va_arg() — Variable Arguments

Return pointer to next argument in argument list
#include <stdarg.h>
typename ***va_arg(***listptr**,** *typename***)**
va_list *listptr***,** *typename***;**

#include <varargs.h>
typename ***va_arg(***listptr**,** *typename***)**
va_list *listptr***,** *typename***;**

va_arg() returns a pointer to the next argument in an argument list. It can be used with functions that take a variable number of arguments, such as **printf()** or **scanf()**, to help write such functions portably. It is always used with **va_end()** and **va_start()** within a function that takes a variable number of arguments.

listptr is of type **va_list**, which is defined in the headers **<stdarg.h>** and **<varargs.h>**. This object must first be initialized by the macro **va_start()**.

typename is the name of the type for which **va_arg()** is to return a pointer. For example, if you wish **va_arg()** to return a pointer to an integer, *typename* should be of type **int**.

va_arg() can only handle "standard" data types, i.e., those data types that can be transformed to pointers by appending an asterisk '*'.

Example

For an example of this macro, see the entry for **stdarg.h**.

See Also

stdarg.h, varargs.h
ANSI Standard, §7.8.1.2

Notes

There are two different versions of **va_arg()**: the ANSI version, which is defined in **<stdarg.h>**; and the UNIX version, which is defined in **<varargs.h>**. For a discussion of how these implementations differ, see the entry for **stdarg.h**.

If there is no next argument for **va_arg()** to handle, or if *typename* is incorrect, then the behavior of **va_arg()** is undefined.

The ANSI Standard demands that **va_arg()** be implemented only as a macro. If its macro definition is suppressed within a program, its behavior is undefined.

va_end() — Variable Arguments

Tidy up after traversal of argument list
#include <stdarg.h>
void va_end(*listptr***)**
va_list *listptr***;**

#include <varargs.h>
void va_end(*listptr***)**
va_list *listptr***;**

va_end() helps to tidy up a function after it has traversed the argument list for a function that takes a variable number of arguments. It can be used with functions that take a variable number of arguments, such as **printf()** or **scanf()**, to help write such functions portably. It should be used with the routines **va_arg()** and **va_start()** from within a function that takes a variable number of arguments.

listptr is of type **va_list()**, which is declared in header **stdarg.h**. *listptr* must first have been initialized by macro **va_start**.

Example

For an example of this function, see the entry for **stdarg.h**.

See Also

stdarg.h, varargs.h
ANSI Standard, §7.8.1.3

Notes

There are two different versions of **va_end()**: the ANSI version, which is defined in **<stdarg.h>**; and the UNIX version, which is defined in **<varargs.h>**. For a discussion of how these implementations differ, see the entry for **stdarg.h**.

If **va_list()** is not initialized by **va_start()**, or if **va_end()** is not called before a function with variable arguments exits, then the behavior of **va_end()** is undefined.

va_start() — Variable Arguments

Point to beginning of argument list
#include <varargs.h>
void va_start(*listptr***)**
va_list *listptr***;**

#include <stdarg.h>
void va_start(*listptr, rightparm***)**
va_list *listptr, type rightparm***;**

va_start() is a macro that points to the beginning of a list of arguments. It can be used with functions that take a variable number of arguments, such as **printf()** or **scanf()**, to help implement them portably. It is always used with **va_arg()** and **va_end()** from within a function that takes a variable number of arguments.

This macro is defined in two different header files, **<stdarg.h>** and **<varargs.h>**. The former header file is the creation of the ANSI C committee, whereas the latter originates with UNIX System V. In both implementations, the first argument is *listptr*, which is of type **va_list**.

The implementation in **<stdarg.h>** (ANSI) adds a second argument, *rightparm*, which is the rightmost parameter preceding the variable arguments in the function's parameter list. Undefined behavior results if any of the following conditions apply to *rightparm*: if it has storage class **register**; if it has a function type or an array type; or if its type is not compatible with the type that results from the default argument promotions.

Example

For an example of this macro, see the entry for **stdarg.h**.

See Also

stdarg.h, varargs.h
ANSI Standard, §7.8.1.1

Notes

For a discussion of how the **<stdarg.h>** and **<varargs.h>** implementations of the variable-argument routines differ, **stdarg.h**.

The ANSI Standard demands that **va_start()** be implemented only as a macro. If the macro definition of **va_start()** is suppressed within a program, the behavior is undefined (and probably unwelcome).

varargs.h — Header File

Declare/define routines for variable arguments
#include <varargs.h>

The header file **<varargs.h>** prototypes and defines the routines used to manage variable arguments. These routines are modelled after those used in UNIX System V. The routines in **varargs.h** were designed to give a C compiler a semi-rational way of dealing with functions (e.g., **printf()**) that can take a variable number of arguments. In brief, these routines consist of the variable-list **typedef va_list**, the parameter declaration **va_dcl**, and the three macros **va_start()**, **va_arg()**, and **va_end()**. The macros respectively start the argument list, fetch the

next member, and end the argument list.

See Also

header files, stdarg.h

Notes

These routines are also implemented in the header file **<stdarg.h>**, which is described in the ANSI Standard. For details on how these implementations differ, see the entry for **stdarg.h**.

vfprintf() — STDIO Function (libc)

Print formatted text into stream
#include <stdarg.h>
#include <stdio.h>
int
vfprintf(*fp***,** *format***,** *arguments***)**
FILE**fp***; char** **format***; va_list** *arguments***;**

vfprintf() constructs a formatted string and writes it into the stream pointed to by *fp*. It translates integers, floating-point numbers, and strings into a variety of text formats. **vfprintf()** can handle a variable list of arguments of various types. It is roughly equivalent to **fprintf()**'s conversion specifier **%r**.

format points to a string that can contain text, character constants, and one or more *conversion specifications*. A conversion specification describes how to convert a particular data type into text. Each conversion specification is introduced with the percent sign '%'. (To print a literal percent sign, use the escape sequence '%%'.) See **printf()** for further discussion of the conversion specification, and for a table of the type specifiers that can be used with **vfprintf()**.

After *format* comes *arguments*. This is of type **va_list**, which is defined in the header file **stdarg.h**. It has been initialized by the macro **va_start()** and points to the base of the list of arguments used by **vfprintf()**. For more information, see the Lexicon entry for **va_arg()**.

arguments should access one argument for each conversion specification in *format*, of the type appropriate to its conversion specification. For example, if *format* contains conversion specifications for an **int**, a **long**, and a string, then *arguments* access three arguments, being, respectively, an **int**, a **long**, and a **char ***. *arguments* can take only the data types acceptable to the macro **va_arg()**, namely, the basic types that can be converted to pointers simply by adding a '*' after the type name. See **va_arg()** for more information on this point.

If there are fewer arguments than conversion specifications, then **vfprintf()**'s behavior is undefined (and probably unwelcome). If there are more, **vfprintf()** evaluates and then ignores every argument without a corresponding conversion specification. If an argument is not of the same type as its corresponding conversion specifier, then the behavior of **vfprintf()** is undefined. Thus, presenting an **int** where **vfprintf()** expects a **char *** may generate unwelcome results.

If it wrote the formatted string correctly, **vfprintf()** returns the number of characters written. Otherwise, it returns a negative number.

See Also

fprintf(), libc, printf(), sprintf(), vprintf(), vsprintf()
ANSI Standard, §7.9.6.7

Notes

vfprintf() can construct a string up to at least 509 characters long.

vi — Command

Clone of Berkeley-style screen editor
vi [*options* **] [+***cmd* **] [** *file1 ... file27* **]**

vi is a link to the editor **elvis**, which is a clone of the UNIX editors **ex** and **vi**. For details on how to run **vi**, see the entry for **elvis** in the Lexicon.

See Also

commands, ed, ex, elvis, me, view

Notes

vidattr() — terminfo Function

Set the terminal's video attributes
#include <curses.h>
vidattr(*newmode***)**
int *newmode***;**

COHERENT comes with a set of functions that let you use **terminfo** descriptions to manipulate a terminal. **vidattr()** sends one or more video attributes to the terminal opened by a call to **setupterm()**. *newmode* is any combination of the macros **A_STANDOUT**, **A_UNDERLINE**, **A_REVERSE**, **A_BLINK**, **A_DIM**, **A_BOLD**, **A_INVIS**, **A_PROTECT**, and **A_ALTCHARSET**, OR'd together. Their names are self-explanatory; all are defined in the header file **curses.h**.

See Also

curses.h, setupterm(), terminfo, vidputs()

vidputs() — terminfo Function

Write video attributes into a function
#include <curses.h>
vidputs(*newmode***,***outc***)**
int *newmode***;**
int (*outc***)();**

COHERENT comes with a set of functions that let you use **terminfo** descriptions to manipulate a terminal. **vidputs()** resets the video attributes of the terminal that had been opened by a call to **setupterm()**.

newmode is any combination of the macros **A_STANDOUT**, **A_UNDERLINE**, **A_REVERSE**, **A_BLINK**, **A_DIM**, **A_BOLD**, **A_INVIS**, **A_PROTECT**, and **A_ALTCHARSET**, OR'd together. Their names are self-explanatory; all are defined in the header file **curses.h**.

outc points to a function that takes a single character as an argument, e.g., **putchar()**.

The related function **vidattr()** resets video attributes without requiring a pointer to a function.

See Also

curses.h, setupterm(), terminfo, vidattr()

view — Command

Screen-oriented viewing utility
view *file1 ... file27*

view is a link to **elvis**, which is a clone of the UNIX **vi/ex** set of editors. Invoking **elvis** through this link forces it to operate solely in read-only mode, just as the UNIX **view** utility operates.

For information on how to use this version of **view**, see the Lexicon page for **elvis**.

