**MINISHELL FUNCTIONS**

**PART I. - READLINE**

**1. readline**

<readline/readline.h>

char \*readline (const char \**prompt*);

* Prints the prompt, but will not return it
* If the prompt is NULL, no prompt will be printed
* The line returned is malloced, thus needs to be freed eventually
* The newline char is removed from the end of the string
* Will return NULL only if the line contains ONLY EOF (otherwise, will be returned as a normal line)

**2. add\_history**

<readline/history.h>

*void* **add\_history** *(const char \*string);*

Place *string* at the end of the history list. The associated data field (if any) is set to NULL.

**3. rl\_clear\_historty**

<readline/readline.h>

*void* **rl\_clear\_history** *(void);*

Clear the history list by deleting all of the entries, in the same manner as the History library’s clear\_history() function. This differs from clear\_history because it frees private data Readline saves in the history list.

**4. rl\_on\_new\_line**

<readline/readline.h>

*int* **rl\_on\_new\_line** *(void)*

Tell the update functions that we have moved onto a new (empty) line, usually after outputting a newline.

More concrete use: <https://stackoverflow.com/questions/71685072/return-readline-to-its-original-state-when-recieving-sigint>

(Basically, it will be needed when the user hits ctrl+C)

**5. rl\_replace\_line**

<readline/readline.h>

*void* **rl\_replace\_line** *(const char \*text, int clear\_undo)*

*Replace the contents of rl\_line\_buffer with text. The point and mark are preserved, if possible. If clear\_undo is non-zero, the undo list associated with the current line is cleared*

rl\_line\_buffer is “the line gathered so far” (a variable of the readline lib. It can be modified, and not just with these dedicated functions).

(Same as rl\_on\_new\_line, will be useful when Ctrl+C is used.)

**6. rl\_redisplay**

<readline/readline.h>

*void* **rl\_redisplay** *(void)*

*Change what’s displayed on the screen to reflect the current contents of rl\_line\_buffer.*

(Like the previous two, used when using Ctrl+C.)

**PART II. - SIGNALS**

**1. signal**

<signal.h>

void ( \***signal**( int *sig*, void (\**handler*)(int) ) )(int);

The signal() function specifies a function to be executed when the program receives a given signal. The parameter *handler* is a pointer to a function that takes one argument of type int and has no return value. This pointer may be the address of a function defined in your program, or one of two macros defined in the header file *signal.h*.

The *handler* argument works in the following ways (assuming that the call to signal() is successful):

* If the *handler* argument is a function pointer, then signal() installs this function as the routine to be called the next time the program receives the signal designated by the integer parameter *sig*.
* If the *handler* argument is equal to the macro SIG\_DFL, then the next time the program receives the specified signal, the default signal handler routine is called. The default handler’s action for most signals is to terminate the program.
* If the *handler* argument is equal to the macro SIG\_IGN, then the specified signal will be ignored.

**2. sigaction**

<signal.h>

int sigaction(int *sig*, const struct *sigaction* \*restrict *ac*t, struct sigaction \*restrict *oact*);

The *sigaction*() function allows the calling process to examine and/or specify the action to be associated with a specific signal.

How is it different from signal()? <https://stackoverflow.com/questions/231912/what-is-the-difference-between-sigaction-and-signal>

(tl;dr, this is probably better to use than signal(), but might be harder to get to work..?)

**3. sigemptyset**

<signal.h>

int sigemptyset(sigset\_t \**set*);

Initializes a signal set *set* to the empty set. All recognized signals are excluded.

sigemptyset() is part of a family of functions that manipulate signal sets. Signal sets are data objects that let a process keep track of groups of signals. For example, a process can create one signal set to record which signals it is blocking, and another signal set to record which signals are pending. Signal sets are used to manipulate groups of signals used by other functions (such as sigprocmask()) or to examine signal sets returned by other functions (such as sigpending()).

Returns 0 on success, -1 on error

**4. sigaddset**

<signal.h>

int sigaddset(sigset\_t \**set*, int *signal*);

Adds a signal to the set of signals already recorded in set.

sigaddset() is part of a family of functions that manipulate signal sets. Signal sets are data objects that let a process keep track of groups of signals. For example, a process can create one signal set to record which signals it is blocking, and another signal set to record which signals are pending. In general, signal sets are used to manipulate groups of signals used by other functions

Applications should call either sigemptyset() or sigfillset() at least once for each object of type sigset\_t prior to any other use of that object.

