

{bmc win_one.bmp}

WOIO Library Documentation

Version 2.03

Copyright (c) by Lucien Cinc

Introduction

[The WOIO package](#)

[WOIO and Dynamic Data Exchange](#)

General Information

[Entry function `main\(\)`](#)

[Stdin, Stdout and Stderr](#)

Automatic File Generation

[Command AUTOGEN](#)

Function Reference

[Categories](#)

Function Reference Categories

Control Functions

<u>more</u>	enable/disable buffered screen output
<u>isbreak</u>	check for ^C pressed

Screen Output Functions

<u>printf</u>	formatted output
<u>puts</u>	display a string
<u>putchar</u>	display a character
<u>putch</u>	display a character (direct video)
<u>perror</u>	display an error message
<u>textcolor</u>	change foreground colours
<u>clrscr</u>	clear the screen
<u>clreol</u>	clear till the end of the line
<u>gotoxy</u>	set the caret screen position
<u>wherex</u>	get the horizontal caret position
<u>wherey</u>	get the vertical caret position
<u>scrwidth</u>	get the screen width
<u>scrheight</u>	get the screen height

Input Functions

<u>scanf</u>	formatted input
<u>gets</u>	get a string
<u>getchar</u>	get a character
<u>getch</u>	get a character (direct keyboard)

Command Line Functions

<u>argc</u>	get the number of command line arguments
<u>argv</u>	retrieve a command line argument
<u>arg_c</u>	Global variable
<u>arg_v</u>	Global variable
<u>argpath</u>	retrieve a command line path
<u>argabs</u>	retrieve an absolute command line path
<u>argtail</u>	retrieve the actual command line tail
<u>argn</u>	get the number of command line switches
<u>args</u>	retrieve all the command line switches
<u>argstr</u>	retrieve a command line string
<u>argnstr</u>	get the number of command line strings
<u>isargstr</u>	check for a command line string

Status Bar Functions

<u>limit</u>	set the status bar target value
<u>inc</u>	update the status bar by a value
<u>empty</u>	clear the status bar percent indicator

File Functions

<u>fillfile</u>	create a file table of all files in a given path
<u>getfile</u>	retrieve a file control block from the file table
<u>getfilepath</u>	retrieve a file path from the file table
<u>getfilename</u>	retrieve a file name from the file table
<u>padfilename</u>	pad a file name suitable for displaying

Path Functions

<u>fillpath</u>	create a path table of all directories
<u>fillpathall</u>	create a path table of all directories for all drives
<u>freepaths</u>	free the path table
<u>getpath</u>	retrieve a path from the path table
<u>totalfiles</u>	total bytes of all files in the path table

Unix Functions

<u>unixpath</u>	convert a DOS path to a Unix path
<u>unixcmd</u>	convert a DOS command to a Unix command
<u>isunix</u>	determine if Unix mode is on or off
<u>todospath</u>	convert a Unix path to a DOS path
<u>todoscmd</u>	convert a Unix command to a DOS command

File Description Functions

<u>opendesc</u>	not yet implemented
<u>closedesc</u>	not yet implemented
<u>appenddesc</u>	not yet implemented
<u>deletedesc</u>	not yet implemented
<u>finddesc</u>	not yet implemented

Environment Functions

<u>getenvironment</u>	get a WinOne environment variable
<u>putenvironment</u>	set a WinOne environment variable

The WOIO package

WOIO is a library that allows programs to interact (ie. perform I/O functions) with the main WinOne window. WOIO is essentially an abstract layer that sits on top of a normal windows program and provides a number of functions, that covers up just what is necessary to write a windows program. In fact a program written using WOIO, will look more like a DOS program than a Windows program. For example, WOIO programs use `main()` as the entry point, just like DOS programs do, and `printf()` will write to the main WinOne window (ie. a virtual screen), just like DOS programs write to the screen.

WOIO greatly simplifies the writing of programs intended for execution by WinOne. Those of you familiar with Windows programming would know just how much code is needed to displaying something as simple as *"hello, world"* inside a window. It takes about 2 pages of code to do this properly, registering a class, set-up an event loop to handle events like `WM_PAINT`, creating a font, etc. Using the WOIO library the same can be accomplished with 5 lines of code :-

```
#include "woio.h"

int main(void)
{
    printf("Hello, World\n");

    return 0;
}
```

There are some additional things that need to be set-up correctly before the above program will work, for example, the compilers switches (eg. Smart Callbacks, the Memory Model, etc...) and a definition file (ie. .DEF file) are needed. All these things will be all discussed later in this document. Apart from these extra things that are needed, some programs will only need to be modified to include this library to compile. However this will NOT generally be the case.

WOIO and Dynamic Data Exchange

WOIO uses a sort of backward Dynamic Data Exchange (DDE) protocol to communicate with WinOne. Essentially, functions are transformed into a message and posted to WinOne. WinOne then processes the message and posts a reply, which signals success or failure. This is the only way one program can communicate with another program and get an instant response.