See Also

commands, ed, elvis, ex, me, vi

Notes

virtual console — Technical Information

COHERENT system of multiple virtual consoles

The "virtual consoles" feature of COHERENT allows you to run multiple sessions from the system console. You can switch between sessions on the console screen using the appropriate keystrokes. If your computer has both monochrome and color video adapters and monitors, you can run multiple sessions on both screens simultaneously.

For this feature to be available, your system must be configured for virtual consoles. Normally, this configuration is done during installation. In addition, virtual console sessions must be *enabled* for logins prior to use. Virtual terminals are most useful when your system is running in multiuser mode.

COHERENT allows up to ten sessions at a given time. All you need to do to access multiple sessions is to hold down the **<Ctrl>** key on the system keyboard and press the digit on the numeric keypad corresponding to the desired active session number. Simultaneously pressing keys **<Ctrl>** and **<.>** (located on the numeric keypad) will take you to the next *open* virtual terminal session. Another means of switching sessions is to hold down the **<Alt>** key and press one of the "function keys". By default, function key **<F10>** takes you to the next *open* virtual terminal session, **<F11>** takes you to the previous *open* virtual terminal session, and **<F12>** toggles between the current and previously selected sessions.

Technical Features

It is not essential to know the following in order to use virtual terminals. We provide this information for advanced users, as well as persons wishing to customize their systems in ways not available under the default scheme used by the COHERENT installation procedure.

Different sessions are accessed by using different device names in directory **/dev.** Like any *character special* device, each virtual terminal screen has a *major* and *minor* number associated with it. The major number for all virtual terminal screens is 2. The device with minor number 0 is initially the console device — this is where output appears during startup and at other times when the system is in single-user mode. Virtual terminals are assigned successive minor numbers. When there are both color and monochrome display adapters on the system, the color sessions are given the lower minor numbers. For example, in a system configured for four color and four monochrome sessions, *logical* devices might be numbered as follows:

```
crwxr-xr-x 1 root   root     2   0 Mon Jun 15 14:51 /dev/console
crwxr-xr-x 1 root   root     2   1 Mon Jun 15 14:51 /dev/vcolor0
crwxr-xr-x 1 root   root     2   2 Mon Jun 15 14:51 /dev/vcolor1
crwxr-xr-x 1 root   root     2   3 Mon Jun 15 14:51 /dev/vcolor2
crwxr-xr-x 1 root   root     2   4 Mon Jun 15 14:51 /dev/vcolor3
crwxr-xr-x 1 root   root     2   5 Mon Jun 15 14:50 /dev/vmono0
crwxr-xr-x 1 root   root     2   6 Mon Jun 15 14:50 /dev/vmono1
crwxr-xr-x 1 root   root     2   7 Mon Jun 15 14:50 /dev/vmono2
crwxr-xr-x 1 root   root     2   8 Mon Jun 15 14:50 /dev/vmono3
```

Alternatively, using *physical* device numbering, successive color-only sessions can be accessed by using minor numbers 64-79, while successive monochrome-only sessions are selected with minor numbers 80-95. The configuration of four color plus four monochrome sessions described above could also be represented as:

```
crwxr-xr-x 1 root   root     2  64 Mon Jun 15 14:51 /dev/color0
crwxr-xr-x 1 root   root     2  65 Mon Jun 15 14:51 /dev/color1
crwxr-xr-x 1 root   root     2  66 Mon Jun 15 14:51 /dev/color2
crwxr-xr-x 1 root   root     2  67 Mon Jun 15 14:51 /dev/color3
crwxr-xr-x 1 root   root     2  80 Mon Jun 15 14:50 /dev/mono0
crwxr-xr-x 1 root   root     2  81 Mon Jun 15 14:50 /dev/mono1
crwxr-xr-x 1 root   root     2  82 Mon Jun 15 14:50 /dev/mono2
crwxr-xr-x 1 root   root     2  83 Mon Jun 15 14:50 /dev/mono3
```

The following diagram summarizes bit assignments in the virtual terminal minor number:

```
7654 3210
 |           1=physical device, 0=logical device
  ||         00=color, 01=mono, 1x=reserved
    ||||     terminal's index number
```

The system initially defaults to a maximum of four color and four monochrome sessions. This may be altered by patching *character* variables **VTVGA** and **VTMONO**. For example, to allow for six color and three monochrome sessions, enter the following command while running as root (note that this will not take effect until after the

system has been rebooted):

```
/conf/patch -v /coherent VTVGA=6:c VTMONO=3:c
```

Running multiple sessions on different virtual consoles requires that logins be enabled for each of the virtual consoles. Each session must have a corresponding entry in file **/etc/ttys**. For example, a system allowing four color and four monochrome sessions would have entries in **/etc/ttys** as follows:

```
0lPconsole
1lPcolor0
1lPcolor1
1lPcolor2
1lPcolor3
1lPmono0
1lPmono1
1lPmono2
1lPmono3
```

Device **/dev/console** must not be enabled when using virtual consoles! Additional lines would be present if logins are enabled for other devices such as serial ports. Commands **enable** and **disable** may be used, as usual, to allow or disallow logins on individual virtual consoles.

When virtual terminals are enabled, kernel output, such as messages about *user traps* or *system panics*, goes to the currently active session (i.e., the session with the cursor showing).

Altering Virtual Consoles

To add, delete, or alter the configuration of virtual consoles, log in as the superuser **root** and type the following commands:

```
cd /etc/conf
console/mkdev
bin/idmkcoh -o /kernel_name
```

where *kernel_name* is what you wish to name the newly built kernel. When you reboot, invoke *kernel_name* in the usual manner and your new configuration will have been implemented.

See Also

Administering COHERENT, console, device drivers, enable, kb.h

Notes

Some confusion can arise when you attempt to install COHERENT to use both color and monochrome consoles.

At installation time, you are asked if you want to install both color and monochrome screens. If you reply "yes," you must select only four multiscreens for each. Otherwise, you will find it difficult to address virtual consoles on both consoles: COHERENT uses the lower function keys for virtual consoles on the color monitor, and the upper function keys for those on the monochrome monitor.

If you have requested two consoles, COHERENT uses the color terminal by default. If you really have only a monochrome monitor plugged into your system, you must invoke the appropriate monochrome virtual console; otherwise, you will nothing on your monitor.

void — C Keyword

Data type

The keyword **void** indicates that the function does not return a value. Using **void** declarations makes programs clearer and is useful in error checking. For example, a function that prints an error message and calls **exit** to terminate a program should be declared **void** because it never returns. A function that performs a calculation and stores its result in a global variable (rather than **return**ing the result), or one that returns no value, should also be declared **void** to prevent the accidental use of the function in an expression.

See Also

C keywords
ANSI Standard, §6.1.2.5

volatile — C Keyword

Qualify an identifier as frequently changing

The type qualifier **volatile** marks an identifier as being frequently changed, either by other portions of the program, by the hardware, by other programs in the execution environment, or by any combination of these. This alerts the translator to re-fetch the given identifier whenever it encounters an expression that includes the identifier. In addition, an object marked as **volatile** must be stored at the point where an assignment to this object takes place.

See Also

C keyword, const
ANSI Standard, §6.5.3

Notes

Although COHERENT recognizes this keyword, the semantics are not implemented in this release. Thus, storage declared to be **volatile** might have references removed by optimizations that the compiler performs. The compiler will generate a warning if the peephole optimizer is enabled and the keyword **volatile** is detected.

vprintf() — STDIO Function (libc)

Print formatted text into standard output stream

```
#include <stdarg.h>
#include <stdio.h>
int
vprintf(format, arguments)
char *format; va_list arguments;
```

vprintf() constructs a formatted string and writes it into the standard output stream. It translates integers, floating-point numbers, and strings into a variety of text formats. **vprintf** can handle a variable list of arguments of various types. It is roughly equivalent to **printf()**'s conversion specifier **%r**.

format points to a string that can contain text, character constants, and one or more *conversion specifications*. A conversion specification defines how a particular data type is converted into a particular text format. Each conversion specification is introduced with the percent sign '%'. (To print a literal percent sign, use the escape sequence '%%'.) See **printf()** for further discussion of the conversion specification and for a table of the type specifiers that can be used with **vprintf()**.

After *format* comes *arguments*. This is of type **va_list**, which is defined in the header **stdarg.h**. It has been initialized by the macro **va_start()** and points to the base of the list of arguments used by **vprintf()**. Each argument must have basic type that can be converted to a pointer simply by adding an '*' after the type name. This is the same restriction that applies to the arguments to the macro **va_arg()**. For more information, see **va_arg()**.

arguments should access one argument for each conversion specification in *format* of the type appropriate to conversion specification. For example, if *format* contains conversion specifications for an **int**, a **long**, and a string, then *arguments* access three arguments, being, respectively, an **int**, a **long**, and a **char** *.

If there are fewer arguments than conversion specifications, then **vprintf**'s behavior is undefined (and probably unwelcome). If there are more, then **vprintf()** evaluates and then ignores every argument without a corresponding conversion specification. If an argument is not of the same type as its corresponding type specification, then the behavior of **vprintf()** is undefined; thus, accessing an **int** where **vprintf()** expects a **char *** may generate unwelcome results.

If it writes the formatted string correctly, **vprintf()** returns the number of characters written; otherwise, it returns a negative number.

See Also

fprintf(), libc, printf(), sprintf(), vfprintf(), vsprintf()
ANSI Standard, §7.9.6.8

Notes

vprintf() can construct a string up to at least 509 characters long.

vsh — Command

Interactive graphical shell
vsh [-d*directory***] [-eirt]**

vsh is the COHERENT system's visual shell. With it, users can use arrow keys or simple keystrokes to perform tasks under the COHERENT, such as change directories, edit files, and execute programs. Each user can program a bank of up to nine function keys to perform complex tasks with a single keystroke. With **vsh**, a naive user can access much of the power of the COHERENT system without having to learn the details of **sh** or **ksh**.

Unlike X or other windowing systems, **vsh** works on a character-based terminal and requires only a modest amount of memory. It does not require a mouse.

