A's to Q's about BASH, the Bourne-Again SHell

Chet Ramey translated in sgml by charles vidal.

Summary: A's to Q's about BASH, the Bourne-Again SHell Originator: *het@odin* URL: *ftp://ftp.cwru.edu/pub/bash/FAQ* Maintainer: *het@po.cwru.edu* (Chet Ramey) This is the Bash FAQ, version 3.8, for Bash version 2.04. This document contains a set of frequently-asked questions concerning Bash, the GNU Bourne-Again Shell. Bash is a freely-available command interpreter with advanced features for both interactive use and shell programming.

Another good source of basic information about shells is the collection of FAQ articles periodically posted to comp.unix.shell.

Questions and comments concerning this document should be sent to *het@po.cwru.edu.*

This document is available for anonymous FTP with the URL

ftp://ftp.cwru.edu/pub/bash/FAQ

The Bash home page is *http://cnswww.cns.cwru.edu/˜chet/bash/bashtop.html* Summary: A's to Q's about BASH, the Bourne-Again SHell Originator: *het@odin* URL: *ftp://ftp.cwru.edu/pub/bash/FAQ* Maintainer: *het@po.cwru.edu* (Chet Ramey)

Contents

1 The Basics

1.1 What is it?

Bash is a Unix command interpreter (shell). It is an implementation of the Posix 1003.2 shell standard, and resembles the Korn and System V shells.

Bash contains a number of enhancements over those shells, both for interactive use and shell programming. Features geared toward interactive use include command line editing, command history, job control, aliases, and prompt expansion. Programming features include additional variable expansions, shell arithmetic, and a number of variables and options to control shell behavior.

Bash was originally written by Brian Fox of the Free Software Foundation. The current developer and maintainer is Chet Ramey of Case Western Reserve University.

1.2 What's the latest version?

The latest version is 2.04, first made available on Friday, 17 March 2000.

1.3 Where can I get it?

Bash is the GNU project's shell, and so is available from the master GNU archive site, *ftp.gnu.org*, and its mirrors. The latest version is also available for FTP from *ftp.cwru.edu*. The following URLs tell how to get version 2.04:

ftp://ftp.gnu.org/pub/gnu/bash/bash-2.04.tar.gz *ftp://ftp.cwru.edu/pub/bash/bash*-2.04.tar.gz

Formatted versions of the documentation are available with the URLs:

ftp://ftp.gnu.org/pub/gnu/bash/bash-doc-2.04.tar.gz *ftp://ftp.cwru.edu/pub/bash/bash-doc*-2.04.tar.gz

1.4 On what machines will bash run?

Bash has been ported to nearly every version of UNIX. All you should have to do to build it on a machine for which a port exists is to type 'configure' and then 'make'. The build process will attempt to discover the version of UNIX you have and tailor itself accordingly, using a script created by GNU autoconf.

More information appears in the file 'INSTALL' in the distribution.

1.5 Will bash run on operating systems other than Unix?

Configuration specifics for Unix-like systems such as QNX and LynxOS are included in the distribution. Bash-2.04 should compile and run on Minix 2.0 (patches were contributed), but I don't believe anyone has built bash-2.x on earlier Minix versions yet.

Bash has been ported to versions of Windows implementing the Win32 programming interface. This includes Windows 95 and Windows NT. The port was done by Cygnus Solutions as part of their CYGWIN project. For more information about the project, look at the URL

http:/sourceware.cygnus.com/cygwin

Cygnus originally ported bash-1.14.7, and that port was part of their early GNU-Win32 (the original name) releases. Cygnus has also done a port of bash-2.02.1 to the CYGWIN environment, and it is available as part of their current release. (They may have upgraded by now.)

Bash-2.04 should require no local Cygnus changes to build and run under CYGWIN.

The Cygnus port works only on Intel machines. There is a port of bash (I don't know which version) to the alpha/NT environment available from

*ftp://ftp.gnustep.org//pub/win*32/bash-alpha-nt-1.01.tar.gz

Softway Systems has ported bash-2.01 to their Interix (nee OpenNT) system, a Unix subsystem for NT that replaces the Microsoft POSIX subsystem. Check out *http://www.interix.com* for more information. Some support for Interix has been incorporated into bash, beginning with Bash-2.03. It should be easier to build bash on Interix now, but Interix users should fetch

ftp://ftp.interix.com/pub/tw/unsup/bash.diffs.tar.gz

and read the README.OpenNT file in that archive. It will detail the arguments 'configure' needs to build on Interix. A configure cache file for Interix is in the bash distribution in cross-build/opennt.cache; copy that to 'config.cache' before starting configure.

D. J. Delorie has ported bash-1.14.7 to run under MS-DOS, as part of the DJGPP project. For more information on the project, see

http://www.delorie.com/djgpp/

I picked up a binary of bash-1.14.7 that is purported to work with the DJGPP V2 environment from

*ftp://ftp.simtel.net/pub/simtelnet/gnu/djgpp/v*2gnu/bsh1147b.zip

The corresponding source is

*ftp://ftp.simtel.net/pub/simtelnet/gnu/djgpp/v*2gnu/bsh1147s.zip

Mark Elbrecht <*nowball3@bigfoot.com*> has sent me notice that bash-2.03 has become available for DJGPP V2. The files are available as:

*ftp://ftp.simtel.net/pub/simtelnet/gnu/djgpp/v*2gnu/bsh203b.zip binary *ftp://ftp.simtel.net/pub/simtelnet/gnu/djgpp/v*2gnu/bsh203d.zip documentation *ftp://ftp.simtel.net/pub/simtelnet/gnu/djgpp/v*2gnu/bsh203s.zip source

Mark has begun to work with bash-2.04.

Ports of bash-1.12 and bash-2.0 are available for OS/2 from

*ftp://hobbes.nmsu.edu/pub/os*2/util/shell/bash_112.zip *ftp://hobbes.nmsu.edu/pub/os*2/util/shell/bash-2.0(253).zip

I haven't looked at either, but the second appears to be a binary-only distribution. Beware.

I have received word that Bash (I'm not sure which version, but I believe that it's at least bash-2.02.1) is the standard shell on BeOS.

1.6 How can I build bash with gcc?

Bash configures to use gcc by default if it is available. Read the file INSTALL in the distribution for more information.

