fescape

1.1

Generated by Doxygen 1.10.0

Chapter 1

File Index

1.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 2

File Documentation

2.1 fescape.c

```
00001 #include "fescape.h"
00002 #include "system-actions.h"
00003
00004 void usage(const char *program) {
00005     printf("Usage: %s [OPTIONS] <ARGUMENTS>\n\n", program);
00006     printf("Options:\n");
00007     printf("  -h, --help        Display this help message and exit\n");
00008     printf("  -r, --repeats     Show repeated non-ASCII chars in brackets\n");
00009     printf("  -n, --newline     Do not filter newline characters\n");
00010     printf("  -o, --octal       Display non-ASCII characters in octal instead of hex\n\n");
00011     printf("Arguments:\n");
00012     printf("  filename(s)       filename(s) to display\n");
00013     printf("  -                 streams from stdin\n");
00014     printf("  no argument       equivalent to -, streams from stdin\n\n");
00015     printf("Examples:\n");
00016     printf("  %s\n", program);
00017     printf("  %s -\n", program);
00018     printf("  %s MyBinaryFile\n", program);
00019     printf("  %s File1 MyBinaryFile2 File3\n\n", program);
00020     // printf("Restrictions:\n");
00021     // printf("  None.\n\n");
00022     // printf("Notes:\n");
00023     // printf("  None.\n");
00024     exit(EXIT_SUCCESS);
00025 } // usage()
00026
00027 void fescape(FILE *input_stream, FILE *output_stream, bool repeats, bool octal, bool filter_newlines)
    {
00028     int current_char;
00029     int saved_char = EOF;
00030     int repeat_count = 1;
00031
00032     while ((current_char = getc(input_stream)) != EOF) {
00033         if (ferror(input_stream)) {
00034             fclose(input_stream);
00035             HANDLE_ERROR("unable to read input stream");
00036         }
00037
00038         // Handle newlines separately when not filtering them
00039         if (!filter_newlines && current_char == '\n') {
00040             if (repeat_count > 1 && repeats && saved_char != '\n') {
00041                 fprintf(output_stream, "[%i]", repeat_count);
00042                 repeat_count = 1;
00043             }
00044             putc(current_char, output_stream);
00045             saved_char = current_char;
00046             continue;
00047         }
00048
00049         if (iscntrl(current_char) || !isprint(current_char)) {
00050             if (current_char == saved_char && repeats) {
00051                 repeat_count++;
00052             } else {
00053                 if (repeat_count > 1 && repeats) {
00054                     fprintf(output_stream, "[%i]", repeat_count);
00055                     repeat_count = 1;
00056                 }
00057                 saved_char = current_char;
```

```
00058                 if (current_char != '\n' || filter_newlines) {
00059                     fprintf(output_stream, octal ? "<%.3o>" : "<0x%02x>", current_char);
00060                 }
00061             }
00062         } else {
00063             if (repeat_count > 1 && repeats) { // Final repeat count for control sequences
00064                 fprintf(output_stream, "[%i]", repeat_count);
00065                 repeat_count = 1;
00066             }
00067             putc(current_char, output_stream);
00068             saved_char = EOF;
00069         }
00070     }
00071     // Handle the case for the last character being repeated
00072     if (repeat_count > 1 && repeats && saved_char != '\n') {
00073         fprintf(output_stream, "[%i]", repeat_count);
00074     }
00075 }
```

2.2 src/fescape.h File Reference

Filter unprintable characters from input stream.

```
#include <ctype.h>
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
```

Functions

- void usage (const char *program)

  Display help to user.
- void fescape (FILE *input_stream, FILE *output_stream, bool repeats, bool octal, bool filter_newlines)

  convert non-ASCII characters to hex or octal representation

2.2.1 Detailed Description

Filter unprintable characters from input stream.

Author

Robert Primmer (https://github.com/rprimmer)

Files that contain non-printable characters mess up the display when printed (e.g., via cat(1)). This program allows the display of such files, substituting hex (or optionally octal) codes for the non-printable characters. Optionally it can show the count for repeated non-printable characters rather than display each repeated hex/octal code.