Graphics Interface

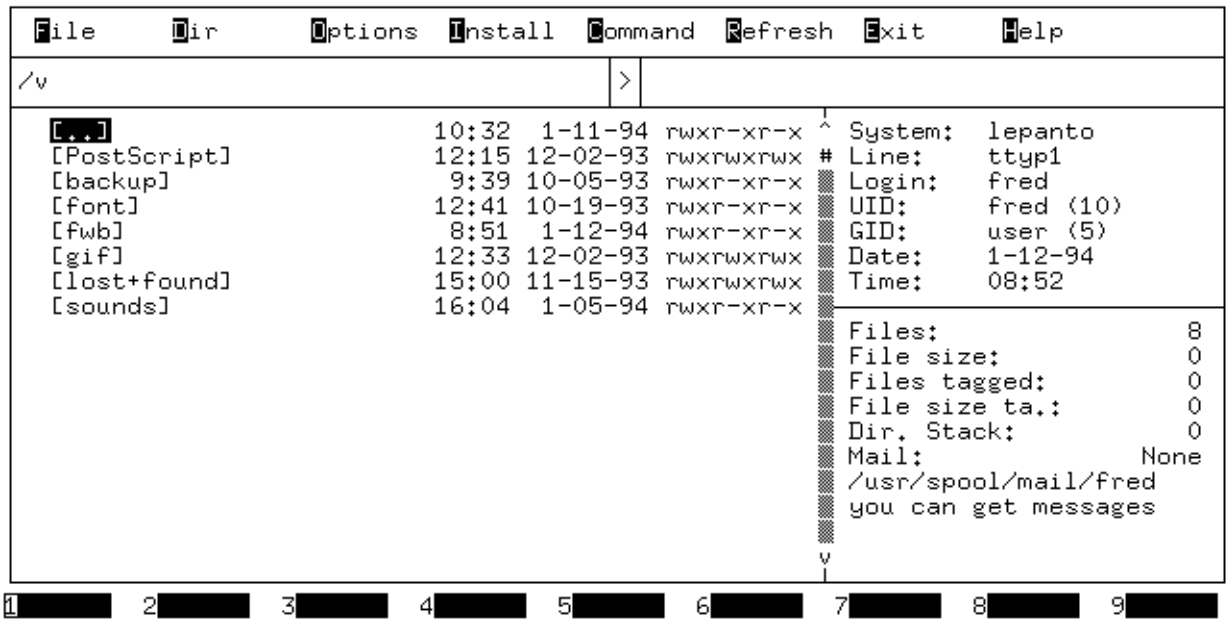
vsh uses the **curses** library and **terminfo** descriptions. To use **vsh**, you must have a **terminfo** description installed for the device upon which you wish to run it, and you must set the environmental variable **TERM** to point correctly to that description. For example, to run **vsh** from your console, you should set **TERM** to **ansipc**; while to run it from a PC that is plugged into a serial port, you should set **TERM** to **vt100**. You must have a **terminfo** description for the device to which you set **TERM**, or **vsh** will behave bizarrely. For more information on devices and how to set them, see the Lexicon entries for **TERM** and **terminfo**. For more information on terminals in general, see the entries for **terminal** and **console**.

To ensure that **TERM** set correctly, you may wish to embed the command **ttytype** in the file **/etc/profile**. For more details, see the Lexicon entry for **ttytype**.

If you have a non-standard terminal or have trouble displaying **vsh**, try invoking it with the options **-e** or **-t**. All of **vsh**'s command-line options are described below.

Main Screen

When you invoke **vsh**, you see a screen that appears as follows:



As you can see, the screen is divided the following six sections, or *windows*:

- The first window, the *Command* Window, is the narrow window that runs across the top of the screen. This window lists the commands that **vsh** can perform. You will enter this window frequently as you work with **vsh**.

- The second window, the *Current Directory* window, names the directory that you are currently in.
- The third window, the *Destination Directory* window, names the default destination directory.
- The fourth window, the *File* Window, extends down the left side of the screen. It lists the contents of the current directory. You will also work frequently in this window.
- The fifth window, the *System* Window, is the upper window on the right side of the screen. It gives information about the system, that is, who is running **vsh**, the device she is running it on, and the current date and time. Your cursor never enters this window.
- The last window, the *Status* Window, gives information about the work you have performed under **vsh**. Again, your cursor will never enter this window.

Across the bottom of the screen are nine "stubs," one each for function keys one through nine. The stub's text indicates the command that **vsh** executes when you press that key.

The following sections discuss each window in detail.

File Window

The file window lists all of the files and directories within the current directory. This is the default window for **vsh**; the cursor ordinarily rests in this window, and you will do most of your work in it.

The leftmost column in the File Window gives the name of each file and directory. Directories are given at the top of the list; they are enclosed within brackets '[ ]'. The other columns give, respectively the time the file or directory was last updated; the date it was last updated; and its permissions. For information on how to interpret the permissions string, see the Lexicon entry for the command **ls**.

The top listing in the File Window is always **[..]**, which represents the current directory's parent directory.

The top listing in this list is highlighted by being shown in reverse video. To move the highlighting bar up and down the list, use the arrow keys. If you press the arrow keys on your keyboard's number pad, be sure to turn the **<NumLock>** key *off*, or the keys will not work as you expect. If you press the (°) key, the bar shifts down one row on the list. Pressing the (ª) key moves the bar up one row.

You can page up or page down by pressing, respectively, the keys **<PgUp>** and **<PgDn>**. The key **<Home>** moves the cursor to the top of the list, and **<End>** moves it to the bottom. If your terminal does not implement these keys, you can use the following control characters:

<ctrl-N>	Next page (like **<PgDn>**)
<ctrl-P>	Previous page (like **<PgUp>**)
<ctrl-A>	Beginning (top) of list (like **<Home>**)
<ctrl-E>	End (bottom) of list (like **<End>**)

Note that if the list of files and directories is too large to fit into the window, moving the bar to the bottom of the window and pressing (°) will scroll the list. If you press the **<End>** key, the row moves to the last row in the list; and if you press **<Home>**, it moves to the top of the list.

A scroll bar runs down the right side of the File Window. As you scroll up and down this window, the scroll bar moves. Note that the position of the scroll bar is proportional to the highlighting bar's position in relation to the entire list of files, not just to its current position within the File Window. This gives you an easy way to see just where you are in the entire file list.

If you position the highlighting bar over the name of a directory and press (¢), **vsh** names that directory in the Current Directory Window and displays its contents in the File Window. For example, if you position the highlighting bar over the entry for directory **[letters]** and press (¢), **vsh** displays the contents of directory **letters** in the File Window. (If you are familiar with the Bourne or Korn shell, this has the same effect as typing the command **cd letters**.) To return to the directory you had just been displaying (that is, the parent directory of **letters**), use the arrow keys to move the highlighting bar to the entry **[..]**; then press (¢). **vsh** changes the contents of the Current Directory Window, and in the File Window erases the contents of **letters** and displays the contents of its parent directory.

If you press (¢) while a file is highlighted instead of a directory, **vsh** does the following:

1. If the file is executable, **vsh** executes it.

2. If the file matches a pattern from the file-action list, **vsh** executes the action from the list with the file as input. The file-action list is in file **$HOME/.vsh**; it looks like:

```
[Mm]akefile:make
*.mk:make -f %F
*.sh:sh %F
*.c:cc -c -O %F
*.sc:sc %F
*.a:ar tv %F | more
*.[1-9]:nroff -man %F | more
*.tar.F:fcat %F | tar xvf -
*.F:fcat %F | more
*.tar.Z:zcat %F | tar xvf -
*.Z:zcat %F | more
```

vsh recognizes most common wildcard characters; for a table of these and their meaning, see the Lexicon entries for **wildcards**. The token **%F** stands for the file that is currently highlighted. For example, in the above example the entry ***.Z:zcat %F** means that if you select a file with the suffix **.Z** (which usually means that a file has been compressed), it passes that file to **zcat** to uncompress and display it. **vsh** defines many defaults for you when it creates this file, which you can use as a model. To change the file-action list, use the **File actions** sub-command of the **Install** command, which is described below.

3. If the file appears to be ASCII **vsh** displays it with the default viewer.

While **vsh** is working, it displays a large letter 'X' in reverse video in the lower right corner of the screen. This shows that **vsh** is doing some internal task. **vsh** cannot accept any commands while the 'X' is displayed, so please be patient.

Also, note that **vsh** cannot handle more than 1,000 files in any given directory. If a directory contains more than 1,000 files, only the first thousand will be available for use.

System Window

The system window is the upper of the two windows on the right side of the screen. The cursor never enters this window; rather, this window simply displays information about your COHERENT system, and how you are currently using it. It contains the following entries:

System: This gives the name of your system, as you (or your COHERENT system administrator) has set it in file **/etc/uucpname**. See the Lexicon's entry for **uucpname** for more details on proper naming conventions for COHERENT systems.

Line: This gives the device by which you are accessing your COHERENT system. If you are working on your system's console device, then you should see **console** on this line; whereas if you are accessing your COHERENT via a PC plugged into serial port **com1l**, you should see **com1l** here. If you are using virtual consoles, the line is shown as **mono[0-8]** or **color[0-8]**. See the Lexicon entries for **console** and **asy** for more information about the devices through which you can access a COHERENT system.

Login: This gives the name under which you logged into COHERENT. For example, if your login identifier is **fred**, then you should see **fred** on this line.

UID: This shows your user-identification number (or UID). This is the unique number by which your COHERENT system knows you, as set in file **/etc/passwd**. For information on the UID and how to set it, see the Lexicon entries for **passwd** and **setuid**.

GID: This gives the number and name of the user group to which you belong. Users on a COHERENT system can be organized into groups; permissions on files can be set to include the members of your group, but exclude all others. For information on groups, see the Lexicon entries for **group** and **setgid**.