Sounds simple enough, so why is it backward?. Well, there is no initialisation or termination involved in the WinOne DDE protocol, that always occurs when using the standard DDE protocol. To understand why there is not, you have to know something about the actual problem that has to be overcome. Assume that WinOne knew nothing about a program that it needed to execute, then after executing it, WinOne would have to wait for some period of time, to see if the executing program tries to initiate a DDE conversation. This waiting time is essentially the problem. For example, how long would WinOne wait before timing out, how can you guarantee that WinOne would not time out, before a program can link via DDE to WinOne. The overhead associated with waiting for some period of time would apply to every program that is executed by WinOne, even programs that do not want to communicate with WinOne. Clearly this is not acceptable, there must be a better way to do things. There needs to be a way to determine before a program is even executed, whether or not, it wants to communicate with WinOne, so that the appropriate course of action can be taken. There is of course and here is how it is done. All windows programs contain a description string that is inserted into the program by the compiler. Description strings usually contain a short description of a program and/or the author's copyright message. This string is set in the Module Definition File (.DEF file) and is inserted into a program at compile time. Knowing this fact, it is possible to check for a predetermined string and thereby know whether or not a program will want to communicate with WinOne before it is even executed.

The entry function main()

WOIO programs have an entry function `main()`, which is similar to DOS programs, but with one important difference, `main()` does not include the `argc` and `argv` arguments, but instead these arguments are implemented as functions calls. The `arg...` family of functions provide a high level of functionality that the standard `argc` and `argv` arguments do not and greatly simplifies the processing of command line arguments. It is not uncommon to see programs that devote whole modules purely for handling command line arguments and this is clearly not necessary when using the WOIO library. It may be interesting to note that these functions are similar to that used internally by WinOne itself. There are 10 `arg...` functions available, which include :-

<u>argc</u>	<u>argn</u>	<u>argnstr</u>	<u>argpath</u>
<u>argv</u>	<u>args</u>	<u>argstr</u>	<u>argabs</u>
<u>argtail</u>		<u>isargstr</u>	

Borland or Turbo C/C++ Compilers for Windows

The WOIO package is compiled using Borland C/C++ for windows. Therefore, to use the WinOne package you will need to use either the Borland or Turbo C/C++ compilers for Windows 3.0 and above.

Memory Model

There are two libraries supplied with the WOIO package. WOIOS.LIB supports the Small Memory model and WOIOM.LIB supports the Medium Memory model.

The Module Definition file (.DEF)

A module definition file (ie. .DEF file) needs to be created, so that WinOne can recognise external commands and take the appropriate course of action. WinOne uses the DESCRIPTION field in the .DEF file to determine whether a program is an external command. For example, consider the following sample .DEF file :-

```
NAME                XXXX
DESCRIPTION          'WinOne Command XXXX - Copyright etc... '
EXETYPE             WINDOWS
CODE                PRELOAD MOVEABLE
DATA                PRELOAD MOVEABLE MULTIPLE
HEAPSIZE            37888
STACKSIZE           6144
```

The DESCRIPTION field must start with 'WinOne Command'. This string is case sensitive. The 'XXXX' part of this field is not currently checked, but will be in latter releases of WinOne, so please include it, in capital letters. The rest of the string is ignored. Insert the .DEF file into your project file and this is all that needs to be done, so that WinOne can determine whether or not your program is an external command and therefore, need special processing.

Command AUTOGEN

Function:

Auto generate the files needed to program external commands.

Syntax:

AUTOGEN

Note:

Command AUTOGEN generates the following files :-

Extension	Description
.CPP	C++ program file
.H	C++ header file
.RC	Resource file
.PRJ	Project file
.DSK	Disk file
.DEF	Module Definition file
CLEAN.BAT	Batch file to clean up program files

AUTOGEN asks the following questions in order to generate the correct files :-

Question	Description
Command Name	Specifies the name of the External Command.
Command Author	Specifies the author of the External Command. This question is optional. Press return to skip the question.
<u>Memory Model (S/M)</u>	Specifies the memory model to use. Enter 'S' to specify the Small Model and enter 'M' to specify the Medium Model.
<u>Compiler Path [D:\TCWIN]</u>	Specifies the directory in which the compiler is located. The default path is D:\TCWIN

All the files are generated in the current directory and AUTOGEN assumes the library files (ie. WOIO.H, WOIOS.LIB and WOIOM.LIB) needed to program external commands are also located in the current directory.