Version

1.1

Date

2024-03-30

Definition in file fescape.h.

2.2.2 Function Documentation

2.2.2.1 fescape()

```
void fescape (
            FILE * input_stream,
            FILE * output_stream,
            bool repeats,
            bool octal,
            bool filter_newlines )
```

convert non-ASCII characters to hex or octal representation

Parameters

input_stream	Input stream to read.
output_stream	Output stream to write.
repeats	If true, display repeated character count.
octal	If true, display control sequences in octal instead of hex.
filter_newlines	If false, do not filter out newline characters.

Definition at line 27 of file fescape.c.

2.2.2.2 usage()

```
void usage (
            const char * program )
```

Display help to user.

Parameters

program	Calling program name

Definition at line 4 of file fescape.c.

2.3 fescape.h

Go to the documentation of this file.

```
00001
00014 #pragma once
00015
00016 #include <ctype.h>
00017 #include <stdio.h>
00018 #include <stdbool.h>
00019 #include <stdlib.h>
00020 #include <string.h>
00021 #include <unistd.h>
00022
00028 void usage(const char *program);
00029
00039 void fescape(FILE *input_stream, FILE *output_stream, bool repeats, bool octal, bool filter_newlines);
```

2.4 main.c

```
00001 #include <getopt.h>
00002 #include <libgen.h>
00003 #include <stdbool.h>
00004 #include <stdio.h>
00005 #include <stdlib.h>
00006 #include <string.h>
00007 #include <unistd.h>
00008
00009 #include "fescape.h"
00010 #include "system-actions.h"
00011
00012 int main(int argc, char **argv) {
00013     char program[PATH_MAX];
00014     basename_r(argv[0], program);
00015     FILE *fp;
00016     bool repeat_count = false;
00017     bool show_octal = false;
00018     bool filter_newlines = true;
00019
00020 #ifdef DEBUG
00021     fprintf(stderr, "%s, %d: argc: %d, optind: %d\n", basename(__FILE__), __LINE__, argc, optind);
00022 #endif // DEBUG
00023
00024     // Handle switches
00025     int option = 0;
00026     int option_index = 0;
00027     static struct option long_options[] = {{"help", no_argument, 0, 'h'},
00028                                            {"repeats", no_argument, 0, 'r'},
00029                                            {"newline", no_argument, 0, 'n'},
00030                                            {"octal", no_argument, 0, 'o'},
00031                                            {0, 0, 0, 0}};
00032     while ((option = getopt_long(argc, argv, "hrno", long_options, &option_index)) != -1) {
00033         switch (option) {
00034         case 'h':
00035             usage(program);
00036             break;
00037         case 'r':
00038             repeat_count = true;
00039             break;
00040         case 'n':
00041             filter_newlines = false;
00042             break;
00043         case 'o':
00044             show_octal = true;
00045             break;
00046         default:
00047             HANDLE_ERROR("invalid switch provided");
00048         }
00049     }
00050
00051 #ifdef DEBUG
00052     fprintf(stderr, "%s, %d: argc: %d, optind: %d\n", basename(__FILE__), __LINE__, argc, optind);
00053 #endif // DEBUG
00054
00055     // Handle arguments and actions
00056     int retval = 0;
00057
00058     if (optind >= argc)
00059         fescape(stdin, stdout, repeat_count, show_octal, filter_newlines);
00060     else
00061         for (; optind < argc; optind++) {
00062             if (strcmp(argv[optind], "-") == 0)
00063                 fescape(stdin, stdout, repeat_count, show_octal, filter_newlines);
00064             else {
00065                 if ((fp = fopen(argv[optind], "r")) == NULL)
00066                     HANDLE_ERROR("fopen: %s, file: %s", strerror(errno), argv[optind]);
00067
00068                 fescape(fp, stdout, repeat_count, show_octal, filter_newlines);
00069                 fprintf(stdout, "\n");
00070                 fclose(fp);
00071             }
00072         }
00073
00074     return ferror(stdout) ? EOF : retval;
00075 }
```

