The Wish Shell - an Internal Look

W. Toomey

Abstract

This report details the data structures and algorithms used in the Wish shell. Note that this report covers an earlier version of Wish than the current version (Version 2.0.41), and does not cover all of the code.
Also note that Wish is a rewrite of another shell, Clam 1.x. Early versions of Wish borrowed some code from Clam, and so it was known as Clam 2.x, but the name was changed as the amount of shared code became minute.
In fact, this whole document is still in rough draft form. However, it's the only guide that I still have to the internals of the 1996 code.

Warren Toomey, e-mail: wkt@tuhs.org
Callum Gibson, e-mail: callum.gibson@db.com

Contents

1 Design Decisions

Version 2 of the Clam shell is a complete rewrite of the shell written by Callum Gibson for his Honours project. As such, Clam 2.0 was written with a new set of design goals in mind. These are:

- Clam 2.0 must be small, in source code and especially executable size. It should fit into the 64K code and 64K data limit under PC-Minix 1.5. Therefore, we opted to rely on as few C library routines as possible; this is the reason that no `stdio` libraries are used. Also, much of the rewritten code is smaller in size than Clam 1.4.
- The shell must be fast, especially in the areas of command line editing, parsing, command line extensions, and finding aliases, variables, history and jobs. The new method of metacharacter expansion, for example, involves less string copying, and helps speed the shell up immensely.
- This version should be portable to **every** Unix and Unix-like Operating System. Examples are: SysVR2, SysVR4, BSD4.x, Xenix, SunOS 3.x and 4.x, Ultrix, Minix and Coherent. Making a shell portable across so many OS platforms is very difficult, and explains the number of `#ifdef`s in the Clam code. Not only should the shell be portable, but it should exhibit the same 'look and feel' on all platforms. This is technically impossible; for example, it is impossible to provide true job control under Minix, Coherent and SysVR2. However, in this instance, we have made some of the job control features available under these systems.
- Clam 2.0 should be POSIX P1003.2 compatible, i.e. it should use the POSIX system interface routines where possible. This is reasonably straight-forward, except when the rationale behind the interface routines is very different from other Unix systems, for example in the area of job control.
- The code should be compileable by both Ansi C and traditional C compilers. This is also reasonably straight-forward.
- The shell should be modular, and be broken up into several blocks, each of which performs a set of logically connected tasks. The inter-module interfaces should be well defined. Clam 1.4 was quite modular, and so retaining modularity in this version was not difficult.
- The code should be well documented and well structured. It should be suitable for 3rd year University students to read and understand. We can only hope that this is so.
- The shell should be easy to compile. To this end, all system dependencies are concentrated in the Makefile, and one header file, and both are documented heavily.
- This version should be compatible with Clam 1.4 where possible, even though it is completely rewritten.
- Clam 2.0 should be useful to the user. To this end, new features have been added to make it more useful.

2 Overview of Clam's Code

As described in the design section, Clam is broken up into several modules, each of which performs a set of logically connected tasks. Each of these modules will be described fully in this report, but a brief overview of each module is given below.
`Alias.c` holds the routines that deal with shell aliases; these are commands that get translated by the shell into another command (or commands). An example is the following alias:

```
alias ls '/bin/ls -lg $*'
```

This aliases the command `ls` to `/bin/ls -lg` with the same arguments as were given to `ls`.

`Builtin.c` finds and executes the builtin routines in the shell. Commands such as `cd`, `source`. `bg` are all shell builtins. Many of the builtins are in files that they are more logically connected with; for example, the `unalias` builtin is in the `alias.c` file.

`Clex.c` performs the command line expansion and selection routines. These routines expand or display words that match the partial word at the end of the command line. `Clex` looks up binaries on the path, files, users, builtins, etc.

`Comlined.c` is the command line editor, which is a very flexible line editor based on Emacs. This version has a sophisticated key binding routine which allows the editor keys to be rebound, giving the user a vi-like (or any editor-like) command line editor.

`Exec.c` actually executes the command, redirecting input/output, and either `exec()`ing or using aliases or builtins.

`File.c` provides file routines, such as file input to the shell, and the `source` builtin.

An overall block diagram with general execution flow is given in Figure 1. The bold lines show the usual execution flow, with `main` controlling the main loop though the command line editor, the metacharacter expansion, the command parsing and execution, and the process job control.

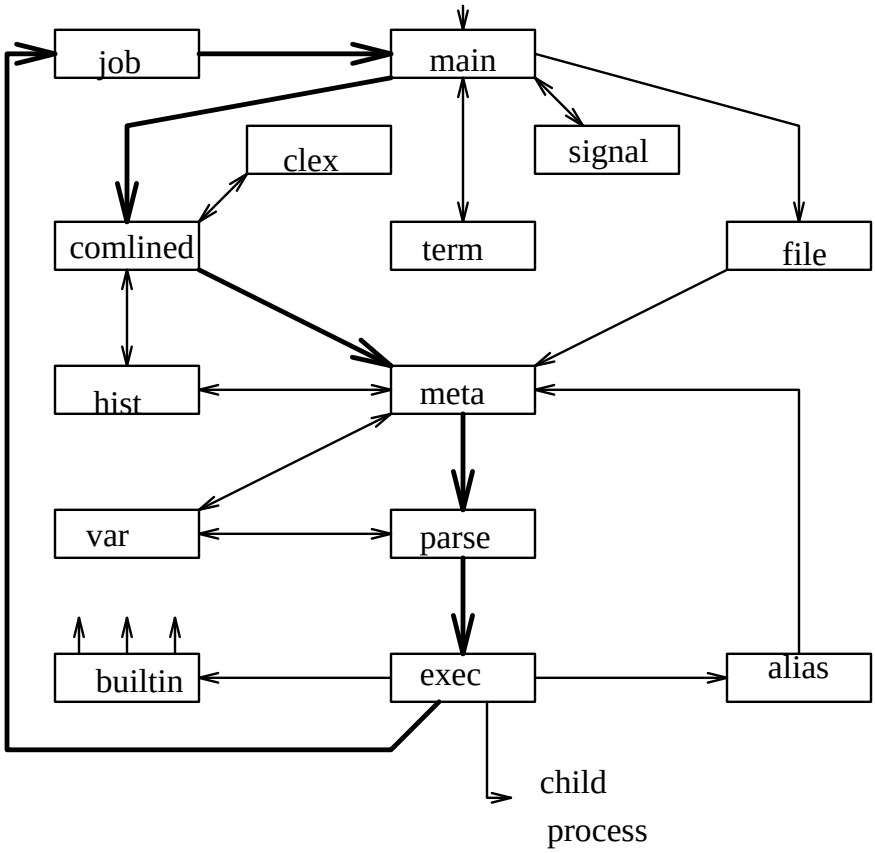


Figure 1: Overall Clam Diagram and Execution Flow

3 C Preprocessor Defines

This describes version 2.0.36 of Clam.
`#define`'d names are used heavily throughout Clam to choose what code to compile according to the Operating System it is being compiled on.