Stdin, Stdout and Stderr

The standard I/O streams `stdin`, `stdout` and `stderr` are not supported by the WOIO library. Instead WOIO provides a number of functions that attempt to simulate these streams. Functions that requires `stdin`, `stdout` or `stderr` to be past as a parameter will NOT work, and should NOT be used. For example, `fputc(c, stdout)`, `putc(c, stdout)`, `fgetc(stdin)` or `getc(stdin)` should not be used. When a program uses these functions, the program will still compile and run, but will not produce the expected result. You may very well be wondering what happens when a program uses these functions, well, the output will most likely end up writing over the desktop window. WOIO supports the following functions, which attempt to mimic the standard I/O functions of the same name as closely as is possible :-

<u>printf</u>	<u>textcolor</u>	<u>scrwidth</u>	<u>scanf</u>
<u>puts</u>	<u>clrscr</u>	<u>scrheight</u>	<u>gets</u>
<u>putchar</u>		<u>clreol</u>	<u>getchar</u>
<u>putch</u>		<u>gotoxy</u>	<u>getch</u>
<u>perror</u>		<u>wherex</u>	
		<u>wherey</u>	

Actually, `stdin`, `stdout` and `stderr` is a special case of a much larger problem. In general, it is not recommended to use any functions from the standard I/O library that uses buffered I/O or file streams (ie. `FILE *stream`). For example `fopen`, `fread`, `fwrite`, `fclose`, `fprintf`, etc..., should not be used, instead un-buffered file I/O should be used, for example `open`, `close`, `read`, `write`, `_dos_open`, `_dos_close`, `_dos_read`, `_dos_write`, `OpenFile`, etc... Un-buffered file I/O will allow yield points to be inserted into the code, so that other tasks can run (ie. multi task).

void more(flag)

BOOL flag /* on and off value */

Enable or disable buffered screen output. When writing to the screen the text will not appear until enough lines have been written to fill one complete screen.

Parameter	Description
flag	When this value is TRUE, then the screen will only update after each screen full. When this value is FALSE, then the screen is updated to show any lines not yet displayed.

Returns

There is no return value.

Comments

When `more()` is set on it should be set off before the program terminates. When `more()` is not used then the screen is updated as it is written to.

Example

```
#include "woio.h"

int main(void)
{
    int ret;

    more(TRUE);      /* enable buffered output */

    ret = dofunction();

    more(FALSE);      /* flush any lines not displayed */

    return ret;      /* error level */
}
```

BOOL isbreak(void)

Determines if ^C has been pressed.

Returns

A non-zero value when ^C has been pressed and zero if it has not been pressed.

Comments

All output to the display is turned OFF when ^C is pressed and the program continues to execute. When ^C has been pressed (ie. `isbreak()` returns TRUE) the program should then perform any cleaning up necessary and exit (ie. by returning from `main()`, do NOT use `exit()` to terminate).

Example

```
#include "woio.h"

/* Loop until ^C is pressed */

int main(void)
{
    printf("press ^C to quit: \n");

    while (1)
        if (isbreak()) /* check for ^C pressed */
            break;

    return 0;
}
```

int printf(fmt, ...)

`printf()` provides formatted output and functions similar to the standard run time library `printf()`.

Example

Comments

For a full description of `printf()` consult your standard run time library reference manual.

`printf()` will only display printable characters.

The following `#define` values can be past as character arguments to change the colour of the text displayed by `printf()` :-

```
#define BLACK          (char )128   /* text colours */
#define RED            (char )129
#define GREEN          (char )130
#define BLUE           (char )131
#define YELLOW         (char )132
#define MAGENTA        (char )133
#define CYAN           (char )134
#define WHITE          (char )135
#define LIGHTGRAY      (char )136
#define LIGHTRED       (char )137
#define LIGHTGREEN     (char )138
#define LIGHTBLUE      (char )139
#define BROWN          (char )140
#define LIGHTMAGENTA   (char )141
#define LIGHTCYAN      (char )142
#define DARKGRAY       (char )143
```

See Also

[puts](#)
[putch](#)
[putchar](#)
[textcolor](#)
[scanf](#)
[gets](#)

Example

Consider the following :-

```
printf("%cHello, world\n%cHow are you?", MAGENTA, BLUE);
```

will display the following :-

```
Hello, world
How are you?
```

void puts(s)

char *s /* character string */

Display a character string along with a CR-LF character combination.

Parameter	Description
s	Address of a NULL terminated character string.

Returns
There is no return value.

Comments
Function `puts()` is streamable, that is, when `stdout` is redirected on the command line to a file, the character string will be written to the file.

Tab characters are padded with space characters.

See Also
[printf](#)
[putchar](#)
[putch](#)
[gets](#)

void putchar(c)

char c /* character */

Display a character directly to the screen.

Parameter	Description
c	Character value.

Returns

There is no return value.

Comments

Function `putchar()` writes directly to the screen, and as a result is not streamable.

Tab characters are padded with space characters.

See Also

[printf](#)
[puts](#)
[putchar](#)
[getch](#)

Example

```
void myerror(char *msg)
{
    textcolor(RED);      /* display message in RED */

    while (*msg)         /* display message */
        putchar(*msg++);

    putchar('\n');
}
```


void putchar(c)

char c /* character */

Display a character.

Parameter	Description
c	Character value.

Returns
There is no return value.

Comments
Function `putchar()` is streamable, that is, when `stdout` is redirected on the command line to a file, the character will be written to the file.

Tab characters are padded with space characters.

