

## Table of Contents

### I. [SDL Guide](#)

#### [Preface](#)

[About SDL](#)

[About SDLdoc](#)

[Credits](#)

#### 1. [The Basics](#)

[Introduction](#)

[Initializing SDL](#)

#### 2. [Graphics and Video](#)

[Introduction to SDL Video](#)

[Using OpenGL With SDL](#)

#### 3. [Input handling](#)

[Handling Joysticks](#)

[Handling the Keyboard](#)

#### 4. [Examples](#)

[Introduction](#)

[Event Examples](#)

[Audio Examples](#)

[CDROM Examples](#)

[Time Examples](#)

### II. [SDL Reference](#)

#### 5. [General](#)

[SDL\\_Init](#) — Initializes SDL

[SDL\\_InitSubSystem](#) — Initialize subsystems

[SDL\\_QuitSubSystem](#) — Shut down a subsystem

[SDL\\_Quit](#) — Shut down SDL

[SDL\\_WasInit](#) — Check which subsystems are initialized

[SDL\\_GetError](#) — Get SDL error string

#### 6. [Video](#)

[SDL\\_GetVideoSurface](#) — returns a pointer to the current display surface

[SDL\\_GetVideoInfo](#) — returns a pointer to information about the video hardware

[SDL\\_VideoDriverName](#) — Obtain the name of the video driver

[SDL\\_ListModes](#) — Returns a pointer to an array of available screen dimensions for the given format and video flags

[SDL\\_VideoModeOK](#) — Check to see if a particular video mode is

supported.

[SDL\\_SetVideoMode](#) — Set up a video mode with the specified width, height and bits-per-pixel.

[SDL\\_UpdateRect](#) — Makes sure the given area is updated on the given screen.

[SDL\\_UpdateRects](#) — Makes sure the given list of rectangles is updated on the given screen.

[SDL\\_Flip](#) — Swaps screen buffers

[SDL\\_SetColors](#) — Sets a portion of the colormap for the given 8-bit surface.

[SDL\\_SetPalette](#) — Sets the colors in the palette of an 8-bit surface.

[SDL\\_SetGamma](#) — Sets the color gamma function for the display

[SDL\\_GetGammaRamp](#) — Gets the color gamma lookup tables for the display

[SDL\\_SetGammaRamp](#) — Sets the color gamma lookup tables for the display

[SDL\\_MapRGB](#) — Map a RGB color value to a pixel format.

[SDL\\_MapRGBA](#) — Map a RGBA color value to a pixel format.

[SDL\\_GetRGB](#) — Get RGB values from a pixel in the specified pixel format.

[SDL\\_GetRGBA](#) — Get RGBA values from a pixel in the specified pixel format.

[SDL\\_CreateRGBSurface](#) — Create an empty SDL\_Surface

[SDL\\_CreateRGBSurfaceFrom](#) — Create an SDL\_Surface from pixel data

[SDL\\_FreeSurface](#) — Frees (deletes) a SDL\_Surface

[SDL\\_LockSurface](#) — Lock a surface for directly access.

[SDL\\_UnlockSurface](#) — Unlocks a previously locked surface.

[SDL\\_LoadBMP](#) — Load a Windows BMP file into an SDL\_Surface.

[SDL\\_SaveBMP](#) — Save an SDL\_Surface as a Windows BMP file.

[SDL\\_SetColorKey](#) — Sets the color key (transparent pixel) in a blittable surface and RLE acceleration.

[SDL\\_SetAlpha](#) — Adjust the alpha properties of a surface

[SDL\\_SetClipRect](#) — Sets the clipping rectangle for a surface.

[SDL\\_GetClipRect](#) — Gets the clipping rectangle for a surface.

[SDL\\_ConvertSurface](#) — Converts a surface to the same format as another surface.

[SDL\\_BlitSurface](#) — This performs a fast blit from the source surface to the destination surface.

[SDL\\_FillRect](#) — This function performs a fast fill of the given rectangle with some color

[SDL\\_DisplayFormat](#) — Convert a surface to the display format

[SDL\\_DisplayFormatAlpha](#) — Convert a surface to the display format

[SDL\\_WarpMouse](#) — Set the position of the mouse cursor.

[SDL\\_CreateCursor](#) — Creates a new mouse cursor.

[SDL\\_FreeCursor](#) — Frees a cursor created with [SDL\\_CreateCursor](#).

[SDL\\_SetCursor](#) — Set the currently active mouse cursor.

[SDL\\_GetCursor](#) — Get the currently active mouse cursor.

[SDL\\_ShowCursor](#) — Toggle whether or not the cursor is shown on the screen.

[SDL\\_GL\\_LoadLibrary](#) — Specify an OpenGL library

[SDL\\_GL\\_GetProcAddress](#) — Get the address of a GL function

[SDL\\_GL\\_GetAttribute](#) — Get the value of a special SDL/OpenGL attribute

[SDL\\_GL\\_SetAttribute](#) — Set a special SDL/OpenGL attribute

[SDL\\_GL\\_SwapBuffers](#) — Swap OpenGL framebuffers/Update Display

[SDL\\_CreateYUVOverlay](#) — Create a YUV video overlay

[SDL\\_LockYUVOverlay](#) — Lock an overlay

[SDL\\_UnlockYUVOverlay](#) — Unlock an overlay

[SDL\\_DisplayYUVOverlay](#) — Blit the overlay to the display

[SDL\\_FreeYUVOverlay](#) — Free a YUV video overlay

[SDL\\_GLattr](#) — SDL GL Attributes

[SDL\\_Rect](#) — Defines a rectangular area

[SDL\\_Color](#) — Format independent color description

[SDL\\_Palette](#) — Color palette for 8-bit pixel formats

[SDL\\_PixelFormat](#) — Stores surface format information

[SDL\\_Surface](#) — Graphical Surface Structure

[SDL\\_VideoInfo](#) — Video Target information

[SDL\\_Overlay](#) — YUV video overlay

## 7. [Window Management](#)

[SDL\\_WM\\_SetCaption](#) — Sets the window title and icon name.

[SDL\\_WM\\_GetCaption](#) — Gets the window title and icon name.

[SDL\\_WM\\_SetIcon](#) — Sets the icon for the display window.

[SDL\\_WM\\_IconifyWindow](#) — Iconify/Minimise the window

[SDL\\_WM\\_ToggleFullScreen](#) — Toggles fullscreen mode

[SDL\\_WM\\_GrabInput](#) — Grabs mouse and keyboard input.

## 8. [Events](#)

[Introduction](#)

[SDL Event Structures.](#)

[Event Functions.](#)

## 9. [Joystick](#)

[SDL\\_NumJoysticks](#) — Count available joysticks.

[SDL\\_JoystickName](#) — Get joystick name.

[SDL\\_JoystickOpen](#) — Opens a joystick for use.

[SDL\\_JoystickOpened](#) — Determine if a joystick has been opened

[SDL\\_JoystickIndex](#) — Get the index of an SDL\_Joystick.

[SDL\\_JoystickNumAxes](#) — Get the number of joystick axes

[SDL\\_JoystickNumBalls](#) — Get the number of joystick trackballs

[SDL\\_JoystickNumHats](#) — Get the number of joystick hats

[SDL\\_JoystickNumButtons](#) — Get the number of joystick buttons

[SDL\\_JoystickUpdate](#) — Updates the state of all joysticks

[SDL\\_JoystickGetAxis](#) — Get the current state of an axis

[SDL\\_JoystickGetHat](#) — Get the current state of a joystick hat

[SDL\\_JoystickGetButton](#) — Get the current state of a given button on a given joystick

[SDL\\_JoystickGetBall](#) — Get relative trackball motion

[SDL\\_JoystickClose](#) — Closes a previously opened joystick

## 10. [Audio](#)

[SDL\\_AudioSpec](#) — Audio Specification Structure

[SDL\\_OpenAudio](#) — Opens the audio device with the desired parameters.

[SDL\\_PauseAudio](#) — Pauses and unpauses the audio callback processing

[SDL\\_GetAudioStatus](#) — Get the current audio state

[SDL\\_LoadWAV](#) — Load a WAVE file

[SDL\\_FreeWAV](#) — Frees previously opened WAV data

[SDL\\_AudioCVT](#) — Audio Conversion Structure

[SDL\\_BuildAudioCVT](#) — Initializes a SDL\_AudioCVT structure for conversion

[SDL\\_ConvertAudio](#) — Convert audio data to a desired audio format.

[SDL\\_MixAudio](#) — Mix audio data

[SDL\\_LockAudio](#) — Lock out the callback function

[SDL\\_UnlockAudio](#) — Unlock the callback function

[SDL\\_CloseAudio](#) — Shuts down audio processing and closes the audio device.

## 11. [CD-ROM](#)

[SDL\\_CDNumDrives](#) — Returns the number of CD-ROM drives on the system.

[SDL\\_CDName](#) — Returns a human-readable, system-dependent identifier for the CD-ROM.

[SDL\\_CDOpen](#) — Opens a CD-ROM drive for access.

[SDL\\_CDStatus](#) — Returns the current status of the given drive.

[SDL\\_CDPlay](#) — Play a CD

[SDL\\_CDPlayTracks](#) — Play the given CD track(s)

[SDL\\_CDPause](#) — Pauses a CDROM

[SDL\\_CDResume](#) — Resumes a CDROM

[SDL\\_CDStop](#) — Stops a CDROM

[SDL\\_CDEject](#) — Ejects a CDROM

[SDL\\_CDClose](#) — Closes a SDL\_CD handle

[SDL\\_CD](#) — CDROM Drive Information

[SDL\\_CDtrack](#) — CD Track Information Structure

## 12. [Multi-threaded Programming](#)

[SDL\\_CreateThread](#) — Creates a new thread of execution that shares its parent's properties.

[SDL\\_ThreadID](#) — Get the 32-bit thread identifier for the current thread.

[SDL\\_GetThreadID](#) — Get the SDL thread ID of a SDL\_Thread

[SDL\\_WaitThread](#) — Wait for a thread to finish.

[SDL\\_KillThread](#) — Gracelessly terminates the thread.

[SDL\\_CreateMutex](#) — Create a mutex

[SDL\\_DestroyMutex](#) — Destroy a mutex

[SDL\\_mutexP](#) — Lock a mutex

[SDL\\_mutexV](#) — Unlock a mutex

[SDL\\_CreateSemaphore](#) — Creates a new semaphore and assigns an initial value to it.

[SDL\\_DestroySemaphore](#) — Destroys a semaphore that was created by

[SDL\\_CreateSemaphore](#).

[SDL\\_SemWait](#) — Lock a semaphore and suspend the thread if the semaphore value is zero.

[SDL\\_SemTryWait](#) — Attempt to lock a semaphore but don't suspend the thread.

[SDL\\_SemWaitTimeout](#) — Lock a semaphore, but only wait up to a specified maximum time.

[SDL\\_SemPost](#) — Unlock a semaphore.

[SDL\\_SemValue](#) — Return the current value of a semaphore.

[SDL\\_CreateCond](#) — Create a condition variable  
[SDL\\_DestroyCond](#) — Destroy a condition variable  
[SDL\\_CondSignal](#) — Restart a thread wait on a condition variable  
[SDL\\_CondBroadcast](#) — Restart all threads waiting on a condition variable  
[SDL\\_CondWait](#) — Wait on a condition variable  
[SDL\\_CondWaitTimeout](#) — Wait on a condition variable, with timeout

### 13. [Time](#)

[SDL\\_GetTicks](#) — Get the number of milliseconds since the SDL library initialization.  
[SDL\\_Delay](#) — Wait a specified number of milliseconds before returning.  
[SDL\\_AddTimer](#) — Add a timer which will call a callback after the specified number of milliseconds has elapsed.  
[SDL\\_RemoveTimer](#) — Remove a timer which was added with [SDL\\_AddTimer](#).  
[SDL\\_SetTimer](#) — Set a callback to run after the specified number of milliseconds has elapsed.