3.1 Unix Classes

There are five major defined names that describe the operating system; only one of these can be defined at any one time. They are:

ATT The system is AT&T's System V Release 2, or similar, such as Xenix. There are no Berkeley enhancements such as job control. Terminfo is used in the terminal routines.

COHERENT The system is Coherent. At the time of writing, this can be defined, but no code uses it. Any Coherent users who wish to port Clam should contact the authors.

MINIX The system is Minix 1.5. Minix is a bit of a hodgepodge of Unix – it uses termcap and some Berkeley `ioctl()`s, but there is no job control.

POSIX The system is POSIX P1003.2 compliant. Clam will use the defined POSIX routines instead of the more usual SysV or BSD routines, and things like job control is handled differently.

UCB The system is BSD4.x, or a derivative of BSD. The usual Berkeley enhancements such as job control are available.

Currently these five defines suffice to cover the major Unix classes; other defines will be invented as we see the need for them.
Some systems, such as Xenix, are similar to one of the major classes above, but are different enough to need different code now and then. Therefore, each Operating System has a defined name that will cause one of the major system names to be defined as well[1]. The following names are currently defined:

GENSYSV A Generic System V Release 2 system. Implies ATT.

SYSVPYR Pyramid dual universe OSx in the System V universe. Implies ATT.

XENIX The Xenix system. Implies ATT.

GENBSD A Generic BSD4.x system. Implies UCB.

BSDPYR Pyramid dual universe OSx in the BSD universe. Implies UCB.

SUN SunOS 3.x and 4.x. Implies UCB. Note that because SunOS 4.x is POSIX compliant, expect to see a separate Sun 4.x name in the future.

3.2 Defined Features

Clam uses defined names to turn on/off compilation of features in the shell. Several of the above system names cause features (such as job control) to be automatically turned on/off – see `header.h` for more information.
The following features can be defined:

PROTO Use Ansi C prototypes when compiling the source. This is turned on when __STDC__ has a non-zero definition; if your compiler has prototypes but doesn't define __STDC__ to be non-zero, define this.

JOB Job control. This needs the BSD (or POSIX) job control signals, and will definitely compile or work without these. It is best to let your defined system name turn this on/off.

POSIXJOB POSIX job control. This automatically defines JOB. Again, let your defined system name turn this on/off.

DEBUG With this defined, debugging code and output statements are compiled into the shell. These tend to be extremely verbose, and usually it's a good idea to turn debugging on only in the file(s) that you are interested in.

Some Clam routines have a variable numbers of arguments; these are compiled to use `stdarg.h` or `varargs.h` or neither, if STDARG, VARARGS or neither are defined. Again, these are usually turned on/off automatically.

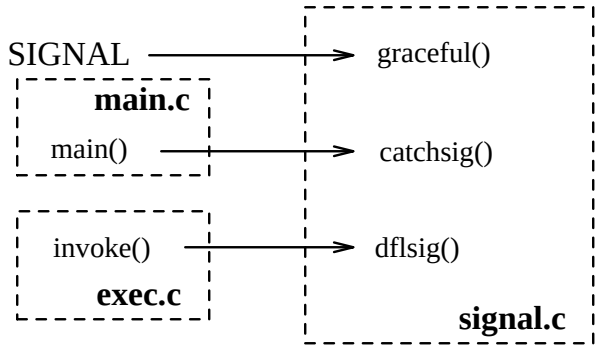
[1] This is done in `header.h`.

3.3 Miscellaneous Defines

The only other defined name of note is NOTYET, which is never defined, and is used to "comment out" pieces of code for testing and quick code changes. Hopefully there are no occurrences of NOTYET in the released code.

4 Signal Handling

This section refers to Clam version 2.0.36.
Every shell has to do some signal handling, to ensure that its children begin with default signals, and to handle the special job control signals if there are any. The file `shell.c` performs the signal handling manipulation, except for the job control signals which are handled in `job.c`.
The general execution flow is given in Figure 2.



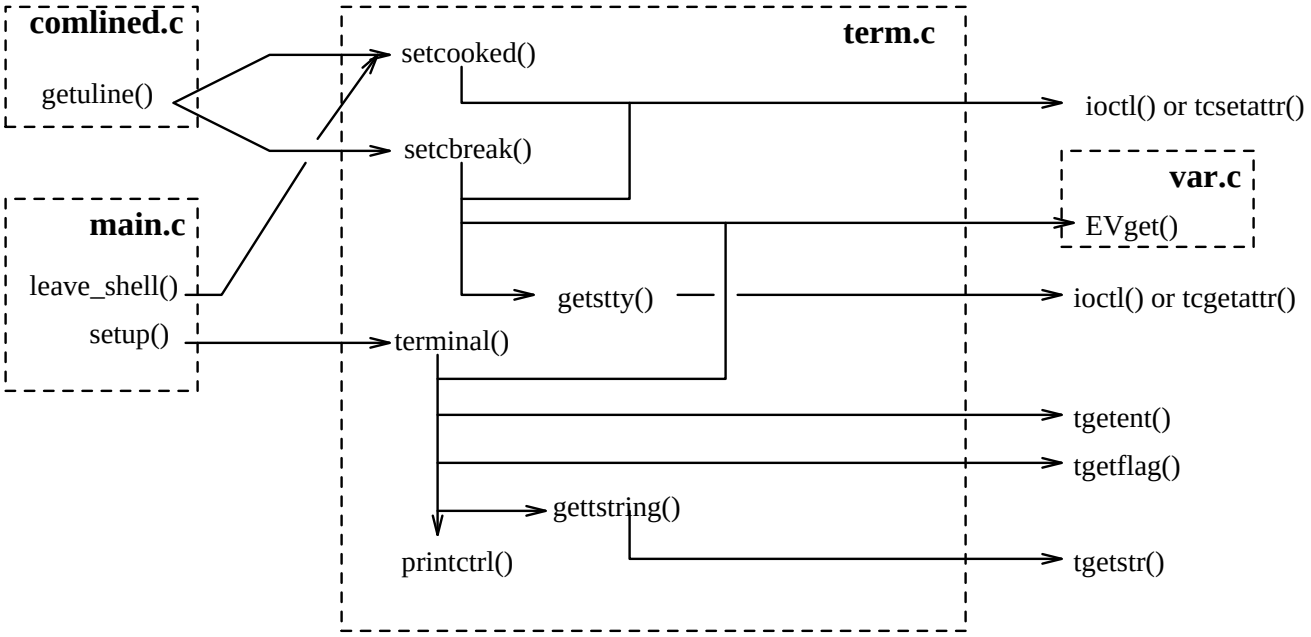
Signal Handling (Version 25)