2.5 system-actions.c

```
00001 // system-actions.c
00002
00003 #include "system-actions.h"
```

```
00004
00005 void handleError(bool fatal, char *file, const char *func, int line, const char *fmt, ...) {
00006     fprintf(stderr, "Error in %s:%s, line %d: ", basename(file), func, line);
00007     va_list args;
00008     va_start(args, fmt);
00009     vfprintf(stderr, fmt, args);
00010     va_end(args);
00011     fprintf(stderr, "\n");
00012
00013     if (fatal)
00014         exit(EXIT_FAILURE);
00015 }
00016
00017 int booleanQuery(const char *prompt) {
00018     char response[10];
00019
00020     printf("%s ", prompt);
00021
00022     if (fgets(response, sizeof(response), stdin) == NULL)
00023         HANDLE_ERROR("failed to read user response");
00024
00025     return (response[0] == 'y' || response[0] == 'Y');
00026 } // booleanQuery()
00027
00028 int fileExists(const char *filename) {
00029     struct stat buffer;
00030     return (stat(filename, &buffer) == 0);
00031 } // fileExists()
00032
00033 int copyFile(const char *src, const char *dest) {
00034     char buffer[BUFSIZ];
00035     size_t bytesRead, bytesWritten;
00036
00037     FILE *source = fopen(src, "rb");
00038     if (source == NULL) {
00039         REPORT_ERROR("fopen: %s, file %s", strerror(errno), src);
00040         return EXIT_FAILURE;
00041     }
00042
00043     FILE *destination = fopen(dest, "wb");
00044     if (destination == NULL) {
00045         fclose(source);
00046         REPORT_ERROR("fopen: %s, file: %s", strerror(errno), dest);
00047         return EXIT_FAILURE;
00048     }
00049
00050     while ((bytesRead = fread(buffer, 1, sizeof(buffer), source)) > 0) {
00051         bytesWritten = fwrite(buffer, 1, bytesRead, destination);
00052         if (bytesWritten < bytesRead) {
00053             fclose(source);
00054             fclose(destination);
00055             REPORT_ERROR("fwrite: %s, file: %s", strerror(errno), dest);
00056             return EXIT_FAILURE;
00057         }
00058     }
00059
00060     if (ferror(source)) {
00061         fclose(source);
00062         fclose(destination);
00063         REPORT_ERROR("read error: %s", src);
00064         return EXIT_FAILURE;
00065     } else if (!feof(source)) {
00066         fclose(source);
00067         fclose(destination);
00068         REPORT_ERROR("unexpected end of file: %s", src);
00069         return EXIT_FAILURE;
00070     }
00071
00072     fclose(source);
00073     fclose(destination);
00074
00075     return EXIT_SUCCESS;
00076 } // copyFile()
00077
00078 int copyFile2(const char *src, const char *dest) {
00079     char buffer[BUFSIZ];
00080     ssize_t bytes_read, bytes_written, total_written;
00081
00082     int source_fd = open(src, O_RDONLY);
00083     if (source_fd == -1) {
00084         REPORT_ERROR("open: %s, file %s", strerror(errno), src);
00085         return EXIT_FAILURE;
00086     }
00087
00088     int dest_fd = open(dest, O_WRONLY | O_CREAT | O_TRUNC, 0644);
00089     if (dest_fd == -1) {
00090         close(source_fd);
```

```
00091         REPORT_ERROR("open: %s, file %s", strerror(errno), dest);
00092         return EXIT_FAILURE;
00093     }
00094
00095     while ((bytes_read = read(source_fd, buffer, sizeof(buffer))) > 0) {
00096         total_written = 0;
00097         do {
00098             bytes_written = write(dest_fd, buffer + total_written, bytes_read - total_written);
00099             if (bytes_written >= 0) {
00100                 total_written += bytes_written;
00101             } else {
00102                 close(source_fd);
00103                 close(dest_fd);
00104                 REPORT_ERROR("write: %s, file: %s", strerror(errno), dest);
00105                 return EXIT_FAILURE;