### **List of Tables**

- 8-1. [SDL Keysym definitions](#)
- 8-2. [SDL modifier definitions](#)

### **List of Examples**

- 1-1. [Initializing SDL](#)
- 2-1. [Initializing the Video Display](#)
- 2-2. [Initializing the Best Video Mode](#)
- 2-3. [Loading and Displaying a BMP File](#)
- 2-4. [getpixel\(\)](#)
- 2-5. [putpixel\(\)](#)
- 2-6. [Using putpixel\(\)](#)
- 2-7. [Initializing SDL with OpenGL](#)
- 2-8. [SDL and OpenGL](#)
- 3-1. [Initializing SDL with Joystick Support](#)
- 3-2. [Querying the Number of Available Joysticks](#)
- 3-3. [Opening a Joystick](#)
- 3-4. [Joystick Axis Events](#)
- 3-5. [More Joystick Axis Events](#)
- 3-6. [Joystick Button Events](#)

- 3-7. [Joystick Ball Events](#)
  - 3-8. [Joystick Hat Events](#)
  - 3-9. [Querying Joystick Characteristics](#)
  - 3-10. [Reading Keyboard Events](#)
  - 3-11. [Interpreting Key Event Information](#)
  - 3-12. [Proper Game Movement](#)
- 

[Next](#)  
SDL Guide

## SDL Library Documentation

[Prev](#)

[Next](#)

---



# I. SDL Guide

## Table of Contents

[Preface](#)

1. [The Basics](#)

2. [Graphics and Video](#)

3. [Input handling](#)

4. [Examples](#)

---

[Prev](#)

[Home](#)

[Next](#)

SDL Library  
Documentation

Preface

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# Preface

**Table of Contents**

[About SDL](#)

[About SDLdoc](#)

[Credits](#)

# About SDL

The SDL library is designed to make it easy to write games that run on Linux, \*BSD, MacOS, Win32 and BeOS using the various native high-performance media interfaces, (for video, audio, etc) and presenting a single source-code level API to your application. SDL is a fairly low level API, but using it, completely portable applications can be written with a great deal of flexibility.

---

[Prev](#)

SDL Guide

[Home](#)

[Up](#)

[Next](#)

About SDLdoc

# SDL Library Documentation

[Prev](#)

Preface

[Next](#)

---

# About SDLdoc

SDLdoc (The SDL Documentation Project) was formed to completely rewrite the SDL documentation and to keep it continually up to date. The team consists completely of volunteers ranging from people working with SDL in their spare time to people who use SDL in their everyday working lives.

The latest version of this documentation can always be found here:  
<http://sdlloc.csn.ul.ie> Downloadable PS, man pages and html tarballs are available at <http://sdlloc.csn.ul.ie/pub/>

---

[Prev](#)

Preface

[Home](#)

[Up](#)

[Next](#)

Credits

# SDL Library Documentation

[Prev](#)

Preface

[Next](#)

---

# Credits

Sam Lantinga, slouken@libsdl.org

Martin Donlon, akawaka@skynet.ie

Mattias Engdegård

Julian Peterson

Ken Jordan

Maxim Sobolev

Wesley Poole

Michael Vance

Andreas Umbach

Andreas Hofmeister

---

[Prev](#)

About SDLdoc

[Home](#)

[Up](#)

[Next](#)

The Basics



## SDL Library Documentation

[Prev](#)

[Next](#)

---

# Chapter 1. The Basics

**Table of Contents**

[Introduction](#)

[Initializing SDL](#)

# Introduction

The SDL Guide section is pretty incomplete. If you feel you have anything to add mail [akawaka@skynet.ie](mailto:akawaka@skynet.ie) or visit <http://akawaka.csn.ul.ie/tne/>.

---

[Prev](#)

Credits

[Home](#)

[Up](#)

[Next](#)

Initializing SDL

# SDL Library Documentation

## Chapter 1. The Basics

[Prev](#)

[Next](#)

---

# Initializing SDL

SDL is composed of eight subsystems - Audio, CDROM, Event Handling, File I/O, Joystick Handling, Threading, Timers and Video. Before you can use any of these subsystems they must be initialized by calling [SDL\\_Init](#) (or [SDL\\_InitSubSystem](#)). `SDL_Init` must be called before any other SDL function. It automatically initializes the Event Handling, File I/O and Threading subsystems and it takes a parameter specifying which other subsystems to initialize. So, to initialize the default subsystems and the Video subsystems you would call:

```
SDL_Init ( SDL_INIT_VIDEO );
```

To initialize the default subsystems, the Video subsystem and the Timers subsystem you would call:

```
SDL_Init ( SDL_INIT_VIDEO | SDL_INIT_TIMER );
```

`SDL_Init` is complemented by [SDL\\_Quit](#) (and [SDL\\_QuitSubSystem](#)). `SDL_Quit` shuts down all subsystems, including the default ones. It should always be called before a SDL application exits.

With `SDL_Init` and `SDL_Quit` firmly embedded in your programmers toolkit you can write your first and most basic SDL application. However, we must be prepared to handle errors. Many SDL functions return a value and indicates whether the function has succeeded or failed, `SDL_Init`, for instance, returns -1 if it could not initialize a subsystem. SDL provides a useful facility that allows you to determine exactly what the problem was, every time an error occurs within SDL an error message is stored which can be retrieved using `SDL_GetError`. Use this often, you can never know too much about an error.

## Example 1-1. Initializing SDL

```
#include "SDL.h" /* All SDL App's need this */
#include <stdio.h>

int main() {

    printf("Initializing SDL.\n");

    /* Initialize defaults, Video and Audio */
```

```
if((SDL_Init(SDL_INIT_VIDEO|SDL_INIT_AUDIO)==-1)) {
    printf("Could not initialize SDL: %s.\n", SDL_GetError());
    exit(-1);
}

printf("SDL initialized.\n");

printf("Quiting SDL.\n");

/* Shutdown all subsystems */
SDL_Quit();

printf("Quiting....\n");

exit(0);
}
```

---

[Prev](#)

The Basics

[Home](#)

[Up](#)

[Next](#)

Graphics and Video

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# Chapter 2. Graphics and Video

## Table of Contents

[Introduction to SDL Video](#)

[Using OpenGL With SDL](#)



# Introduction to SDL Video

Video is probably the most common thing that SDL is used for, and so it has the most complete subsystem. Here are a few examples to demonstrate the basics.

## Initializing the Video Display

This is what almost all SDL programs have to do in one way or another.

### Example 2-1. Initializing the Video Display

```
SDL_Surface *screen;

/* Initialize the SDL library */
if( SDL_Init(SDL_INIT_VIDEO) < 0 ) {
    fprintf(stderr,
           "Couldn't initialize SDL: %s\n", SDL_GetError());
    exit(1);
}

/* Clean up on exit */
atexit(SDL_Quit);

/*
 * Initialize the display in a 640x480 8-bit palettized mode,
 * requesting a software surface
 */
screen = SDL_SetVideoMode(640, 480, 8, SDL_SWSURFACE);
if ( screen == NULL ) {
    fprintf(stderr, "Couldn't set 640x480x8 video mode: %s\n",
           SDL_GetError());
    exit(1);
}
```

## Initializing the Best Video Mode

If you have a preference for a certain pixel depth but will accept any other, use `SDL_SetVideoMode` with `SDL_ANYFORMAT` as below. You can also use `SDL_VideoModeOK()` to find the native video mode that is closest to the mode you request.

### Example 2-2. Initializing the Best Video Mode

```
/* Have a preference for 8-bit, but accept any depth */
screen = SDL_SetVideoMode(640, 480, 8, SDL_SWSURFACE|SDL_ANYFORM
if ( screen == NULL ) {
    fprintf(stderr, "Couldn't set 640x480x8 video mode: %s\n",
              SDL_GetError());
    exit(1);
}
printf("Set 640x480 at %d bits-per-pixel mode\n",
       screen->format->BitsPerPixel);
```

# Loading and Displaying a BMP File

The following function loads and displays a BMP file given as argument, once SDL is initialised and a video mode has been set.

## Example 2-3. Loading and Displaying a BMP File

```
void display_bmp(char *file_name)
{
    SDL_Surface *image;

    /* Load the BMP file into a surface */
    image = SDL_LoadBMP(file_name);
    if (image == NULL) {
        fprintf(stderr, "Couldn't load %s: %s\n", file_name, SDL_GetError());
        return;
    }

    /*
     * Palettized screen modes will have a default palette (a standard
     * 8*8*4 colour cube), but if the image is palettized as well we
     * use that palette for a nicer colour matching
     */
    if (image->format->palette && screen->format->palette) {
        SDL_SetColors(screen, image->format->palette->colors, 0,
            image->format->palette->ncolors);
    }

    /* Blit onto the screen surface */
    if(SDL_BlitSurface(image, NULL, screen, NULL) < 0)
        fprintf(stderr, "BlitSurface error: %s\n", SDL_GetError());

    SDL_UpdateRect(screen, 0, 0, image->w, image->h);

    /* Free the allocated BMP surface */
    SDL_FreeSurface(image);
}
```

## Drawing Directly to the Display

The following two functions can be used to get and set single pixels of a surface. They are carefully written to work with any depth currently supported by SDL. Remember to lock the surface before calling them, and to unlock it before calling any other SDL functions.

To convert between pixel values and their red, green, blue components, use `SDL_GetRGB()` and `SDL_MapRGB()`.

### Example 2-4. `getpixel()`

```
/*
 * Return the pixel value at (x, y)
 * NOTE: The surface must be locked before calling this!
 */
Uint32 getpixel(SDL_Surface *surface, int x, int y)
{
    int bpp = surface->format->BytesPerPixel;
    /* Here p is the address to the pixel we want to retrieve */
    Uint8 *p = (Uint8 *)surface->pixels + y * surface->pitch + x * b

    switch(bpp) {
    case 1:
        return *p;

    case 2:
        return *(Uint16 *)p;

    case 3:
        if(SDL_BYTEORDER == SDL_BIG_ENDIAN)
            return p[0] << 16 | p[1] << 8 | p[2];
        else
            return p[0] | p[1] << 8 | p[2] << 16;

    case 4:
        return *(Uint32 *)p;

    default:
        return 0;          /* shouldn't happen, but avoids warnings */
    }
}
```

### Example 2-5. `putpixel()`

```

/*
 * Set the pixel at (x, y) to the given value
 * NOTE: The surface must be locked before calling this!
 */
void putpixel(SDL_Surface *surface, int x, int y, Uint32 pixel)
{
    int bpp = surface->format->BytesPerPixel;
    /* Here p is the address to the pixel we want to set */
    Uint8 *p = (Uint8 *)surface->pixels + y * surface->pitch + x * b

    switch(bpp) {
    case 1:
        *p = pixel;
        break;

    case 2:
        *(Uint16 *)p = pixel;
        break;

    case 3:
        if(SDL_BYTEORDER == SDL_BIG_ENDIAN) {
            p[0] = (pixel >> 16) & 0xff;
            p[1] = (pixel >> 8) & 0xff;
            p[2] = pixel & 0xff;
        } else {
            p[0] = pixel & 0xff;
            p[1] = (pixel >> 8) & 0xff;
            p[2] = (pixel >> 16) & 0xff;
        }
        break;

    case 4:
        *(Uint32 *)p = pixel;
        break;
    }
}

```

The following code uses the `putpixel()` function above to set a yellow pixel in the middle of the screen.

