh** *NN***>** with every command, where *NN* is the number of the current command. This is useful when you have enabled

the history feature, as described below.

To print a prompt that includes your local site name, include the variable **$PWD** (described below) in the definition of **PS1**. For example,

```
PS1='$PWD>'
```

prints the current directory as your prompt, just like MS-DOS does. To include your system's name, read the contents of file **/etc/uucpname**, as follows:

```
SITE=`cat /etc/uucpname`
PS1='$SITE!!$PWD>'
```

This form of the prompt is quite useful when you are working on networked machines and may not always be sure just what system you are working on. Note that two exclamation points are necessary; as noted above, **ksh** expands one '!' into the number of the current command.

Finally, to include the command number with site name and current directory, do the following:

```
SITE=`cat /etc/uucpname`
PS1='$SITE!!$PWD !>'
```

This will give you a very long prompt, but one with much information in it.

PS2 Second prompt string, usually '>'. **ksh** prints it when it expects more input, such as when an open quotation-mark has been typed but a close quotation-mark has not been typed, or within a shell construct.

PWD The present working directory, i.e., the directory within which you are now working.

SECONDS
The number of seconds since the current shell was started.

SHELL The full path name of the shell that you are now executing.

TERM The name of the type of terminal you are now using, as used by various programs for reading the file **/etc/termcap**.

TIMEZONE
The current timezone you are located in, as set in your **.profile**. This is an interesting and powerful variable; see its entry in the Lexicon for details.

USER The login-identifier of the user, i.e., you.

The following special forms substitute parameters conditionally:

${*name***-***token***}**
Substitite *name* if it is set; if it is not, substitute *token*.

${*name***=***token***}**
Substitute *name* if it is set; if it is not set, substitute *token* and set *name* to equal *token*.

${*name***+***token***}**
Substitute *token* if *name* is set.

${*name***?***token***}**
Substitute *name* if it is set; if it is not, print *token* and exit from the shell.

To unset an environmental variable, use the command **unset**.

Command Output Substitution

ksh can use the output of a command as shell input (as command arguments, for example) by enclosing the command in grave characters ` `. For example, to list the contents of the directories named in file **dirs**, use the command

```
ls -l `cat dirs`
```

Constructs

ksh lets you control the execution of programs through the following constructs. It recognizes a construct only if it occurs unquoted as the first token of a command. This implies that a separator must precede each reserved word in the following constructs; for example, newline or ';' must precede **do** in the **for** construct.

LEXICON

break *[n]*
Exit from **for**, **until**, or **while**. If *n* is given, exit from *n* levels.

case *token* **in [** *pattern [| pattern] ...***)** *sequence***;;] ... esac**
Check *token* against each *pattern*, and execute *sequence* associated with the first matching *pattern*.

continue *[n]*
Branch to the end of the *n*th enclosing **for**, **until**, or **while** construct.

for *name [* **in** *token ...]* **do** *sequence* **done**
Execute *sequence* once for each *token*. On each iteration, *name* takes the value of the next *token*. If the **in** clause is omitted, **$@** is assumed. For example, to list all files ending with **.c:**

```
for i in *.c
do
      cat $i
done
```

if *seq1* **then** *seq2 [* **elif** *seq3* **then** *seq4] ... [* **else** *seq5]* **fi**
Execute *seq1*. If the exit status is zero, execute *seq2*; if not, execute the optional *seq3* if given. If the exit status of *seq3* is zero, then execute *seq4*, and so on. If the exit status of all tested sequences is nonzero, execute *seq5*.

time *sequence*
Time how long it takes *sequence* to execute. When *sequence* has finished executing, the time is displayed on the standard output.

while *sequence1 [* **do** *sequence2]* **done**
Execute *sequence2* as long as the execution of *sequence1* results in an exit status of zero.

(*sequence***)**
Execute *sequence* within a subshell. This allows *sequence* to change the current directory, for example, and not affect the enclosing environment.

{*sequence*}
Braces simply enclose a *sequence*.

Built-in Commands

ksh executes most commands via the **fork** system call, which creates a new process. See the Lexicon articles on **fork()** and **exec** for details on these calls. **ksh** also has the following commands built into itself.

. *script* Read and execute commands from *script*. Positional parameters are not allowed. **ksh** searches the directories named in the environmental variable **PATH** to find the given *script*.

: *[token ...]*
A colon ':' indicates a "partial comment". **ksh** normally ignores all commands on a line that begins with a colon, except for redirection and such symbols as **$**, **{**, **?**, etc.