00106             }
00107         } while (bytes_read > total_written);
00108     }
00109
00110     if (bytes_read == -1) {
00111         close(source_fd);
00112         close(dest_fd);
00113         REPORT_ERROR("read error: %s", src);
00114         return EXIT_FAILURE;
00115     }
00116
00117     close(source_fd);
00118     close(dest_fd);
00119
00120     return EXIT_SUCCESS;
00121 } // copyFile2()
00122
00123 int lsFiles(const char *dirname, const char *files) {
00124     DIR *dir = opendir(dirname);
00125
00126     if (dir == NULL) {
00127         REPORT_ERROR("opendir: %s, file: %s", strerror(errno), dirname);
00128         return EXIT_FAILURE;
00129     }
00130
00131     struct dirent *entry;
00132     struct stat file_stat;
00133     char full_path[PATH_MAX];
00134
00135     while ((entry = readdir(dir)) != NULL) {
00136         if (fnmatch(files, entry->d_name, 0) == 0) {
00137
00138             if (dirname[strlen(dirname) - 1] == '/')
00139                 snprintf(full_path, sizeof(full_path), "%s%s", dirname, entry->d_name);
00140             else
00141                 snprintf(full_path, sizeof(full_path), "%s/%s", dirname, entry->d_name);
00142
00143             if (lstat(full_path, &file_stat) == 0) {
00144                 printf("%s ", full_path);
00145                 printf("Owner: %s ", getpwuid(file_stat.st_uid)->pw_name);
00146                 printf("Group: %s ", getgrgid(file_stat.st_gid)->gr_name);
00147                 printf("Size: %lld ", (long long)file_stat.st_size);
00148                 printf("Last modified: %s", ctime(&file_stat.st_mtime));
00149             } else {
00150                 closedir(dir);
00151                 REPORT_ERROR("lstat: %s, file: %s", strerror(errno), full_path);
00152                 return EXIT_FAILURE;
00153             }
00154         }
00155     }
00156
00157     return (closedir(dir));
00158 } // lsFiles()
00159
00160 int fileInfo(const char *filepath) {
00161     struct stat fileStat;
00162     if (lstat(filepath, &fileStat) < 0) {
00163         REPORT_ERROR("lstat: %s, file: %s", strerror(errno), filepath);
00164         return EXIT_FAILURE;
00165     }
00166
00167     printf("Information for %s\n", filepath);
00168     printf("---------------------------\n");
00169     printf("File Size: \t\t%lld bytes\n", (long long)fileStat.st_size);
00170     printf("Number of Links: \t%lu\n", (unsigned long)fileStat.st_nlink);
00171     printf("File inode: \t\t%lu\n", (unsigned long)fileStat.st_ino);
00172
00173     printf("File Permissions: \t");
00174     printf((S_ISDIR(fileStat.st_mode)) ? "d" : (S_ISLNK(fileStat.st_mode)) ? "l" :
    (S_ISFIFO(fileStat.st_mode)) ? "p" :
00175             (S_ISSOCK(fileStat.st_mode)) ? "s" : (S_ISCHR(fileStat.st_mode)) ? "c" :
    (S_ISBLK(fileStat.st_mode)) ? "b" : "-");
```

```
00176     printf((fileStat.st_mode & S_IRUSR) ? "r" : "-");
00177     printf((fileStat.st_mode & S_IWUSR) ? "w" : "-");
00178     printf((fileStat.st_mode & S_IXUSR) ? ((fileStat.st_mode & S_ISUID) ? "s" : "x") :
00179            ((fileStat.st_mode & S_ISUID) ? "S" : "-"));
00180     printf((fileStat.st_mode & S_IRGRP) ? "r" : "-");
00181     printf((fileStat.st_mode & S_IWGRP) ? "w" : "-");
00182     printf((fileStat.st_mode & S_IXGRP) ? ((fileStat.st_mode & S_ISGID) ? "s" : "x") :
00183            ((fileStat.st_mode & S_ISGID) ? "S" : "-"));
00184     printf((fileStat.st_mode & S_IROTH) ? "r" : "-");
00185     printf((fileStat.st_mode & S_IWOTH) ? "w" : "-");
00186     printf((fileStat.st_mode & S_IXOTH) ? ((fileStat.st_mode & S_ISVTX) ? "t" : "x") :
00187            ((fileStat.st_mode & S_ISVTX) ? "T" : "-"));
00188     printf("\n");
00189