### Example 2-6. Using `putpixel()`

```

/* Code to set a yellow pixel at the center of the screen */

int x, y;
Uint32 yellow;

/* Map the color yellow to this display (R=0xff, G=0xFF, B=0x00)

```

```
    Note:  If the display is palettized, you must set the palette
*/
yellow = SDL_MapRGB(screen->format, 0xff, 0xff, 0x00);

x = screen->w / 2;
y = screen->h / 2;

/* Lock the screen for direct access to the pixels */
if ( SDL_MUSTLOCK(screen) ) {
    if ( SDL_LockSurface(screen) < 0 ) {
        fprintf(stderr, "Can't lock screen: %s\n", SDL_GetError());
        return;
    }
}

putpixel(screen, x, y, yellow);

if ( SDL_MUSTLOCK(screen) ) {
    SDL_UnlockSurface(screen);
}
/* Update just the part of the display that we've changed */
SDL_UpdateRect(screen, x, y, 1, 1);

return;
```

---

[Prev](#)

Initializing SDL

[Home](#)

[Up](#)

[Next](#)

Using OpenGL With SDL

**SDL Library Documentation**  
Chapter 2. Graphics and Video

[Prev](#)

[Next](#)

---



# Using OpenGL With SDL

SDL has the ability to create and use OpenGL contexts on several platforms(Linux/X11, Win32, BeOS, MacOS Classic/Toolbox, MacOS X, FreeBSD/X11 and Solaris/X11). This allows you to use SDL's audio, event handling, threads and times in your OpenGL applications (a function often performed by GLUT).

## Initialisation

Initialising SDL to use OpenGL is not very different to initialising SDL normally. There are three differences; you must pass `SDL_OPENGL` to [SDL\\_SetVideoMode](#), you must specify several GL attributes (depth buffer size, framebuffer sizes) using [SDL\\_GL\\_SetAttribute](#) and finally, if you wish to use double buffering you must specify it as a GL attribute, *not* by passing the `SDL_DOUBLEBUF` flag to `SDL_SetVideoMode`.

### Example 2-7. Initializing SDL with OpenGL

```
/* Information about the current video settings. */
const SDL_VideoInfo* info = NULL;
/* Dimensions of our window. */
int width = 0;
int height = 0;
/* Color depth in bits of our window. */
int bpp = 0;
/* Flags we will pass into SDL_SetVideoMode. */
int flags = 0;

/* First, initialize SDL's video subsystem. */
if( SDL_Init( SDL_INIT_VIDEO ) < 0 ) {
    /* Failed, exit. */
    fprintf( stderr, "Video initialization failed: %s\n",
            SDL_GetError( ) );
    quit_tutorial( 1 );
}

/* Let's get some video information. */
info = SDL_GetVideoInfo( );

if( !info ) {
    /* This should probably never happen. */
    fprintf( stderr, "Video query failed: %s\n",
            SDL_GetError( ) );
    quit_tutorial( 1 );
}

/*
 * Set our width/height to 640/480 (you would
 * of course let the user decide this in a normal
 * app). We get the bpp we will request from
 * the display. On X11, VidMode can't change
 * resolution, so this is probably being overly
```

```

    * safe. Under Win32, ChangeDisplaySettings
    * can change the bpp.
    */
width = 640;
height = 480;
bpp = info->vfmt->BitsPerPixel;

/*
 * Now, we want to setup our requested
 * window attributes for our OpenGL window.
 * We want *at least* 5 bits of red, green
 * and blue. We also want at least a 16-bit
 * depth buffer.
 *
 * The last thing we do is request a double
 * buffered window. '1' turns on double
 * buffering, '0' turns it off.
 *
 * Note that we do not use SDL_DOUBLEBUF in
 * the flags to SDL_SetVideoMode. That does
 * not affect the GL attribute state, only
 * the standard 2D blitting setup.
 */
SDL_GL_SetAttribute( SDL_GL_RED_SIZE, 5 );
SDL_GL_SetAttribute( SDL_GL_GREEN_SIZE, 5 );
SDL_GL_SetAttribute( SDL_GL_BLUE_SIZE, 5 );
SDL_GL_SetAttribute( SDL_GL_DEPTH_SIZE, 16 );
SDL_GL_SetAttribute( SDL_GL_DOUBLEBUFFER, 1 );

/*
 * We want to request that SDL provide us
 * with an OpenGL window, in a fullscreen
 * video mode.
 *
 * EXERCISE:
 * Make starting windowed an option, and
 * handle the resize events properly with
 * glViewport.
 */
flags = SDL_OPENGL | SDL_FULLSCREEN;

/*
 * Set the video mode
 */
if( SDL_SetVideoMode( width, height, bpp, flags ) == 0 ) {
    /*
     * This could happen for a variety of reasons,
     * including DISPLAY not being set, the specified
     * resolution not being available, etc.

```

```
    */  
    fprintf( stderr, "Video mode set failed: %s\n",  
            SDL_GetError( ) );  
    quit_tutorial( 1 );  
}
```

# Drawing

Apart from initialisation, using OpenGL within SDL is the same as using OpenGL with any other API, e.g. GLUT. You still use all the same function calls and data types. However if you are using a double-buffered display, then you must use [SDL\\_GL\\_SwapBuffers\(\)](#) to swap the buffers and update the display. To request double-buffering with OpenGL, use [SDL\\_GL\\_SetAttribute](#) with `SDL_GL_DOUBLEBUFFER`, and use [SDL\\_GL\\_GetAttribute](#) to see if you actually got it.

A full example code listing is now presented below.

## Example 2-8. SDL and OpenGL

```
/*
 * SDL OpenGL Tutorial.
 * (c) Michael Vance, 2000
 * briareos@loki games.com
 *
 * Distributed under terms of the LGPL.
 */

#include <SDL/SDL.h>
#include <GL/gl.h>
#include <GL/glu.h>

#include <stdio.h>
#include <stdlib.h>

static GLboolean should_rotate = GL_TRUE;

static void quit_tutorial( int code )
{
    /*
     * Quit SDL so we can release the fullscreen
     * mode and restore the previous video settings,
     * etc.
     */
    SDL_Quit( );

    /* Exit program. */
    exit( code );
}
```

```

static void handle_key_down( SDL_keysym* keysym )
{
    /*
     * We're only interested if 'Esc' has
     * been pressed.
     *
     * EXERCISE:
     * Handle the arrow keys and have that change the
     * viewing position/angle.
     */
    switch( keysym->sym ) {
    case SDLK_ESCAPE:
        quit_tutorial( 0 );
        break;
    case SDLK_SPACE:
        should_rotate = !should_rotate;
        break;
    default:
        break;
    }
}

static void process_events( void )
{
    /* Our SDL event placeholder. */
    SDL_Event event;

    /* Grab all the events off the queue. */
    while( SDL_PollEvent( &event ) ) {

        switch( event.type ) {
        case SDLK_KEYDOWN:
            /* Handle key presses. */
            handle_key_down( &event.key.keysym );
            break;
        case SDL_QUIT:
            /* Handle quit requests (like Ctrl-c). */
            quit_tutorial( 0 );
            break;
        }

    }
}

static void draw_screen( void )
{

```

```

/* Our angle of rotation. */
static float angle = 0.0f;

/*
 * EXERCISE:
 * Replace this awful mess with vertex
 * arrays and a call to glDrawElements.
 *
 * EXERCISE:
 * After completing the above, change
 * it to use compiled vertex arrays.
 *
 * EXERCISE:
 * Verify my windings are correct here ;).
 */
static GLfloat v0[] = { -1.0f, -1.0f, 1.0f };
static GLfloat v1[] = { 1.0f, -1.0f, 1.0f };
static GLfloat v2[] = { 1.0f, 1.0f, 1.0f };
static GLfloat v3[] = { -1.0f, 1.0f, 1.0f };
static GLfloat v4[] = { -1.0f, -1.0f, -1.0f };
static GLfloat v5[] = { 1.0f, -1.0f, -1.0f };
static GLfloat v6[] = { 1.0f, 1.0f, -1.0f };
static GLfloat v7[] = { -1.0f, 1.0f, -1.0f };
static GLubyte red[] = { 255, 0, 0, 255 };
static GLubyte green[] = { 0, 255, 0, 255 };
static GLubyte blue[] = { 0, 0, 255, 255 };
static GLubyte white[] = { 255, 255, 255, 255 };
static GLubyte yellow[] = { 0, 255, 255, 255 };
static GLubyte black[] = { 0, 0, 0, 255 };
static GLubyte orange[] = { 255, 255, 0, 255 };
static GLubyte purple[] = { 255, 0, 255, 0 };

/* Clear the color and depth buffers. */
glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

/* We don't want to modify the projection matrix. */
glMatrixMode( GL_MODELVIEW );
glLoadIdentity( );

/* Move down the z-axis. */
glTranslatef( 0.0, 0.0, -5.0 );

/* Rotate. */
glRotatef( angle, 0.0, 1.0, 0.0 );

if( should_rotate ) {
    if( ++angle > 360.0f ) {
        angle = 0.0f;
    }
}

```

```

    }
}

/* Send our triangle data to the pipeline. */
glBegin( GL_TRIANGLES );

glColor4ubv( red );
glVertex3fv( v0 );
glColor4ubv( green );
glVertex3fv( v1 );
glColor4ubv( blue );
glVertex3fv( v2 );

glColor4ubv( red );
glVertex3fv( v0 );
glColor4ubv( blue );
glVertex3fv( v2 );
glColor4ubv( white );
glVertex3fv( v3 );

glColor4ubv( green );
glVertex3fv( v1 );
glColor4ubv( black );
glVertex3fv( v5 );
glColor4ubv( orange );
glVertex3fv( v6 );

glColor4ubv( green );
glVertex3fv( v1 );
glColor4ubv( orange );
glVertex3fv( v6 );
glColor4ubv( blue );
glVertex3fv( v2 );

glColor4ubv( black );
glVertex3fv( v5 );
glColor4ubv( yellow );
glVertex3fv( v4 );
glColor4ubv( purple );
glVertex3fv( v7 );

glColor4ubv( black );
glVertex3fv( v5 );
glColor4ubv( purple );
glVertex3fv( v7 );
glColor4ubv( orange );
glVertex3fv( v6 );

```



```

glColor4ubv( yellow );
glVertex3fv( v4 );
glColor4ubv( red );
glVertex3fv( v0 );
glColor4ubv( white );
glVertex3fv( v3 );

glColor4ubv( yellow );
glVertex3fv( v4 );
glColor4ubv( white );
glVertex3fv( v3 );
glColor4ubv( purple );
glVertex3fv( v7 );

glColor4ubv( white );
glVertex3fv( v3 );
glColor4ubv( blue );
glVertex3fv( v2 );
glColor4ubv( orange );
glVertex3fv( v6 );

glColor4ubv( white );
glVertex3fv( v3 );
glColor4ubv( orange );
glVertex3fv( v6 );
glColor4ubv( purple );
glVertex3fv( v7 );

glColor4ubv( green );
glVertex3fv( v1 );
glColor4ubv( red );
glVertex3fv( v0 );
glColor4ubv( yellow );
glVertex3fv( v4 );

glColor4ubv( green );
glVertex3fv( v1 );
glColor4ubv( yellow );
glVertex3fv( v4 );
glColor4ubv( black );
glVertex3fv( v5 );

glEnd( );

/*
 * EXERCISE:
 * Draw text telling the user that 'Spc'
 * pauses the rotation and 'Esc' quits.
 * Do it using vetors and textured quads.