Figure 2: Signal Handling Execution Flow

When the shell starts, `catchsig()` is called, which sets the signal handler for most of the signals to be `graceful()`; when this routine receives a signal, it simply prints out the signal number, and kills the shell. `Catchsig()` also sets the signal handler for the job control signal SIGTSTP to `stopjob()`, which is detailed in the Job Control section.
Just before a new child is `exec()`'d, `dflsig()` is called, which sets all of the child's signal handlers back to SIG_DFL.
Note that both `catchsig()` and `dflsig()` depend on the word MAXSIG begin defined to the maximum number of signals available on the system. This should be defined automatically in `header.h`.

5 Terminal Handling

This section refers to Clam version 2.0.36.
the file `term.c` handles two parts of the terminal-related work in Clam: obtaining termcap/terminfo strings for use by the command line editor, and setting the mode of the terminal to either cooked or cbreak. The third type of terminal-related work, controlling the process group of the terminal, is handled by the job control code, and is discussed in the relevant section.
The general execution flow is given in Figure 2.



Terminal Handling (Version 36)

Figure 3: Terminal Handling Execution Flow

5.1 Termcap/Terminfo Information

The command line editor needs to able to move the cursor around the screen, delete characters and perform scrolling. To do this, it needs to know about the special escape sequences for the terminal upon which it will be doing this work. These can be obtained from the `termcap` library routines[2].
Clam uses the following global character pointers to point to the termcap strings:

- `bs` holds the sequence used to destructively backspace.
- `nd` holds the sequence which to morve the cursor right non-destructively.
- `cl` holds the sequence which homes the cursor and clears the screen.
- `cd` holds the sequence which clears the screen from the curdor down.
- `up` holds the sequence which moves the cursor up one line.
- `so` holds the standout sequence, which usually causes text to go bold, inversed, or underlined.
- `se` holds the sequence to terminate standout mode, bringing the mode back to normal.
- `beep` holds the "bell" sequence, which is used to indicate a user error while in the command line editor.

`terminal()` is called when Clam starts to set up these strings. It first gets the value of the environment variable `TERM`, and calls `tgetent()` to get all the termcap strings for that terminal. Then `gettstring()` is called with the name of the individual strings wanted. It uses `tgetstr()` to get the strings, checks to see if there is a string, and removes any time delay characters from the beginning of the string.
Non-BSD systems usually have a `terminfo` library rather than a `termcap` library. Fortunately, the former emulates enough of the routines in the latter to allow Clam to compile with no `#ifdef`s in this part of the code.

[2] As far as I can tell, Clam should be portable to a Version 7 Unix system, as long as the `termcap` and `dirent` libraries exist on the system.

5.2 Cooked and Raw Modes

blah blah

6 Metacharacter Expansion

This section refers to Clam version 2.0.38.
As with all shells, Clam provides metacharacter expansion to minimise typing. The metacharacter section of clam is kept in `meta.c`. It takes a line typed by a user (or from a file), alters the content of the line according to the metacharacters used on the line, and presents the parser with a series of 'tokens' that are more easily understood by the parser.

6.1 The Metacharacters

Most shell provide many sorts of metacharacters. The list of metacharacters and their order of expansion in Clam is given in following table:

Single quotes (`'`): Single quotes are used to turn *all* metacharacter expansion off inside the quotes. All the characters inside the quotes will be left 'as is' and passed directly to the parser.

Back Quotes (`` ` ``): Back quotes work in exactly the same way as single quotes. However, instead of protecting the contents inside the back quotes, the contents are *executed* (as if they were a command) and the output from the command substituted on the line. For example, the command `size 'which clam'` finds the program '`clam`' on your path, and determines the sizes of the sections of the executable.

Backslash: The backslash is used to directly quote the *next* character, turning off any metacharacter meaning for that letter. As an example, to pass the string `Hello I'm a crazy guy!` to the parser without any meta expansion, you would type it as `'Hello I'\''m a crazy guy!'`.

Whitespace: Whitespace characters (spaces and tabs) are used to separate the input line into words for the parser; each whitespace separated word becomes a separate *argument* to the program that is eventually run, including the zeroth argument i.e the name of the program. Quoting characters such as the ones above are ignored when producing words, so that `'Testing  testing'\,'one''two''three'` is in fact one word, which happens to contain spaces.

Bang (`!`): The bang character is used to retrieve previous commands from the history list; for example, `!vi` will be replaced by the last command that began with the letters `vi`, `!5` will be replaced by the 5th command; `!!` means the last command and `!$` will eventually mean all of the last line except for the first word. The bang is a metacharacter everywhere on the input line, except inside quotes, backquotes and after a backslash of course.

Tilde: The tilde is used as a shorthand for directories. Usually `~name` means the home directory of the user `name`, although you can use the `tilde` builtin to define your own shorthands.

Dollar: The dollar is used to replace the following variable name with its value, e.g `$HOME` is replaced by the value of the variable `HOME`. There are several special variables whose name is one character in length; their values are as follows:

`$$`	The process-id of the shell or alias
`$#`	The number of arguments to the shell or alias
`$?`	The exit status of the last command
`$0 - $9`	The first 10 arguments to the shell or alias
`$*`	All the arguments to the shell or alias except the zeroth

Other variable names longer than 1 character are expanded normally; a variable name ends with a non `isalnum()` character. If this character is a left bracket '`[`', the characters immediately after the bracket are converted to an integer and used to find a field in the variable. For example, if `COW` has the value `a b c d e last`, then `$COW[4]` would evaluate to the string `e`.

Star (*): The star matches zero or more characters in a file name. Thus `*.c` will evaluate to the list of files that end in `.c`.

Question (?): The question matches exactly one character in a file name. So `fred.?` may expand to `fred.c fred.o fred.p`.

Left Bracket: The characters between a left and right bracket form a set of characters that can be matched at the position where they occur in a filename; a minus sign is used to form an ascending list of characters. Thus, `fre[A-Z].c` may expand to `freQ.c freT.c freY.c`.

The star, question mark, and left bracket have the same precedence.