00190     printf("Last access time: \t%s", ctime(&fileStat.st_atime));
00191     printf("Last modification time: %s", ctime(&fileStat.st_mtime));
00192     printf("Last status change time: %s", ctime(&fileStat.st_ctime));
00193
00194     struct passwd *pw = getpwuid(fileStat.st_uid);
00195     struct group *gr = getgrgid(fileStat.st_gid);
00196     printf("File Owner: \t\t%s (%d)\n", pw->pw_name, fileStat.st_uid);
00197     printf("File Group: \t\t%s (%d)\n", gr->gr_name, fileStat.st_gid);
00198     printf("Block Size: \t\t%ld bytes\n", (long)fileStat.st_blksize);
00199
00200     printf("File Type: \t\t");
00201     if (S_ISREG(fileStat.st_mode))
00202         printf("Regular file\n");
00203     else if (S_ISDIR(fileStat.st_mode))
00204         printf("Directory\n");
00205     else if (S_ISCHR(fileStat.st_mode))
00206         printf("Character device\n");
00207     else if (S_ISBLK(fileStat.st_mode))
00208         printf("Block device\n");
00209     else if (S_ISFIFO(fileStat.st_mode))
00210         printf("FIFO\n");
00211     else if (S_ISLNK(fileStat.st_mode))
00212         printf("Symbolic link\n");
00213     else if (S_ISSOCK(fileStat.st_mode))
00214         printf("Socket\n");
00215     else
00216         printf("Unknown\n");
00217
00218     return EXIT_SUCCESS;
00219 } // fileInfo()
00220
00221 int checkProcess(const char *process_name) {
00222     char command[128];
00223     snprintf(command, sizeof(command), "pgrep %s", process_name);
00224
00225     FILE *pipe = popen(command, "r");
00226     if (pipe == NULL) {
00227         REPORT_ERROR("popen: %s, process name: %s", strerror(errno), process_name);
00228         return EXIT_FAILURE;
00229     }
00230
00231     char buffer[256];
00232     if (fgets(buffer, sizeof(buffer), pipe) == NULL)
00233         fprintf(stderr, "Warning: the %s process is not running.\n", process_name);
00234     else {
00235         pid_t pid = atoi(strtok(buffer, "\n")); // Extract first PID
00236         printf("The %s process is running with PID(s): %d", process_name, pid);
00237
00238         // Check for additional PIDs
00239         while (fgets(buffer, sizeof(buffer), pipe) != NULL) {
00240             pid = atoi(strtok(buffer, "\n"));
00241             printf(" %d", pid);
00242         }
00243         printf("\n");
00244     }
00245
00246     return (pclose(pipe));
00247 } // checkProcess()
00248
00249 int displayProcess(const char *process_name) {
00250     char command[128];
00251     snprintf(command, sizeof(command), "ps aux | grep %s | grep -v grep", process_name);
00252
00253     FILE *pipe = popen(command, "r");
00254     if (pipe == NULL) {
00255         REPORT_ERROR("popen: %s, process name: %s", strerror(errno), process_name);
00256         return EXIT_FAILURE;
00257     }
00258
00259     char buffer[256];
00260     while (fgets(buffer, sizeof(buffer), pipe) != NULL)
00261         printf("%s", buffer);
00262
```

```
00263     return (pclose(pipe));
00264 } // displayProcess()
00265
00266 int validateDNSname(const char *dns_name) {
00267     regex_t regex;
00268     int result;
00269     const char *dns_regex = "^([a-zA-Z0-9]([-a-zA-Z0-9]{0,61}[a-zA-Z0-9])?\\.)+[a-zA-Z]{2,}$";
00270
00271     // Compile the regular expression
00272     result = regcomp(&regex, dns_regex, REG_EXTENDED | REG_NOSUB);
00273     if (result) {
00274         REPORT_ERROR("regex: %s, DNS name: %s", strerror(errno), dns_name);
00275         return EXIT_FAILURE;
00276     }
00277
00278     // Execute the regular expression
00279     result = regexec(&regex, dns_name, 0, NULL, 0);
00280     regfree(&regex); // Free memory allocated to the pattern buffer by regcomp
00281
00282     return result;
00283 } // validateDNSname
```