```

```

    */

/*
 * Swap the buffers. This this tells the driver to
 * render the next frame from the contents of the
 * back-buffer, and to set all rendering operations
 * to occur on what was the front-buffer.
 *
 * Double buffering prevents nasty visual tearing
 * from the application drawing on areas of the
 * screen that are being updated at the same time.
 */
SDL_GL_SwapBuffers( );
}

static void setup_opengl( int width, int height )
{
    float ratio = (float) width / (float) height;

    /* Our shading model--Gouraud (smooth). */
    glShadeModel( GL_SMOOTH );

    /* Culling. */
    glCullFace( GL_BACK );
    glFrontFace( GL_CCW );
    glEnable( GL_CULL_FACE );

    /* Set the clear color. */
    glClearColor( 0, 0, 0, 0 );

    /* Setup our viewport. */
    glViewport( 0, 0, width, height );

    /*
     * Change to the projection matrix and set
     * our viewing volume.
     */
    glMatrixMode( GL_PROJECTION );
    glLoadIdentity( );
    /*
     * EXERCISE:
     * Replace this with a call to glFrustum.
     */
    gluPerspective( 60.0, ratio, 1.0, 1024.0 );
}

int main( int argc, char* argv[] )
{
    /* Information about the current video settings. */

```

```

const SDL_VideoInfo* info = NULL;
/* Dimensions of our window. */
int width = 0;
int height = 0;
/* Color depth in bits of our window. */
int bpp = 0;
/* Flags we will pass into SDL_SetVideoMode. */
int flags = 0;

/* First, initialize SDL's video subsystem. */
if( SDL_Init( SDL_INIT_VIDEO ) < 0 ) {
    /* Failed, exit. */
    fprintf( stderr, "Video initialization failed: %s\n",
            SDL_GetError( ) );
    quit_tutorial( 1 );
}

/* Let's get some video information. */
info = SDL_GetVideoInfo( );

if( !info ) {
    /* This should probably never happen. */
    fprintf( stderr, "Video query failed: %s\n",
            SDL_GetError( ) );
    quit_tutorial( 1 );
}

/*
 * Set our width/height to 640/480 (you would
 * of course let the user decide this in a normal
 * app). We get the bpp we will request from
 * the display. On X11, VidMode can't change
 * resolution, so this is probably being overly
 * safe. Under Win32, ChangeDisplaySettings
 * can change the bpp.
 */
width = 640;
height = 480;
bpp = info->vfmt->BitsPerPixel;

/*
 * Now, we want to setup our requested
 * window attributes for our OpenGL window.
 * We want *at least* 5 bits of red, green
 * and blue. We also want at least a 16-bit
 * depth buffer.
 *
 * The last thing we do is request a double
 * buffered window. '1' turns on double

```

```

* buffering, '0' turns it off.
*
* Note that we do not use SDL_DOUBLEBUF in
* the flags to SDL_SetVideoMode. That does
* not affect the GL attribute state, only
* the standard 2D blitting setup.
*/
SDL_GL_SetAttribute( SDL_GL_RED_SIZE, 5 );
SDL_GL_SetAttribute( SDL_GL_GREEN_SIZE, 5 );
SDL_GL_SetAttribute( SDL_GL_BLUE_SIZE, 5 );
SDL_GL_SetAttribute( SDL_GL_DEPTH_SIZE, 16 );
SDL_GL_SetAttribute( SDL_GL_DOUBLEBUFFER, 1 );

/*
* We want to request that SDL provide us
* with an OpenGL window, in a fullscreen
* video mode.
*
* EXERCISE:
* Make starting windowed an option, and
* handle the resize events properly with
* glViewport.
*/
flags = SDL_OPENGL | SDL_FULLSCREEN;

/*
* Set the video mode
*/
if( SDL_SetVideoMode( width, height, bpp, flags ) == 0 ) {
    /*
    * This could happen for a variety of reasons,
    * including DISPLAY not being set, the specified
    * resolution not being available, etc.
    */
    fprintf( stderr, "Video mode set failed: %s\n",
             SDL_GetError( ) );
    quit_tutorial( 1 );
}

/*
* At this point, we should have a properly setup
* double-buffered window for use with OpenGL.
*/
setup_opengl( width, height );

/*
* Now we want to begin our normal app process--
* an event loop with a lot of redrawing.
*/

```

```
while( 1 ) {
    /* Process incoming events. */
    process_events( );
    /* Draw the screen. */
    draw_screen( );
}

/*
 * EXERCISE:
 * Record timings using SDL_GetTicks() and
 * and print out frames per second at program
 * end.
 */

/* Never reached. */
return 0;
}
```

---

[Prev](#)

Graphics and Video

[Home](#)

[Up](#)

[Next](#)

Input handling

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# Chapter 3. Input handling

**Table of Contents**

[Handling Joysticks](#)

[Handling the Keyboard](#)

# Handling Joysticks



## Initialization

The first step in using a joystick in a SDL program is to initialize the Joystick subsystems of SDL. This done by passing the `SDL_INIT_JOYSTICK` flag to [SDL\\_Init](#). The joystick flag will usually be used in conjunction with other flags (like the video flag) because the joystick is usually used to control something.

### Example 3-1. Initializing SDL with Joystick Support

```
if (SDL_Init( SDL_INIT_VIDEO | SDL_INIT_JOYSTICK ) < 0)
{
    fprintf(stderr, "Couldn't initialize SDL: %s\n", SDL_GetError());
    exit(1);
}
```

This will attempt to start SDL with both the video and the joystick subsystems activated.

## Querying

If we have reached this point then we can safely assume that the SDL library has been initialized and that the Joystick subsystem is active. We can now call some video and/or sound functions to get things going before we need the joystick. Eventually we have to make sure that there is actually a joystick to work with. It's wise to always check even if you know a joystick will be present on the system because it can also help detect when the joystick is unplugged. The function used to check for joysticks is [SDL\\_NumJoysticks](#).

This function simply returns the number of joysticks available on the system. If it is at least one then we are in good shape. The next step is to determine which joystick the user wants to use. If the number of joysticks available is only one then it is safe to assume that one joystick is the one the user wants to use. SDL has a function to get the name of the joysticks as assigned by the operations system and that function is [SDL\\_JoystickName](#). The joystick is specified by an index where 0 is the first joystick and the last joystick is the number returned by `SDL_NumJoysticks - 1`. In the demonstration a list of all available joysticks is printed to stdout.

### Example 3-2. Querying the Number of Available Joysticks

```
printf("%i joysticks were found.\n\n", SDL_NumJoysticks() );
printf("The names of the joysticks are:\n");

for( i=0; i < SDL_NumJoysticks(); i++ )
{
    printf("    %s\n", SDL_JoystickName(i));
}
```

## Opening a Joystick and Receiving Joystick Events

SDL's event driven architecture makes working with joysticks a snap. Joysticks can trigger 4 different types of events:

- [SDL\\_JoyAxisEvent](#) Occurs when an axis changes
- [SDL\\_JoyBallEvent](#) Occurs when a joystick trackball's position changes
- [SDL\\_JoyHatEvent](#) Occurs when a hat's position changes
- [SDL\\_JoyButtonEvent](#) Occurs when a button is pressed or released

Events are received from all joysticks opened. The first thing that needs to be done in order to receive joystick events is to call [SDL\\_JoystickEventState](#) with the `SDL_ENABLE` flag. Next you must open the joysticks that you want to receive events from. This is done with the [SDL\\_JoystickOpen](#) function. For the example we are only interested in events from the first joystick on the system, regardless of what it may be. To receive events from it we would do this:

### Example 3-3. Opening a Joystick

```
SDL_Joystick *joystick;  
  
SDL_JoystickEventState(SDL_ENABLE);  
joystick = SDL_JoystickOpen(0);
```

If we wanted to receive events for other joysticks we would open them with calls to `SDL_JoystickOpen` just like we opened joystick 0, except we would store the `SDL_Joystick` structure they return in a different pointer. We only need the joystick pointer when we are querying the joysticks or when we are closing the joystick.

Up to this point all the code we have is used just to initialize the joysticks in order to read values at run time. All we need now is an event loop, which is something that all SDL programs should have anyway to receive the systems quit events. We must now add code to check the event loop for at least some of the above mentioned events. Let's assume our event loop looks like this:

```
SDL_Event event;  
/* Other initialization code goes here */
```

```

/* Start main game loop here */

while(SDL_PollEvent(&event))
{
    switch(event.type)
    {
        case SDL_KEYDOWN:
            /* handle keyboard stuff here */
            break;

        case SDL_QUIT:
            /* Set whatever flags are necessary to */
            /* end the main game loop here */
            break;
    }
}

/* End loop here */

```

To handle Joystick events we merely add cases for them, first we'll add axis handling code. Axis checks can get kinda of tricky because alot of the joystick events received are junk. Joystick axis have a tendency to vary just a little between polling due to the way they are designed. To compensate for this you have to set a threshold for changes and ignore the events that have'nt exceeded the threshold. 10% is usually a good threshold value. This sounds a lot more complicated than it is. Here is the Axis event handler:

### Example 3-4. Joystick Axis Events

```

case SDL_JOYAXISMOTION: /* Handle Joystick Motion */
if ( ( event.jaxis.value < -3200 ) || (event.jaxis.value > 3200)
{
    /* code goes here */
}
break;

```

Another trick with axis events is that up-down and left-right movement are two different sets of axes. The most important axis is axis 0 (left-right) and axis 1 (up-down). To handle them seperatly in the code we do the following:

### Example 3-5. More Joystick Axis Events

```

case SDL_JOYAXISMOTION: /* Handle Joystick Motion */
if ( ( event.jaxis.value < -3200 ) || (event.jaxis.value > 3200)
{
    if( event.jaxis.axis == 0)

```

```

    {
        /* Left-right movement code goes here */
    }

    if( event.jaxis.axis == 1)
    {
        /* Up-Down movement code goes here */
    }
}
break;

```

Ideally the code here should use *event.jaxis.value* to scale something. For example lets assume you are using the joystick to control the movement of a spaceship. If the user is using an analog joystick and they push the stick a little bit they expect to move less than if they pushed it a lot. Designing your code for this situation is preferred because it makes the experience for users of analog controls better and remains the same for users of digital controls.

If your joystick has any additional axis then they may be used for other sticks or throttle controls and those axis return values too just with different *event.jaxis.axis* values.

Button handling is simple compared to the axis checking.

### Example 3-6. Joystick Button Events

```

case SDL_JOYBUTTONDOWN: /* Handle Joystick Button Presses */
if ( event.jbutton.button == 0 )
{
    /* code goes here */
}
break;

```

Button checks are simpler than axis checks because a button can only be pressed or not pressed. The `SDL_JOYBUTTONDOWN` event is triggered when a button is pressed and the `SDL_JOYBUTTONUP` event is fired when a button is released. We do have to know what button was pressed though, that is done by reading the *event.jbutton.button* field.

Lastly when we are through using our joysticks we should close them with a call to [SDL\\_JoystickClose](#). To close our opened joystick 0 we would do this at the end of our program:

```
SDL_JoystickClose(joystick);
```

## Advanced Joystick Functions

That takes care of the controls that you can count on being on every joystick under the sun, but there are a few extra things that SDL can support. Joyballs are next on our list, they are alot like axis with a few minor differences. Joyballs store relative changes unlike the the absolute postion stored in a axis event. Also one trackball event contains both the change in x and they change in y. Our case for it is as follows:

### Example 3-7. Joystick Ball Events

```
case SDL_JOYBALLMOTION: /* Handle Joyball Motion */
if( event.jball.ball == 0 )
{
    /* ball handling */
}
break;
```

The above checks the first joyball on the joystick. The change in position will be stored in *event.jball.xrel* and *event.jball.yrel*.

Finally we have the hat event. Hats report only the direction they are pushed in. We check hat's position with the bitmasks:

```
SDL_HAT_CENTERED
SDL_HAT_UP
SDL_HAT_RIGHT
SDL_HAT_DOWN
SDL_HAT_LEFT
```

Also there are some predefined combinations of the above:

```
SDL_HAT_RIGHTUP
SDL_HAT_RIGHTDOWN
SDL_HAT_LEFTUP
SDL_HAT_LEFTDOWN
```

Our case for the hat may resemble the following:

### Example 3-8. Joystick Hat Events

```

case SDL_JOYHATMOTION: /* Handle Hat Motion */
if ( event.jhat.hat & SDL_HAT_UP )
{
    /* Do up stuff here */
}

if ( event.jhat.hat & SDL_HAT_LEFT )
{
    /* Do left stuff here */
}

if ( event.jhat.hat & SDL_HAT_RIGHTDOWN )
{
    /* Do right and down together stuff here */
}
break;

```

In addition to the queries for number of joysticks on the system and their names there are additional functions to query the capabilities of attached joysticks:

[SDL\\_JoystickNumAxes](#) Returns the number of joystick axes  
[SDL\\_JoystickNumButtons](#) Returns the number of joystick buttons  
[SDL\\_JoystickNumBalls](#) Returns the number of joystick balls  
[SDL\\_JoystickNumHats](#) Returns the number of joystick hats

To use these functions we just have to pass in the joystick structure we got when we opened the joystick. For Example:

### Example 3-9. Querying Joystick Characteristics

```

int number_of_buttons;
SDL_Joystick *joystick;

joystick = SDL_JoystickOpen(0);
number_of_buttons = SDL_JoystickNumButtons(joystick);

```

This block of code would get the number of buttons on the first joystick in the system.