**5. kill**

<signal.h>

int kill(pid\_t *pid*, int *sig*);

<https://www.ibm.com/docs/en/zos/3.1.0?topic=functions-kill-send-signal-process>

**PART III. - TERMIOS**

What is termios?

The termios functions describe a general terminal interface that

is provided to control asynchronous communications ports.

**The termios structure**

Many of the functions described here have a *termios\_p* argument

that is a pointer to a *termios* structure. This structure

contains at least the following members:

tcflag\_t c\_iflag; /\* input modes \*/

tcflag\_t c\_oflag; /\* output modes \*/

tcflag\_t c\_cflag; /\* control modes \*/

tcflag\_t c\_lflag; /\* local modes \*/

cc\_t c\_cc[NCCS]; /\* special characters \*/

More info:

<https://blog.nelhage.com/2009/12/a-brief-introduction-to-termios/>

**1. tcsetattr**

<termios.h>

int tcsetattr(int *fildes*, int *optional\_actions*, const struct termios \**termios\_p*);

The *tcsetattr*() function shall set the parameters associated with the terminal referred to by the open file descriptor *fildes* (an open file descriptor associated with a terminal) from the **termios** structure referenced by *termios\_p*

**2. tcgetattr**

<termios.h>

int tcgetattr(int fildes, struct termios \*termptr);

Gets a termios structure, which contains control information for a terminal associated with fildes. It stores that information in a memory location that termptr points to.

(So, basically the opposite of tcsetattr?)

**3. tgetent**

<termcap.h>

int tgetent (char \**buffer*, char \**termtype*);

This entry is pure gold and must be read:

<https://www.gnu.org/software/termutils/manual/termcap-1.3/html_chapter/termcap_2.html#SEC4>

But essentially, it returns a descriptor for your terminal. The termtype variable you can get with *getenv* from TERM

**4. tgetflag**

<termcap.h>

int tgetflag (char \**name*);

Use tgetflag to get a boolean value. If the capability *name* is present in the terminal description, tgetflag returns 1; otherwise, it returns 0.

What is a capability? Each piece of information recorded in a terminal description is called a *capability*. Each defined terminal capability has a two-letter code name and a specific meaning.

Once you have found the proper terminal description with tgetent, your application program must *interrogate* it for various terminal capabilities. You must specify the two-letter code of the capability whose value you seek.

Capability values can be numeric, boolean (capability is either present or absent) or strings. Any particular capability always has the same value type; for example, `co' always has a numeric value, while `am' (automatic wrap at margin) is always a flag, and `cm' (cursor motion command) always has a string value. The documentation of each capability says which type of value it has.

**5. tgetnum**

<termcap.h>

int tgetnum (char \**name*);

Use tgetnum to get a capability value that is numeric. The argument *name* is the two-letter code name of the capability. If the capability is present, tgetnum returns the numeric value (which is nonnegative). If the capability is not mentioned in the terminal description, tgetnum returns -1.

**6. tgetstr**

<termcap.h>

char \*tgetstr (char \**name*, char \*\**area*);

Use tgetstr to get a string value. It returns a pointer to a string which is the capability value, or a null pointer if the capability is not present in the terminal description. There are two ways tgetstr can find space to store the string value: tgetstr can allocate it, or you can do it yourself. More details at:

<https://www.gnu.org/software/termutils/manual/termcap-1.3/html_chapter/termcap_2.html#SEC5>

**7. tgoto:**

<termcap.h>

char \*tgoto (char \**cstring*, int *hpos*, int *vpos*)

In the same family of functions as the previous tget ones, but it handles cursor motion.

<https://www.gnu.org/software/termutils/manual/termcap-1.3/html_chapter/termcap_2.html#SEC16>

**8. tputs**

<termcap.h>

int tputs (char \**string*, int *nlines*, int (\**outfun*) ());

*Padding* means outputting null characters following a terminal display command that takes a long time to execute. The terminal description says which commands require padding and how much; the function tputs, described below, outputs a terminal command while extracting from it the padding information, and then outputs the padding that is necessary.

<https://www.gnu.org/software/termutils/manual/termcap-1.3/html_chapter/termcap_2.html#SEC11>

**PART IV. - MISC.**

**1. getenv**

Maybe stdlib?

char \*getenv(const char \*name);

Returns a specific environmental variable, like PATH, TERM. (so, no need to take it as \*\*env main function argument!). Returns NULL on failure

**2. getcwd**

<unistd.h>

char \*getcwd(char \**buf*, size\_t *size*);

Returns the absolute pathname of the current working directory.