1.7 How can I make bash my login shell?

Some machines let you use 'chsh' to change your login shell. Other systems use 'passwd -s' or 'passwd -e'. If one of these works for you, that's all you need. Note that many systems require the full pathname to a shell to appear in /etc/shells before you can make it your login shell. For this, you may need the assistance of your friendly local system administrator.

If you cannot do this, you can still use bash as your login shell, but you need to perform some tricks. The basic idea is to add a command to your login shell's startup file to replace your login shell with bash.

For example, if your login shell is csh or tcsh, and you have installed bash in /usr/gnu/bin/bash, add the following line to /.login:

```
if ( -f /usr/gnu/bin/bash ) exec /usr/gnu/bin/bash --login
```

(the '–login' tells bash that it is a login shell).

It's not a good idea to put this command into /.cshrc, because every csh you run without the '-f' option, even ones started to run csh scripts, reads that file. If you must put the command in /.cshrc, use something like

```
if ( $?prompt ) exec /usr/gnu/bin/bash --login
```

to ensure that bash is exec'd only when the csh is interactive.

If your login shell is sh or ksh, you have to do two things.

First, create an empty file in your home directory named '.bash_profile'. The existence of this file will prevent the exec'd bash from trying to read /.profile, and re-execing itself over and over again. /.bash_profile is the first file bash tries to read initialization commands from when it is invoked as a login shell.

Next, add a line similar to the above to /.profile:

```
[ -f /usr/gnu/bin/bash ] && exec /usr/gnu/bin/bash --login
```

This will cause login shells to replace themselves with bash running as a login shell. Once you have this working, you can copy your initialization code from /.profile to /.bash_profile.

I have received word that the recipe supplied above is insufficient for machines running CDE. CDE has a maze of twisty little startup files, all slightly different.

If you cannot change your login shell in the password file to bash, you will have to (apparently) live with CDE using the shell in the password file to run its startup scripts. If you have changed your shell to bash, there is code in the CDE startup files (on Solaris, at least) to do the right thing.

'dtterm' claims to use $SHELL as the default program to start, so if you can change $SHELL in the CDE startup files, you should be able to use bash in your terminal windows.

Setting DTSOURCEPROFILE in /.dtprofile will cause the 'Xsession' program to read your login shell's startup files. You may be able to use bash for the rest of the CDE programs by setting SHELL to bash in /.dtprofile as well, but I have not tried this.

You can use the above 'exec' recipe to start bash when not logging in with CDE by testing the value of the DT variable:

```
if [ -n "$DT" ]; then
        [ -f /usr/gnu/bin/bash ] && exec /usr/gnu/bin/bash --login
fi
```

1.8 I just changed my login shell to bash, and now I can't FTP into my

machine. Why not?

You must add the full pathname to bash to the file /etc/shells. As noted in the answer to the previous question, many systems require this before you can make bash your login shell.

Most versions of *ftpd* use this file to prohibit 'special' users such as 'uucp' and 'news' from using FTP.

1.9 What's the 'POSIX 1003.2 standard'?

POSIX is a name originally coined by Richard Stallman for a family of open system standards based on UNIX. There are a number of aspects of UNIX under consideration for standardization, from the basic system services at the system call and C library level to applications and tools to system administration and management. Each area of standardization is assigned to a working group in the 1003 series.

The POSIX Shell and Utilities standard has been developed by IEEE Working Group 1003.2 (POSIX.2). It concentrates on the command interpreter interface and utility programs commonly executed from the command line or by other programs. An initial version of the standard has been approved and published by the IEEE, and work is currently underway to update it.

Bash is concerned with the aspects of the shell's behavior defined by POSIX.2. The shell command language has of course been standardized, including the basic flow control and program execution constructs, I/O redirection and pipelining, argument handling, variable expansion, and quoting.

The 'special' builtins, which must be implemented as part of the shell to provide the desired functionality, are specified as being part of the shell; examples of these are 'eval' and 'export'. Other utilities appear in the sections of POSIX.2 not devoted to the shell which are commonly (and in some cases must be) implemented as builtin commands, such as 'read' and 'test'. POSIX.2 also specifies aspects of the shell's interactive behavior as part of the UPE, including job control and command line editing. Only vi-style line editing commands have been standardized; emacs editing commands were left out due to objections.

A10) What is the bash 'posix mode'?

Although bash is an implementation of the POSIX.2 shell specification, there are areas where the bash default behavior differs from that spec. The bash 'posix mode' changes the bash behavior in these areas so that it obeys the spec more closely.

Posix mode is entered by starting bash with the –posix option or executing 'set -o posix' after bash is running.

The specific aspects of bash which change when posix mode is active are listed in the file CWRU/POSIX.NOTES in the bash distribution. They are also listed in a section in the Bash Reference Manual.

2 The latest version

2.1 What's new in version 2.04?

Bash-2.04 contains the following new features (see the manual page for complete descriptions and the CHANGES and NEWS files in the bash-2.04 distribution):

- Programmable word completion with the new 'complete' and 'compgen' builtins; examples are provided in examples/complete/complete-examples
- 'history' has a new '-d' option to delete a history entry
- 'bind' has a new '-x' option to bind key sequences to shell commands
- The prompt expansion code has new '\j' and '\l' escape sequences
- The 'no_empty_cmd_completion' shell option, if enabled, inhibits command completion when TAB is typed on an empty line
- 'help' has a new '-s' option to print a usage synopsis
- New arithmetic operators: var++, var–, ++var, –var, expr1,expr2 (comma)
- New ksh93-style arithmetic for command: for ((expr1 ; expr2; expr3)); do list; done
- 'read' has new options: '-t', '-n', '-d', '-s'
- The redirection code handles several filenames specially: /dev/fd/N, /dev/stdin, /dev/stdout, /dev/stderr
- The redirection code now recognizes /dev/tcp/HOST/PORT and /dev/udp/HOST/PORT and tries to open a TCP or UDP socket, respectively, to the specified port on the specified host
- The ${!prefix*} expansion has been implemented
- A new FUNCNAME variable, which expands to the name of a currently-executing function
- The GROUPS variable is no longer readonly
- A new shopt 'xpg_echo' variable, to control the behavior of echo with respect to backslash-escape sequences at runtime
- The NON_INTERACTIVE_LOGIN_SHELLS #define has returned The version of Readline released with Bash-2.04, Readline-4.1, has several new features as well:

- Parentheses matching is always compiled into readline, and controllable with the new 'blink-matching-paren' variable
- The history-search-forward and history-search-backward functions now leave point at the end of the line when the search string is empty, like reverse-search-history, and forward-search-history
- A new function for applications: rl_on_new_line_with_prompt()
- New variables for applications: rl_already_prompted, and rl_gnu_readline_p

A short feature history dating from bash-2.0:

Bash-2.03 had very few new features, in keeping with the convention that odd-numbered releases provide mainly bug fixes. A number of new features were added to Readline, mostly at the request of the Cygnus folks.