---

[Prev](#)

Using OpenGL With SDL

[Home](#)

[Up](#)

[Next](#)

Handling the Keyboard



# SDL Library Documentation

## Chapter 3. Input handling

[Prev](#)

[Next](#)

---

# Handling the Keyboard

# Keyboard Related Structures

It should make it a lot easier to understand this tutorial if you are familiar with the data types involved in keyboard access, so I'll explain them first.

## SDLKey

SDLKey is an enumerated type defined in `SDL/include/SDL_keysym.h` and detailed [here](#). Each SDLKey symbol represents a key, `SDLK_a` corresponds to the 'a' key on a keyboard, `SDLK_SPACE` corresponds to the space bar, and so on.

## SDLMod

SDLMod is an enumerated type, similar to SDLKey, however it enumerates keyboard modifiers (Control, Alt, Shift). The full list of modifier symbols is [here](#). SDLMod values can be AND'd together to represent several modifiers.

## SDL\_keysym

```
typedef struct{
    Uint8 scancode;
    SDLKey sym;
    SDLMod mod;
    Uint16 unicode;
} SDL_keysym;
```

The `SDL_keysym` structure describes a key press or a key release. The *scancode* field is hardware specific and should be ignored unless you know what you're doing. The *sym* field is the SDLKey value of the key being pressed or released. The *mod* field describes the state of the keyboard modifiers at the time the key press or release occurred. So a value of `KMOD_NUM | KMOD_CAPS | KMOD_LSHIFT` would mean that Numlock, Capslock and the left shift key were all pressed (or enabled in the case of the lock keys). Finally, the *unicode* field stores the 16-bit unicode value of the key.

**Note:** It should be noted and understood that this field is only valid when the `SDL_keysym` is describing a key press, not a key release. Unicode values only make sense on a key press because the unicode value describes

an international character and only key presses produce characters. More information on Unicode can be found at [www.unicode.org](http://www.unicode.org)

**Note:** Unicode translation must be enabled using the [SDL\\_EnableUNICODE](#) function.

## SDL\_KeyboardEvent

```
typedef struct{
    Uint8 type;
    Uint8 state;
    SDL_keysym keysym;
} SDL_KeyboardEvent;
```

The `SDL_KeyboardEvent` describes a keyboard event (obviously). The *key* member of the [SDL\\_Event](#) union is a `SDL_KeyboardEvent` structure. The *type* field specifies whether the event is a key release (`SDL_KEYUP`) or a key press (`SDL_KEYDOWN`) event. The *state* is largely redundant, it reports the same information as the *type* field but uses different values (`SDL_RELEASED` and `SDL_PRESSED`). The *keysym* contains information of the key press or release that this event represents (see above).

## Reading Keyboard Events

Reading keyboard events from the event queue is quite simple (the event queue and using it is described [here](#)). We read events using `SDL_PollEvent` in a `while()` loop and check for `SDL_KEYUP` and `SDL_KEYDOWN` events using a switch statement, like so:

### Example 3-10. Reading Keyboard Events

```
SDL_Event event;
.
.
/* Poll for events. SDL_PollEvent() returns 0 when there are no  *
/* more events on the event queue, our while loop will exit when  *
/* that occurs.                                               *
while( SDL_PollEvent( &event ) ){
    /* We are only worried about SDL_KEYDOWN and SDL_KEYUP events */
    switch( event.type ){
        case SDL_KEYDOWN:
            printf( "Key press detected\n" );
            break;

        case SDL_KEYUP:
            printf( "Key release detected\n" );
            break;

        default:
            break;
    }
}
.
.
```

This is a very basic example. No information about the key press or release is interpreted. We will explore the other extreme out our first full example below - reporting all available information about a keyboard event.

## A More Detailed Look

Before we can read events SDL must be initialised with [SDL\\_Init](#) and a video mode must be set using [SDL\\_SetVideoMode](#). There are, however, two other functions we must use to obtain all the information required. We must enable unicode translation by calling `SDL_EnableUNICODE(1)` and we must convert `SDLKey` values into something printable, using [SDL\\_GetKeyName](#)

**Note:** It is useful to note that unicode values  $< 0x80$  translate directly a characters ASCII value. This is used in the example below

### Example 3-11. Interpreting Key Event Information

```
#include "SDL.h"

/* Function Prototypes */
void PrintKeyInfo( SDL_KeyboardEvent *key );
void PrintModifiers( SDLMod mod );

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

    SDL_Event event;
    int quit = 0;

    /* Initialise SDL */
    if( SDL_Init( SDL_INIT_VIDEO ) < 0){
        fprintf( stderr, "Could not initialise SDL: %s\n", SDL_GetError() );
        exit( -1 );
    }

    /* Set a video mode */
    if( !SDL_SetVideoMode( 320, 200, 0, 0 ) ){
        fprintf( stderr, "Could not set video mode: %s\n", SDL_GetError() );
        SDL_Quit();
        exit( -1 );
    }

    /* Enable Unicode translation */
    SDL_EnableUNICODE( 1 );

    /* Loop until an SDL_QUIT event is found */
    while( !quit ){
```

```

/* Poll for events */
while( SDL_PollEvent( &event ) ){

    switch( event.type ){
        /* Keyboard event */
        /* Pass the event data onto PrintKeyInfo() */
        case SDL_KEYDOWN:
        case SDL_KEYUP:
            PrintKeyInfo( &event.key );
            break;

        /* SDL_QUIT event (window close) */
        case SDL_QUIT:
            quit = 1;
            break;

        default:
            break;
    }

}

}

/* Clean up */
SDL_Quit();
exit( 0 );
}

/* Print all information about a key event */
void PrintKeyInfo( SDL_KeyboardEvent *key ){
    /* Is it a release or a press? */
    if( key->type == SDL_KEYUP )
        printf( "Release:- " );
    else
        printf( "Press:- " );

    /* Print the hardware scancode first */
    printf( "Scancode: 0x%02X", key->keysym.scancode );
    /* Print the name of the key */
    printf( ", Name: %s", SDL_GetKeyName( key->keysym.sym ) );
    /* We want to print the unicode info, but we need to make */
    /* sure its a press event first (remember, release events */
    /* don't have unicode info */
    if( key->type == SDL_KEYDOWN ){
        /* If the Unicode value is less than 0x80 then the */
        /* unicode value can be used to get a printable */
        /* representation of the key, using (char)unicode. */
        printf(", Unicode: " );
    }
}

```

```

        if( key->keysym.unicode < 0x80 && key->keysym.unicode >
            printf( "%c (0x%04X)", (char)key->keysym.unicode,
                key->keysym.unicode );
        }
        else{
            printf( "? (0x%04X)", key->keysym.unicode );
        }
    }
    printf( "\n" );
    /* Print modifier info */
    PrintModifiers( key->keysym.mod );
}

/* Print modifier info */
void PrintModifiers( SDLMod mod ){
    printf( "Modifers: " );

    /* If there are none then say so and return */
    if( mod == KMOD_NONE ){
        printf( "None\n" );
        return;
    }

    /* Check for the presence of each SDLMod value */
    /* This looks messy, but there really isn't */
    /* a clearer way. */
    if( mod & KMOD_NUM ) printf( "NUMLOCK " );
    if( mod & KMOD_CAPS ) printf( "CAPSLOCK " );
    if( mod & KMOD_LCTRL ) printf( "LCTRL " );
    if( mod & KMOD_RCTRL ) printf( "RCTRL " );
    if( mod & KMOD_RSHIFT ) printf( "RSHIFT " );
    if( mod & KMOD_LSHIFT ) printf( "LSHIFT " );
    if( mod & KMOD_RALT ) printf( "RALT " );
    if( mod & KMOD_LALT ) printf( "LALT " );
    if( mod & KMOD_CTRL ) printf( "CTRL " );
    if( mod & KMOD_SHIFT ) printf( "SHIFT " );
    if( mod & KMOD_ALT ) printf( "ALT " );
    printf( "\n" );
}

```





```
}  
:  
.
```

At first glance you may think this is a perfectly reasonable piece of code for the task, but it isn't. Like I said keyboard events only occur when a key changes state, so the user would have to press and release the left cursor key 100 times to move the alien 100 pixels to the left.

To get around this problem we must not use the events to change the position of the alien, we use the events to set flags which are then used in a separate section of code to move the alien. Something like this:

### Example 3-12. Proper Game Movement

```
/* Alien screen coordinates */  
int alien_x=0, alien_y=0;  
int alien_xvel=0, alien_yvel=0;  
.  
.  
/* Initialise SDL and video modes and all that */  
.  
/* Main game loop */  
/* Check for events */  
while( SDL_PollEvent( &event ) ){  
    switch( event.type ){  
        /* Look for a keypress */  
        case SDLK_KEYDOWN:  
            /* Check the SDLKey values and move change the coord  
            switch( event.key.keysym.sym ){  
                case SDLK_LEFT:  
                    alien_xvel = -1;  
                    break;  
                case SDLK_RIGHT:  
                    alien_xvel = 1;  
                    break;  
                case SDLK_UP:  
                    alien_yvel = -1;  
                    break;  
                case SDLK_DOWN:  
                    alien_yvel = 1;  
                    break;  
                default:  
                    break;  
            }  
            break;  
        /* We must also use the SDLK_KEYUP events to zero the x *
```

```

/* and y velocity variables. But we must also be          *
/* careful not to zero the velocities when we shouldn't*
case SDL_KEYUP:
    switch( event.key.keysym.sym ){
        case SDLK_LEFT:
            /* We check to make sure the alien is moving
            /* to the left. If it is then we zero the
            /* velocity. If the alien is moving to the
            /* right then the right key is still press
            /* so we don't touch the velocity
            if( alien_xvel < 0 )
                alien_xvel = 0;
            break;
        case SDLK_RIGHT:
            if( alien_xvel > 0 )
                alien_xvel = 0;
            break;
        case SDLK_UP:
            if( alien_yvel < 0 )
                alien_yvel = 0;
            break;
        case SDLK_DOWN:
            if( alien_yvel > 0 )
                alien_yvel = 0;
            break;
        default:
            break;
    }
    break;

default:
    break;
}
}
.
.
/* Update the alien position */
alien_x += alien_xvel;
alien_y += alien_yvel;

```

As can be seen, we use two extra variables, `alien_xvel` and `alien_yvel`, which represent the motion of the ship, it is these variables that we update when we detect keypresses and releases.

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# Chapter 4. Examples

## Table of Contents

[Introduction](#)

[Event Examples](#)

[Audio Examples](#)

[CDROM Examples](#)

[Time Examples](#)

# Introduction

For the moment these examples are taken directly from the old SDL documentation. By the 1.2 release these examples should hopefully deal with most common SDL programming problems.