See Also

[printf](#)
[puts](#)
[putch](#)
[getchar](#)

Example

```
void charmsg(char *msg)
{
    while (*msg)                /* display message */
        putchar(*msg++);

    putchar('\n');
}
```

void perror(msg)

char *msg /* character string */

Display an error message.

Parameter	Description
msg	Address of a NULL terminated character string that contains the message to display.

Returns
There is no return value.

Comments
The message along with two CR-LF character combinations is written to `stderr`. `Stderr` is not streamable, that is, when `stdout` is redirected on the command line to a file, `stderr` will still write to the screen.

RED is used for the foreground colour.

See Also
[printf](#)
[puts](#)
[putchar](#)

void textcolor(col)

char col /* colour value */

Set the current text colour.

Parameter	Description
col	Range value, that specifies the colour to set.

Returns

There is no return value.

Comments

The following colours are #defined in the WOIO.H header file :-

```
#define BLACK          (char )128   /* text colours */
#define RED            (char )129
#define GREEN          (char )130
#define BLUE           (char )131
#define YELLOW         (char )132
#define MAGENTA        (char )133
#define CYAN           (char )134
#define WHITE          (char )135
#define LIGHTGRAY      (char )136
#define LIGHTRED       (char )137
#define LIGHTGREEN     (char )138
#define LIGHTBLUE      (char )139
#define BROWN         (char )140
#define LIGHTMAGENTA   (char )141
#define LIGHTCYAN      (char )142
#define DARKGRAY       (char )143
```

There are no blinking or bold characters.

See Also

[printf](#)

void clrscr(void)

Clear the screen.

Returns

There is no return value

Comments

When the screen is cleared, the contents of the screen are NOT moved to the scroll back buffer.

void clreol(void)

Clear till the end of the current line.

Returns

There is no return value

void gotoxy(x, y)

```
int x          /* co-ordinate */
int y          /* co-ordinate */
```

Position the caret on the screen.

Parameter	Description
x	Co-ordinate on the horizontal x-axis.
y	Co-ordinate on the vertical y-axis.

Returns

There is no return value

Comments

The first character on the screen is at co-ordinate 1, 1.

See Also

[wherex](#)
[wherey](#)

Example

```
#include "woio.h"
#include <string.h>

/*
    Display the string "Hello, World"
    centred on the screen
*/

int main(void)
{
    char *s;

    s = "Hello, World";    /* string to display */
    clrscr();

    gotoxy((scrwidth() - strlen(s)) / 2, scrheight() / 2);
    printf("%c%s", WHITE, s);

    gotoxy(0, scrheight());

    return 0;              /* error level */
}
```

int wherex(void)

Determine the horizontal location of the caret.

Returns

Co-ordinate of the caret on the horizontal x-axis.

Comments

Co-ordinates start from 1.

See Also

[wherey](#)

[gotoxy](#)

int wherey(void)

Determine the vertical location of the caret.

Returns

Co-ordinate of the caret on the vertical y-axis.

Comments

Co-ordinate start from 1.

See Also

[wherex](#)

[gotoxy](#)

int scrwidth(void)

Determine the screen width in characters.

Returns

The screen width in characters.

See Also

[scrheight](#)

int scrheight(void)

Determine the screen height in characters.

Returns

The screen height in characters.

See Also

[scrwidth](#)

int scanf(fmt, ...)

`scanf()` provides formatted input and functions the same as the standard run time library `scanf()`.

Comments

For a full description of `scanf()` consult your standard run time library reference manual.

See Also

[gets](#)
[getch](#)
[getchar](#)
[printf](#)

char *gets(s)

char *s /* character string */

Get a character string without the CR-LF character combination.

Parameter	Description
s	Address of a character array to store the string. This array must be at least 80 characters in size.

Returns

On success, it returns the address of the character array, where the NULL terminated character string is stored. On end of file (ie. EOF) or on error, NULL is returned.

Comments

Function `gets()` is streamable, that is, when `stdin` is redirected on the command line from a file, the character string will be read from the file and will not be echoed to the screen.

Tab characters are converted to a single space character, unless `stdin` has been redirected on the command line.

See Also

[scanf](#)
[getch](#)
[getchar](#)
[puts](#)

Example

```
#include "woio.h"
#include <dos.h>

/* Determine whether a file exists */

int prompt_open(void)
{
    char buf[80];
    int handle;

    printf("%cEnter filename:%c ", WHITE, LIGHTGRAY);

    if (gets(buf))                    /* get a filename */
        if (_dos_open(buf, 0, &handle)
            return handle;           /* opened file */

    return 0;                        /* failed to open file */
}

int main(void)
{
    int handle;

    if ((handle = prompt_open()) != 0) {
        printf("File exists\n");
        _dos_close(handle);        /* close the file */
        return 1;
    }
}
```

```
    }  
    return 0;  
}
```

int getchar(void)

Get a character .

Returns

On success, a character value is returned, on error or end of file, a value of EOF (ie. -1) is returned.

Comments

Function `getchar()` is streamable, that is, when `stdin` is redirected on the command line from a file, the characters will be read from the file and will not be echoed to the screen.