A complete comment: if **#** is the first character on a line, **ksh** ignores all text that follows on that line.

alias *[name***=***value ...]*
When called without arguments, **alias** prints all aliases and their values. When called with a *name* but no associated value, then it prints the value of *name*. When called with a *name* and *value* combination, it associated *value* with *name*.

For example, the command

```
alias logout='exit'
```

binds the token **logout** to the command **exit**: hereafter, whenever you type **logout**, it will be as if you typed the **exit** command.

Note that when you define an alias, you should be careful not to write one that is self-referring, or **ksh** will go into an infinite loop when it tries to expand the alias. For example, the definition:

```
# DO NOT DO THIS!
alias ls='ls -CF'
```

will send **ksh** into an infinite loop, as it tries infinitely to replace **ls** with **ls**. Rather, use the definition:

```
# THIS IS CORRECT
alias ls='/bin/ls -CF'
```

or

```
# THIS TOO IS CORRECT
alias ls='  ls -CF'
```

In the latter example, note the spaces between the first grave character and the **ls**.

ksh has a number of aliases set by default. See the section **Aliases**, below, for details.

bind *[-**m**] [key_sequence**=**binding_name ...]*
When called without arguments, list the current set of key bindings for MicroEMACS-style editing of command lines. When called with arguments, bind the *key_sequence* to *binding_name.*

For example, the command

```
bind '^[^H'=delete-word-backward
```

binds the editing command **delete-word-backward** to the key sequence **<esc><backspace>**. Note that the carat characters in this command are literally that, not the shell's representation of a literal **<esc>** or **<backspace>** character.

When called with the **-m** option, bind more than one *binding_name* to a given *key_sequence*. This lets you build keyboard macros, to perform complex editing tasks with one or two keystrokes.

For details, see the sections below on command-line editing.

builtin *command*
Execute *command* as a built-in command.

cd *dir* Change the working directory to *dir.* If no argument is given, change to the home directory as set by the environmental variable **HOME**. When invoked, it also changes the environmental variables **PWD** and **OLDPWD**.

Using a hyphen '-' as the argument causes **ksh** to change to the previous directory, i.e., the one indicated by shell variable **OLDPWD**. In effect, this swaps **OLDPWD** and **PWD**, thus allowing you to flop back and forth easily between two directories.

echo *token ...*
Echo *token* onto the standard output. **ksh** replaces the command **echo** with the alias **echo='print'**.

eval *[token ...]*
Evaluate each *token* and treat the result as shell input.

exec *[command]*
Execute *command* directly rather than as a subprocess. This terminates the current shell.

exit *[status]*
Set the exit status to *status*, if given, then terminate; otherwise, the previous status is used.

export *[name ...]*
ksh executes each command in an *environment*, which is essentially a set of shell variable names and corresponding string values. It inherits an environment when invoked, and normally it passes the same environment to each command it invokes. **export** specifies that the shell should pass the modified value of each given *name* to the environment of subsequent commands. When no *name* is given, **ksh** prints the name of each variable marked for export.

export *VARIABLE**=**value*
This form of the **export** command sets *VARIABLE* to *value*, and exports it. Thus, the command

```
export FOO=bar
```

is equivalent to the commands:

```
FOO=bar
export FOO
```

fc *[-**l**] [-**n**] [first [last]]*
Draw the previously executed commands *first* through *last* back for manipulation and possible execution. *first* and *last* can be referenced either by their history numbers, or by a string with which the command in question begins. Normally, the commands are pulled into an editor for manipulation before they are executed; the editor is defined by the environmental variable **FCEDIT** (default, **ed**). The commands in question are executed as soon as you exit from the editor. Option **-l** lists the command(s) on **stdout**, and so suppresses the editing feature. Option **-n** inhibits the default history numbers.

function *funcname* { *script* }
Define function **funcname** for the shell to execute. For example the following defines function **get_name** for the shell:

```
function get_name {
      echo -n Please enter your name ...
      read name
      return 0
}
```

When **ksh** encounters **get_name**, it runs the above-defined function, rather than trying to find **get_name** on the disk. Note that the return status can be any valid status and can be checked in the code that called **get_name** by reading the shell variable **$?** (described above), or by using the function as the argument to an **if** statement. This allows you to build constructs like the following:

```
if get_name; then
      do_something
else
      do_something_else
fi
```

To list all defined functions, type the alias **functions**. To receive detailed information on a defined function, use the command **type** *name* where *name* is the name of the function in which you are interested.