Other characters are recognised as *tokens*, which have special importance to the parser. These are the semicolon (;), the pipe (|), the less than and greater than signs (< and >), and the ampersand (&). If you wish these characters to be left alone, place them in single quotes.

6.2 Overview of the Code

Metacharacter expansion is broken into the following steps:

- `Meta_1()` takes the user line and breaks it up into the tokens given above. At the same time, history is expanded when the '!' character is encountered. The tokens are not necessarily broken on whitespace boundaries; the token boundaries are created to make it easier for the next section to expand metacharacters.
- `Meta_2()` scans through the token list from `meta_1()` and expands tokens if it finds a match with the metacharacters above. To do this, it uses the routines `tilde()`, `dollar()`, `backquot()`, and `star()`. Once metacharacter expansion on each token is done, `joinup()` is used to join tokens together so that the token boundaries are on whitespace.

6.3 Data Structures

Clam uses a linked list of *tokens* to do metacharacter expansion, which it forms out of the line provided by the user. There are several reasons for this:

- Using a linked list instead of a traditional character array allows efficient manipulation and insertion into the user line.
- Pointers to words can be kept in the nodes of the list, minimising the string copying needed with character arrays.
- Tokenising the line at an early stage allows succeeding stages to bypass tokens that they must not perform expansion on.

`Meta.c` uses the following structure to hold the linked list of tokens:

```
struct candidate
        { char *name;                   /* The file's name */
          struct candidate *next;       /* Next field in linked list */
          int mode;                     /* Type of token */
        };
```

The routines in `meta.c` ignore nodes where `name==NULL` and `mode==0`. This feature is used to ease the replacement of tokens in the linked list, and is discussed later.
The `mode` is a very complex bitfield. For `candidate` nodes where `name!=NULL`, the `mode` has the following values:

```
#define C_SPACE         002             /* Word has a space on the end */
#define C_DOLLAR        004             /* Word starts with a $ sign */
#define C_QUOTE         010             /* Word in single quotes */
#define C_BACKQUOTE     040             /* Word in backquotes */
```

The first value indicates whether the word was malloc'd. The other values indicate if the word is quoted, or begins with a dollar sign. For example, the following user line

```
alpha 'beta' $gamma`delta` 'epsi$lon' $phi$kappa\mu
```

is expressed by the linked list:

```
alpha -> beta -> gamma -> delta -> epsi$lon -> phi -> kappa -> m -> u
  S      S | Q     D      S | B     S | Q       D       D
```

where S, D, Q, and B stand for the appropriate defines above.
However, special words are used by the command parser, and they are separate, non-malloc'd, non-quoted words, where the values used above can be reused. They have one or more of the following values:

```
#define C_WORDMASK      01700           /* Token bits must fall in here */

#define C_SEMI           0100           /* The word is a semicolon */
#define C_DOUBLE        C_SEMI          /* Add this to 'double' the symbol */
#define C_PIPE           0200           /* The word is a pipe */
#define C_DBLPIPE        0300           /* The word is a double pipe */
#define C_AMP            0400           /* The word is an ampersand */
#define C_DBLAMP         0500           /* The word is a double ampersand */
#define C_LT             0600           /* The word is a less-than. */
#define C_LTLT           0700           /* The word is two less-thans. */
#define C_FD               03           /* File descriptor bits */
#define C_GT            01000           /* The word is a greater-than. Bits */
                                        /* in the mask C_FD hold the fd (1-2) */
#define C_GTGT          01100           /* The word is two greater-thans.Bits */
                                        /* in the mask C_FD hold the fd (1-2) */
```

Many of the token can appear 'doubled', and if so, the `C_DOUBLE` bit is turned on. For tokens `C_GT` and `C_GTGT`, the lowest 2 bits hold the file descriptor.
For example, the following user line

```
; | <><<&&&>&
```

is expressed as the following linked list:

```
C_SEMI -> C_PIPE -> C_LT -> C_GT -> C_LTLT -> C_DBLAMP -> C_AMP -> C_GTGT
                            | 1                                    | 2
```

Note that the value of the `C_FD` bits for `C_GT` and `C_GTGT`, if unspecified, default to 1. Also note that none of the tokens have `C_SPACE` on. The command parser assumes that all of these tokens implicitly have `C_SPACE` on. All of the metacharacter expansion routines ignore nodes with these tokens, and they are left for the command parser.

6.4 Global Variables

The metacharacter expansion routines form a reasonably independent part of Clam. However, because their output list is needed by the command parser, and they re-use a data structure used by `clex.c`, there are a few global variables. These are:

```
extern struct candidate carray[MAXCAN]; /* The array holding the list */
extern int ncand;                       /* Holds the # of candidates */
struct candidate *wordlist;             /* The word list for the parser */
```

The carray holds the tokens for the command parser. A user line can be expanded up to MAXCAN tokens only. Ncand indicates which array element will be used next. Wordlist points to the first node in the token linked list. Note also that ncand also is the index of the first unused carray element, and this variable is often used by the metacharacter routines to add words into the token list.
At this point it may be asked 'Why use an array to hold the token linked list when it can be malloc'd on the fly?'. The answer to this is that qsort() is used in several places in meta.c to sort a meta-expansion into alphabetical order, and qsort() can only operate on arrays. Apart from this, there is no need to use an array.
It is *not* true to say that ncand holds the number of tokens in the token list. This is because the metacharacter routines can be called *more than once* at the same time, and there may be more than one token list in the carray at the same time. This occurs, for example, when the source builtin is used to execute a file, or when an alias needs to be expanded. Thus, once metacharacter expansion has finished, wordlist and ncand indicate the beginning and end of the current token list.

6.5 Meta_1()

Meta_1() breaks a user line up into the initial tokens, as described above. This is fairly straightforward. The input line is scanned for characters that may form tokens or end words. These characters are space, tab, newline, backslash, single quote, back quote, semicolon, pipe, less-than, greater-than, ampersand and dollar.
If the character is a quote (or backquote), a matching quote is sought, and addword() called with the quote-stripped word and the appropriate quote bit in mode. If no matching quote is found, meta_1() prints an error and returns.
If the character is a word terminator, such as a space or tab, the word is left until the loop begins again. This time, a > old and the word is added using addword().
Parser tokens are dealt with individually, although some effort has been made to minimise code duplication. Doubled tokens are recognised in a simple manner. If a greater-than sign is found, the value of the file descriptor is found as well, or the default value of 1 is given.
The recognition of dollar signs is interesting. They are parsed in the same manner as whitespace, but at the end of each loop the variable lastdollar is set to C_DOLLAR if the matched char was a dollar sign. This value is OR'd with the mode of the word on the next loop iteration.
Note that addword() is usually called with a malc value of FALSE. Thus, none of the characters in the user line are ever copied, and meta_1() ensures that all words are terminated with an EOS so they can be used without malloc'ing and strcpy'ing.