When `stdin` has NOT been redirected on the command line then the following applies :-

1. Characters are echoed to the screen.
2. Tab characters are converted to single space characters,
3. Carriage return characters are converted to new line characters (ie. `'\r'` mapped to `'\n'`).
4. All non-printable characters are ignored.

See Also

[scanf](#)
[getch](#)
[putch](#)
[putchar](#)

int getch(void)

Get a character from the keyboard

Returns

A character value.

Comments

Function `getch()` read characters from the keyboard, and as a result is not streamable.

Characters read are not echoed to the screen.

There is no character mapping. (ie. `'\r'` is NOT mapped to `'\n'`);

See Also

[scanf](#)

[putch](#)

[getchar](#)

[putchar](#)

int argc(void)

Determines the number of command line arguments.

Returns

The number of command line arguments.

Comments

Command line strings (eg. "This is a string") are considered as command line arguments.
Command line switches are not considered as part of the command line arguments.

See Also

[argv](#)

[argn](#)

[args](#)

[argstr](#)

[argnstr](#)

char *argv(index)

int index /* command line argument */

Retrieve a command line argument.

Parameter	Description
index	Specifies which argument to retrieve. Specifying an index of 0 retrieves the programs name. Command line arguments start from an index of 1.

Returns

On success it returns the address of a NULL terminated string containing the argument. On error it returns a NULL.

Comments

The argument is stored in a static buffer and is over-written each time this function is called. This function cannot be used to retrieve command line switches.

See Also

[argc](#)
[argn](#)
[args](#)
[argstr](#)
[argnstr](#)

Example

```
#include "woio.h"
#include <stdlib.h>

/* Sum all value on the command line */

int main(void)
{
    long total;
    int i, n;

    total = 0;          /* zero total */
    if ((n = argc()) == 0) {
        perror("nothing to sum");
        return 1;
    }

    for (i = 0; i < n; i++)

        total += atol(argv(i + 1));

    printf("%ctotal=%c%ld\n", WHITE, YELLOW, total);

    return 0;          /* error level */
}
```

int arg_c
char *arg_v[]

Global variables that contains the number of command line arguments (ie. `arg_c`) and the actual command line arguments (ie. `arg_v`).

Comments

`arg_c` and `arg_v` is provided for compatibility with the standard library `argc` and `argv`, which is past to a normal C or C++ `main()`, and has the following format :-

```
#include "stdio.h"

int main(int argc, char *argv[])
{
}
```

and a WinOne external command `main()`, has the following format :-

```
#include "woio.h"

int main(void)
{
    /* arg_c is used instead of argc */
    /* arg_v is used instead of argv */
}
```

When using `arg_c` and `arg_v`, avoid using the `arg...()` family of functions, since `arg_c` and `arg_v` do not separate command line arguments and command line switches.

See Also

[argc](#)
[argv](#)
[argn](#)
[args](#)
[argstr](#)
[argnstr](#)

char *argpath(index)

int index /* command line argument */

Retrieve a command line argument and convert it to a full path name.

Parameter	Description
index	Specifies which argument to retrieve. Specifying an index of 0 retrieves the current directory, as a full path name. Command line arguments start from an index of 1.

Returns

On success it returns the address of a NULL terminated string containing the full path name. On error it returns a NULL.

Comments

The full path name is stored in a static buffer and is over-written each time this function is called.

Full path names are made up of the following components :-

drive:\directory\filename

Component	When not Specified
drive	Current drive is used.
directory	Current directory is used. Also relative directories are converted to absolute directories.
filename	*.* is used. Wildcard characters are allowed in the filename.

See Also

[argc](#)
[argv](#)
[argabs](#)

Examples

The following examples assume the current directory is C:\WINDOWS :-

Argument	Full path
C:	C:\WINDOWS*.*
C:\	C:*.*
\DOS\	C:\DOS*.*
NOTEPAD.EXE	C:\WINDOWS\NOTEPAD.EXE
.EXE	C:\WINDOWS.EXE
WHAT	C:\WINDOWS\WHAT.
.	C:\WINDOWS*.*
.	C:\WINDOWS*.*
..	C:*.*

char *argabs(index)

int index /* command line argument */

Retrieve a command line argument and convert it to an absolute path name.

Parameter	Description
-----------	-------------

index Specifies which argument to retrieve. Specifying an index of 0 retrieves the current directory, as an absolute path name. Command line arguments start from an index of 1.

Returns

On success it returns the address of a NULL terminated string containing the absolute path name. On error it returns a NULL.

Comments

The absolute path name is stored in a static buffer and is over-written each time this function is called.

Absolute path names are made up of the following components :-

drive:\directory\filename

Component	When not Specified
drive	Current drive is used.
directory	Current directory is used. Also relative directories are converted to absolute directories.
filename	The previous directory name becomes the filename. Wildcard characters are allowed in filename.