getopts *optstring name [arg ...]*
Parse the *arg*s to *command*. See the Lexicon entry for **getopts** for details.

hash *[-**r**] [name ...]*
When called without arguments, **hash** lists the path names of all hashed commands. When called with *name* **hash** check to see if it is an executable command, and if so adds it to the shell's hash list. The **-r** option removes *name* from the hash list.

kill *[-**l**] [signal] process ...*
Send *signal* to *process*. The default signal is **TERM**, which terminates the process. *signal* may either be a number or a mnemonic as **#define**d in header file **<signal.h>**. When called with the **-l** option, it lists all known types of signals. See the Lexicon entry for **kill** for details.

let *[expression]*
Evaluate each *expression*. This command returns zero if *expression* evaluates to non-zero (i.e., fails), and returns non-zero if it evalutes to zero (i.e., succeeds). This is useful for evaluating expressions before actually executing them.

print *[-**nreu***n**] [argument ...]*
Print each *argument* on the standard output, separated by spaces and terminated with a newline. Option **-n** suppresses printing of the newline. Option **-u***n* redirects output from the standard output to file descriptor *n*.

Note that each *argument* can contain the following standard C escape characters: **\b**, **\f**, **\n**, **\r**, **\v**, and **\###**. See the Lexicon article on **C Language** for details each character's meaning. The option **-r** inhibits this feature, and the **-e** option re-enables it.

read *name ...*
Read a line from the standard input and assign each token of the input to the corresponding shell variable *name*. If the input contains fewer tokens than the *name* list, assign the null string to extra variables. If the input contains more tokens, assign the last *name* the remainder of the input.

readonly *[name ...]*
Mark each shell variable *name* as a read-only variable. Subsequent assignments to read-only variables will not be permitted. With no arguments, print the name and value of each read-only variable.

return *[status]*
Return *status* to the parent process.

set *[-***aefhkmnuvx** *[-***o** *keyword] [name ...]]*
Set listed flag. The **-o** option sets *keyword*, where *keyword* is a shell option.

When used with one or more *names*, this command sets shell variables *name* to values of positional parameters beginning with **$1**.

For example, the command

```
set -h -o emacs ignoreeof
```

performs the following: turns on hashing for all commands, turns on MicroEMACS-style command-line editing, and turns off exiting upon EOF (that is, you must type **exit** to exit from the shell). **set** commands are especially useful when embedded in your **.profile**, where they can customize **ksh** to your preferences.

For details of this command, see its Lexicon entry.

shift Rename positional parameter **1** to current value of **$2,** and so on.

test *[option] [expression]*
Check *expression* for condition *option.* This is a useful and complex command, with more options than can be listed here. See its Lexicon entry for details.

times Print on the standard output a summary of processing time used by the current shell and all of its child processes.

trap *[command] [n ...]*
Execute *command* if **ksh** receives signal *n.* If *command* is omitted, reset traps to original values. To ignore a signal, pass null string as *command.* With *n* zero, execute *command* when the shell exits. With no arguments, print the current trap settings.

typeset *[-***firx***] [+***firx***] [name [=value] ...]*
When called without an argument, this command lists all variables and their attributes.

When called with an option but without a *name*, it lists all variables that have the specified attribute; **-** tells **typeset** to list the value of each variable and **+** tells it not to.

When called with one or more *names,* it gives *name* to the listed attribute. If *name* is associated with a *value,* **typeset** also assigns the *value* to it.

typeset recognizes the following attributes:

- **-i** Store variable's value as an integer
- **-f** List function instead of variable
- **-r** Make the variable read-only
- **-x** Export variable to the environment

umask *[nnn]*
Set user file creation mask to *nnn.* If no argument is given, print the current file creation mask.

unalias *name ...*
Remove the alias for each *name*.

wait *[pid]*
Hold execution of further commands until process *pid* terminates. If *pid* is omitted, wait for all child processes. If no children are active, this command finishes immediately.

whence *[-***v***] name ...*
List the type of command for each *name.* When called with the **-v** option, also list functions and aliases.

Aliases

ksh implements as aliases a number of commands that **sh** calls as separate executable programs. The **echo** alias, for instance, does everything that **/bin/echo** does, but **ksh** does not have to **fork()** and **exec()** simply to echo a token. Other aliases, like **pwd**, work by printing the contents of shell variables. The command **/bin/pwd** still works should you prefer it, but you must request it by its full path name should you not wish to use the much faster alias version.