---

[Prev](#)

Handling the Keyboard

[Home](#)

[Up](#)

[Next](#)

Event Examples

# SDL Library Documentation

## Chapter 4. Examples

[Prev](#)

[Next](#)

---

# Event Examples



## Filtering and Handling Events

```
#include <stdio.h>
#include <stdlib.h>

#include "SDL.h"

/* This function may run in a separate event thread */
int FilterEvents(const SDL_Event *event) {
    static int boycott = 1;

    /* This quit event signals the closing of the window */
    if ( (event->type == SDL_QUIT) && boycott ) {
        printf("Quit event filtered out -- try again.\n");
        boycott = 0;
        return(0);
    }
    if ( event->type == SDL_MOUSEMOTION ) {
        printf("Mouse moved to (%d,%d)\n",
            event->motion.x, event->motion.y);
        return(0);    /* Drop it, we've handled it */
    }
    return(1);
}

int main(int argc, char *argv[])
{
    SDL_Event event;

    /* Initialize the SDL library (starts the event loop) */
    if ( SDL_Init(SDL_INIT_VIDEO) < 0 ) {
        fprintf(stderr,
            "Couldn't initialize SDL: %s\n", SDL_GetError());
        exit(1);
    }

    /* Clean up on exit, exit on window close and interrupt */
    atexit(SDL_Quit);

    /* Ignore key events */
    SDL_EventState(SDL_KEYDOWN, SDL_IGNORE);
    SDL_EventState(SDL_KEYUP, SDL_IGNORE);

    /* Filter quit and mouse motion events */
    SDL_SetEventFilter(FilterEvents);

    /* The mouse isn't much use unless we have a display for referen
```

```

if ( SDL_SetVideoMode(640, 480, 8, 0) == NULL ) {
    fprintf(stderr, "Couldn't set 640x480x8 video mode: %s\n",
        SDL_GetError());
    exit(1);
}

/* Loop waiting for ESC+Mouse_Button */
while ( SDL_WaitEvent(&event) >= 0 ) {
    switch (event.type) {
        case SDL_ACTIVEEVENT: {
            if ( event.active.state & SDL_APPACTIVE ) {
                if ( event.active.gain ) {
                    printf("App activated\n");
                } else {
                    printf("App iconified\n");
                }
            }
        }
        break;

        case SDL_MOUSEBUTTONDOWN: {
            Uint8 *keys;

            keys = SDL_GetKeyState(NULL);
            if ( keys[SDLK_ESCAPE] == SDL_PRESSED ) {
                printf("Bye bye...\n");
                exit(0);
            }
            printf("Mouse button pressed\n");
        }
        break;

        case SDL_QUIT: {
            printf("Quit requested, quitting.\n");
            exit(0);
        }
        break;
    }
}
/* This should never happen */
printf("SDL_WaitEvent error: %s\n", SDL_GetError());
exit(1);
}

```

---

[Prev](#)

Examples

[Home](#)

[Up](#)

[Next](#)

Audio Examples

# SDL Library Documentation

## Chapter 4. Examples

[Prev](#)

[Next](#)

---

# **Audio Examples**

## Opening the audio device

```
SDL_AudioSpec wanted;
extern void fill_audio(void *udata, Uint8 *stream, int len);

/* Set the audio format */
wanted.freq = 22050;
wanted.format = AUDIO_S16;
wanted.channels = 2; /* 1 = mono, 2 = stereo */
wanted.samples = 1024; /* Good low-latency value for callback */
wanted.callback = fill_audio;
wanted.userdata = NULL;

/* Open the audio device, forcing the desired format */
if ( SDL_OpenAudio(&wanted, NULL) < 0 ) {
    fprintf(stderr, "Couldn't open audio: %s\n", SDL_GetError())
    return(-1);
}
return(0);
```

## Playing audio

```
static Uint8 *audio_chunk;
static Uint32 audio_len;
static Uint8 *audio_pos;

/* The audio function callback takes the following parameters:
   stream: A pointer to the audio buffer to be filled
   len:    The length (in bytes) of the audio buffer
*/
void fill_audio(void *udata, Uint8 *stream, int len)
{
    /* Only play if we have data left */
    if ( audio_len == 0 )
        return;

    /* Mix as much data as possible */
    len = ( len > audio_len ? audio_len : len );
    SDL_MixAudio(stream, audio_pos, len, SDL_MIX_MAXVOLUME);
    audio_pos += len;
    audio_len -= len;
}

/* Load the audio data ... */

;;;;;

audio_pos = audio_chunk;

/* Let the callback function play the audio chunk */
SDL_PauseAudio(0);

/* Do some processing */

;;;;;

/* Wait for sound to complete */
while ( audio_len > 0 ) {
    SDL_Delay(100);      /* Sleep 1/10 second */
}
SDL_CloseAudio();
```

---

[Prev](#)

Event Examples

[Home](#)

[Up](#)

[Next](#)

CDROM Examples

# SDL Library Documentation

## Chapter 4. Examples

[Prev](#)

[Next](#)

---

# **CDROM Examples**



## Listing CD-ROM drives

```
#include "SDL.h"

/* Initialize SDL first */
if ( SDL_Init(SDL_INIT_CDROM) < 0 ) {
    fprintf(stderr, "Couldn't initialize SDL: %s\n",SDL_GetError);
    exit(1);
}
atexit(SDL_Quit);

/* Find out how many CD-ROM drives are connected to the system */
printf("Drives available: %d\n", SDL_CDNumDrives());
for ( i=0; i<SDL_CDNumDrives(); ++i ) {
    printf("Drive %d:  \"%s\"\n", i, SDL_CDName(i));
}
```

## Opening the default drive

```
SDL_CD *cdrom;
CDstatus status;
char *status_str;

cdrom = SDL_CDOpen(0);
if ( cdrom == NULL ) {
    fprintf(stderr, "Couldn't open default CD-ROM drive: %s\n",
            SDL_GetError());
    exit(2);
}

status = SDL_CDStatus(cdrom);
switch (status) {
    case CD_TRAYEMPTY:
        status_str = "tray empty";
        break;
    case CD_STOPPED:
        status_str = "stopped";
        break;
    case CD_PLAYING:
        status_str = "playing";
        break;
    case CD_PAUSED:
        status_str = "paused";
        break;
    case CD_ERROR:
        status_str = "error state";
        break;
}
printf("Drive status: %s\n", status_str);
if ( status >= CD_PLAYING ) {
    int m, s, f;
    FRAMES_TO_MSFC(cdrom->cur_frame, &m, &s, &f);
    printf("Currently playing track %d, %d:%2.2d\n",
        cdrom->track[cdrom->cur_track].id, m, s);
}
```

## Listing the tracks on a CD

```
SDL_CD *cdrom;          /* Assuming this has already been set..
int i;
int m, s, f;

SDL_CDStatus(cdrom);
printf("Drive tracks: %d\n", cdrom->numtracks);
for ( i=0; i<cdrom->numtracks; ++i ) {
    FRAMES_TO_MSF(cdrom->track[i].length, &m, &s, &f);
    if ( f > 0 )
        ++s;
    printf("\tTrack (index %d) %d: %d:%2.2d\n", i,
cdrom->track[i].id, m, s);
}
```

## Play an entire CD

```
SDL_CD *cdrom;          /* Assuming this has already been set..

// Play entire CD:
if ( CD_INDRIVE(SDL_CDStatus(cdrom)) )
    SDL_CDPlayTracks(cdrom, 0, 0, 0, 0);

// Play last track:
if ( CD_INDRIVE(SDL_CDStatus(cdrom)) ) {
    SDL_CDPlayTracks(cdrom, cdrom->numtracks-1, 0, 0, 0);
}

// Play first and second track and 10 seconds of third track
if ( CD_INDRIVE(SDL_CDStatus(cdrom)) )
    SDL_CDPlayTracks(cdrom, 0, 0, 2, CD_FPS * 10);
```

---

[Prev](#)

Audio Examples

[Home](#)

[Up](#)

[Next](#)

Time Examples

# SDL Library Documentation

## Chapter 4. Examples

[Prev](#)

[Next](#)

---

# **Time Examples**

# Time based game loop

```
#define TICK_INTERVAL    30

static Uint32 next_time;

Uint32 time_left(void)
{
    Uint32 now;

    now = SDL_GetTicks();
    if(next_time <= now)
        return 0;
    else
        return next_time - now;
}

/* main game loop */

    next_time = SDL_GetTicks() + TICK_INTERVAL;
    while ( game_running ) {
        update_game_state();
        SDL_Delay(time_left());
        next_time += TICK_INTERVAL;
    }
```

---

[Prev](#)

CDROM Examples

[Home](#)

[Up](#)

[Next](#)

SDL Reference

# SDL Library Documentation

## Chapter 4. Examples

[Prev](#)

[Next](#)

---



# **Video Examples**

## Initializing the video display

```
SDL_Surface *screen;

/* Initialize the SDL library */
if( SDL_Init(SDL_INIT_VIDEO) < 0 ) {
    fprintf(stderr,
        "Couldn't initialize SDL: %s\n", SDL_GetError());
    exit(1);
}

/* Clean up on exit */
atexit(SDL_Quit);

/* Initialize the display in a 640x480 8-bit palettized mode */
screen = SDL_SetVideoMode(640, 480, 8, SDL_SWSURFACE);
if ( screen == NULL ) {
    fprintf(stderr, "Couldn't set 640x480x8 video mode: %s\n",
        SDL_GetError());
    exit(1);
}
```

## Initializing the best video mode

```
/* Have a preference for 8-bit, but accept any depth */
screen = SDL_SetVideoMode(640, 480, 8, SDL_SWSURFACE|SDL_ANYFORM
if ( screen == NULL ) {
    fprintf(stderr, "Couldn't set 640x480x8 video mode: %s\n",
        SDL_GetError());
    exit(1);
}
printf("Set 640x480 at %d bits-per-pixel mode\n",
    screen->format->BitsPerPixel);
```

## Loading and displaying a BMP file

```
SDL_Surface *image;
SDL_Rect dest;
int ncolors, i;
SDL_Color *colors;

/* Load the BMP file into a surface */
image = SDL_LoadBMP("sample.bmp");
if ( image == NULL ) {
    fprintf(stderr, "Couldn't load sample.bmp: %s\n",
            SDL_GetError());
    return;
}

/* Set the display colors -- SDL_SetColors() only does something
   palettized displays, but it doesn't hurt anything on HiColor
   TrueColor displays.
   If the display colors have already been set, this step can be
   skipped, and the library will automatically map the image to
   the current display colors.
*/
if ( image->format->palette ) {
    ncolors = image->format->palette->ncolors;
    colors = (SDL_Color *)malloc(ncolors*sizeof(SDL_Color));
    memcpy(colors, image->format->palette->colors, ncolors);
}
else {
    int r, g, b;

    /* Allocate 256 color palette */
    ncolors = 256;
    colors = (SDL_Color *)malloc(ncolors*sizeof(SDL_Color));

    /* Set a 3,3,2 color cube */
    for ( r=0; r<8; ++r ) {
        for ( g=0; g<8; ++g ) {
            for ( b=0; b<4; ++b ) {
                i = ((r<<5)|(g<<2)|b);
                colors[i].r = r<<5;
                colors[i].g = g<<5;
                colors[i].b = b<<6;
            }
        }
    }
}
/* Note: A better way of allocating the palette might be
   to calculate the frequency of colors in the image
```

```
        and create a palette based on that information.
    */
}
/* Set colormap, try for all the colors, but don't worry about i
SDL_SetColors(screen, colors, 0, ncolors);
free(colors);

/* Blit onto the screen surface */
dest.x = 0;
dest.y = 0;
dest.w = image->w;
dest.h = image->h;
SDL_BlitSurface(image, NULL, screen, &dest);

SDL_UpdateRects(screen, 1, &dest);

/* Free the allocated BMP surface */
SDL_FreeSurface(image);
return;
```

## Drawing directly to the display

```
/* Code to set a yellow pixel at the center of the screen */

Sint32  X, Y;
Uint32  pixel;
Uint8   *bits, bpp;

/* Map the color yellow to this display (R=0xFF, G=0xFF, B=0x00)
   Note: If the display is palettized, you must set the palette
   */
pixel = SDL_MapRGB(screen->format, 0xFF, 0xFF, 0x00);

/* Calculate the framebuffer offset of the center of the screen
   if ( SDL_MUSTLOCK(screen) ) {
       if ( SDL_LockSurface(screen) < 0 )
           return;
   }
   bpp = screen->format->BytesPerPixel;
   X = screen->w/2;
   Y = screen->h/2;
   bits = ((Uint8 *)screen->pixels)+Y*screen->pitch+X*bpp;

/* Set the pixel */
switch(bpp) {
    case 1:
        *((Uint8 *)bits) = (Uint8)pixel;
        break;
    case 2:
        *((Uint16 *)bits) = (Uint16)pixel;
        break;
    case 3: { /* Format/endian independent */
        Uint8 r, g, b;

        r = (pixel>>screen->format->Rshift)&0xFF;
        g = (pixel>>screen->format->Gshift)&0xFF;
        b = (pixel>>screen->format->Bshift)&0xFF;
        *((bits)+screen->format->Rshift/8) = r;
        *((bits)+screen->format->Gshift/8) = g;
        *((bits)+screen->format->Bshift/8) = b;
        }
        break;
    case 4:
        *((Uint32 *)bits) = (Uint32)pixel;
        break;
}
```

```
/* Update the display */
if ( SDL_MUSTLOCK(screen) ) {
    SDL_UnlockSurface(screen);
}
SDL_UpdateRect(screen, X, Y, 1, 1);

return;
```