See Also

[argc](#)
[argv](#)
[argpath](#)

Example

The following examples assume the current directory is C:\WINDOWS :-

Argument	Full path
C:	C:\WINDOWS.
C:\	C:\.
\DOS\	C:\DOS.
NOTEPAD.EXE	C:\WINDOWS\NOTEPAD.EXE
.EXE	C:\WINDOWS.EXE
WHAT	C:\WINDOWS\WHAT.
.	C:\WINDOWS*.*
.	C:\WINDOWS.
..	C:\.

char *argtail(void)

Retrieve the actual command line tail.

Returns

Address of a NULL terminated string containing the command line tail.

Comments

The actual command line tail does not include any redirection arguments. Also, the tail is stored in a static buffer and is over-written each time this function is called.

See Also

[argc](#)

argv
argn
args

int argn(void)

Determines the number of command line switches.

Returns

The number of switches.

See Also

[argc](#)

[argv](#)

[args](#)

char *args(void)

Retrieve the command line switches.

Returns

Address of a NULL terminated string containing all the switches.

Comments

Switches are stored in a static buffer and is over-written each time this function is called.

The string returned can be empty, when there are no command line switches.

See Also

[argc](#)

[argv](#)

[argn](#)

char *argstr(index)

int index /* command line argument */

Retrieve a command line argument and convert it from a string.

Parameter	Description
index	Specifies which argument to retrieve. Command line arguments start from an index of 1.

Returns

On success it returns the address of a NULL terminated string containing the converted command line string. On error it returns a NULL.

Comments

Strings are converted by removing the first and last double-quote marks characters, and by replacing any pair of double-quote marks that appear inside the string with a single double-quote mark character.

This function converts the actual command line argument and then returns it. After a command line argument is converted it is no longer considered a command line string.

The converted command line string is stored in a static buffer and is over-written each time this function is called.

See Also

[argc](#)
[argv](#)
[argnstr](#)
[isargstr](#)

Example

Converting the following string "This is a ""string""." returns This is a "string".

int argnstr(void)

Determines the number of command line arguments, that can be converted from a string.

Returns

The number of command line arguments that can be converted from a string.

See Also

[argc](#)

[argv](#)

[argstr](#)

[isargstr](#)

BOOL isargstr(index)

int index /* command line argument */

Determines whether a command line argument can be converted from a string.

Parameter	Description
index	Specifies which argument to check. Command line arguments start from an index of 1.

Returns

On success, a non-zero value when a command line argument can be converted from a string and zero otherwise.

See Also

[argc](#)
[argv](#)
[argnstr](#)
[argstr](#)

void limit(upper)

unsigned long upper /* upper limit */

Set the Status Bar upper limit (ie. target value) to reach.

Parameter	Description
upper	Specifies the upper limit to reach.

Returns

There is no return value.

Comments

This will display 0 in the Percent indicator, the next time the display is updated.

See Also

[inc](#)
[empty](#)

void inc(value)

unsigned long value /* value to increment by */

Increment the current Status Bar total.

Parameter	Description
value	Specifies a value to be added to the current Status Bar total.

Returns

There is no return value.

Comments

Function `inc()` may be called many times before a percentage is calculated and displayed, since the Status Bar is updated once every second.

See Also

[limit](#)
[empty](#)

void empty(void)

Clear the Status Bar Percent indicator.

Returns

There is no return value.

Comments

The Status Bar Percent indicator is blanked unconditionally.

See Also

[limit](#)

[inc](#)

int fillfile(path, attr)

char *path /* directory path */
unsigned int attr /* file attributes */

Create a File Table of all the files in a given path and with a given attribute.

Parameter	Description												
path	Address of a NULL terminated character string containing a path. This path can be relative or absolute and can contain Wildcard characters any where in the filename part of the path.												
attr	DOS File attribute (defined in DOS.H), include the following :- <table><tr><td>FA_RDONLY</td><td>Read-only</td></tr><tr><td>FA_HIDDEN</td><td>Hidden file</td></tr><tr><td>FA_SYSTEM</td><td>System file</td></tr><tr><td>FA_LABEL</td><td>Volume label</td></tr><tr><td>FA_DIREC</td><td>Directory</td></tr><tr><td>FA_ARCH</td><td>Archive</td></tr></table>	FA_RDONLY	Read-only	FA_HIDDEN	Hidden file	FA_SYSTEM	System file	FA_LABEL	Volume label	FA_DIREC	Directory	FA_ARCH	Archive
FA_RDONLY	Read-only												
FA_HIDDEN	Hidden file												
FA_SYSTEM	System file												
FA_LABEL	Volume label												
FA_DIREC	Directory												
FA_ARCH	Archive												

Returns

On success, it returns the number of files that match the path and attribute specified, otherwise, it returns a value of zero, when no files match.

Comments

Function `fillfile()` replaces `findfirst()`, `findnext()`, `_dos_findfirst()` and `_dos_findnext()`, since these functions do not support extended wildcard card characters.

WinOne allows wildcard characters to be placed anywhere inside a filename and be correctly interpreted.

All files that meet the specifies requirements are places inside a table (ie. File Table) which is over-written with each call to this function. Use the `getfile...()` functions to access the information stored in this table.