A new shopt option, 'restricted_shell', so that startup files can test whether or not the shell was started in restricted mode Filename generation is now performed on the words between (and) in compound array assignments (this is really a bug fix) OLDPWD is now auto-exported, as POSIX.2 requires ENV and BASH_ENV are read-only variables in a restricted shell Bash may now be linked against an already-installed Readline library, as long as the Readline library is version 4 or newer All shells begun with the '–login' option will source the login shell startup files, even if the shell is not interactive

There were lots of changes to the version of the Readline library released along with Bash-2.03. For a complete list of the changes, read the file CHANGES in the Bash-2.03 distribution.

Bash-2.02 contained the following new features:

a new version of malloc (based on the old GNU malloc code in previous bash versions) that is more page-oriented, more conservative with memory usage, does not 'orphan' large blocks when they are freed, is usable on 64-bit machines, and has allocation checking turned on unconditionally POSIX.2-style globbing character classes ([:alpha:], [:alnum:], etc.) POSIX.2-style globbing equivalence classes POSIX.2-style globbing collating symbols the ksh [[...]] extended conditional command the ksh egrep-style extended pattern matching operators a new 'printf' builtin the ksh-like $(command substitution, which is equivalent to $(cat filename) new tilde prefixes that expand to directories from the directory stack new '**' arithmetic operator to do exponentiation case-insensitive globbing (filename expansion) menu completion a la tcsh 'magic-space' history expansion function like tcsh the readline inputrc 'language' has a new file inclusion directive ($include)

Bash-2.01 contained only a few new features:

new 'GROUPS' builtin array variable containing the user's group list new bindable readline commands: history-and-alias-expand-line and alias-expand-line

Bash-2.0 contained extensive changes and new features from bash-1.14.7. Here's a short list:

- new 'time' reserved word to time pipelines, shell builtins, and shell functions
- one-dimensional arrays with a new compound assignment statement, appropriate expansion constructs and modifications to some of the builtins (read, declare, etc.) to use them
- new quoting syntaxes for ANSI-C string expansion and locale-specific string translation
- new expansions to do substring extraction, pattern replacement, and indirect variable expansion
- new builtins: 'disown' and 'shopt'

- new variables: HISTIGNORE, SHELLOPTS, PIPESTATUS, DIRSTACK, GLOBIGNORE, MACHTYPE, BASH_VERSINFO
- special handling of many unused or redundant variables removed (e.g., $notify, $glob_dot_filenames, $no_exit_on_failed_exec)
- dynamic loading of new builtin commands; many loadable examples provided new prompt expansions: \a, \e, \n, \H, \T, \@, \v, \V
- history and aliases available in shell scripts
- new readline variables: enable-keypad, mark-directories, input-meta, visible-stats, disable-completion, comment-begin
- new readline commands to manipulate the mark and operate on the region
- new readline emacs mode commands and bindings for ksh-88 compatibility
- updated and extended builtins
- new DEBUG trap
- expanded (and now documented) restricted shell mode

implementation stuff:

- autoconf-based configuration
- nearly all of the bugs reported since version 1.14 have been fixed
- most builtins converted to use builtin 'getopt' for consistency
- most builtins use -p option to display output in a reusable form (for consistency)
- grammar tighter and smaller (66 reduce-reduce conflicts gone)
- lots of code now smaller and faster
- test suite greatly expanded

2.2 Are there any user-visible incompatibilities between bash-2.04 and

bash-1.14.7?

There are a few incompatibilities between version 1.14.7 and version 2.04. They are detailed in the file COMPAT in the bash-2.04 distribution.

3 Differences from other Unix shells

3.1 How does bash differ from sh, the Bourne shell?

This is a non-comprehensive list of features that differentiate bash from the SVR4.2 shell. The bash manual page explains these more completely.

Things bash has that sh does not:

- long invocation options
- '!' reserved word to invert pipeline return value
- 'time' reserved word to time pipelines and shell builtins
- the 'function' reserved word
- the 'select' compound command and reserved word
- arithmetic for command: for ((expr1 ; expr2; expr3)); do list; done
- new $'...' and $"..." quoting
- the $(...) form of command substitution
- the $(form of command substitution, equivalent to $(cat filename)
- the ${#param} parameter value length operator
- the ${!param} indirect parameter expansion operator
- the ${!param*} prefix expansion operator
- the ${param:length: $offset]\}parametersubstringoperatorthe\$\{param/pat[/string]\}parameterpatternsubstitutionoperator$

- ```
  expansions to perform substring removal (${p%[%]w}, ${p#[#]w})
  expansion of positional parameters beyond $9 with ${num}
  variables: BASH, BASH_VERSION, BASH_VERSINFO, UID, EUID, REPLY,
             TIMEFORMAT, PPID, PWD, OLDPWD, SHLVL, RANDOM, SECONDS,
             LINENO, HISTCMD, HOSTTYPE, OSTYPE, MACHTYPE, HOSTNAME,
             ENV, PS3, PS4, DIRSTACK, PIPESTATUS, HISTSIZE, HISTFILE,
             HISTFILESIZE, HISTCONTROL, HISTIGNORE, GLOBIGNORE, GROUPS,
             PROMPT_COMMAND, FCEDIT, FIGNORE, IGNOREEOF, INPUTRC,
             SHELLOPTS, OPTERR, HOSTFILE, TMOUT, FUNCNAME, histchars,
             auto_resume
  ```

- DEBUG trap
- variable arrays with new compound assignment syntax
- redirections:
- , &>, >|
  ```

- prompt string special char translation and variable expansion
- auto-export of variables in initial environment
- command search finds functions before builtins
- bash return builtin will exit a file sourced with '.'