# Fastest possible surface blit

There are three different ways you can draw an image to the screen:

1. Create a surface and use [SDL\\_BlitSurface](#) to blit it to the screen
2. Create the video surface in system memory and call [SDL\\_UpdateRect](#)
3. Create the video surface in video memory and call [SDL\\_LockSurface](#)

The best way to do this is to combine methods:

```
#include <stdio.h>
#include <stdlib.h>
#include "SDL.h"
#include "SDL_timer.h"

void ComplainAndExit(void)
{
    fprintf(stderr, "Problem: %s\n", SDL_GetError());
    exit(1);
}

int main(int argc, char *argv[])
{
    SDL_PixelFormat fmt;
    SDL_Surface *screen, *locked;
    SDL_Surface *imagebmp, *image;
    SDL_Rect dstrect;
    int i;
    Uint8 *buffer;

    /* Initialize SDL */
    if ( SDL_Init(SDL_INIT_VIDEO) < 0 ) {
        ComplainAndExit();
    }
    atexit(SDL_Quit);

    /* Load a BMP image into a surface */
    imagebmp = SDL_LoadBMP("image.bmp");
    if ( imagebmp == NULL ) {
        ComplainAndExit();
    }

    /* Set the video mode (640x480 at native depth) */
    screen = SDL_SetVideoMode(640, 480, 0, SDL_HWSURFACE|SDL_FULLSCREEN);
    if ( screen == NULL ) {
```



```

        ComplainAndExit();
    }

    /* Set the video colormap */
    if ( imagebmp->format->palette != NULL ) {
        SDL_SetColors(screen,
            imagebmp->format->palette->colors, 0,
            imagebmp->format->palette->ncolors);
    }

    /* Convert the image to the video format (maps colors) */
    image = SDL_DisplayFormat(imagebmp);
    SDL_FreeSurface(imagebmp);
    if ( image == NULL ) {
        ComplainAndExit();
    }

    /* Draw bands of color on the raw surface */
    if ( SDL_MUSTLOCK(screen) ) {
        if ( SDL_LockSurface(screen) < 0 )
            ComplainAndExit();
    }
    buffer=(Uint8 *)screen->pixels;
    for ( i=0; i<screen->h; ++i ) {
        memset(buffer, (i*255)/screen->h,
            screen->w*screen->format->BytesPerPixel);
        buffer += screen->pitch;
    }
    if ( SDL_MUSTLOCK(screen) ) {
        SDL_UnlockSurface(screen);
    }

    /* Blit the image to the center of the screen */
    dstrect.x = (screen->w-image->w)/2;
    dstrect.y = (screen->h-image->h)/2;
    dstrect.w = image->w;
    dstrect.h = image->h;
    if ( SDL_BlitSurface(image, NULL, screen, &dstrect) < 0 ) {
        SDL_FreeSurface(image);
        ComplainAndExit();
    }
    SDL_FreeSurface(image);

    /* Update the screen */
    SDL_UpdateRects(screen, 1, &dstrect);

    SDL_Delay(5000);          /* Wait 5 seconds */
    exit(0);
}

```

---

---

[Prev](#)

Examples

[Home](#)

[Up](#)

[Next](#)

Event Examples

## SDL Library Documentation

[Prev](#)

[Next](#)

---

## II. SDL Reference

### Table of Contents

- 5. [General](#)
- 6. [Video](#)
- 7. [Window Management](#)
- 8. [Events](#)
- 9. [Joystick](#)
- 10. [Audio](#)
- 11. [CD-ROM](#)
- 12. [Multi-threaded Programming](#)
- 13. [Time](#)

---

[Prev](#)

[Home](#)

[Next](#)

Time Examples

General

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# Chapter 5. General

## Table of Contents

[SDL\\_Init](#) — Initializes SDL

[SDL\\_InitSubSystem](#) — Initialize subsystems

[SDL\\_QuitSubSystem](#) — Shut down a subsystem

[SDL\\_Quit](#) — Shut down SDL

[SDL\\_WasInit](#) — Check which subsystems are initialized

[SDL\\_GetError](#) — Get SDL error string

Before SDL can be used in a program it must be initialized with [SDL\\_Init](#). `SDL_Init` initializes all the subsystems that the user requests (video, audio, joystick, timers and/or cdrom). Once SDL is initialized with `SDL_Init` subsystems can be shut down and initialized as needed using [SDL\\_InitSubSystem](#) and [SDL\\_QuitSubSystem](#).

SDL must also be shut down before the program exits to make sure it cleans up correctly. Calling [SDL\\_Quit](#) shuts down all subsystems and frees any resources allocated to SDL.

---

[Prev](#)

SDL Reference

[Home](#)

[Up](#)

[Next](#)

SDL\_Init

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# SDL\_Init



## **Name**

SDL\_Init -- Initializes SDL

## Synopsis

```
#include "SDL.h"
```

```
int SDL_Init(Uint32 flags);
```

## Description

Initializes SDL. This should be called before all other SDL functions. The *flags* parameter specifies what part(s) of SDL to initialize.

SDL_INIT_TIMER	Initializes the <a href="#">timer</a> subsystem.
SDL_INIT_AUDIO	Initializes the <a href="#">audio</a> subsystem.
SDL_INIT_VIDEO	Initializes the <a href="#">video</a> subsystem.
SDL_INIT_CDROM	Initializes the <a href="#">cdrom</a> subsystem.
SDL_INIT_JOYSTICK	Initializes the <a href="#">joystick</a> subsystem.
SDL_INIT_EVERYTHING	Initialize all of the above.
SDL_INIT_NOPARACHUTE	Prevents SDL from catching fatal signals.
SDL_INIT_EVENTTHREAD	

## **Return Value**

Returns -1 on an error or 0 on success.

## See Also

[SDL\\_Quit](#), [SDL\\_InitSubSystem](#)

---

[Prev](#)

General

[Home](#)

[Up](#)

[Next](#)

SDL\_InitSubSystem

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_InitSubSystem**

## **Name**

SDL\_InitSubSystem -- Initialize subsystems



## Synopsis

```
#include "SDL.h"
```

```
int SDL_InitSubSystem(Uint32 flags);
```

## Description

After SDL has been initialized with [SDL\\_Init](#) you may initialize uninitialized subsystems with `SDL_InitSubSystem`. The *flags* parameter is the same as that used in [SDL\\_Init](#).

## Examples

```
/* Seperating Joystick and Video initialization. */
SDL_Init(SDL_INIT_VIDEO);
.
.
SDL_SetVideoMode(640, 480, 16, SDL_DOUBLEBUF|SDL_FULLSCREEN);
.
/* Do Some Video stuff */
.
.
/* Initialize the joystick subsystem */
SDL_InitSubSystem(SDL_INIT_JOYSTICK);

/* Do some stuff with video and joystick */
.
.
.
/* Shut them both down */
SDL_Quit();
```

## **Return Value**

Returns -1 on an error or 0 on success.

## See Also

[SDL\\_Init](#), [SDL\\_Quit](#), [SDL\\_QuitSubSystem](#)

---

[Prev](#)

SDL\_Init

[Home](#)

[Up](#)

[Next](#)

SDL\_QuitSubSystem

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_QuitSubSystem**

## **Name**

SDL\_QuitSubSystem -- Shut down a subsystem



## Synopsis

```
#include "SDL.h"
```

```
void SDL_QuitSubSystem(Uint32 flags);
```

## Description

`SDL_QuitSubSystem` allows you to shut down a subsystem that has been previously initialized by [SDL\\_Init](#) or [SDL\\_InitSubSystem](#). The *flags* tells `SDL_QuitSubSystem` which subsystems to shut down, it uses the same values that are passed to [SDL\\_Init](#).

## See Also

[SDL\\_Quit](#), [SDL\\_Init](#), [SDL\\_InitSubSystem](#)

---

[Prev](#)

SDL\_InitSubSystem

[Home](#)

[Up](#)

[Next](#)

SDL\_Quit

## SDL Library Documentation

[Prev](#)

[Next](#)

---

**SDL\_Quit**

## **Name**

SDL\_Quit -- Shut down SDL

## Synopsis

```
#include "SDL.h"
```

```
void SDL_Quit(void);
```

## Description

`SDL_Quit` shuts down all SDL subsystems and frees the resources allocated to them. This should always be called before you exit. For the sake of simplicity you can set `SDL_Quit` as your `atexit` call, like:

```
SDL_Init(SDL_INIT_VIDEO|SDL_INIT_AUDIO);
atexit(SDL_Quit);
.
```

**Note:** While using `atexit` maybe be fine for small programs, more advanced users should shut down SDL in their own cleanup code. Plus, using `atexit` in a library is a sure way to crash dynamically loaded code



## See Also

[SDL\\_QuitSubsystem](#), [SDL\\_Init](#)

---

[Prev](#)

SDL\_QuitSubSystem

[Home](#)

[Up](#)

[Next](#)

SDL\_WasInit

## SDL Library Documentation

[Prev](#)

[Next](#)

---

**SDL\_WasInit**

## **Name**

SDL\_WasInit -- Check which subsystems are initialized

## Synopsis

```
#include "SDL.h"
```

```
Uint32 SDL_WasInit(Uint32 flags);
```

## Description

`SDL_WasInit` allows you to see which SDL subsystems have been [initialized](#). *flags* is a bitwise OR'd combination of the subsystems you wish to check (see [SDL\\_Init](#) for a list of subsystem flags).

## **Return Value**

SDL\_WasInit returns a bitwised OR'd combination of the initialized subsystems.

## Examples

```
/* Here are several ways you can use SDL_WasInit() */

/* Get init data on all the subsystems */
Uint32 subsystem_init;

subsystem_init=SDL_WasInit(SDL_INIT_EVERYTHING);

if(subsystem_init&SDL_INIT_VIDEO)
    printf("Video is initialized.\n");
else
    printf("Video is not initialized.\n");

/* Just check for one specific subsystem */

if(SDL_WasInit(SDL_INIT_VIDEO)!=0)
    printf("Video is initialized.\n");
else
    printf("Video is not initialized.\n");

/* Check for two subsystems */

Uint32 subsystem_mask=SDL_INIT_VIDEO|SDL_INIT_AUDIO;

if(SDL_WasInit(subsystem_mask)==subsystem_mask)
    printf("Video and Audio initialized.\n");
else
    printf("Video and Audio not initialized.\n");
```



## See Also

[SDL\\_Init](#), [SDL\\_Subsystem](#)

---

[Prev](#)

SDL\_Quit

[Home](#)

[Up](#)

[Next](#)

SDL\_GetError

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_GetError**

## **Name**

SDL\_GetError -- Get SDL error string

## Synopsis

```
#include "SDL/SDL.h"
```

```
char *SDL_GetError(void);
```

## **Description**

`SDL_GetError` returns a NULL terminated string containing information about the last internal SDL error.

## Return Value

SDL\_GetError returns a string containing the last error.