2.6 src/system-actions.h File Reference

Common functions and system actions.

```
#include <dirent.h>
#include <errno.h>
#include <fcntl.h>
#include <fnmatch.h>
#include <grp.h>
#include <libgen.h>
#include <limits.h>
#include <pwd.h>
#include <regex.h>
#include <stdarg.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <time.h>
#include <unistd.h>
```

Macros

- #define HANDLE_ERROR(fmt, ...) handleError(true, __FILE__, __func__, __LINE__, fmt, ##__VA_ARGS↩ __)
- #define REPORT_ERROR(fmt, ...) handleError(false, __FILE__, __func__, __LINE__, fmt, ##__VA_ARGS↩ __)

Functions

- void handleError (bool fatal, char *file, const char *func, int line, const char *fmt,...)

  Common error handling routine.
- int booleanQuery (const char *prompt)

Query user for yes or no.

- int fileExists (const char *filename)

  Check for file existence.
- int copyFile (const char *src, const char *dest)

  Make a copy of a file. Uses fread(3) & fwrite(3).
- int copyFile2 (const char *src, const char *dest)

  Make a copy of a file. Uses read(2) & write(2).
- int lsFiles (const char *dirname, const char *files)

  List files in a directory.
- int fileInfo (const char *filepath)

  Display information about a file.
- int checkProcess (const char *process_name)

  Check if a process is currently running.
- int displayProcess (const char *process_name)

  Display info on a running process.
- int validateDNSname (const char *dns_name)

  DNS name must start & end with a letter or a number and can only contain letters, numbers, and hyphens.

2.6.1 Detailed Description

Common functions and system actions.

Author

Robert Primmer (https://github.com/rprimmer)

Version

1.3

Date

2024-04-02

Definition in file system-actions.h.

2.6.2 Macro Definition Documentation

2.6.2.1 HANDLE_ERROR

```
#define HANDLE_ERROR(
              fmt,
              ...  ) handleError(true, __FILE__, __func__, __LINE__, fmt, ##__VA_ARGS__)
```

Definition at line 37 of file system-actions.h.

2.6.2.2 REPORT_ERROR

```
#define REPORT_ERROR(
              fmt,
              ... ) handleError(false, __FILE__, __func__, __LINE__, fmt, ##__VA_ARGS__)
```

Definition at line 38 of file system-actions.h.

2.6.3 Function Documentation

2.6.3.1 booleanQuery()

```
int booleanQuery (
             const char * prompt )
```

Query user for yes or no.

Parameters

prompt	Message to be displayed to user.

Returns

int Return true if user entered y or Y.

Definition at line 17 of file system-actions.c.

2.6.3.2 checkProcess()

```
int checkProcess (
             const char * process_name )
```

Check if a process is currently running.

Parameters

process_name	Process to look for.

Returns

int Return status.

Definition at line 221 of file system-actions.c.

2.6.3.3 copyFile()

```
int copyFile (
             const char * src,
             const char * dest )
```

Make a copy of a file. Uses fread(3) & fwrite(3).

Parameters

src	File to be copied.
dest	Filename of copy.

Returns

int Return status.

Definition at line 33 of file system-actions.c.

2.6.3.4 copyFile2()

```
int copyFile2 (
            const char * src,
            const char * dest )
```

Make a copy of a file. Uses read(2) & write(2).