LEXICON

ksh sets the following aliases by default. If you wish, you can use the built-in command **unalias** to make one or all of them go away.

```
echo=print
false=let
functions=typeset -f
history=fc -l
integer=typeset -i
login=exec login
newgrp=exec newgrp
pwd=print -r $PWD
true=:
type=whence -v
```

The alias **history** is especially useful when you are using the Korn shell's history feature. When invoked with no argument, it prints the last 13 commands you typed. When invoked with one numeric argument, it lists the command that corresponds to that argument; for example

```
history 106
```

prints the 106th command you entered (assuming that you've entered that many). When used with two numeric arguments, it prints the range of commands between the two arguments; for example

```
history 10 99
```

prints the tenth through the 99th commands you entered.

Job Control

ksh lets you manipulate and monitor background jobs via its *job control* commands.

The following commands manipulate background jobs:

jobs Display information about all controlled jobs. Information is in the following format:

> **%***num* **[+-]** *pid status command*

where *num* indicates the job number, '+' indicates the current job, '-' indicates the previous job, *pid* is the job's process identifier, *status* shows the status of the job (e.g., Running, Done, Killed), and *command* is the command description. Note that **ksh** only checks for changes in job status when waiting for a command to complete.

kill *[-signal] pid ...*
Described above.

wait *[pid]*
Hold execution of further commands until process *pid* terminates. See its Lexicon entry for details.

The following '%' syntax can be used with the above commands:

%+ Select the current job.

%- Select the previous job.

%*num* Select the job with job number *num*.

%*string* Select the most recently invoked job whose command begins with *string*.

%?*string*
Select the most recently invoked job whose command contains *string*.

vi-Style Command-line Editing

ksh, has built into it an editing feature that lets you recall and edit commands using **vi**-style editing commands. When you have finished editing, simply typing `(¢)` dispatches the command for re-execution.

To turn on **vi**-style editing, use the command

```
set -o vi
```

The following table gives each input-mode command:

**** Escape the next erase or kill character.

<ctrl-D> This character (EOF) terminates **ksh** if the current line is empty. Note that the command

```
alias logout='exit'
```

neutralizes this effect of EOF.

<ctrl-H> Delete previous character — that is, the character to the left.

<ctrl-V> Quote the next character. You can use this to embed editing and kill characters within a command.

<ctrl-W> Delete the previous word. A "word" is any clump of text delineated by white space.

<ctrl-J>
<ctrl-M>
(↵) Execute this line.

The following table gives each editing-mode command:

[count] **k**
[count] **-** Get previous command from the history buffer.

[count] **j**
[count] **+** Get next command from the history buffer.

[count] **G** Get command *count* from the history buffer. Default is the least recently entered command.

/*string* Search the history buffer for the most recently entered command that contains *string*. If *string* is NULL, use the previous string. *string*. must be terminated by **<ctrl-M>** or **<ctrl-J>**.

?*string* Same as **/**, except that **ksh** seeks the least recently entered command.

n Repeat the previous search.

N Repeat the last search, but in the opposite direction.

[count] **l** Move right *count* characters (default, one).

[count] **w** Move forward *count* alphanumeric words (default, one).

[count] **W** Move forward *count* blank-separated words (default, one).

[count] **e** Move forward to the end of the *count*'th word.

[count] **E** Move forward to the end of the *count*'th blank-separated word.

[count] **h** Move left *count* characters (default, one).

[count] **b** Move back *count* words.

[count] **B** Move back *count* blank-separated words.

0 Move cursor to start of line.

^ Move cursor to start of line.

$ Move cursor to end of line.

[count] **f** *c* Move rightward to the *count*'th occurrence of character *c*.

[count] **B** *c* Move leftward to the *count*'th occurrence of character *c*.

[count] **t** *c* Move rightward *almost* to the *count*'th occurrence of character *c* (default, one). Same as **f***c* followed by **h**.

[count] **T** *c* Move leftward *almost* to the *count*'th occurrence of character *c* (default, one). Same as **F***c* followed by **l**.

; Repeats the last **f**, **F**, **t**, or **T** command.

LEXICON

, Reverse of **;**.

a Enter input mode and enter text after the current character.

A Append text to the end of the line; same as **$a**.

*[count]***c***c*
c*[count]c* Delete current character through character *c* and then execute input line.

s Same as **cc**.

*[count]***d***motion*
d*[count]c* Delete current character through the character *c*.

D Delete current character through the end of line. Same as **d$**.

i Enter input mode and insert text before the current character.