The table is sorted in alphabetical order.

See Also

[getfile](#)
[getfilepath](#)
[getfilename](#)

BOOL getfile(index, pff)

int index /* index into File Table */
struct ffbk *pff /* DOS file control block structure */

Retrieve a DOS file control block structure from the File Table.

Parameter	Description
index	Specifies which ffbk to retrieve from the File Table. Entries in the File Table start from an index of 0.
pff	Address of a DOS file control block structure (defined in DIR.H) :-

```
struct ffbk {  
    char ff_reserved[21];    /* reserved by DOS */  
    char ff_attrb;           /* attribute found */  
    int  ff_ftime;           /* file time */  
    int  ff_fdate;           /* file date */  
    long ff_fsize;           /* file size */  
    char ff_name[13];        /* found file name */  
};
```

Returns

On success, it returns a non-zero value and the file control block structure is filled. On error zero is returned.

Comments

Use the `fillfile()` to fill the File Table before using `getfile()`.

See Also

[fillfile](#)
[getfilepath](#)
[getfilename](#)

Example

```
#include "woio.h"  
#include <dir.h>  
  
/* Display a file listing */  
  
int dir(char *path)  
{  
    int i, n;  
    struct ffbk ffbk;  
  
    printf("%cDirectory of %s\n\n", WHITE, path);  
  
    if ((n = fillfile(path, 0)) == 0) {  
        perror("No files found");  
        return 1;  
    }  
  
    for (i = 0; i < n; i++) {  
        if (getfile(i, &ffbk) == FALSE) {
```

```

        perror("Bad index");
        return 1;
    }

    printf(" %c%-13s%c%9ld\n",
        GREEN, ffblk.ff_name,
        YELLOW, ffblk.ff_fsize); /* display file names */

}

return 0;        /* all done */
}

int main(void)
{
    char *path;
    int ret;

    if (argc() > 1) { /* check number of arguments */
        perror("Too many or few arguments");
        return 1;
    }

    more(TRUE);    /* buffered screen output */

    if ((path = argpath(argc())) == NULL) {
        perror("Path or file not found");
        return 1;
    }

    ret = dir(path);

    more(FALSE);   /* flush output */

    return ret;    /* error level */
}

```


char *getfilepath(index)

int index /* index into File Table */

Retrieve a file path from the File Table.

Parameter	Description
index	Specifies which file path to retrieve from the File Table. Entries in the File Table start from an index of 0.

Returns

On success it returns the address of a NULL terminated string containing the path name. On error it returns a NULL.

Comments

Use the `fillfile()` to fill the File Table before using `getfilepath()`.

Path names are stored in a static buffer and is over-written each time this function is called.

The path returned may not contain a fully qualified path name.

See Also

[fillfile](#)
[getfile](#)
[getfilename](#)

char *getfilename(index)

int index /* index into File Table */

Retrieve a file name from the File Table.

Parameter

index

Description

Specifies which file name to retrieve from the File Table. Entries in the File Table start from an index of 0.

Returns

On success it returns the address of a NULL terminated string containing the file name. On error it returns a NULL.

Comments

Use the `fillfile()` to fill the File Table before using `getfilename()`.

File names are stored in a static buffer and is over-written each time this function is called.

See Also

[fillfile](#)

[getfile](#)

[getfilepath](#)

char *padfilename(path)

char *path /* character string */

Pad a file name so that it is suitable for displaying.

Parameter	Description
path	Address of a NULL terminated character string contain a path name.

Returns

The address of a character string containing the padded file name.

Comments

Only the file name is padded and returned, the rest of the path is discarded.

The padded file name is stored in a static buffer and is over-written each time this function is called.

int fillpath(path)

char *path /* directory path */

Create a Path Table containing all the directories and sub-directories starting from the specified path.

Parameter	Description
path	Address of a NULL terminated character string containing a path. This path can be relative or absolute. The file name part of the path is expected and ignored.

Returns

On success, it returns the number of directories and sub-directories found, otherwise, it returns a value of zero, when no directories are found.

Comments

The directories . and . . are not included in the table.

A new Path Table is allocated with each call to this function. Use `freepaths()` to release the memory allocated for the table, when it is no longer needed.

The table is sorted in alphabetical order

See Also

[fillpathall](#)
[freepaths](#)
[getpath](#)

int fillpathall(void)

Create a Path Table containing all the directories and sub-directories for all drives that are non-removable.

Returns

On success, it returns the number of directories and sub-directories found, otherwise, it returns a value of zero, when no directories are found.

Comments

The directories `.` and `..` are not included in the table.

A new Path Table is allocated with each call to this function. Use `freepaths()` to release the memory allocated for the table, when it is no longer needed.

The table is sorted in alphabetical order

See Also

[fillpath](#)
[freepaths](#)
[getpath](#)

void freepaths(void)

Release the memory allocated to store the Path Table.

Returns

There is no return value.