Parameters

src	File to be copied.
dest	Filename of copy.

Returns

int Return status.

Definition at line 78 of file system-actions.c.

2.6.3.5 displayProcess()

```
int displayProcess (
            const char * process_name )
```

Display info on a running process.

Parameters

process_name	Process to look for.

Returns

int Return status.

Definition at line 249 of file system-actions.c.

2.6.3.6 fileExists()

```
int fileExists (
            const char * filename )
```

Check for file existence.

Parameters

filename	File to check.

Returns

int Return true of file exists.

Definition at line 28 of file system-actions.c.

2.6.3.7 fileInfo()

```
int fileInfo (
            const char * filepath )
```

Display information about a file.

Parameters

filepath	File to stat.

Returns

int Return status.

Definition at line 160 of file system-actions.c.

2.6.3.8 handleError()

```
void handleError (
            bool fatal,
            char * file,
            const char * func,
            int line,
            const char * fmt,
             ...  )
```

Common error handling routine.

Parameters

fatal	If true, exit program, else returns to the caller.
file	C filename (translation unit) of caller.
func	Function name of caller.
line	Line number in translation unit.
fmt	Optional parameters can be provided (va_list).

Definition at line 5 of file system-actions.c.

2.6.3.9 lsFiles()

```
int lsFiles (
            const char * dirname,
            const char * files )
```

List files in a directory.

Parameters

dirname	Directory housing files.
files	Files to list.

Returns

int Return status.

Definition at line 123 of file system-actions.c.

2.6.3.10 validateDNSname()

```
int validateDNSname (
            const char * dns_name )
```

DNS name must start & end with a letter or a number and can only contain letters, numbers, and hyphens.

Parameters

dns_name	DNS name to check.

Returns

int Return status.

Definition at line 266 of file system-actions.c.

2.7 system-actions.h

Go to the documentation of this file.

```
00001
00009 #ifndef SYSTEM_ACTIONS_H
00010 #define SYSTEM_ACTIONS_H
00011
00012 #include <dirent.h>
00013 #include <errno.h>
00014 #include <fcntl.h>
00015 #include <fnmatch.h>
00016 #include <grp.h>
```

```
00017 #include <libgen.h>
00018 #include <limits.h>
00019 #include <pwd.h>
00020 #include <regex.h>
00021 #include <stdarg.h>
00022 #include <stdbool.h>
00023 #include <stdio.h>
00024 #include <stdlib.h>
00025 #include <string.h>
00026 #include <sys/stat.h>
00027 #include <sys/types.h>
00028 #include <sys/wait.h>
00029 #include <time.h>
00030 #include <unistd.h>
00031
00032 // ##__VA_ARGS__ is a GNU extension that still works if __VA_ARGS__ is empty,
00033 // which supports calling the macro with just a string or with additional format arguments.
00034 // Modern compilers support this so I didn't want to clutter the code with a bunch of
00035 // #ifdef __GNUC__ conditionals just for the sake of some ancient compiler from long long ago.
00036 // __func__ was introduced in C99.
00037 #define HANDLE_ERROR(fmt, ...) handleError(true, __FILE__, __func__, __LINE__, fmt, ##__VA_ARGS__)
00038 #define REPORT_ERROR(fmt, ...) handleError(false, __FILE__, __func__, __LINE__, fmt, ##__VA_ARGS__)
00039
00049 void handleError(bool fatal, char *file, const char *func, int line, const char *fmt, ...);
00050
00057 int booleanQuery(const char *prompt);
00058
00065 int fileExists(const char *filename);
00066
00074 int copyFile(const char *src, const char *dest);
00075
00083 int copyFile2(const char *src, const char *dest);
00084
00092 int lsFiles(const char *dirname, const char *files);
00093
00100 int fileInfo(const char *filepath);
00101
00108 int checkProcess(const char *process_name);
00109
00116 int displayProcess(const char *process_name);
00117
00124 int validateDNSname(const char *dns_name);
00125
00126 #endif /* SYSTEM_ACTIONS_H */
```

Index