I Enter input mode and insert text before the first word on the line.

*[count]***P** Place the previous text modification before the cursor.

*[count]***p** Place the previous text modification after the cursor.

R Enter input mode and overwrite characters on the line.

r*c* Replace the current character with character *c*.

*[count]***x** Delete the current character.

*[count]***X** Delete the preceding character.

*[count]***.** Repeat the previous text modification command

~ Invert the case of the current character and advance the cursor.

*[count]_* Append the *count*'th word from the previous command and enter input mode (default, last word).

* Attempt file-name generate on the current word. If a match is found, replace the current word with the match and enter input mode.

u Undo the last text-modification command.

U Restore the current line to its original state.

*[count]***v** Execute command

```
fc -e ${VISUAL:-${EDITOR:-vi}}
```

<ctrl-L> Line feed and print the current line.

<ctrl-J>
<ctrl-M>
(¢) Execute the current line.

Same as **I#<return>**.

Command Completion

ksh supports *command completion*. This feature permits you to invoke a command by typing only a fraction of it; **ksh** fleshes out the command, based on what commands you have already entered.

To invoke command completion, set the following in **.profile** or **.kshrc**:

```
set -h -o emacs
```

or:

```
set -h -o vi
```

This turns on hashing and tracking. It also turn on command-line editing: the former command turns on MicroEMACS-style editing, whereas the latter turn on **vi**-style editing.

As an example, say that you type the following commands:

```
compress foo.tar
ps alx
df -t
```

With MicroEMACS-style editing, if you type **<ctrl-X>?**, you then see the commands you typed in alphabetical order:

```
compress     df     ps
```

If you want to re-invoke the **compress** command without having to type all of it, you can use either type **<ctrl-R>** followed by 'c' to use the shell's reverse-search capabilities; or you can type 'c' followed by **<esc><esc>** to have the shell's command-completion facility complete the command.

If you use the reverse-incremental search, you get the entire command line as you had typed it. Additional uses of **<ctrl-R>** while already in search mode tell **ksh** to search further back in its history list of commands.

If, however, you use the command completion, you get only the command. So, to continue the example, if you type the letter 'c' followed by **<esc><esc> ksh** displays the word **compress**, followed by a **<space>**, and awaits more input.

In general, the reverse-search is better if you wish to re-execute an entire command; but command completion is better if you want just the command name.

Under **vi**-style editing, you can also use command completion. To complete a command, type '*' while in edit mode; or type **<esc>*** while in input mode.

File-Name Completion

ksh also lets you "complete" file names and directory names, just like you complete command names. With MicroEMACS-style editing, the file-completion command is **<esc><esc>**; with **vi**-style editing, the file-completion command is '*' (in edit mode) or **<esc>*** (in input mode).

If you are entering a file name and have specified enough of the name in order to specify a unique file, typing the file-completion command completes the file name or directory name. If you have not typed enough, **ksh** remains silent; type more characters of the file name, then again try the file-completion command. If you enter a bogus file name or directory name, **ksh** beeps to indicate that it cannot complete the given name. When **ksh** completes a file name, it then prints a space character. This indicates that the string names a file (rather than a directory); the space character lets you begin immediately to type the next argument. When **ksh** completes a directory name, it appends a slash ('/') instead of a space character, and waits for you to type the next part of the path name.

For example, if you type

```
ls -l /usr/spool/uucp
```

followed by **<esc><esc>**, nothing happens because of the ambiguity between directory names **/usr/spool/uucp/** and **/usr/spool/uucppublic/**.

If you then type the letter 'p', the command now appears:

```
ls -l /usr/spool/uucpp
```

Typing **<esc><esc>** now expands it out to

```
ls -l /usr/spool/uucppublic/
```

which is the name you desire. Note that **ksh** appends the trailing slash and waits for more.

A file-name completion example is:

```
more /usr/lib/uucp/P
```

followed by **<esc><esc>**; this yields:

```
more /usr/lib/uucp/Permissions
```

which saves you eight keystrokes.

.profile and .kshrc

When a user of the Korn shell logs into COHERENT, **ksh** first executes the script **/etc/profile**. This sets up default environmental variables for every user on the system, such as the default **PATH** and the default **TERM** variables.

Next, **ksh** executes the script **.profile** in the user's home directory. You can customize this file to suit your preferences. For example, you can set up a customized **PATH**, define aliases, and have the shell execute some

programs automatically (such as **calendar** or **fortune**).