6.6 Addword()

The expansion of the '!' character is done in addword(), which is passed a word/token from meta_1(). If the first character is a '!', and the word is not quoted, gethist() is called with the word, to find the last command that matches the word. The resulting string is passed back to meta_1() for further expansion. Thus, meta_1() is recursive. This recursion means that history must be saved after '!' has been expanded, otherwise the following set of commands would result in an infinite loop:

```
% /bin/ls
% !!             /* Do the last command */
% !!             /* Do the last command */
```

because gethist("!!") will return "!!", which causes the loop.
If the word doesn't start with a '!', it is simply passed to addcarray(), a general purpose routine that appends the word at the end of the token list.
Once meta_1() has tokenised the user line successfully, doline() saves the tokenised line in the history list, via expline(), which takes a token list and converts it back into a string.

6.7 Meta_2()

This routine is really a front-end for other routines, tilde(), dollar(), backquot(), and star(). The token list is scanned, and words in single quotes and special tokens are skipped. The remaining tokens are checked for the '~', '*', '?', '[' characters, and for modes with C_DOLLAR.

If the token begins with a '~', `tilde()` is called with the token, to expand the token to a full pathname. `Meta_2()` then replaces the token with the returned pathname.
If the token's mode contains a `C_DOLLAR`, `dollar()` is called with a pointer to the token. `Meta_2()` cannot simply replace the token as with `tilde()`, because 'variable' tokens can expand to several words. Thus a node pointer is passed to `dollar()` so that several nodes can be inserted.
`Dollar()` may often split the token it is given into two halves. For example:

```
ls -> -l -> HOME/*.c
 S     S     D
```

is split by `dollar()` into:

```
ls -> -l -> /home/staff/fred -> /*.c
 S     S
```

If left as is the expansion of the star character will be incorrect. Therefore, `joinup()` is called at this stage to join up words that are split by the `dollar()` routine.
After `joinup()`, each token is checked for a `C_BACKQUOTE` mode. I'll write about that later. Then the tokens are checked for the characters '*', '?' and '['. If one of these is found, `finddir()` is called to find the directory component (if any) if the word. Then `matchdir()` is called to place filenames that match the metacharacters into the `carray`. Once this is done, `qsort()` sorts the files found into alphabetical order, and they are linked into the token list.

The bits after this I haven't checked yet – Warren

6.8 Tilde()

`Tilde()` takes a string beginning with '~', and returns a strings containing the expanded filename of the input. For example:

- `/file` becomes `/u1/staff/user/file` if the user's home directory is `/u1/staff/user`.
- `short` becomes `/usr/local/abc` if the tilde list contains this word pair (see the `tilde` builtin).
- `fred/file` becomes `/u1/staff/fred/file` if fred's home directory is `/u1/staff/fred`.

`Tilde()` works in the order given above. If the input begins with '~/' then the user's directory name is placed in `dir`. Otherwise a match is sought in the tilde list, and `/etc/passwd`.
If a match is found in the tilde list, the replacement is written into `dir`. If not, and a match is found in `/etc/passwd`, the pathname from there is written into `dir`.
Finally, the unused part of the input (anything after a '/' sign) is appended to `dir`, and is returned.

6.9 Dollar()

`Dollar()` replaces a word with the value of the variable named by the word. It is a very messy routine because some variables expand to many words, and the shell also has implicit variables that do not exist in the variable list.
The list of implicit variables known by Clam are:

- `$$` - The process id of the shell.
- `$#` - The number of arguments to the shell.
- `$n` - The nth argument to the shell, where n is 0 to 9.
- `$*` - All the arguments to the shell.

When `dollar()` starts, it sets the output string to length 0. It then checks the input word `cand` for a match on the implicit variables above, and malloc's `value` if a match is found. Note that for `$*`, the arguments are placed at the end of the `carray`, and linked into the token list, with the `cand` node set to NULL, which will be skipped in later passes.
If no match is found, the variable name is pointed to by `doll`, and `EVget()` called to find the value.
Finally the value, if found, is placed in the token list. This is difficult if the word passed to `dollar()` was not simply a variable name, in which case the variable value and the remainder must be strcat'd together.

7 Job Control

This section refers to Clam version 2.0.39.

7.1 Concepts

Clam gives the user an easier and more flexible method of finding and manipulating her processes than `ps(1)` and `kill(1)`; this method is known as *job control*.
Each new process which is started by Clam is given a **job number**, plus other attributes that hold the current status of the job. At any stage, the user can terminate the job, stop the job, or restart the job if it has already been stopped. The user can also print out a list of the jobs still existing, and their state.
A job can have one of three states, not including the termination state where is exits and is removed from the system. These three states are:

Foreground The process is running, and Clam is waiting for it to stop or terminate before returning control back to the user.

Background The process is running, but Clam is not waiting for it to stop or terminate, and has gone back to accepting more commands from the user.

Stopped The process is not executing, but still exists. It will stay in this state until the user moves it to the foreground or background.

7.2 Job Control Primitives

Job Control primitives in Clam are broken into 2 categories: those which are builtin commands in the shell, and those which occur when the user types certain keys; the latter are needed when a foreground process is running, and the user cannot use the shell builtins.
The job builtins are:

`jobs` Show the list of existing jobs, their name (such as `ls` or `vi`, their process-id, and their current status.

`bg` Move a stopped job into the background.

`fg` Move a background or stopped job into the foreground.

`kill` Send a given signal to a job.

`stop` Stop a background job.

The latter four builtins take as argument either the process-id of the job, or the job number (with a `%` prepended). If no argument is given, Clam uses the *current* job as the argument – this is the job last created or last manipulated.
As well as the keys that normally send signals to the foreground process, such as `^C` and `^\`, the user can define a key (usually `^Z`) which stops the foreground process and returns control to Clam.

Clam provides four levels of job control:

- No job control at all, except for job listing,
- Job control as provided using `ptrace()` under Version7 and System V,

- Job control as provided by Berkeley, and
- Job control as provided by POSIX.

with the features and complexity increasing down the list. The following sections describe the data structures used to hold job information, and job control methods for each of these variants.