```
builtins: cd -/-L/-P, exec -l/-c/-a, echo -e/-E, hash -p.
          export -n/-f/-p/name=value, pwd -L/-P,
          read -e/-p/-a/-t/-n/-d/-s,
          readonly -a/-f/name=value, trap -l, set +o,
          set -b/-m/-o option/-h/-p/-B/-C/-H/-P,
          unset -f/-v, ulimit -m/-p/-u,
          type -a/-p/-t, suspend -f, kill -n,
          test -o optname/s1 == s2/s1 < s2/s1 > s2/-nt/-ot/-ef/-O/-G/-S
```

- bash reads /.bashrc for interactive shells, $ENV for non-interactive
- bash restricted shell mode is more extensive
- bash allows functions and variables with the same name
- brace expansion
- tilde expansion
- arithmetic expansion with $((...)) and 'let' builtin the '[[...]]'*extendedconditionalcommand*
- process substitution
- aliases and alias/unalias builtins
- local variables in functions and 'local' builtin
- readline and command-line editing with programmable completion
- command history and history/fc builtins
- csh-like history expansion
- other new bash builtins: bind, command, compgen, complete, builtin, declare/typeset, dirs, enable, fc, help, history, logout, popd, pushd, disown, shopt, printf
- exported functions
- filename generation when using output redirection (command >a*)
- POSIX.2-style globbing character classes
- POSIX.2-style globbing equivalence classes
- POSIX.2-style globbing collating symbols
- egrep-like extended pattern matching operators

- case-insensitive pattern matching and globbing
- variable assignments preceding commands affect only that command, even for builtins and functions posix mode
- redirection to /dev/fd/N, /dev/stdin, /dev/stdout, /dev/stderr, /dev/tcp/host/port, /dev/udp/host/port

Things sh has that bash does not: uses variable SHACCT to do shell accounting includes 'stop' builtin (bash can use alias stop='kill -s STOP') 'newgrp' builtin turns on job control if called as 'jsh' $TIMEOUT (like bash $TMOUT) '^' is a synonym for '|' new SVR4.2 sh builtins: mldmode, priv

Implementation differences:

- redirection to/from compound commands causes sh to create a subshell
- bash does not allow unbalanced quotes; sh silently inserts them at EOF
- bash does not mess with signal 11
- sh sets (euid, egid) to (uid, gid) if -p not supplied and uid < 100
- bash splits only the results of expansions on IFS, using POSIX.2 field splitting rules; sh splits all words on IFS
- sh does not allow MAILCHECK to be unset (?)
- sh does not allow traps on SIGALRM or SIGCHLD
- bash allows multiple option arguments when invoked (e.g. -x -v); sh allows only a single option argument ('sh -x -v' attempts to open a file named '-v', and, on SunOS 4.1.4, dumps core. On Solaris 2.4 and earlier versions, sh goes into an infinite loop.)
- sh exits a script if any builtin fails; bash exits only if one of the POSIX.2 'special' builtins fails

3.2 How does bash differ from the Korn shell, version ksh88?

Things bash has or uses that ksh88 does not:

- long invocation options
- '!' reserved word
- arithmetic for command: for ((expr1 ; expr2; expr3)); do list; done
- posix mode and posix conformance
- command hashing
- tilde expansion for assignment statements that look like $PATH
- process substitution with named pipes if /dev/fd is not available
- the ${!param} indirect parameter expansion operator

  - the ${!param*} prefix expansion operator
  - the ${param:length: *offset]}parametersubstringoperatorthe${param/pat[/string]}parameterpatternsubstitutionoperator*

- variables:

```
BASH, BASH_VERSION, BASH_VERSINFO, UID, EUID, SHLVL,
                  TIMEFORMAT, HISTCMD, HOSTTYPE, OSTYPE, MACHTYPE,
                  HISTFILESIZE, HISTIGNORE, HISTCONTROL, PROMPT_COMMAND,
                  IGNOREEOF, FIGNORE, INPUTRC, HOSTFILE, DIRSTACK,
                  PIPESTATUS, HOSTNAME, OPTERR, SHELLOPTS, GLOBIGNORE,
                  GROUPS, FUNCNAME, histchars, auto_resume
```

  - prompt expansion with backslash escapes and command substitution
  - redirection: &> (stdout and stderr)
  - more extensive and extensible editing and programmable completion
  - builtins: bind, builtin, command, declare, dirs, echo -e/-E, enable, exec -l/-c/-a, fc -s, export -n/-f/-p, hash, help, history, jobs -x/-r/-s, kill -s/-n/-l, local, logout, popd, pushd, read -e/-p/-a/-t/-n/-d/-s, readonly -a/-n/-f/-p, set -o braceexpand/-o histexpand/-o interactive-comments/ -o notify/-o physical/-o posix/-o hashall/-o onecmd/ -h/-B/-C/-b/-H/-P, set +o, suspend, trap -l, type, typeset -a/-F/-p, ulimit -u, umask -S, alias -p, shopt, disown, printf, complete, compgen
  - '!' csh-style history expansion
  - POSIX.2-style globbing character classes
  - POSIX.2-style globbing equivalence classes
  - POSIX.2-style globbing collating symbols
  - egrep-like extended pattern matching operators
  - case-insensitive pattern matching and globbing
  - '**' arithmetic operator to do exponentiation
  - redirection to /dev/fd/N, /dev/stdin, /dev/stdout, /dev/stderr

Things ksh88 has or uses that bash does not:

  - tracked aliases
  - variables: ERRNO, FPATH, EDITOR, VISUAL
  - co-processes (|&, >, <)
  - weirdly-scoped functions
  - typeset +f to list all function names without definitions
  - text of command history kept in a file, not memory

- builtins: alias -x, cd old new, fc -e -, newgrp, print, read -p/-s/-u/var?prompt, set -A/-o gmacs/ -o bgnice/-o markdirs/-o nolog/-o trackall/-o viraw/-s, typeset -H/-L/-R/-A/-ft/-fu/-fx/-l/-u/-t, whence

Implementation differences:

- ksh runs last command of a pipeline in parent shell context
- bash has brace expansion by default (ksh88 compile-time option)
- bash has fixed startup file for all interactive shells; ksh reads $ENV
- bash has exported functions
- bash command search finds functions before builtins

3.3 Which new features in ksh-93 are not in bash, and which are?

New things in ksh-93 not in bash-2.04:

- associative arrays
- floating point arithmetic
- math library functions
- ${!name*sub*]}*nameofsubscriptforassociativearray*‘.’*isallowedinvariablenamestocreateahierarchicalnamespace*

- more extensive compound assignment syntax

- discipline functions
- ‘sleep’ and ‘getconf’ builtins (bash has loadable versions)
- typeset -n and ‘nameref’ variables
- KEYBD trap
- variables: .sh.edchar, .sh.edmode, .sh.edcol, .sh.edtext, .sh.version, .sh.name, .sh.subscript, .sh.value, HISTEDIT
- backreferences in pattern matching
- print -f (bash uses printf)
- ‘fc’ has been renamed to ‘hist’
- ‘.’ can execute shell functions

New things in ksh-93 present in bash-2.04:

```
for (( expr1; expr2; expr3 )) ; do list; done - arithmetic for command
?:, ++, --, `expr1 , expr2' arithmetic operators
expansions: ${!param}, ${param:offset[:len]}, ${param/pat[/str]},
            ${!param*}
compound array assignment
the `!' reserved word
loadable builtins -- but ksh uses `builtin' while bash uses `enable'
`command', `builtin', `disown' builtins
new $'...' and $"..." quoting
FIGNORE (but bash uses GLOBIGNORE), HISTCMD
set -o notify/-C
changes to kill builtin
read -A (bash uses read -a)
read -t/-d
trap -p
exec -c/-a
`.' restores the positional parameters when it completes
POSIX.2 `test'
umask -S
unalias -a
command and arithmetic substitution performed on PS1, PS4, and ENV
command name completion
ENV processed only for interactive shells
```

4 Why does bash do some things differently than other Unix shells?

4.1 Why does bash run a different version of 'command' than

'which command' says it will?

On many systems, 'which' is actually a csh script that assumes you're running csh. In tcsh, 'which' and its cousin 'where' are builtins. On other Unix systems, 'which' is a perl script that uses the PATH environment variable.

The csh script version reads the csh startup files from your home directory and uses those to determine which 'command' will be invoked. Since bash doesn't use any of those startup files, there's a good chance that your bash environment differs from your csh environment. The bash 'type' builtin does everything 'which' does, and will report correct results for the running shell. If you're really wedded to the name 'which', try adding the following function definition to your .bashrc:

```
which()
{
        builtin type "$@"
}
```

If you're moving from tcsh and would like to bring 'where' along as well, use this function:

```
where()
{
        builtin type -a "$@"
}
```

4.2 Why doesn't bash treat brace expansions exactly like csh?

The only difference between bash and csh brace expansion is that bash requires a brace expression to contain at least one unquoted comma if it is to be expanded. Any brace-surrounded word not containing an unquoted comma is left unchanged by the brace expansion code. This affords the greatest degree of sh compatibility.

Bash, ksh, zsh, and pd-ksh all implement brace expansion this way.

4.3 Why doesn't bash have csh variable modifiers?

Posix has specified a more powerful, albeit somewhat more cryptic, mechanism cribbed from ksh, and bash implements it.

${parameter%word} Remove smallest suffix pattern. The WORD is expanded to produce a pattern. It then expands to the value of PARAMETER, with the smallest portion of the suffix matched by the pattern deleted.

```
x=file.c
echo ${x%.c}.o
-->file.o
```

${parameter%%word}

Remove largest suffix pattern. The WORD is expanded to produce a pattern. It then expands to the value of PARAMETER, with the largest portion of the suffix matched by the pattern deleted.

```
x=posix/src/std
echo ${x%%/*}
-->posix
```

${parameter#word} Remove smallest prefix pattern. The WORD is expanded to produce a pattern. It then expands to the value of PARAMETER, with the smallest portion of the prefix matched by the pattern deleted.

```
x=$HOME/src/cmd
echo ${x#$HOME}
-->/src/cmd
```

${parameter##word} Remove largest prefix pattern. The WORD is expanded to produce a pattern. It then expands to the value of PARAMETER, with the largest portion of the prefix matched by the pattern deleted.

```
x=/one/two/three
echo ${x##*/}
-->three
```

Given

```
a=/a/b/c/d
b=b.xxx
csh                     bash            result
---                     ----            ------
$a:h                    ${a%/*}            /a/b/c
$a:t                    ${a##*/}           d
$b:r                    ${b%.*}            b
$b:e                    ${b##*.}           xxx
```

4.4 How can I make my csh aliases work when I convert to bash?

Bash uses a different syntax to support aliases than csh does. The details can be found in the documentation. We have provided a shell script which does most of the work of conversion for you; this script can be found in ./examples/misc/alias-conv.sh. Here is how you use it:

Start csh in the normal way for you. (e.g., 'csh')

Pipe the output of 'alias' through 'alias-conv.sh', saving the results into 'bash_aliases':

```
alias | alias-conv.sh >bash_aliases
```

Edit 'bash_aliases', carefully reading through any created functions. You will need to change the names of some csh specific variables to the bash equivalents. The script converts $cwd to $PWD, $term to $TERM, $home to $HOME, $user to $USER, and $prompt to $PS1. You may also have to add quotes to avoid unwanted expansion.

For example, the csh alias:

```
alias cd 'cd \!*; echo $cwd'
```

is converted to the bash function:

```
cd () { command cd "$@"; echo $PWD ; }
```

The only thing that needs to be done is to quote $PWD:

```
cd () { command cd "$@"; echo "$PWD" ; }
```

Merge the edited file into your /.bashrc.

There is an additional, more ambitious, script in examples/misc/cshtobash that attempts to convert your entire csh environment to its bash equivalent. This script can be run as simply 'cshtobash' to convert your normal interactive environment, or as 'cshtobash /.login' to convert your login environment.

4.5 How can I pipe standard output and standard error from one command to

another, like csh does with '|&'?

Use