Finally, **ksh** executes the script named in the environmental variable **ENV** whenever you invoke a shell. By custom, this script is named **.kshrc** and is kept in your home directory, but you name it anything you wish. This file should define how you want the shell itself to function. In this way, you can ensure that your settings will be available to all subshells, as well as to your login shell. If you wish to hide these settings from subshells, just conclude your **.kshrc** with the command:

```
unset ENV
```

For more information, see the Lexicon articles **profile**, **.profile**, and **.kshrc**.

Example

The following C code creates a program called **splurt.c**. It demonstrates numbered redirection of **ksh**, by writing to five streams without opening them. Compile it with the command:

```
cc -o splurt splurt.c
```

To call it from the command line, you could type a command of the form:

```
splurt      3> splurt3 4> splurt4 5> splurt5 6> splurt6 7> splurt7
```

This will redirect the **splurt**'s output into files **splurt3** through **splurt7**.

```
#include <stdio.h>
main()
{
      int i;
      char buf[50];

      for(i = 3; i < 8; i++) {
            sprintf(buf, "For fd %d\n", i);
            write(i, buf, strlen(buf));
      }
}
```

Files

/etc/profile — System-wide initial commands
$HOME/.kshrc — Set up user-specific environment
$HOME/.profile — User-specific initial commands
/dev/null — For background input

See Also

bind, commands, dup(), environ, exec, fork(), getopts, jobs, kill, .kshrc, login, newgrp, profile, set, sh, signal(), test, Using COHERENT, vsh, wait

For a list of commands associated with **ksh**, see the **Shell Commands** section of the **Commands** Lexicon article.

Introduction to sh, the Bourne Shell, tutorial

Notes

Note that the queue of previously issued commands is stored in memory, not on disk.

This version of **ksh** does not support variable arrays.

The Mark Williams version of **ksh** is based on the public-domain version of the Korn shell, which in turn is based on the public-domain version of the seventh edition Bourne shell written by Charles Forsyth and modified by Eric Gisin, Ron Natalie, Arnold Robbins, Doug Gwyn, and Erik Baalbergen.

KSH_VERSION — Environmental Variable

List current version of Korn shell

The Korn shell stores its current version in environmental variable **KSH_VERSION**.

See Also

environmental variables, ksh

.kshrc — System Administration

Set personal environment for Korn shell

Whenever you invoke the Korn shell **ksh**, it executes the script named in the environmental variable **ENV**. By custom, this is usually the file **${HOME}/.kshrc**.

To ensure that **.kshrc** is executed whenever you log in, insert the line

```
export ENV=${HOME}/.kshrc
```

into your **.profile**.

.kshrc should include all items that you wish to have known to all of the shells that you invoke — both the login shell and all subshells. These should include aliases, environmental variables, and the **set** commands that you use to fine-tune the operation of the shell. If you wish to define items in your login shell but hide them from subshells, simply place them in your **.profile** instead of your **.kshrc**. For example, the command

```
set -o emacs
```

turns on MicroEMACS-style command-line editing for all of your subshells when you insert it into your **.kshrc**, but turns it on only for your login shell if you insert it only into your **.profile**.

The following gives a sample **.kshrc**:

```
# Set the main prompt (PS1) to be the machine (i.e., site) name, the
# tty name (i.e., session name) and the current directory.  The
# second-level prompt (PS2) used for multi-line commands is much simpler.
SITE=‘cat /etc/uucpname‘
TTY=‘tty | sed s/^.....///‘
PS1=’$SITE $TTY $PWD: ’
PS2=’MORE> ’

# Turn on hashing, tracking, and filename completion (-h), EMACS-like
# command-line editing, and ignore end-of-file (<ctrl-D>) as a way to
# log out.
set -h -o emacs ignoreeof

#
# Add the word "logout" as an alias for "exit".
#
alias logout=’exit’

# Add EMACS command line editing command "delete-word-backword" bound
# to the key sequence <Esc><Backspace>. Note that there are four
# characters inside the apostrophes; the shell interprets a ^
# followed by a character as meaning <Ctrl> character.
bind ’^[^H’=delete-word-backward

#  Select MicroEMACS as the default editor to use with "fc" commands
FCEDIT=emacs
```

See Also

Administering COHERENT, ENV, ksh, profile, .profile, Using COHERENT

ktty.h — Header File

Kernel portion of tty structure
#include <sys/ktty.h>

The header file **ktty.h** defines the kernel’s portion of the teletypewriter (tty) structure. It also defines a set of test macros that can be used to test for specific conditions.

See Also

header files