Date: This gives today's date (or rather, what your COHERENT system thinks today's date is).

Time: This gives what your system thinks the current time is. If your system's time is not set correctly, then the time shown here will not be correct. For information on how to set the system time, see the Lexicon entries for the commands **ATclock** and **date**.

The time can also vary depending upon what time zone your COHERENT system thinks it's located in. For information on timezones and how to set them correctly, see the Lexicon entry for **TIMEZONE**.

Command Window

The Command Window is the top window, and stretches across the width of the screen. This window gives you access to **vsh**'s commands. Some commands in the command window actually open an entire menu of commands, with which you can perform all manner of work.

The command window contains the following entries. For convenience, the following displays the entries vertically; the actual window displays them horizontally:

File
Directory
Options
Install
Command
Refresh
Exit
Help

When the cursor is in the File Window (which is the default) and you wish to execute one of the commands in the Command Window, press its initial letter. For example, to execute the **Refresh** command, press **R**.

Note that the commands on this window are in two groups. A command's behavior differs, depending upon which group it belongs to.

The commands **File**, **Directory**, **Options**, and **Install** display a drop-down menu when you invoke it. That is because they have more than one option available under it. If you do not wish to invoke any of the sub-commands on that menu, you can do either of the following: You can press the **<Esc>** key, which erases the drop down-menu and returns you to the File Window; or you can press the (æ) or (Æ) keys, which move you to the command in this group that lies, respectively, to the left or to the right of the current command. For example, suppose that you were in the File Window, and you pressed **F**, to invoke the **File** command. **vsh** would move the cursor into the Command Window, and display the File Command's drop-down window, which displays its sub-commands. If you then pressed the **<Esc>** key, **vsh** would return you to the File Window. If you pressed the (Æ) key, **vsh** would erase the **File** command's drop-down window and display, instead, the drop-down window for the **Directory** command. If, however, you pressed the (æ) key, **vsh** would erase the **File** command's drop-down window and display, instead, the drop-down window for the **Help** command. As you can see, **vsh** "wraps-around" the cursor — it considers the command at the far right to be to the left of the command to the far left, and vice versa. This concept is a little difficult to grasp when you read about it, but once you try it, it will quickly become clear.

Please note that **vsh** delays for one second its reaction to the **<esc>** key. The **curses** function **wgetch()**, which is used to read the keyboard, needs this delay so it can distinguish between the **<esc>** key and the other function keys, which all of which start with an **<esc>**. So, please be patient.

The other group of commands are the commands **Command**, **Refresh**, **Exit**, and **Help** each have only option, so when you invoke one of them, it immediately begins to execute that option. When you access one of these commands through the (Æ) and (æ) keys, each displays a drop-down menu that shows its one option.

The following describes each command in detail.

File Pressing **F** invokes the **File** command. This displays a drop-down menu that lists a set of sub-commands. These sub-commands let you manipulate files; with them, you can edit a file, create a file, change its permissions, rename it, erase it, print it, or do other common tasks.

To invoke a sub-command, you can do either of the following: Press the letter in the sub-command that is underlined (each sub-command has its own unique letter with which you can invoke it); or use the (ª) and (°) keys to move the highlighting bar to that command, and then press (¢).

The following discusses each sub-command in detail:

Copy This sub-command copies a file. Please note that the behavior of this subcommand depend upon whether you have tagged files.

If you have tagged one or more files, **vsh** opens a pop-up window that requests the path name of a directory. By default, **vsh** displays the destination directory, if you have set one. When you enter the path name, **vsh** copies every tagged file into that directory.

If you have not tagged any files, **vsh** opens a pop-up window that requests that you enter a file name or a path name. Again, if you have set a destination directory, the window displays it by default. If you enter only a file name into this window, **vsh** copies the highlighted file into the

newly named file in the current directory; if you have named an existing file, **vsh** prompts you before it overwrites that file. If you enter a path name, **vsh** copies the highlighted file into the directory you have named; the copied file retains its current name. If, however, you enter both a file name and a path name, then **vsh** copies the highlighted file into the directory you named, and gives it the file name that you entered.

Note that this command will not overwrite a file that you do not own; nor will it create a new file in a directory in which you do not have write permission, or copy a file on which you do not have read permission. For more information on copying files under COHERENT, see the Lexicon entry for the command **cp**.

Move This sub-command prompts you for the name of a directory; if you have set a destination directory, **vsh** displays it by default. When you confirm the destination, **vsh** then moves all tagged files into it. (If no files are tagged, **vsh** moves only the highlighted file. For more information on tagging, see the entry for the sub-command **Tag**, below.) The files retain their names in the new directory.

This command does not move a file for which you do not have read permission, or move a file into a directory into which you do not have write permission; nor will it move a file into a non-existent directory (of course). For details on moving files, see the Lexicon entry for the command **mv**.

Delete This sub-command deletes the tagged files. (If no files are tagged, then it deletes only the highlighted file. For more information on tagging, see the entry for the sub-command **Tag**, below.) It will prompt you to confirm that you really do want to delete the file or files in question. With regard to the mass deletion of tagged files, this sub-command lets you choose whether to do a mass deletion or delete files one at a time.

Note that this sub-command will not delete a file that you do not own. For details on deleting files, see the Lexicon entry for the command **rm**.

Rename
This sub-command lets you rename the highlighted file. It opens a pop-up window that shows the current name of the file, and prompts you to type the new name. Press **<Esc>** to abort this sub-command, or type the new name and press (¢).

It does not work with directories. It will not let you rename a file that you do not own. For details on renaming a file, see the Lexicon entry for the command **mv**.

Execute
This sub-command executes the highlighted file. **vsh** prompts you to type the arguments you wish to pass this file, then invokes the file with those arguments.

Note that **vsh** will not execute a file for which you do not have execute permission.

Access This sub-command lets you change the manner in which every tagged file can be accessed. (If no files are tagged, the default is the highlighted file.) When you invoke it, **vsh** displays the following pop-up window for each tagged file:

```
Change access f file filename

Owner
Read [x]       Write[x]        Execute[ ]

Group
Read [x]       Write[ ]        Execute[ ]

World
Read [x]       Write[ ]        Execute[ ]

Special
Set UID [ ]    Set GID [ ]     Set sticky[ ]
```

An 'x' in a field means that that permission is turned on; a blank means that it is turned off. Use the arrow keys to move to the cursor the field whose status you wish to change, then enter a space or 'x' to, respectively, turn off or turn on that given permission. To abort this command, press **<Esc>**.

For information what permission fields mean, see the Lexicon entry for **ls**. Note that you can reset permissions only on the files you own.

Owner This lets you change the owner and group that owns each tagged file. If no files are tagged, then this applies only to the highlighted file. When you invoke this sub-command, **vsh** opens a pop-up window that shows the user and group that own a file: type the name of the user or group you want to own the file. **vsh** repeats this step for each tagged file. To abort this command, press **<Esc>**.

For details on changing ownership of a file, see the Lexicon entries for the command **chown** and **chgrp**. Note that only the superuser **root** can run this command.

Print This passes every tagged file to the print spooler for printing. To change the default print spooler, use the **Install** command's **Print spooler** sub-command.

Note that **vsh** simply passes the file to the spooler for printing; you cannot use this to process a file before printing it. If you try to use this feature of **vsh** to print a file on a PostScript printer, the printer will hang. We suggest that you use the **Command** feature to print a file on a specialized printer; it's a little more difficult, but it works. Another approach is to use the spooler **lp** and prepare a special backend script to do the processing automatically. For details on how to do that, see the Lexicon entries for **lp** and **printer**.

View This sub-command invokes the default viewer to display the contents of every tagged file. If you try to view the contents of a binary file, the results may not be what you expect.

Note that **vsh** will not display a file for which you do not have read permission. To change the default viewer, use the **Install** command's **File viewer** sub-command.

Edit This sub-command invokes the text editor to edit every tagged file. If no files are tagged, then edit only the highlighted file.

The default text editor is **vi**, which can create problems for persons who do not know how to exit from that editor. For a quick brush-up on **vi**, see the Lexicon entry for **elvis**. To change the default text editor, use the **Install** command's **Editor** sub-command. Note that COHERENT will not let you edit a file for which you do not have read permission.

Edit new
This sub-command prompts you to type the name of a file, then invokes the editor for that file. This can be a new file (that is, one that does not yet exist in the current directory), or a file that already exists.

Note that if you do try to edit a binary file, you may find yourself running into difficulties.

Touch This "touches" every tagged file — that is, it changes the date and time that the file was last modified, just the same as if you had just edited it.

Note that you cannot touch a file for which you do not have write permission. For more information on touching files, see the Lexicon entry for the command **touch**.

Tag all This sub-command "tags" every file in the current directory. This lets you do mass moves or deletions of files. When you tag a file, **vsh** updates the entries **Files tagged** and **File size ta.** in the Status Window, to reflect the number and total size of the files you have just tagged. It also prints an asterisk next to the tagged file.

When the cursor is in the File Window, you can toggle tagging on the highlighted file by pressing the space bar. Note that the highlighted file is implicitly tagged, whether an as asterisk appears next to it or not. For details, see the section on the Status Window, below.

Untag all
This sub-command untags all files that are tagged in the current directory. As noted above, you can toggle the tagging of the highlighted file by pressing the space bar. This command updates the Status Window to reflect your changes.

Select This sub-command opens a pop-up window and lets you enter a regular expression; it then tags all files that match the expression. For example, if you enter ***.c**, then this sub-command tags all files that end in the string **.c**.

File type
This sub-command prints a summary of information about the type of the highlighted file.