**3. chdir**

<unistd.h>

int chdir(const char \*path);

The **chdir** command is a system function (system call) that is used to change the current working directory. On some systems, this command is used as an alias for the shell commandcd. chdir changes the current working directory of the calling process to the directory specified in path.

**4. stat**

<sys/stat.h>

int stat(const char \*restrict *path*, struct stat \*restrict *buf*);

The stat() function shall obtain information about the named file and write it to the area pointed to by the buf argument. The path argument points to a pathname naming a file. Read, write, or execute permission of the named file is not required.

Works through symlinks as well.

**5. lstat**

<sys/stat.h>

int lstat(const char \*restrict *path*, struct stat \*restrict *buf*);

Similar to stat, but does not work with symlinks. “In that case *lstat*() shall return information about the link, while *stat*() shall return information about the file the link references.”

**6. fstat**

<sys/stat.h>

int fstat(int *fildes*, struct stat \**buf*);

Similar to stat, but gives info on an open fd.

**7. opendir**

<dirent.h>

DIR \*opendir(const char \**dirname*);

Opens a directory.

**8. readdir**

<sys/types.h>

<dirent.h>

struct dirent \*readdir(DIR \**dirp*);

int readdir\_r(DIR \**dirp*, struct dirent \**entry*, struct dirent \*\**result*);

The type **DIR**, which is defined in the header [*<dirent.h>*](https://pubs.opengroup.org/onlinepubs/7908799/xsh/dirent.h.html), represents a *directory stream*, which is an ordered sequence of all the directory entries in a particular directory. Directory entries represent files; files may be removed from a directory or added to a directory asynchronously to the operation of *readdir()*.

The *readdir()* function returns a pointer to a structure representing the directory entry at the current position in the directory stream specified by the argument *dirp*, and positions the directory stream at the next entry. It returns a null pointer upon reaching the end of the directory stream. The structure *dirent* defined by the [*<dirent.h>*](https://pubs.opengroup.org/onlinepubs/7908799/xsh/dirent.h.html) header describes a directory entry.

**9. closedir**

<dirent.h>

int closedir(DIR \**dirp*);

Closes a directory.

**10. ioctl**

<stropts.h>

int ioctl(int *fildes*, int *request*, ... /\* arg \*/);

The system call ioctl() is provided for device-specific custom commands (such as format, reset and shutdown) that are not provided by standard system calls.

The ioctl function performs the generic I/O operation *command* on *filedes*.

A third argument is usually present, either a single number or a pointer to a structure. The meaning of this argument, the returned value, and any error codes depends upon the command used.

**11. wait3 & wait4**

<sys/wait.h>

<sys/time.h>

<sys/resource.h>

pid\_t wait3(int \**stat\_loc*, int *options*, struct rusage \**rusage*);

pid\_t wait4(pid\_t *pid*, int \**stat\_loc*, int *options*, struct rusage \**rusage*);

The wait3() function delays its caller until a signal is received or one of its child processes terminates or stops due to tracing. If any child process has died or stopped due to tracing and this has not already been reported, return is immediate, returning the process ID and status of one of those children. If that child process has died, it is discarded. If there are no children, -1 is returned immediately. If there are only running or stopped but reported children, the calling process is blocked.

The wait4() function is an extended interface. With a *pid* argument of 0, it is equivalent to wait3(). If *pid* has a nonzero value, then wait4() returns status only for the indicated process ID, but not for any other child processes. The status can be evaluated using the macros listed on the wstat() reference page.

<https://www.mkssoftware.com/docs/man3/wait3.3.asp>

**12. isatty**

<unistd.h>

int isatty(int *fildes*);

The *isatty*() function shall test whether *fildes*, an open file descriptor, is associated with a terminal device. 1 is returned if it is a terminal, otherwise, 0 is returned.

**13. ttyname**

<unistd.h>

char \*ttyname(int *fd*);

The *ttyname*() function shall return a pointer to a string containing a null-terminated pathname of the terminal associated with file descriptor *fildes*.

**14. ttyslot**

<stdlib.h>

int ttyslot(void);

The ttyslot() function returns the index of the current user's entry in the utmpx database. The current user's entry is an entry for which the ut\_line member matches the name of a terminal device associated with any of the process's file descriptors 0, 1 or 2. The ttyname() function is used to obtain the terminal device. The "/dev/" part returned by ttyname() is not used when searching the utmpx database member ut\_line.

**PART V. - “PIPEX” FUNCTIONS**