7.3 Data Structures

Clam uses a **job** structure to manipulate the jobs that are currently being executed. This structure is given below:

```
struct job
{
  int jobnumber;                /* The job number */
  int pid;                      /* The pid of the job */
  char *name;                   /* Job's argv[0]; */
  char *dir;                    /* The job's working directory */
#ifdef UCBJOB
  union wait status;            /* Job's status */
#else
  int status;                   /* Job's status */
#endif
  bool changed;                 /* Changed since last CLE? */
  struct job *next;             /* Pointer to next job */
};
```

Note that under Berkeley Unix the `status` field is a union and not an integer; both, however, holds the state of the job.
Clam maintains a linked lists of jobs using the `next` field in the job structure, and this is kept in increasing order of `jobnumber`. When a child process is created, a new job structure is created, filled in and added to the list; initially, `status` is RUNFG or RUNBG, indicating a child running in the foreground or background, and `changed` is 0, indicating the status has not changed yet. When a child dies, its job structure is eventually removed, and its jobnumber becomes available to be refilled.
Clam also has a single pointer `currentjob` that points to the job last referenced.

7.4 No Job Control

In early versions of Unix (before BSD Unix and System V Release 4), there was no concept of a stopped process. Therefore any job control was limited to showing the user what jobs were running, and being able to send signals to a process via a job number instead of its process-id.
Many of the job routines in Clam deal with the creation, deletion of jobs, and the modification of their status. These are common to all job control versions, and essentially form the routines used when there is no job control.

`Findjob()` takes a process-id and returns a pointer to the job structure which has that process-id, or NULL if no job structure is found. `Pidfromjob()` takes a job number and returns the process-id of the corresponding job; this is used by `bg()`, `fg()` and `Kill()` to send signals to a process when the given argument is a job number. When a process terminates, its job structure and job number must be removed from the job list. This is peformed by `rmjob()`.

When a new child is created by the `invoke()` routine, it calls `addjob()` to create a new job structure for the child, and adds the job to the job list, making the new job the `currentjob`. `Addjob()` creates a job structure, and searches along the job list looking for a place to insert/append the new job (remember, the job list is in ascending job number). When an unused job number is found, the job is given that number and inserted in the list; otherwise the new job is appended to the job list and given the next available job number.

If the job is being executed in the foreground, Clam must wait until it has finished execution. `Waitfor()` is called, which uses the `wait()` system call to collect change in status information about its children. Each time `wait()` returns, the new status is stored in the appropriate job's structure, and its `changed` flag is set. When the current job's status changes, `waitfor()` returns, and execution passes back to the command line.
`Waitfor()`'s argument, nominally the process-id upon which to wait, takes three forms. If the pid is $>$ 0, `waitfor()` waits until the process with that pid changes state. If pid is 0, `waitfor()` loops collecting the change of state of any processes, until there are no more processes that have changed state. Finally, if pid is negative, we wait for `abs(pid)` to exit; if it has stopped, we must keep restarting it until it finishes. The latter two variants of `waitfor()` cannot be achieved under non job control Unixes. To begin with, `wait()` blocks until there is a process that does change state – therefore `waitfor()` won't return immediately when pid is 0 and no processes have changed state. Secondly, `waitfor(-pid) == waitfor(pid)`, as there is no stopped state. The algorithm for `waitfor()` with no job control is thus:

```
void waitfor(pid)
  int pid;
{
  struct job *thisjob;
  int wpid;
  bool subshell=FALSE;
  int status;

  if (pid<0) { pid= -pid; subshell= TRUE; }
  while (1)
  {
    if (pid == 0) return; /* We can't waitfor(0) */
    wpid = wait(&status); /* Get a child's status */
    if (wpid == -1 || wpid == 0) break; /* If an error, return */
    thisjob = findjob(wpid);
    if (thisjob == NULL) continue;
    thisjob->status = status; /* Change the job's status */
    thisjob->changed = TRUE;
    if (pid == wpid) return; /* If waitfor() this one, return */
  }
}
```

Two builtins are available when there is job control: `jobs`, and `kill`.
`jobs` prints out the status of each of the jobs in the current job list, and is performed by the `joblist()` routine. This routine is also called after `waitfor()` returns to print out the status of the **changed** jobs, except for the current job if it exited normally.
One of the downfalls of the synchronous `wait()` is that information about the death of the shell's children is not sent asynchronously to the shell. This means that the information displayed by `joblist()` was only valid when the last `waitfor()` returned, and may now be out of date. Even worse, `joblist()` cannot call `wait()` to get the new information, as this will cause the shell to 'hang' if no children have changed state.
`Joblist()` is fairly straightforward. It uses several preprocessor macros to determine the current status of each job. The definitions of the macros depends upon the flavour of Unix under which the shell is compiled, but at least makes the code less ifdef'd and easier to keep consistent. These macros are:

WIFSTOPPED This is true is the job has been stopped.

WIFSIGNALED This is true is the job received a signal and terminated.

WIFEXITED This is true is the job `exit(2)`'d.

WRUNFG This is true is the job is running in the foreground.

WRUNBG This is true is the job is running in the background.

Other macros can evaluate the signal received, the exit status etc:

WSTOPSIG Evaluated if WIFSTOPPED, this returns the signal that stopped the job.

WTERMSIG Evaluated if WIFSIGNALED, this returns the signal that terminated the job.

WEXITSTATUS Evaluated if WIFEXITED, this returns the exit status of the job.

WIFCORE Evaluated if WIFSIGNALED, this is true if the job core dumped because of the signal.

The second builtin is `kill`, performed by the `Kill()` routine, which takes the process to be 'killed', and the signal name to be sent, and sends that process the signal. The process may be identified by a process-id, or by a job number, which is converted into a process-id by `getpidfromjob()`. Similarly, the signal name or signal number may be given, and the name is converted into the appropriate number. If no signal is named, `Kill()` will use the default of `SIGKILL`.

7.5 Version 7 Job Control

Job control can be achieved under early versions of Unix, such as Version 7, by using the `ptrace(2)` system call in a manner not intended by its designers.
`Ptrace()` was designed to allow a parent process to trace the excution of a child process, stopping the child under certain conditions, examining or modifying the contents of the child's memory, and restarting the child. We use the stopping/restarting abilities of `ptrace()` to provide job control.

To permit a child process to be stopped, it must inform the parent that it wants its execution to be traced, which it does by `ptrace(0,0,0,0)`. Fortunately, this can be done after the `fork()` and before the `exec()` in the `invoke()` routine, so that none of our programs need to be modified to add the `ptrace()`.
When a traced process is executing, it is stopped under the following conditions:

- the process receives a signal, or
- the process `exec()`s

If the shell is `wait()`ing on the child, it will be informed that the child has stopped, and can determine the signal that caused the process to stop (SIGTRAP in the case that the process `exec()`d). It is then able, using `ptrace()` with various arguments, to terminate or restart the process. At the same time, the shell can also deliver the signal to the restarted process, or not deliver the signal[3].
Version 7 job control, thus, is not so much a matter of stopping a process when requested to by the user, as ensuring that the process is always restarted, except when the user wants it to stop.