See Also

[fillpath](#)

[fillpathall](#)

char *getpath(index)

int index /* index into the Path Table */

Retrieve a path name from the Path Table.

Parameter	Description
index	Specifies which path to retrieve from the Path Table. Entries in the Path Table start from an index of 0.

Returns

On success it returns the address of a NULL terminated string containing the path name. On error it returns a NULL.

Comments

All the path names returned contains * . * for the file name part..

Use the `fillpath()` or `fillpathall()` to fill the Path Table before using `getpath()`.

Path names are stored in a static buffer and is over-written each time this function is called.

See Also

[fillpath](#)
[fillpathall](#)
[freepaths](#)

int totalfiles(void)

Get the total number of bytes for all the files in the directories contained in the Path Table.

Returns

The total number of bytes for all the files in the Path Table.

Comments

Use the `fillpath()` or `fillpathall()` to fill the Path Table before using `totalfiles()`.

The total number of bytes includes all files, irrespective of the file attributes.

See Also

[fillpath](#)

[fillpathall](#)

char *unixpath(path)

char *path /* DOS path to convert */

Convert a DOS path name to a Unix path name, when Unix mode is enabled.

Parameter	Description
path	Address of a NULL terminated character string containing a DOS path. This path can be relative or absolute.

Returns

The address of a NULL terminated string containing the Unix path name, when Unix mode is enabled, otherwise, the DOS path name is returned.

Comments

Path names are stored in a static buffer and is over-written each time this function is called.

When Unix mode is not enabled, then the DOS path name is simply copied into the static buffer and is not converted.

See Also

[unixcmd](#)
[isunix](#)
[todospath](#)
[todoscmd](#)

char *unixcmd(cmd)

char *cmd /* DOS command to convert */

Convert a DOS command line to a Unix command line, when Unix mode is enabled.

Parameter	Description
cmd	Address of a NULL terminated character string containing a DOS command line.

Returns

The address of a NULL terminated string containing the Unix command line, when Unix mode is enabled, otherwise, the DOS command line is returned.

Comments

Command lines are stored in a static buffer and is over-written each time this function is called.

When Unix mode is not enabled, then the DOS command line is simply copied into the static buffer and is not converted.

See Also

[unixpath](#)
[isunix](#)
[todospath](#)
[todoscmd](#)

int isunix(void)

Determines whether Unix mode is on or off.

Returns

A value greater than zero when Unix is on and zero when Unix is off.

See Also

[unixpath](#)

[unixcmd](#)

[todospath](#)

[todoscmd](#)

char *todopath(path)

char *path /* Unix path to convert */

Convert a Unix path name to a DOS path name, when Unix mode is enabled.

Parameter	Description
path	Address of a NULL terminated character string containing a Unix path. This path can be relative or absolute.

Returns

The address of a NULL terminated string containing the DOS path name, when Unix mode is enabled, otherwise, the Unix path name is returned.

Comments

Path names are stored in a static buffer and is over-written each time this function is called.

When Unix mode is not enabled, then the Unix path name is simply copied into the static buffer and is not converted.

Typically, `isunix()` will be called before `todopath()` to determine whether or not a path should be converted.

See Also

[unixpath](#)
[unixcmd](#)
[isunix](#)
[todoscmd](#)

char *todoscmd(cmd)

char *cmd /* Unix command to convert */

Convert a Unix command line to a DOS command line, when Unix mode is enabled.

Parameter	Description
cmd	Address of a NULL terminated character string containing a Unix command line.

Returns

The address of a NULL terminated string containing the DOS command line, when Unix mode is enabled, otherwise, the Unix command line is returned.

Comments

Command lines are stored in a static buffer and is over-written each time this function is called.

When Unix mode is not enabled, then the Unix command line is simply copied into the static buffer and is not converted.

Typically, `isunix()` will be called before `todoscmd()` to determine whether or not a command line should be converted.

See Also

[unixpath](#)
[unixcmd](#)
[isunix](#)
[todospath](#)

char *getenvironment(name)

char *name /* name of environment variable to retrieve */

Retrieve an environment variable from the WinOne environment space.

Parameter	Description
name	Address of a NULL terminated character string that contains the name of the environment variable to retrieve.

Returns

The address of a NULL terminated character string, where the environment variable is stored. On error a NULL value is returned.

Comments

The environment variable is stored in a static buffer, that is over-written each time this function is called.

See Also

[putenvironment](#)

int putenvironment(name)

char *name /* environment string */

Place an environment string into the WinOne environment space.

Parameter	Description
name	Address of a NULL terminated character string that contains the environment string to place into the WinOne environment space.

Returns

Value of zero. On error, a value of -1 is returned

Comments

The parameter name is duplicated using `malloc()` by WinOne and therefore, does not need to be a static or malloced and not freed, which is the case with the standard library function `putenv()`.

Environment strings have the form:-

```
VARNAME=ENVSTRING
```

To delete an environment variable, exclude the `ENVSTRING`. For example, to delete the environment variable `AVAR` :-

```
putenvironment("AVAR=");
```

See Also

[getenvironment](#)