File info
This sub-command opens a pop-up window that displays the following information about the highlighted file or directory:

```
Filename
Filetype
I-Node
Links
Owner UID
Owner GID
access
modification
status changed
```

Filename is the name of the file. *Filetype* is its type, e.g., directory or regular file. *I-Node* gives the number of this file's i-node; for information on what an i-node is, see its entry in the Lexicon. *Links* gives the number of links to the file. For information on what a link is, see the Lexicon entries for **ln** and **link()**. *Owner UID* and *Owner GID* identify the owner and group that own this file. For information on what the UID and GID are, see the Lexicon entries for **setuid** and **setgid**. *access*, *modification*, and *status changed* give, respectively, the date and time the file was last accessed, last modified, or last had its status changed.

Directory
Pressing **D** invokes the **Directory** command. This displays a drop-down menu that lists a set of sub-commands. These sub-commands let you manipulate directories; with them, you can create a directory, remove a directory, change permissions, and other common tasks. You can also manipulate a "directory stack," which lets you jump quickly from one directory to another without having to retype its name.

The following discusses each sub-command in detail:

Change
This lets you change the current directory. When you invoke this subcommand, **vsh** displays the following pop-up window:

```
Enter destination path
```

Type the full path name of the directory you wish to enter. If this directory does not exist, or if you cannot access it, **vsh** leaves you in the current directory; otherwise, it moves you to the requested directory.

Home This moves you to your home directory.

User's Home
This moves you to the home directory of another user. When you invoke this sub-command, **vsh** asks you to name the user whose home directory you wish to enter. To abort, press **<Esc>**. If the user you enter does not exist or if you do not have permission to read her home directory, **vsh** leaves you in the current directory; otherwise, **vsh** moves you into that user's home directory.

Set dest
Set the destination directory. This directory is saved in your **.vsh** file, and is restored the next time you invoke **vsh**.

Push The next three sub-commands makes it easy for you to maneuver your way around the COHERENT file system. The work by using what is called a "directory stack". In effect, you can tell **vsh** to remember the directory you are in (this is termed "pushing" the directory onto the stack); then, when you have switched to another directory, you can returned to this directory simply by "popping" this pushed directory from the directory stack. This lets you move around among directories without having to retype them continually.

The **Push** sub-command pushes the current directory onto the directory stack. When you push a directory, **vsh** increments the number next to the entry **Dir. Stack** in the Status Window. This tells you how many directories you have pushed onto the directory stack.

Pop & cd
This sub-command moves you to the last directory you pushed onto the directory stack. It also removes that directory from directory stack. When you pop a directory from the directory stack, **vsh** decrements the number next to the entry **Dir. Stack** in the Status Window. This tells you how many directories remain on the directory stack.

Note that directories are popped in the order opposite from that in which they were entered. For example, if you pushed directory **/usr/bin/sys** onto the directory stack, then directory **/usr/lib/mail**, then **/bin**, invoking the **Pop** sub-command will return you to directory **/bin**, then to **/usr/lib/mail**, and finally to directory **/usr/include/sys**.

Switch This command switches the current directory and the top entry in the directory stack.

Copy This copies the highlighted directory plus all of its contents into another directory whose name you type into a pop-up window. It behaves much like the command **cpdir**.

Delete This deletes the highlighted directory. It does not work with files. If the directory has files in it, **vsh** will prompt you and ask if you want the directory to vanish. If you answer 'Y', then **vsh** removes it, files and all — just as if you had executed the command **rm -rf**.

vsh will not delete a directory that you do not own.

Rename
This sub-command renames the highlighted directory. **vsh** opens a pop-up window and prompts you to type the new name of the directory. Press **<Esc>** to abort this sub-command. Note that you can rename only directories that you own. This sub-command does not work with files.

Create This sub-command creates a new directory in the current directory. **vsh** prompts you for the name of the new directory, and then creates it. Note that you can create a directory only if you have write permission in the current directory.

Access This lets you reset the access permission on the highlighted directory. This is the directory equivalent of the **File** command's **Access** sub-command.

Owner This lets you reset the user and group that own a given directory. This is the directory equivalent of the **File** command's **Owner** sub-command. Note that only the superuser **root** can run this command.

Read new
This tells **vsh** to re-read the current directory. **vsh** copies the contents of the current directory into memory for its own use; thus, if other people manipulated the directory and its contents after **vsh** read its contents, what you see in the File Window will not reflect the true state of affairs in that directory. If you are working with a directory that is being manipulated by one or more other people, you should issue this command from time to time, to ensure that you are working with an accurate image of the directory's contents.

Switch CWD
This command switches the current working directory with the destination directory.

Switch TOS
This switches the destination directory with the directory on top of the stack.

Info This is the same as the **File info** sub-command under the **File** command, described above.

Options
Pressing **O** invokes the **Options** command. Its sub-commands let you perform common system tasks. The following discusses each sub-command in detail:

Shell This command invokes an interactive shell. When you exit from the shell (either by typing **exit** or **<ctrl-D>**), you will be returned to **vsh**.

By default, **vsh** invokes the Bourne shell **sh**; to change the default shell, use the **Shell** sub-command under the **Install** command, which is described below.

Lock terminal
This command locks your terminal. When the terminal is locked, no command can entered into it; this lets you walk away from your terminal briefly without worrying whether anyone (e.g., your cat) will do anything untoward under your login. The terminal remains locked until you retype the

secret password that you entered when you invoked this sub-command

When you invoke this sub-command, a pop-up window appears with the following:

```
Lock Enter Password
```

vsh prints a '#' to echo each character that you type. If you wish to abort the **Lock** sub-command, press **<Esc>**. When you have finished entering your password, press (¢). When you have entered the password, the following window appears:

```
This Terminal is locked!

Enter Password to unlock
or hit return to logoff
```

Type the password to return to **vsh**. If you (or someone else) presses (¢), you will be logged out of COHERENT.

Messages

This sub-command lets you receive or ignore messages. A message can be sent to your terminal by another user or another process; for example, the **mail** command may send a prompt to your screen when new mail is received.

When you invoke this sub-command, **vsh** displays the following pop-up window:

```
Do you want to receive messages ?

       Yes         No
```

Use the (Æ) and (æ) keys to select the option you want, then press (¢). When you change your message status, the information in the Status Window changes. For example, when you turn off messaging, the following appears at the bottom of the Status Window:

```
You can't get messages
```

For information on how COHERENT sends messages to your terminal, see the Lexicon entry for **mesg**. Also, see the description of the Status Window, below.

Online manual

This lets you select an entry from the COHERENT system's on-line manual pages. When you invoke this sub-command, **vsh** displays the following pop-up window:

```
Enter topic, chapter is optional :

Topic:

Chapter:
```

Type the title of the Lexicon entry that interests you; for example, to see the Lexicon entry for the command **vsh**, enter **vsh** in the **Topic** slot, then type (¢). Do not enter anything into the **Chapter** slot; this does not apply to the COHERENT system. You will see on your screen the Lexicon entry that you are now reading. If you change your mind, press **<Esc>** to abort this command.

Note that if you did not install or uncompress the manual pages when you installed your COHERENT system, this sub-command will not work. For more information on the COHERENT manual pages, see the Lexicon entries for the commands **help** and **man**.

System news

Display news about your current system. By default, this invokes the COHERENT command **msgs**.

Internet news

Invoke a reader for Internet news. By default, this command invokes **rn**, should you have it installed.

Electronic mail
Invoke your mail reader. By default, this invokes **mail**.

Install Pressing **I** invokes the **Install** command. Its sub-commands let you modify some of **vsh**'s default behaviors; in particular, it lets you program your function keys to execute some tasks you select with one keystroke. The following discusses each sub-command in detail:

Display
This command lets you customize appearance of **vsh**. When you invoke this sub-command, **vsh** displays the following pop-up window:

```
Display Attributes

Menubar
Menu color
Menu attribute
Dialog box
```

The entry **Menubar** lets you select the display attribute for the menu bar, which can be one of **bold**, **underline**, or **reverse**.

The entry **Menu color** lets you set the menu color, which can be either **normal** or **reverse**. (This may vary, depending on the type of terminal you are using.)

The entry **Menu attribute** lets you set the display attribute for pulldown menus, which can be one of **bold**, **underline**, **bold**, or **normal**.

Finally, the entry **Dialog box** lets you set the display attribute for dialogue boxes, which can be one of **bold**, **underline**, or **both**.

The best way to see what these commands do is to try them out. As mentioned above, the behavior may change from device to device, depending upon the type of terminal that you are using.

Function keys
This lets you "program" up to nine function keys, so you can invoke selected commands easily. Each user can have her own list of programmed function keys.

When you invoke this sub-command, **vsh** displays the following drop-down menu:

```
Function keys

Function key 1
Function key 2
   ...
Function key 9
```

Press 1 through 9 to program the corresponding function key (or use the (ª) and (°) keys to move then highlighting bar, then press (¢)). **vsh** asks you to enter the label for the function key and the command you want that function key to invoke. When you have finished, the new label will appear in the corresponding function-key tag at the bottom of the screen; and when you press that function key, **vsh** executes the corresponding command.

For example, to make the game **chase** one of your function key entries, do the following: First, press **I** to invoke the **Install** command. The press **k** to invoke the **Function keys** sub-command. When the function-keys drop-down menu appears, press **2**, for function-key **F2**. When the label pop-window appears, type **chase** into the first slot, which holds the label Press **<Tab>** to jump to the second slot, which holds the command to execute, then type **/usr/games/chase**. When you have done typing, press (¢).

As you can see, the **F2** stub at the bottom of the screen shows **chase**; and when you press **F2**, **vsh** launches you into **chase**. You can program the first nine function keys to work in the same way.

You can embed the token **%F** as a placeholder for the current file. For example, to count the number of lines in the current file, put the following command into a function-key definition:

```
wc -l %F
```

Because some computers still do not have function keys (e.g., the NeXT machine), you can also use the number keys to execute commands installed on the function keys.

By the way, for information on the highly amusing game **chase**, see its entry in the Lexicon.