Restarting stopped processes is straightforward. Stopping a running foreground process, however, is difficult, as there is no terminal key that, when pressed, will inform the shell to top the process; indeed, the shell is most likely blocked `wait()`ing for the process to terminate.
Two keys that do affect the execution of a foreground process are *int* (usually `^C` or DEL), which sends a SIGINT to the process, and *quit* (usually `^\`), which sends a SIGQUIT to the process. The latter cannot be caught or ignored by the process, and the delivery of SIGQUIT causes the process to terminate, usually with a core dump. However, when a process is being traced, pending signals are *not* delivered; instead, the process is stopped, and the parent informed about the pending signal. The parent can choose to terminate or restart the process, delivering or ignoring the signal as described above.

Therefore, with `^C` being frequently used, and `^\` rarely used, we decided to reinterpret the meaning of `^\` and SIGQUIT to mean "stop the process". The SIGQUIT from `^\` is never delivered by the shell, but all other signals (including the SIGINT from `^C`) are delivered. Users can then re-bind the *quit* key with `stty(1)` to be the more traditional *stop* key, `^Z`.
The `waitfor()` routine must be modified to restart a process stopped on any signal but SIGQUIT, and deliver any signal except SIGTRAP, which is caused when the process `exec()`s. The code is shown below:

[3] See the manual for `ptrace(2)`.

```
void waitfor(pid)
  int pid;
{
  struct job *thisjob;
  int wpid;
  bool subshell=FALSE;
  int stopsig;
  int status;

  if (pid<0) { pid= -pid; subshell= TRUE; }
  while (1)
  {
    if (pid == 0) return;
    wpid = wait(&status);               /* Get a process' status */
    if (wpid == -1 || wpid == 0) break;
    thisjob = findjob(wpid);
    if (thisjob == NULL) continue;
    if (WIFSTOPPED(status))             /* If it was stopped, */
     { stopsig= WSTOPSIG(status);
       if (stopsig!=SIGQUIT)            /* ignore any SIGQUITs, */
        { if (stopsig== SIGTRAP)
            ptrace(7,wpid,1,0);         /* keep it going after exec */
          else
            ptrace(7,wpid,1,stopsig);   /* or deliver the signal */
          continue;
        }
     }     /* SIGQUIT falls out to below where job is marked stopped */
    thisjob->status = status;
    thisjob->changed = TRUE;
    if (pid == wpid) return;
  }
}
```

WIth a method of stopping foreground jobs now available, we can introduce the builtins used to stop a background job, and to move a stopped job to the foreground or background.
`Stop`, ostensibly a Clam builtin, is in fact implemented by the alias `kill -QUIT $*`, which sends a SIGQUIT to the job, stopping it.
The `bg` builtin is implemented by `bg()`, which converts its arguments to a process-id, and then restarts that process by doing `ptrace(7,pid,1,0)`; the job's status is marked RUNBG, and the current job pointer is updated.
The `fg` builtin is implemented by `fg()`. `Fg` can bring to the foreground either a background or a stopped job. The former is already running; the latter must be restarted with `ptrace(7,pid,1,0)`. Once the job is running, the current job pointer is updated, and the pid returned so that the shell can `waitfor()` the new foreground job.

Finally, two more modifications must be made to the shell before job control can work. The `kill` builtin cannot deliver a signal to a stopped job. Instead, if the specified job is stopped, `Kill()` must restart the job and deliver the signal with `ptrace(7,pid,1,signal)`.
Secondly, when a background job is `invoke()`d, its `exec()` will cause it to be stopped on a SIGTRAP. However, the shell will never `waitfor()` the job, and so it will remain stopped until `waitfor()` is called when a foreground job is `exec()`d. Therefore, `invoke()` must be modified to restart an `exec()`d background job with `ptrace(7,pid,1,0)`.

7.6 POSIX Job Control

Although Berkeley job control predates POSIX job control, they can be considered variants of each other. We prefer to describe POSIX job control in detail, as it will be the job control of choice on Unix systems in the near future, and describe the differences between Berkeley and POSIX job control in another section.

I'm rewriting the POSIX & Berkeley bits now.

7.7 Berkeley Job Control

The Berkeley job control provides new system calls `wait3()`, `setpgrp()` and `getpgrp()`, and several new signals, to overcome the synchronous `wait()` problem mentioned above, and to provide better control over the user's terminal. To enable this form of job contrl, Clam is compiled with the `-DJOB` preprocessor definition.
The new signals are:

- `SIGCHLD` This signal indicates that one of the process' children has changed status. It differs from the System V `SIGCLD` in that the status is not lost, and can be retrieved with the `wait3()` system call.
- `SIGSTOP` When received, this causes a process to stop its execution.
- `SIGCONT` A stopped process can be restarted if it receives this signal.
- `SIGTSTP` This is sent to the process controlling the terminal if a particular key is pressed (usually `^Z`), and is taken to mean that the user wants the current job to be stopped.
- `SIGCHLD` This signal indicates that one of the process' children has changed status. It differs from the System V `SIGCLD` in that the status is not lost, and can be retrieved with the `wait3()` system call.
- `SIGTTIN` The process receiving this has tried to read a character from the user's terminal, and has been prevented from doing this.
- `SIGTTOU` The process receiving this has tried to send a character to the user's terminal, and has been prevented from doing this.

Their use is discussed in the next sections.

7.7.1 Asynchronous Child Status Reporting

`SIGCHLD` and `wait3()` together allow Clam to keep the status of its jobs up to date. In Berkeley job control, Clam catches `SIGCHLD` in the `checkjobs()` routine, which, when called, uses `wait3()` to get the changed status of the child that caused the `SIGCHLD`, storing this in the appropriate job structure, and setting the `changed` flag in the structure too.
In the `waitfor()` routine, instead of using `wait()` or `wait3()`, Clam calls `pause()`, which pauses the shell until after the next signal arrives and is handled; this is usually `SIGCHLD`. After `pause()` returns, the `currentjob`'s status is checked, and if it has changed, `waitfor()` returns, passing execution back to the command line.
Therefore, because the children's status change is now handled by an asynchronous signal handler, instead of the shell proper, the status information about the shell's children is kept up to date.