```
command 2>&1 | command2
```

The key is to remember that piping is performed before redirection, so file descriptor 1 points to the pipe when it is duplicated onto file descriptor 2.

4.6 Now that I've converted from ksh to bash, are there equivalents to

ksh features like autoloaded functions and the 'whence' command?

There are features in ksh-88 and ksh-93 that do not have direct bash equivalents. Most, however, can be emulated with very little trouble.

ksh-88 feature Bash equivalent ———— ———— compiled-in aliases set up aliases in .bashrc; some ksh aliases are bash builtins (hash, history, type) coprocesses named pipe pairs (one for read, one for write) typeset +f declare -F cd, print, whence function substitutes in examples/functions/kshenv autoloaded functions examples/functions/autoload is the same as typeset -fu read var?prompt read -p prompt var

ksh-93 feature Bash equivalent ———— ———— sleep, getconf Bash has loadable versions in examples/loadables ${.sh.version} $BASH_VERSION print -f printf hist alias fc=hist $HISTEDIT $FCEDIT

5 How can I get bash to do certain things, and why does bash do

things the way it does?

5.1 Why is the bash builtin 'test' slightly different from /bin/test?

The specific example used here is [! x -o x], which is false.

Bash's builtin 'test' implements the Posix.2 spec, which can be summarized as follows (the wording is due to David Korn):

Here is the set of rules for processing test arguments.

- 0 Args: False
- 1 Arg: True iff argument is not null.
- 2 Args: If first arg is !, True iff second argument is null. If first argument is unary, then true if unary test is true Otherwise error.
- 3 Args: If second argument is a binary operator, do binary test of $1 $3 If first argument is !, negate two argument test of $2 $3 If first argument is '(' and third argument is ')', do the one-argument test of the second argument. Otherwise error.
- 4 Args: If first argument is !, negate three argument test of $2 $3 $4. Otherwise unspecified

- 5 or more Args: unspecified. (Historical shells would use their current algorithm).

The operators -a and -o are considered binary operators for the purpose of the 3 Arg case.

As you can see, the test becomes (not (x or x)), which is false.

5.2 Why does bash sometimes say 'Broken pipe'?

If a sequence of commands appears in a pipeline, and one of the reading commands finishes before the writer has finished, the writer receives a SIGPIPE signal. Many other shells special-case SIGPIPE as an exit status in the pipeline and do not report it. For example, in:

```
ps -aux | head
```

'head' can finish before 'ps' writes all of its output, and ps will try to write on a pipe without a reader. In that case, bash will print 'Broken pipe' to stderr when ps is killed by a SIGPIPE.

You can build a version of bash that will not report SIGPIPE errors by uncommenting the definition of DONT_REPORT_SIGPIPE in the file config-top.h.

5.3 When I have terminal escape sequences in my prompt, why does bash

wrap lines at the wrong column?

Readline, the line editing library that bash uses, does not know that the terminal escape sequences do not take up space on the screen. The redisplay code assumes, unless told otherwise, that each character in the prompt is a 'printable' character that takes up one character position on the screen.

You can use the bash prompt expansion facility (see the PROMPTING section in the manual page) to tell readline that sequences of characters in the prompt strings take up no screen space.

Use the \[escape to begin a sequence of non-printing characters, and the \] escape to signal the end of such a sequence.

5.4 If I pipe the output of a command into 'read variable', why doesn't

the output show up in $variable when the read command finishes?

This has to do with the parent-child relationship between Unix processes.

Each element of a pipeline runs in a separate process, a child of the shell running the pipeline. A subprocess cannot affect its parent's environment. When the 'read' command sets the variable to the input, that variable is set only in the subshell, not the parent shell. When the subshell exits, the value of the variable is lost.

Many pipelines that end with 'read variable' can be converted into command substitutions, which will capture the output of a specified command. The output can then be assigned to a variable:

```
grep ^gnu /usr/lib/news/active | wc -l | read ngroup
```

can be converted into

```
        ngroup=$(grep ^gnu /usr/lib/news/active | wc -l)
```

This does not, unfortunately, work to split the text among multiple variables, as read does when given multiple variable arguments. If you need to do this, you can either use the command substitution above to read the output into a variable and chop up the variable using the bash pattern removal expansion operators or use some variant of the following approach.

Say /usr/local/bin/ipaddr is the following shell script:

```
#! /bin/sh
host `hostname` | awk '/address/ {print $NF}'
```

Instead of using

```
        /usr/local/bin/ipaddr | read A B C D
```

to break the local machine's IP address into separate octets, use

```
        OIFS="$IFS"
        IFS=.
        set -- $(/usr/local/bin/ipaddr)
        IFS="$OIFS"
        A="$1" B="$2" C="$3" D="$4"
```

Beware, however, that this will change the shell's positional parameters. If you need them, you should save them before doing this.

This is the general approach – in most cases you will not need to set $IFS to a different value.

5.5 I have a bunch of shell scripts that use backslash-escaped characters

in arguments to 'echo'. Bash doesn't interpret these characters. Why not, and how can I make it understand them?

This is the behavior of echo on most Unix System V machines.

The bash builtin 'echo' is modeled after the 9th Edition Research Unix version of 'echo'. It does not interpret backslash-escaped characters in its argument strings by default; it requires the use of the -e option to enable the interpretation. The System V echo provides no way to disable the special characters; the bash echo has a -E option to disable them.

There is a configuration option that will make bash behave like the System V echo and interpret things like '\t' by default. Run configure with the –enable-usg-echo-default option to turn this on. Be aware that this will cause some of the tests run when you type 'make tests' to fail.

There is a shell option, 'xpg_echo', settable with 'shopt' that will change the behavior of echo at runtime. Enabling this option turns on expansion of backslash-escape sequences.

5.6 Why doesn't a while or for loop get suspended when I type ^Z?

This is a consequence of how job control works on Unix. The only thing that can be suspended is the process group. This is a single command or pipeline of commands that the shell forks and executes.

When you run a while or for loop, the only thing that the shell forks and executes are any commands in the while loop test and commands in the loop bodies. These, therefore, are the only things that can be suspended when you type ^Z.

If you want to be able to stop the entire loop, you need to put it within parentheses, which will force the loop into a subshell that may be stopped (and subsequently restarted) as a single unit.

5.7 What about empty for loops in Makefiles?

It's fairly common to see constructs like this in automatically-generated Makefiles:

```
SUBDIRS = @SUBDIRS@
        ...
subdirs-clean:
        for d in $SUBDIRS; do \
                ( cd $$d && ${MAKE} ${MFLAGS} clean ) \
        done
```

When SUBDIRS is empty, this results in a command like this being passed to bash:

```
        for d in ; do
                ( cd $$d && ${MAKE} ${MFLAGS} clean )
        done
```

This is a syntax error. If the reserved word 'in' is present, a word must follow it before the semicolon or newline. The language in the manual page referring to the list of words being empty refers to the list after it is expanded. There must be at least one word following the 'in' when the construct is parsed.

The idiomatic Makefile solution is something like:

```
SUBDIRS = @SUBDIRS@
subdirs-clean:
        subdirs=$SUBDIRS ; for d in $$subdirs; do \
                ( cd $$d && ${MAKE} ${MFLAGS} clean ) \
        done
```

6 Things to watch out for on certain Unix versions

6.1 Why can't I use command line editing in my 'cmdtool'?

The problem is 'cmdtool' and bash fighting over the input. When scrolling is enabled in a cmdtool window, cmdtool puts the tty in 'raw mode' to permit command-line editing using the mouse for applications that cannot do it themselves. As a result, bash and cmdtool each try to read keyboard input immediately, with neither getting enough of it to be useful.

This mode also causes cmdtool to not implement many of the terminal functions and control sequences appearing in the 'sun-cmd' termcap entry. For a more complete explanation, see that file examples/suncmd.termcap in the bash distribution.

'xterm' is a better choice, and gets along with bash much more smoothly.

If you must use cmdtool, you can use the termcap description in examples/suncmd.termcap. Set the TERMCAP variable to the terminal description contained in that file, i.e.

TERMCAP='Mu|sun-cmd:am:bs:km:pt:li#34:co#80:cl=^L:ce=\E[K:cd=\E[J:rs=\E[s:'

Then export TERMCAP and start a new cmdtool window from that shell. The bash command-line editing should behave better in the new cmdtool. If this works, you can put the assignment to TERMCAP in your bashrc file.

6.2 I built bash on Solaris 2. Why do globbing expansions and filename

completion chop off the first few characters of each filename?

This is the consequence of building bash on SunOS 5 and linking with the libraries in /usr/ucblib, but using the definitions and structures from files in /usr/include.

The actual conflict is between the dirent structure in /usr/include/dirent.h and the struct returned by the version of 'readdir' in libucb.a (a 4.3-BSD style 'struct direct').

Make sure you've got /usr/ccs/bin ahead of /usr/ucb in your $PATH when configuring and building bash. This will ensure that you use /usr/ccs/bin/cc or acc instead of /usr/ucb/cc and that you link with libc before libucb.

If you have installed the Sun C compiler, you may also need to put /usr/ccs/bin and /opt/SUNWspro/bin into your $PATH before /usr/ucb.

6.3 Why does bash dump core after I interrupt username completion or

' user' tilde expansion on a machine running NIS?

This is a famous and long-standing bug in the SunOS YP (sorry, NIS) client library, which is part of libc.

The YP library code keeps static state – a pointer into the data returned from the server. When YP initializes itself (setpwent), it looks at this pointer and calls free on it if it's non-null. So far, so good.

If one of the YP functions is interrupted during getpwent (the exact function is interpretwithsave()), and returns NULL, the pointer is freed without being reset to NULL, and the function returns. The next time getpwent is called, it sees that this pointer is non-null, calls free, and the bash free() blows up because it's being asked to free freed memory.

The traditional Unix mallocs allow memory to be freed multiple times; that's probably why this has never been fixed. You can run configure with the '–without-gnu-malloc' option to use the C library malloc and avoid the problem.

6.4 I'm running SVR4.2. Why is the line erased every time I type '@'?

The '@' character is the default 'line kill' character in most versions of System V, including SVR4.2. You can change this character to whatever you want using 'stty'. For example, to change the line kill character to control-u, type

```
stty kill ^U
```

where the '^' and 'U' can be two separate characters.

6.5 Why does bash report syntax errors when my C News scripts use a

redirection before a subshell command?

The actual command in question is something like

```
< file ( command )
```

According to the grammar given in the POSIX.2 standard, this construct is, in fact, a syntax error. Redirections may only precede 'simple commands'. A subshell construct such as the above is one of the shell's 'compound commands'. A redirection may only follow a compound command.

This affects the mechanical transformation of commands that use 'cat' to pipe a file into a command (a favorite Useless-Use-Of-Cat topic on comp.unix.shell). While most commands of the form

```
cat file | command
```

can be converted to '< file command', shell control structures such as loops and subshells require 'command < file'.

The file CWRU/sh-redir-hack in the bash-2.04 distribution is an (unofficial) patch to parse.y that will modify the grammar to support this construct. It will not apply with 'patch'; you must modify parse.y by hand. Note that if you apply this, you must recompile with -DREDIRECTION_HACK. This introduces a large number of reduce/reduce conflicts into the shell grammar.

6.6 Why can't I use vi-mode editing on Red Hat Linux 6.1?

The short answer is that Red Hat screwed up.

The long answer is that they shipped an /etc/inputrc that only works for emacs mode editing, and then screwed all the vi users by setting INPUTRC to /etc/inputrc in /etc/profile.

The short fix is to do one of the following: remove or rename /etc/inputrc, set INPUTRC= /.inputrc in /.bashrc (or .bash_profile, but make sure you export it if you do), remove the assignment to INPUTRC from /etc/profile, add

```
set keymap emacs
```

to the beginning of /etc/inputrc, or bracket the key bindings in /etc/inputrc with these lines

```
$if mode=emacs
        [...]
$endif
```

7 How can I get bash to do certain common things?

7.1 How can I get bash to read and display eight-bit characters?

This is a process requiring several steps.

First, you must ensure that the ‘physical’ data path is a full eight bits. For xterms, for example, the ‘vt100’ resources ‘eightBitInput’ and ‘eightBitOutput’ should be set to ‘true’.

Once you have set up an eight-bit path, you must tell the kernel and tty driver to leave the eighth bit of characters alone when processing keyboard input. Use ‘stty’ to do this:

```
stty cs8 -istrip -parenb
```

For old BSD-style systems, you can use

```
stty pass8
```

You may also need

```
stty even odd
```

Finally, you need to tell readline that you will be inputting and displaying eight-bit characters. You use readline variables to do this. These variables can be set in your .inputrc or using the bash ‘bind’ builtin. Here’s an example using ‘bind’:

```
bash$ bind 'set convert-meta off'
bash$ bind 'set meta-flag on'
bash$ bind 'set output-meta on'
```

The ‘set’ commands between the single quotes may also be placed in /.inputrc.

7.2 How do I write a function 'x' to replace builtin command 'x', but

still invoke the command from within the function?

This is why the 'command' and 'builtin' builtins exist. The 'command' builtin executes the command supplied as its first argument, skipping over any function defined with that name. The 'builtin' builtin executes the builtin command given as its first argument directly.

For example, to write a function to replace 'cd' that writes the hostname and current directory to an xterm title bar, use something like the following:

```
cd()
{
        builtin cd "$@" && xtitle "$HOST: $PWD"
}
```

This could also be written using 'command' instead of 'builtin'; the version above is marginally more efficient.

7.3 How can I find the value of a shell variable whose name is the value

of another shell variable?

Versions of Bash newer than Bash-2.0 support this directly. You can use

```
${!var}
```

For example, the following sequence of commands will echo 'z':

```
var1=var2
var2=z
echo ${!var1}
```

For sh compatibility, use the 'eval' builtin. The important thing to remember is that 'eval' expands the arguments you give it again, so you need to quote the parts of the arguments that you want 'eval' to act on.

For example, this expression prints the value of the last positional parameter:

```
eval echo \"\$\{$#\}\"
```

The expansion of the quoted portions of this expression will be deferred until 'eval' runs, while the '$#' will be expanded before 'eval' is executed. In versions of bash later than bash-2.0,

```
echo ${!#}
```

does the same thing.

7.4 How can I make the bash 'time' reserved word print timing output that

looks like the output from my system's /usr/bin/time?

The bash command timing code looks for a variable 'TIMEFORMAT' and uses its value as a format string to decide how to display the timing statistics.

The value of TIMEFORMAT is a string with '%' escapes expanded in a fashion similar in spirit to printf(3). The manual page explains the meanings of the escape sequences in the format string.

If TIMEFORMAT is not set, bash acts as if the following assignment had been performed:

```
TIMEFORMAT=$'\nreal\t%3lR\nuser\t%3lU\nsys\t%3lS'
```

The POSIX.2 default time format (used by 'time -p command') is

```
TIMEFORMAT=$'real %2R\nuser %2U\nsys %2S'
```

The BSD /usr/bin/time format can be emulated with:

```
TIMEFORMAT=$'\t%1R real\t%1U user\t%1S sys'
```

The System V /usr/bin/time format can be emulated with:

```
TIMEFORMAT=$'\nreal\t%1R\nuser\t%1U\nsys\t%1S'
```

The ksh format can be emulated with:

```
TIMEFORMAT=$'\nreal\t%2lR\nuser\t%2lU\nsys\t%2lS'
```

7.5 How do I get the current directory into my prompt?

Bash provides a number of backslash-escape sequences which are expanded when the prompt string (PS1 or PS2) is displayed. The full list is in the manual page.

The \w expansion gives the full pathname of the current directory, with a tilde (' ') substituted for the current value of $HOME. The \W expansion gives the basename of the current directory. To put the full pathname of the current directory into the path without any tilde subsitution, use $PWD. Here are some examples:

```
PS1='\w$ '      # current directory with tilde
PS1='\W$ '      # basename of current directory
PS1='$PWD$ '    # full pathname of current directory
```

The single quotes are important in the final example to prevent $PWD from being expanded when the assignment to PS1 is performed.

7.6 How can I rename "*.foo" to "*.bar"?

Use the pattern removal functionality described in D3. The following 'for' loop will do the trick:

```
for f in *.foo; do
        mv $f ${f%foo}bar
done
```

7.7 How can I translate a filename from uppercase to lowercase?

The script examples/functions/lowercase, originally written by John DuBois, will do the trick. The converse is left as an exercise.

7.8 How can I write a filename expansion (globbing) pattern that will match

all files in the current directory except "." and ".."?

You must have set the 'extglob' shell option using 'shopt -s extglob' to use this:

```
echo .!(.|) *
```

A solution that works without extended globbing is given in the Unix Shell FAQ, posted periodically to comp.unix.shell.

8 Where do I go from here?

8.1 How do I report bugs in bash, and where should I look for fixes and

advice?

Use the 'bashbug' script to report bugs. It is built and installed at the same time as bash. It provides a standard template for reporting a problem and automatically includes information about your configuration and build environment.

'bashbug' sends its reports to bug-*ash@gnu.org*, which is a large mailing list gatewayed to the usenet newsgroup gnu.bash.bug.

Bug fixes, answers to questions, and announcements of new releases are all posted to gnu.bash.bug. Discussions concerning bash features and problems also take place there.

To reach the bash maintainers directly, send mail to bash-*aintainers@gnu.org.*

8.2 What kind of bash documentation is there?

First, look in the doc directory in the bash distribution. It should contain at least the following files:

bash.1 an extensive, thorough Unix-style manual page builtins.1 a manual page covering just bash builtin commands bashref.texi a reference manual in GNU tex'info format bashref.info an info version of the reference manual FAQ this file article.ms text of an article written for The Linux Journal readline.3 a man page describing readline

Postscript, HTML, and ASCII files created from the above source are available in the documentation distribution.

There is additional documentation available for anonymous FTP from host *ftp.cwru.edu* in the 'pub/bash' directory.

Cameron Newham and Bill Rosenblatt have written a book on bash, published by O'Reilly and Associates. The book is based on Bill Rosenblatt's Korn Shell book. The title is "Learning the Bash Shell", and the ISBN number is 1-56592-147-X. Look for it in fine bookstores near you. This book covers bash-1.14, but has an appendix describing some of the new features in bash-2.0.

A second edition of this book is available, published in January, 1998. The ISBN number is 1-56592-347-2. Look for it in the same fine bookstores or on the web.

8.3 What's coming in future versions?

These are features I plan to include in a future version of bash.

a bash debugger (a minimally-tested version is included with bash-2.04) associative arrays

8.4 What's on the bash 'wish list' for future versions?

These are features that may or may not appear in a future version of bash.

breaking some of the shell functionality into embeddable libraries a module system like zsh's, using dynamic loading like builtins better internationalization using GNU 'gettext' an option to use external files for the long 'help' text date-stamped command history a bash programmer's guide with a chapter on creating loadable builtins a better loadable interface to perl with access to the shell builtins and variables (contributions gratefully accepted)

8.5 When will the next release appear?

The next version will appear sometime in 2000 or 2001. Never make predictions.