Shell This sub-command lets you set the default shell that **vsh** runs when you invoke its **Shell** command. When you invoke this sub-command, **vsh** displays the following pop-up menu:

```
Enter command to run a shell
(Coherent default is '/bin/sh')

      /bin/sh
```

Type the shell that you want, either **/bin/sh** or **/bin/ksh**, and press (¢). (You can enter another program if you like, but you may get some strange results if you do.) For information on each shell, see its entry in the Lexicon.

Editor This lets you set the editor that **vsh** invokes when you select the **Edit** sub-command under the **File** command. When you invoke the **Editor** sub-command, **vsh** displays the following pop-up window:

```
Enter command to run an editor
(Coherent default is 'vi')

      vi
```

Type the editor that you want, one of **ed**, **me**, or **vi**; then press (¢). For information on each editor, see its entry in the Lexicon.

Print spooler

This lets you set the spooler that **vsh** invokes when you select the **Print** sub-command under the **File** command. When you invoke the **Print spooler** sub-command, **vsh** displays the following pop-up window:

```
Enter command to run a print-spooler
(Coherent default is 'lpr -B')

      lpr -B
```

Enter the spooler that you want. For more information on the spooling commands available under COHERENT, see the Lexicon entry **printer**.

Beginning with release 2.7 of **vsh**, this feature works with pipes. **vsh** understands that the token **%F** represents the current file. For example, if you have a PostScript printer, you will want every file to be processed by the command **prps** before you print it. Thus, enter the command:

```
prps %F | hpr -B
```

This tells **vsh** to filter each file through **prps** and pipe the output to the laser-printer spooler **hpr**.

Some of this functionality may not be necessary under COHERENT release 4.2, which implements the System-V **lp** print spooler. See the Lexicon article **printer** for details.

File viewer

This lets you set the viewer that **vsh** invokes when you select the **View** sub-command under the **File** command. When you invoke the **File viewer** sub-command, **vsh** displays the following pop-up window:

```
Enter command to run a file view utility
(Coherent default is 'more')

      more
```

Enter one of **more** or **scat**. For information on how these commands differ, see their entries in the Lexicon.

File action

As noted above, **vsh** has a list of default actions that it takes when you select a file of a given type. For example, if you invoke the **File** command, move the highlighting bar to a file with the suffix **.c** and press (¢), **vsh** by default invokes the C compiler **cc** to compile that file.

vsh stores in the file **$HOME/.vsh** the list of its default actions. The **File actions** sub-command invokes a special editor so you can edit this list.

When you invoke this option, **vsh** displays the following pop-up window:

```
Edit actions list
Configure action
```

Use the (ª) and (°) keys to move the highlighting bar to the item you want; then press (¢).

When you select **Edit actions list**, **vsh** displays a pop-up window that contains all of the default actions. The syntax of the default actions is described above. Use the (ª) and (°) keys to move the highlighting bar to the action you wish to edit. To erase the current line, press **<ctrl-D>**; to open a new line, press **<ctrl-I>**.

To modify the line that is currently highlighted, press (¢). When you do so, the highlighting bar disappears and a cursor appears. Use the (æ) and (Æ) keys to move the cursor to the point you wish to change; typing inserts new text into the command, whereas pressing **<Backspace>** erases text. When you have finished modifying the current line, press (¢). To abort modifying the current line, press **<Esc>**.

When you have finished modifying the action list, press **<Esc>**. **vsh** records your changes into file **$HOME/.vsh**, and returns you to the File window.

When you select the option **Configure action**, **vsh** displays a window with the prompt

```
Show file actions before execution ?
```

The cursor is under the response **y**, for **yes**. If you accept this option, **vsh** will prompt you for your confirmation before it performs a default action. If you want **vsh** simply to go ahead and perform its default without asking for your approval, press the (Æ) key to move the cursor to the option **n**, for **no**, and press (¢).

Sys. news reader

Tell **vsh** what system news program you want it to invoke by default.

Internet news

Tell **vsh** what Internet news reader you want it to invoke by default.

Electronic mail

Tell **vsh** what mail reader you want it to invoke by default.

Command

The command **Command** lets you send a command directly to a COHERENT shell. This lets you invoke commands that ordinarily are not available through **vsh**.

Suppose, for example, that you decided you wanted to play a session of the game **tetris**, and that you have not yet programmed **tetris** as one of your function keys. Press **C** to invoke **Command**. **vsh** moves the cursor moves to the bottom of the screen, and erases the row of boxes that describe the function keys. You can now type the command you want, in this case **/usr/games/tetris**. To run the command, press (¢); to abort entering a command and return to **vsh**, type **<Esc>**.

When you press (¢), **vsh** runs the command you typed. When you have finished playing **tetris** and have exited from it, **vsh** clears the screen and displays the message:

```
Hit any key to continue ...
```

When you press a key, **vsh** redraws itself on your screen and returns the cursor to the File Window.

(By the way, the COHERENT version of **tetris** is available as part of COHware volumes 2 and 3. For information on obtaining COHware, see the release notes that came with your copy of COHERENT.)

Command remembers the last 40 commands that you have issued. To invoke a command that you previous issued through **Command**, press the (ª) key. The last command you issued will appear in the command slot. If you continue to press the (ª) key, others commands appear, in reverse order from when you issued them. If you overshoot the command that you want to re-run, press the (º) key to walk back down the list of previously issued commands. When you find the previously issued command that you wish to rerun, just press (¢) and **vsh** runs it again.

You can also edit a previously issued command. The following gives lists the available editing commands:

←	Move the cursor one character to the left
→	Move the cursor one character to the right
<del>	Delete the character to the left
<backspace>	Delete the character to the left
<ctrl-D>	Delete the character over the cursor
<ctrl-P>	Go to last character of the command
<ctrl-N>	Go to first character of the command

A command can use environmental variables, such as **$HOME**. **vsh** will expand all environmental variables correctly before it tries to execute the command.

You can also embed the following tokens in a command:

%F	Represent the currently highlighted file
%T	Represent all tagged files
%D	Represent the destination directory

For example, the command

```
cp %T %D
```

copies all tagged files into the destination directory.

Refresh

The command **Refresh** redraws the screen. It does no other work. This is helpful if your screen has become jumbled or scrambled for any reason — such as a message being written onto your screen by another user.

To invoke this command, type **R**. **vsh** pauses very briefly, then the screen flickers as **vsh** redraws. If the screen had been confused for any reason, invoking this command should restore to its proper state. If you need to refresh the screen while a pop-up menu or a pop-up window is active, press **<cntl-L>**.

Exit The command **Exit** exits you from **vsh**. To exit from **vsh**, press **E**. In response, **vsh** pops the following window onto your screen:

```
Do you really want to quit?

       Yes       No
```

The window is in reverse video, for emphasis. The option **Yes** is underlined, to show that it is the default choice. If you really do wish to exit, press (¢); and **vsh** returns to the COHERENT shell.

If you changed your mind, however, and do not wish to exit, press the (Æ) key to change the option; this will shift the underlining from option **Yes** to option **No**. Pressing enter at this point selects the **No** option; **vsh** in response removes the pop-up window from the screen and returns you to the File Window.

If you change your mind again, though, and really do wish to exit, then press the (æ) key. The underlining shifts back to the **Yes** option; and when you press (¢) you exit from **vsh** and return to the shell.

Status Window

The Status Window is the lower window on the right side of the screen. The cursor never enters this window; rather, this window gives information about how **vsh** is functioning, and in particular about the files that are currently displayed in the File Window.

The Status Window contains the following entries:

Files This gives the number of files being shown in File Window. Note that this is all files that can be scrolled through that window, *not* the files that are shown in that window at this moment.

File size
This gives the total size, in bytes, of all files available through in the File Window.

Files tagged
This gives the number of files that you have tagged. See the description of the **File** command, above, for details.

File size ta.
This gives the total size of all tagged files. See the description of the **File** command, above, for details.

Dir. Stack
This gives the number of directories that currently reside on the directory stack. As noted above, you can "push" directories onto the directory stack or "pop" them from it. By doing so, you have an easy way to jump about from one directory to another, without having to type directory names repeatedly. See the above description of the **Directory** command for more details.

You can have a maximum of ten directories on the stack.

Mail This line indicates whether you have mail waiting to be read. If you don't, this line will say

```
None
```

whereas if you do, the line will say

```
Avail
```

and flash at you. If new mail arrives, **vsh** flashes

```
New
```

in that slot.

mailbox
This line gives the name of your mailbox — that is, the file that **mail** reads.

messages
This indicates whether your terminal can receive messages — e.g., whether a message will pop up on your screen if someone wishes to communicate with you via the **write** command. For more information on how to change the message status of your terminal, see the Lexicon entry for the command **mesg**.

Function Keys

The bottom of the screen show nine small boxes in reverse video. These are labelled **F1** through **F9**. If you have defined the key using the **Function Key** command, **vsh** displays the box the tag that you gave that key.

For example, in our above example we set key **F1** to run the command **ps -a**, and gave it the tag **ps**. At the bottom of the screen, the box labelled **F1** should show **ps**.

For more details, see the description of the **Function Key** command, above.

Configuration File

vsh reads the file **$HOME/.vsh** to configure itself.

A typical **.vsh** file reads as follows:

```
cwd=/v/fwb
shell=/bin/ksh
editor=me
print-spooler=hpr -B
view=more
make=make
me-disp-attr=reverse
pd-disp-color=normal
pd-disp-attr=bold
se-disp-attr=underline
pfkey1=  mail mail
pfkey9=tetris /usr/games/tetris
cmd=
      tetris
      tetris
      echo foo
```

cwd points to the current working directory, that is, the directory in which you have last worked with **vsh**. **vsh** returns you to that directory when you next invoke the shell.

shell, **editor**, **print-spooler**, **view**, and **make** give, respectively, the shell, editor, print-spooler, viewer, and make utility that you selected with the **Install** command. If you change one of these values, the behavior of **vsh** changes to reflect the change. For example, if you change the line

```
editor=me
```

to

```
editor=ed
```

then **vsh** will invoke **ed** the next time you request the **File** command' **Edit** sub-command.

me-disp-attr, **pd-disp-color**, **pd-disp-attr**, and **se-disp-attr** give the display features for, respectively, the menu bar, the menu color, the menu attribute, and the dialogue box.