7.8 Stopping and Restarting Children

One of the features in Berkeley job control is that it allows a user to temporarily stop a running process, start new processes, and eventually restart the stopped process. The two signals `SIGSTOP` and `SIGCONT`, when sent to a process, stop and restart it.

However, the shell must be told by the user when to send these signals to which particular process. This is done by having a `SIGTSTP` sent to the shell when the user types a special key on the keyboard, usually `^Z`. When `SIGTSTP` arrives, the shell can handle it in whatever way it likes.
In Clam, `SIGTSTP` is caught by `stopjob()`, which finds the process-id of the current job, and sends a `SIGSTOP` to that job. This causes the process to change state, which causes `checkjobs()` to execute. If the job was running in the foreground, `waitfor()` will see the current job change state, and return execution to the command line.
Once a child has been stopped, it can be restarted by sending it a `SIGCONT`. This is done by the `bg()` and `fg()` routines, described in the next section.

7.9 Terminal Rights

A new concept in Berkeley Unix is that of a group of processes that have the right to read from the keyboard or write to the screen. Under System V, any descendant of the shell has this right.
Berkeley Unix provides two system calls to determine the terminal rights of a process. The first is `setpgrp(pid, pgrp)`, which sets the **process group** of the given process. The second is `getpgrp(pid)`, which returns the process group of the given process.
Only one process group at any time has the rights to the terminals. Processes outside this group are sent `SIGTTIN` if they try to read from the terminal, and `SIGTTOU` if they try to write to the terminal. They also receive a `SIGSTOP` signal to stop their execution. This also causes a `SIGCHLD` to be sent to the shell, indicating that the child has changed state.
In Clam, the routine that creates a new process, `invoke()`, changes the process group of the child to a new group if the child is a background process. Otherwise, the process is left in its current process group. When a background process tries to read or write to the terminal, the actions described above are performed, and the child is now stopped. The user can leave the child stopped, or move it into the background or foreground, using the `bg` or `fg` builtins, performed by `bg()` or `fg()`. `bg()` puts the process into a process group other than the group with terminal rights, and `SIGCONT`s it. `fg()` brings the process back into the process group with the terminal rights, `SIGCONT`s it, sets the current job to that process, and calls `waitfor()`, to wait for the child's status to change.
Thus, processes can be moved from the foreground and background, and can be stopped by the user, or are stopped when they terminal I/O.

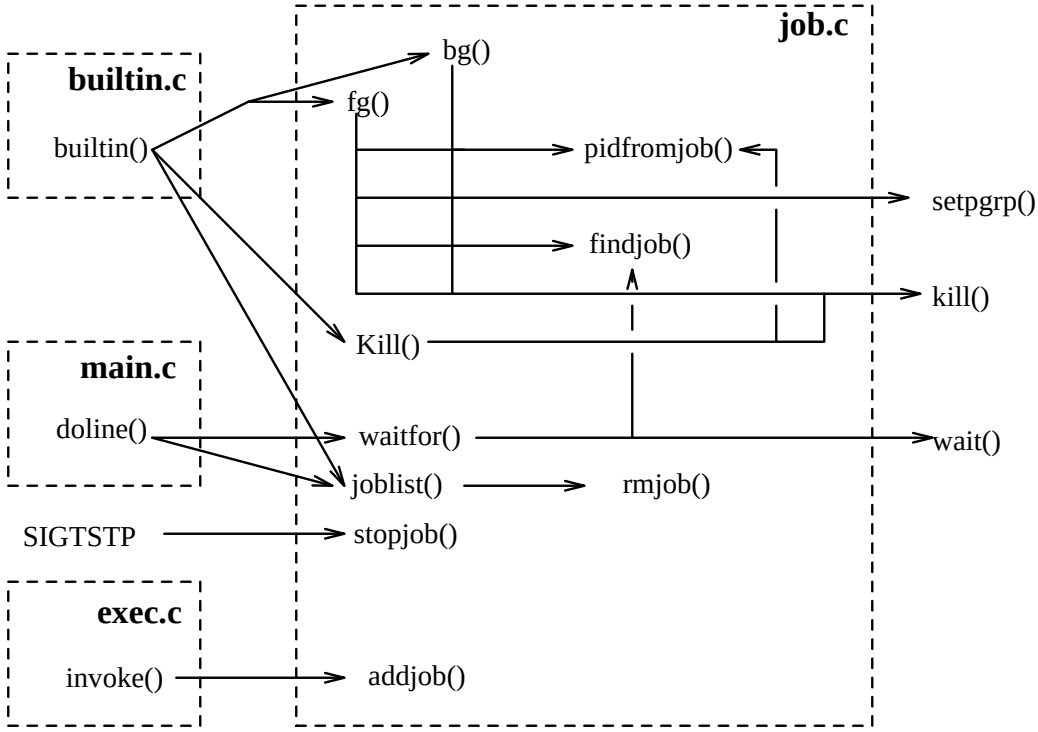
7.10 POSIX Job Control

POSIX uses the same signals for job control as Berkeley, but has a more complex view on job control, and as such, has defined several more system calls to deal with it. POSIX also recommends a particular method of job control which involves the shell passing ownership of the terminal to the current job, and reinheriting the terminal when the process is stopped or dies. See the definition of job control in the Rationale and Notes section of the POSIX standard.
Clam, on the other hand, uses a variant of the foreground/background method used under Berkeley. There are minor differences, such as the fact that the `setpgrp()` system call is replaced by `setpgid()` under POSIX. However, foreground processes when `fork()ed` stay in the same process group as the shell, and background processes are moved to a new process group.
The main difference is that under POSIX, a stopped background process cannot be brought into the process group of the shell. Instead, the shell must move control of the terminal **and** itself into the background process' process group. The actual code from `fg()` is shown below, where `pid` is the process-id of the stopped background process:

```
#ifdef POSIXJOB
  if (tcsetpgrp(0,pid) == -1)   /* Move the terminal to that pgrp */
    {
     perror("fg tcsetpgrp"); return (1);
    }
  if (setpgid(0, pid) == -1)    /* Set shell's process group to the pid's */
    {
     perror("fg setpgid"); return (1);
    }
```

```
#else
  if (setpgrp(pid, getpgrp(0)) == -1)   /* set process group to the shell's */
  {
    perror("fg setpgrp"); return (1);
  }
#endif
                                        /* and finally wake them up */
  kill(pid, SIGCONT);
```

The general job control execution flow (for BSD and POSIX variants) is shown in Figure 4.



Job Control (Version 25)

Figure 4: Job Control Execution Flow

This picture was commented out, so maybe it is now incorrect. – Warren