The lines **pfkey1** through **pfkey9** set the behavior of the function keys. The first seven characters after the equal sign '=' give the text that appears in stub at the bottom of the screen. Everything after the first seven characters describes the command to be executed when you press that function key.

The text that follows the line **cmd=** lists the commands that you have executed with the command **Command**. You can embed the following tokens in a command:

%F Represent the currently highlighted file
%T Represent all tagged files
%D Represent the destination directory

These are used just as they are with the **Command** command, described above.

Command-line Options

vsh recognizes the following options:

-d*directory*
Enter **vsh** and begin to work in *directory*. If no *directory* is named, then begin work in the current directory **vsh** normally begins in the last directory used in your last **vsh** session.

-e Do not use the graphic character set. This option coarsens the appearance of **vsh**, but gives it a fighting chance to run on cheap terminals that do not implement the full alternate character set of the DEC VT-100 terminal.

-i Restrict the user's ability to run the **Install** command. In this mode, **vsh** can be used as a restricted shell, especially if it is embedded in **/etc/passwd**.

-r Restrict the shell. This option turns off the following:

- The command **Command**
- No interactive shell can be called from the **Options** menu
- Most options from the **Directory** menu
- Most options from the **Install** menu

LEXICON

This lets the system administrator restrict the activity of users fairly strongly.

-t This command-line option tells **vsh** to assume the entire VT-100 mapping. This is useful with terminals whose system definitions are incomplete, or the alternate character set is ignored.

Files

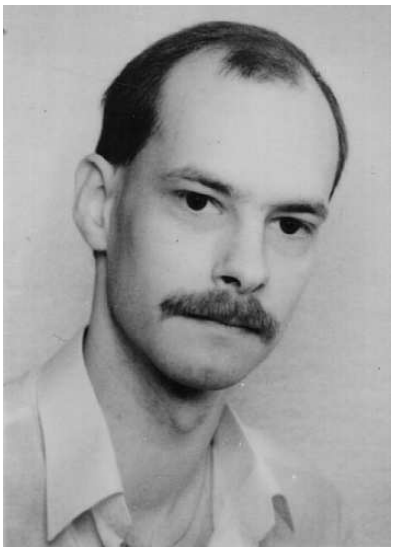
$HOME/.vsh — Configuration file

See Also

commands, ksh, sh, terminfo, ttytype, Using COHERENT

Notes

vsh was written by Udo Munk:



To reach Udo, send e-mail to udo@mwc.com.

vsprintf() — STDIO Function (libc)

Print formatted text into string

```
#include <stdarg.h>
#include <stdio.h>
int
vsprintf(string, format, arguments)
char *string, *format; va_list arguments;
```

vsprintf() constructs a formatted string in the area pointed to by *string*. It translates integers, floating-point numbers, and strings into a variety of text formats. **vsprintf()** can handle a variable list of arguments of various types. It is roughly equivalent to the '%r' conversion specifier to **sprintf()**.

format points to a string that can contain text, character constants, and one or more *conversion specifications*. A conversion specification describes how to convert a particular data type into a particular text format. Each conversion specification is introduced with the percent sign '%'. (To print a literal percent sign, use the escape sequence '%%'.) See **printf()** for further discussion of the conversion specification and for a table of the type specifiers that can be used with **vsprintf()**.

After *format* comes *arguments*. This is of type **va_list**, which is defined in the header file **stdarg.h**. It has been initialized by the macro **va_start()** and points to the base of the list of arguments used by **vsprintf()**. For more information, see **va_arg()**.

arguments should access one argument for each conversion specification in *format* of the type appropriate to the conversion specification. For example, if *format* contains conversion specifications for an **int**, a **long**, and a string, then *arguments* access three arguments, being, respectively, an **int**, a **long**, and a **char** *. If there are fewer arguments than conversion specifications, then **vsprintf()**'s behavior is undefined (and probably unwelcome). If

there are more, **vsprintf()** evaluates and then ignores every argument without a corresponding conversion specification. If an argument is not of the same type as its corresponding type specification, then the behavior of **vsprintf()** is undefined; thus, accessing an **int** where **vsprintf** expects a **char** * may generate unwelcome results.

If it writes the formatted string correctly, **vsprintf()** returns the number of characters written; otherwise, it returns a negative number.

See Also

fprintf(), libc, printf(), sprintf(), vprintf(), vsprintf()
ANSI Standard, §7.9.6.9

Notes

vsprintf() can construct a string up to at least 509 characters long.

vtkb — Device Driver

Non-configurable keyboard driver, virtual consoles

The device driver **vtkb** drives the keyboard on your system's console — that is, the keyboard that is plugged directly into your computer.

This driver recognizes the standard 83-, 101-, and 102-key AT-protocol keyboards, using the keyboard layout used in the United States. These codes are "hard-wired" into the driver. Unlike the other COHERENT keyboard driver, **vtnkb**, you cannot modify these settings.

vtkb is, in general, more robust than **vtnkb** in handling inexpensive keyboards that do not conform fully to accepted standards.

For details on how to select a given keyboard driver, see the Lexicon entry for **keyboard**.

See Also

device drivers, keyboard, vtnkb

vtnkb — Device Driver

Configurable keyboard driver, virtual consoles

The device driver **vtnkb** drives the keyboard on your system's console — that is, the keyboard that is plugged directly into your computer.

Unlike the related driver **vtkb**, **vtnkb** uses a loadable translation table to interpret keystrokes. This permits you to use any number of national keyboard mappings on your COHERENT system without changing the kernel in any way. You can select among any number of configuration programs stored in directory **/conf/kbd**, or you can create your own keymapping table to suit your preferences.

To change the layout and function-key bindings, run one of the keyboard-mapping programs kept in directory **/conf/kbd**. This directory contains the C source code for the mapping tables, as well as a **Makefile** that helps you rebuild the mapping programs. This rest of this article describes the structure of the driver **vtnkb**, and describes how you can write or modify a keyboard-mapping program.

Internal Structure of the Driver

vtnkb understands the following "shift" and "lock" keys:

scroll	Scroll lock
num	Keypad **<num>** lock
caps	**<shift>** or **<caps>** lock
lalt	Left **<alt>** key
ralt	Right **<alt>** key
lshift	Left **<shift>** key
rshift	Right **<shift>** key
lctrl	Left **<ctrl>** key
rctrl	Right **<ctrl>** key
altgr	**<alt graphic>** key (non-U.S. keyboards)

vtnkb records the internal shift state, as defined by the current positions of the shift and lock keys. The shift state is a logical combination of internal states **SHIFT**, **CTRL**, **ALT**, and **ALT_GR**. The **<lshift>** and **<rshift>** keys combine to form the current **SHIFT** state for non-alphabetic keys. Alphabetic keys generally use the current state

of the **<caps-lock>** key plus the left and right **<shift>** keys. Numeric keys on the keypad generally use the state of the **<num lock>** key plus the left and right **<shift>** keys. The left and right **<ctrl>** keys form the internal **CTRL** state. Likewise, the left and right **<alt>** keys form the internal **ALT** state. Note that 102-key keyboards generally replace the right **<alt>** key with the **<altgr>** (alt graphics) key, to allow access to the alternate graphics characters found on some keyboards.

vtnkb lets you configure or read the internal mapping tables via the following requests to **ioctl()**, as defined in header file **<sgtty.h>**:

TIOCGETF	Get function key bindings
TIOCSETF	Set function key bindings
TIOCGETKBT	Get keyboard table bindings
TIOCSETKBT	Set keyboard table bindings

Requests **TIOCGETF** and **TIOCSETF** reference a data structure of type **FNKEY**, which is defined in header file **<sys/kb.h>**. Structure member **k_fnval** is a character array that contains a series of contiguous function key/value bindings; the end of the bindings is marked by manifest constant **DELIM**. You can use any value other than **DELIM** as part of a function-key binding. Structure member **k_nfkeys** indicates how many function keys have associated entries in **k_fnval**. Function keys are numbered from zero through **k_nfkeys**-1.

How To Write a Keyboard Table

The main difference between the keyboard drivers **vtnkb** and **vtkb** is that **vtnkb** uses a "supplemental" process to interpret keystrokes. You can re-construct the interface to the **vtnkb** driver by modifying a keyboard-mapping file and then using a support module to link that file to the driver.

As noted above, directory **/conf/kbd** contains the source code for a series of such supplemental programs. These programs differ from each other only in the keyboard binding or mapping tables each uses; by selecting one such program and linking it into **vtnkb**, you can switch easily from the standard keyboard layout used in the U.S. to any of a number of layouts used in other countries. **/conf/kbd** also contains compiled executables, and a **Makefile** that you use to construct the executables from the corresponding source files.

The keyboard-mapping file is a C program that consists of initialized tables and strings. In addition, several header files provide the scan codes and other constants required for the key tables. This format makes the file easy to edit, and also lets you enter characters in several different formats.

The support module, in turn, performs several tasks. These include scanning the keyboard-mapping file for errors, reformatting the table for use by the device driver, and passing the reformatted table to the driver.

A keyboard-mapping source file consists primarily of three data structures that you must modify to support a given keyboard mapping. The first, and simplest, of the structures is **tbl_name**. This is a character string that describes the keyboard. For example, the stock 101-key U.S. AT keyboard mapping file **/conf/kbd/us.c** initializes this string to:

```
"U.S. AT keyboard table"
```

The second data structure, **kbtbl**, is an array of key-mapping entries. It has one entry (or row) for each possible key location. Each entry in this structure consists of 11 fields, which hold, respectively, the key number, nine possible mapping values, and a mode field. The following example is for physical key location 3 from key-mapping source file **/conf/kbd/belgian.c**:

```
{ K_3, 0x82, '2', none, none, 0x82, '2', '~', none, '~', O|T },
```

Field 1 contains the *scan code set 3* code value for the desired key. Header file **<sys/kbscan.h>** contains manifest constants of the form **K_***nnn* that map the AT keyboard's *physical* key number *nnn* to the corresponding scan code set 3 value generated by the keyboard. In the above example, **K_3** corresponds to key location three.

Fields 2 through 10 contain the key mappings corresponding to the following shift states, as follows:

2 base or unshifted
3 **SHIFT**
4 **CONTROL**
5 **CONTROL+SHIFT**
6 **ALT**
7 **ALT+SHIFT**
8 **ALT+CONTROL**
9 **ALT+CONTROL+SHIFT**
10 **ALT_GRAPHIC**

For "regular" keys, the values for these nine fields are eight-bit characters; for "function" or "shift" keys, they are special values. The symbolic constant **none** indicates that you want no output when the key is pressed in the specified shift state.

In the case of a function key, the value specified is the number of the desired function key. Header file **<sys/kb.h>** defines a set of symbolic constants of the form **f***n*, where *n* is the number of the desired function key. You should use these constants; they will improve the readability of your code, and they will protect your keyboard mapping source files from any future changes in the structure of the keyboard driver.

In the case of a "shift" key, all nine entries must be identical and must consist of one of the following symbolic constants: **scroll**, **num**, **caps**, **lalt**, **ralt**, **lshift**, **rshift**, **lctrl**, **rctrl**, or **altgr**. These are defined in the header file **<sys/kb.h>**. Note that 83-key XT-layout keyboards only have one "control" and "alt" key, so not every shift-key combination may be possible on your target keyboard.

The last (11th) field in the key entry is the "mode" field. The following symbolic constants specify the mode of the current key:

S The specified key is a "shift" or "lock" key. Note that all entries in array **k_val** must be identical for a "shift" or "lock" key to work correctly.

F The specified key is a "function" or special key. The value of all elements of array **k_val** must specify a function key number.

O The specified key is "regular" and requires no special processing.

C The **<caps-lock>** key affects this key.

M Make: generate an interrupt only upon key "make" (i.e., when the key is depressed). This mode is useful for keys that do not repeat. Note that using this mode with a "shift" key stops you from unshifting upon release of the key!

T Typematic: generate an interrupt when the key is depressed, and generate subsequent key-depression interrupts while the key is depressed. The rate at which interrupts are generated is specified by the typematic rate of the keyboard. This type is usually associated with a "regular" key.

MB Make/Break: generate an interrupt when the key is depressed and when it is released. No additional interrupts are generated no matter how long the key is depressed. This mode is used for "shift" keys.

TMB Typematic/Make/Break: generate an interrupt when the key is first depressed; generate subsequent key depression interrupts while the key remains depressed; and generate an interrupt when the key is released.

The above example specifies a mode field of **O|T**, which corresponds to a "regular" key with typematic repeat, and no special handling of the "lock" keys.

The last data structure, **funkey**, consists of an array of function-key initializers, one per function key. The initializers are simple, quoted character strings delimited by either hexadecimal value **0xFF**, octal value **\377**, or symbolic constant **DELIM**. Note that any other value can be used as part of a function-key binding. Function keys are numbered starting at zero.

Function keys are useful not only in the classic sense of the programmable function keys on the keyboard, but also as a general purpose mechanism for binding arbitrary length character sequences to a given key. For example, physical key location 16 is usually associated with the **<tab>** and **<back tab>** on the AT keyboard; and **/conf/kbd/us.c** sets the key mapping table entry for key 16 as follows:

```
{ K_16, f42, f43, none, none, f42, f43, none, none, none, F|T },
```

For traditional reasons, the **<back tab>** key outputs the sequence **<esc>[Z** whereas the **<tab>** key simply outputs the horizontal-tab character **<ctrl-I>**. Because at least one of the mapping values for this key is more than one character long, the key must be defined as a "function" key and all entries for the the key must correspond to function-key numbers. In this example, function key number 42 was chosen for **<tab>**, and function key number 43 was chosen for **<back tab>**. The constant **none** indicates that you want no output when the key is pressed in the specified shift state. The corresponding **funkey** initialization entries for function keys **f42** and **f43** are as follows:

```
/* 42 */      "\t\377",          /* Tab */
/* 43 */      "\033[Z\377",      /* Back Tab */
```

We strongly recommend that you comment your function-key bindings.

You can also change function-key bindings via the command **fnkey**. This command lets you temporarily alter one or more function-key mappings without changing your key-mapping sources.

Examples

Prior to the release of the 101- and 102-key, enhanced-layout AT keyboards, the **<ctrl>** key was positioned to the left of 'A' key. Most terminals also locate the **<ctrl>** key there. The first example shows how to swap the left **<ctrl>** key and the **<caps-lock>** key on a 101- and 102-key keyboard. The **<caps-lock>** key is physical key 30, whereas the left **<ctrl>** key is physical key 58. Their respective entries in file **/conf/kbd/us.c** source file are as follows:

```
{ K_30, caps, caps, caps, caps, caps, caps, caps, caps, caps,  S|M },
{ K_58, lctrl,lctrl,lctrl,lctrl,lctrl,lctrl,lctrl,lctrl,lctrl, S|MB },
```

Note that the **<caps-lock>** key is defined with mode **M** as it is a "lock" key. The keyboard will interrupt only on key depressions, because releasing a "lock" key has no effect. The left **<ctrl>** key is defined with mode **MB** as it is a "shift" key. The keyboard generates an interrupt on both key depression and key release, because the driver must track the state of this key.

To swap the aforementioned keys, simply change all occurrences of **caps** to **lctrl** and vice-versa, as well as swapping the mode fields. After making the changes, the entries now appear as:

```
{ K_30, lctrl,lctrl,lctrl,lctrl,lctrl,lctrl,lctrl,lctrl,lctrl, S|MB },
{ K_58, caps, caps, caps, caps, caps, caps, caps, caps, caps,  S|M  },
```

The second example converts a 101- or 102-key keyboard table to support an XT-style 83-key keyboard layout. The following section summarizes the "typical" differences found when comparing the two keyboard layouts. Needless to say, given the extreme variety in keyboard designs, your mileage may vary:

Location	*101/102 Value*	*83-key Value*	*Comments*
14	**none**	Various	Keyboard-specific
30	**caps**	**lctrl**	
58	**lctrl**	**lalt**	
64	**rctrl**	**caps**	
65	**none**	**F2**	Function Key
66	**none**	**F4**	Function Key
67	**none**	**F6**	Function Key
68	**none**	**F8**	Function Key
69	**none**	**F10**	Function Key
70	**none**	**F1**	Function Key
71	**none**	**F3**	Function Key
72	**none**	**F5**	Function Key
73	**none**	**F7**	Function Key
74	**none**	**F9**	Function Key
90	**num**	**Esc**	
95	**/**	**num**	
100	*	**scroll**	
105	**-**	**none**	**<SysReq>** not used
106	**+**	*	
107	**none**	**-**	
108	**<Enter>**	**+**	
110	**esc**	**none**	Not on XT layout
112-123	**F1-F12**	**none**	Not on XT layout
124	**none**	**none**	**<PrtScr>** not used
125	**scroll**	**none**	Not on XT layout
126	**none**	**none**	**<Pause>** not used

Building New Binaries

After you have modified an existing keyboard-mapping table, use the following commands to rebuild the corresponding executables:

```
cd /conf/kbd
su root
make
```

If you have created a new keyboard mapping table, you must edit **/conf/kbd/Makefile**. Duplicate an existing entry from the **Makefile**, and change the duplicated name to match the name of your new keyboard-mapping table. After you have finished your editing, build an executable from your source file by simply executing the above series of commands.

To load your new keyboard table, simply type the name of the executable that corresponds to your keyboard-mapping file. For example, if you just built executable **french** from source file **french.c**, type the following command:

```
/conf/kbd/french
```

If the keyboard-support module finds an error, it will print an appropriate message. If it finds no errors, it will update the internal tables of the **vtnkb** keyboard driver, reprogram the keyboard, and print a message of the form:

```
Loaded French AT keyboard table
```

To ensure that the keyboard-support module is loaded automatically when you boot your COHERENT system, edit file **drvld.all** to name the module you wish to use. For example, to ensure that the French keyboard table is loaded automatically when you boot your system, insert the following command into **/etc/drvld.all**:

```
/conf/kbd/french
```

Disabling <Ctrl><Alt><Del>

By convention, function-key 0, when enabled, causes the computer system to reboot. This function key is usually bound to the key sequence **<ctrl><alt><del>**, but you can disable it by setting the value of driver-variable **KBBOOT** to zero. The installation script for configuring your console asks you if you want to turn off this feature; and if so, it sets **KBBOOT** to the correct value.

Problems With Incompatible Keyboards

If you are experiencing problems with respect to key mappings, and you installed one of the loadable keyboard mapping tables from the keyboard selection menu, you may have an incompatible keyboard. Please note that the COHERENT **vtnkb** driver (and loadable tables) only work with well-engineered keyboards, such as those built by IBM, Cherry, MicroSwitch, or Keytronics; it may not work correctly with a poorly engineered "clone" keyboard.

The preferred action is to replace your keyboard with one made by one of the above-named manufacturers. If, however, you wish to use a "clone" keyboard with COHERENT, your best bet is to re-install COHERENT and select the **vtkb** driver instead of **vtnkb**. **vtkb** is not loadable and supports only the U.S., German, and French keyboard layouts. For details on how to replace **vtnkb** with **vtkb**, see the Lexicon entry for **keyboard**.

See Also

device drivers, keyboard, vtkb