---

[Prev](#)

SDL\_WasInit

[Home](#)

[Up](#)

[Next](#)

Video

## SDL Library Documentation

[Prev](#)

[Next](#)

---



# Chapter 6. Video

## Table of Contents

- [SDL\\_GetVideoSurface](#) — returns a pointer to the current display surface
- [SDL\\_GetVideoInfo](#) — returns a pointer to information about the video hardware
- [SDL\\_VideoDriverName](#) — Obtain the name of the video driver
- [SDL\\_ListModes](#) — Returns a pointer to an array of available screen dimensions for the given format and video flags
- [SDL\\_VideoModeOK](#) — Check to see if a particular video mode is supported.
- [SDL\\_SetVideoMode](#) — Set up a video mode with the specified width, height and bits-per-pixel.
- [SDL\\_UpdateRect](#) — Makes sure the given area is updated on the given screen.
- [SDL\\_UpdateRects](#) — Makes sure the given list of rectangles is updated on the given screen.
- [SDL\\_Flip](#) — Swaps screen buffers
- [SDL\\_SetColors](#) — Sets a portion of the colormap for the given 8-bit surface.
- [SDL\\_SetPalette](#) — Sets the colors in the palette of an 8-bit surface.
- [SDL\\_SetGamma](#) — Sets the color gamma function for the display
- [SDL\\_GetGammaRamp](#) — Gets the color gamma lookup tables for the display
- [SDL\\_SetGammaRamp](#) — Sets the color gamma lookup tables for the display
- [SDL\\_MapRGB](#) — Map a RGB color value to a pixel format.
- [SDL\\_MapRGBA](#) — Map a RGBA color value to a pixel format.
- [SDL\\_GetRGB](#) — Get RGB values from a pixel in the specified pixel format.
- [SDL\\_GetRGBA](#) — Get RGBA values from a pixel in the specified pixel format.
- [SDL\\_CreateRGBSurface](#) — Create an empty SDL\_Surface
- [SDL\\_CreateRGBSurfaceFrom](#) — Create an SDL\_Surface from pixel data
- [SDL\\_FreeSurface](#) — Frees (deletes) a SDL\_Surface
- [SDL\\_LockSurface](#) — Lock a surface for directly access.
- [SDL\\_UnlockSurface](#) — Unlocks a previously locked surface.
- [SDL\\_LoadBMP](#) — Load a Windows BMP file into an SDL\_Surface.
- [SDL\\_SaveBMP](#) — Save an SDL\_Surface as a Windows BMP file.
- [SDL\\_SetColorKey](#) — Sets the color key (transparent pixel) in a blittable surface and RLE acceleration.
- [SDL\\_SetAlpha](#) — Adjust the alpha properties of a surface
- [SDL\\_SetClipRect](#) — Sets the clipping rectangle for a surface.
- [SDL\\_GetClipRect](#) — Gets the clipping rectangle for a surface.

[SDL\\_ConvertSurface](#) — Converts a surface to the same format as another surface.

[SDL\\_BlitSurface](#) — This performs a fast blit from the source surface to the destination surface.

[SDL\\_FillRect](#) — This function performs a fast fill of the given rectangle with some color

[SDL\\_DisplayFormat](#) — Convert a surface to the display format

[SDL\\_DisplayFormatAlpha](#) — Convert a surface to the display format

[SDL\\_WarpMouse](#) — Set the position of the mouse cursor.

[SDL\\_CreateCursor](#) — Creates a new mouse cursor.

[SDL\\_FreeCursor](#) — Frees a cursor created with `SDL_CreateCursor`.

[SDL\\_SetCursor](#) — Set the currently active mouse cursor.

[SDL\\_GetCursor](#) — Get the currently active mouse cursor.

[SDL\\_ShowCursor](#) — Toggle whether or not the cursor is shown on the screen.

[SDL\\_GL\\_LoadLibrary](#) — Specify an OpenGL library

[SDL\\_GL\\_GetProcAddress](#) — Get the address of a GL function

[SDL\\_GL\\_GetAttribute](#) — Get the value of a special SDL/OpenGL attribute

[SDL\\_GL\\_SetAttribute](#) — Set a special SDL/OpenGL attribute

[SDL\\_GL\\_SwapBuffers](#) — Swap OpenGL framebuffers/Update Display

[SDL\\_CreateYUVOverlay](#) — Create a YUV video overlay

[SDL\\_LockYUVOverlay](#) — Lock an overlay

[SDL\\_UnlockYUVOverlay](#) — Unlock an overlay

[SDL\\_DisplayYUVOverlay](#) — Blit the overlay to the display

[SDL\\_FreeYUVOverlay](#) — Free a YUV video overlay

[SDL\\_GLAttr](#) — SDL GL Attributes

[SDL\\_Rect](#) — Defines a rectangular area

[SDL\\_Color](#) — Format independent color description

[SDL\\_Palette](#) — Color palette for 8-bit pixel formats

[SDL\\_PixelFormat](#) — Stores surface format information

[SDL\\_Surface](#) — Graphical Surface Structure

[SDL\\_VideoInfo](#) — Video Target information

[SDL\\_Overlay](#) — YUV video overlay

SDL presents a very simple interface to the display framebuffer. The framebuffer is represented as an offscreen surface to which you can write directly. If you want the screen to show what you have written, call the [update](#) function which will guarantee that the desired portion of the screen is updated.

Before you call any of the SDL video functions, you must first call

`SDL_Init(SDL_INIT_VIDEO)`, which initializes the video and events in the SDL library. Check the return code, which should be 0, to see if there were any errors in starting up.

If you use both sound and video in your application, you need to call `SDL_Init(SDL_INIT_AUDIO | SDL_INIT_VIDEO)` before opening the sound device, otherwise under Win32 DirectX, you won't be able to set full-screen display modes.

After you have initialized the library, you can start up the video display in a number of ways. The easiest way is to pick a common screen resolution and depth and just initialize the video, checking for errors. You will probably get what you want, but SDL may be emulating your requested mode and converting the display on update. The best way is to [query](#), for the best video mode closest to the desired one, and then [convert](#) your images to that pixel format.

SDL currently supports any bit depth  $\geq 8$  bits per pixel. 8 bpp formats are considered 8-bit palettized modes, while 12, 15, 16, 24, and 32 bits per pixel are considered "packed pixel" modes, meaning each pixel contains the RGB color components packed in the bits of the pixel.

After you have initialized your video mode, you can take the surface that was returned, and write to it like any other framebuffer, calling the update routine as you go.

When you have finished your video access and are ready to quit your application, you should call "`SDL_Quit()`" to shutdown the video and events.

---

[Prev](#)

SDL\_GetError

[Home](#)

[Up](#)

[Next](#)

SDL\_GetVideoSurface

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_GetVideoSurface**

## **Name**

SDL\_GetVideoSurface -- returns a pointer to the current display surface

## Synopsis

```
#include "SDL.h"
```

```
SDL_Surface *SDL_GetVideoSurface(void);
```

## **Description**

This function returns a pointer to the current display surface. If SDL is doing format conversion on the display surface, this function returns the publicly visible surface, not the real video surface.



## See Also

[SDL\\_Surface](#)

---

[Prev](#)

Video

[Home](#)

[Up](#)

[Next](#)

SDL\_GetVideoInfo

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_GetVideoInfo**

## **Name**

SDL\_GetVideoInfo -- returns a pointer to information about the video hardware

## Synopsis

```
#include "SDL.h"
```

```
SDL_VideoInfo *SDL_GetVideoInfo(void);
```

## Description

This function returns a read-only pointer to [information](#) about the video hardware. If this is called before [SDL\\_SetVideoMode](#), the *vfmt* member of the returned structure will contain the pixel format of the "best" video mode.

## See Also

[SDL\\_SetVideoMode](#), [SDL\\_VideoInfo](#)

---

[Prev](#)

SDL\_GetVideoSurface

[Home](#)

[Up](#)

[Next](#)

SDL\_VideoDriverName

## SDL Library Documentation

[Prev](#)

[Next](#)

---



**SDL\_VideoDriverName**

## **Name**

SDL\_VideoDriverName -- Obtain the name of the video driver

## Synopsis

```
#include "SDL.h"
```

```
char *SDL_VideoDriverName(char *namebuf, int maxlen);
```

## Description

The buffer pointed to by *namebuf* is filled up to a maximum of *maxlen* characters (include the NULL terminator) with the name of the initialised video driver. The driver name is a simple one word identifier like "x11" or "windib".

## Return Value

Returns NULL if video has not been initialised with `SDL_Init` or a pointer to *namebuf* otherwise.

## See Also

[SDL\\_Init](#) [SDL\\_InitSubSystem](#)

---

[Prev](#)

SDL\_GetVideoInfo

[Home](#)

[Up](#)

[Next](#)

SDL\_ListModes

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_ListModes**



## **Name**

SDL\_ListModes -- Returns a pointer to an array of available screen dimensions for the given format and video flags

## Synopsis

```
#include "SDL.h"
```

```
SDL_Rect **SDL_ListModes(SDL_PixelFormat *format, Uint32 flags);
```

## Description

Return a pointer to an array of available screen dimensions for the given format and video flags, sorted largest to smallest. Returns `NULL` if there are no dimensions available for a particular format, or `-1` if any dimension is okay for the given format.

If *format* is `NULL`, the mode list will be for the format returned by [SDL\\_GetVideoInfo\(\)](#)->*vfmt*. The *flag* parameter is an OR'd combination of [surface](#) flags. The flags are the same as those used [SDL\\_SetVideoMode](#) and they play a strong role in deciding what modes are valid. For instance, if you pass `SDL_HWSURFACE` as a flag only modes that support hardware video surfaces will be returned.

## Example

```
SDL_Rect **modes;
int i;
.
.
.

/* Get available fullscreen/hardware modes */
modes=SDL_ListModes(NULL, SDL_FULLSCREEN|SDL_HWSURFACE);

/* Check is there are any modes available */
if(modes == (SDL_Rect **)0){
    printf("No modes available!\n");
    exit(-1);
}

/* Check if or resolution is restricted */
if(modes == (SDL_Rect **)-1){
    printf("All resolutions available.\n");
}
else{
    /* Print valid modes */
    printf("Available Modes\n");
    for(i=0;modes[i];++i)
        printf("  %d x %d\n", modes[i]->w, modes[i]->h);
}
.
.
```

## See Also

[SDL\\_SetVideoMode](#), [SDL\\_GetVideoInfo](#), [SDL\\_Rect](#), [SDL\\_PixelFormat](#)

---

[Prev](#)

SDL\_VideoDriverName

[Home](#)

[Up](#)

[Next](#)

SDL\_VideoModeOK

## SDL Library Documentation

[Prev](#)

[Next](#)

---

**SDL\_VideoModeOK**

## **Name**

SDL\_VideoModeOK -- Check to see if a particular video mode is supported.



## Synopsis

```
#include "SDL.h"
```

```
int SDL_VideoModeOK(int width, int height, int bpp, Uint32 flags);
```

## Description

SDL\_VideoModeOK returns 0 if the requested mode is not supported under any bit depth, or returns the bits-per-pixel of the closest available mode with the given width, height and requested [surface](#) flags (see [SDL\\_SetVideoMode](#)).

The bits-per-pixel value returned is only a suggested mode. You can usually request and bpp you want when [setting](#) the video mode and SDL will emulate that color depth with a shadow video surface.

The arguments to SDL\_VideoModeOK are the same ones you would pass to [SDL\\_SetVideoMode](#)

## Example

```
SDL_Surface *screen;
Uint32 bpp;
.
.
.
printf("Checking mode 640x480@16bpp.\n");
bpp=SDL_VideoModeOK(640, 480, 16, SDL_HWSURFACE);

if(!bpp){
    printf("Mode not available.\n");
    exit(-1);
}

printf("SDL Recommends 640x480@%dbpp.\n", bpp);
screen=SDL_SetVideoMode(640, 480, bpp, SDL_HWSURFACE);
.
.
```

## See Also

[SDL\\_SetVideoMode](#), [SDL\\_GetVideoInfo](#)

---

[Prev](#)

SDL\_ListModes

[Home](#)

[Up](#)

[Next](#)

SDL\_SetVideoMode

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_SetVideoMode**

## **Name**

SDL\_SetVideoMode -- Set up a video mode with the specified width, height and bits-per-pixel.

## Synopsis

```
#include "SDL.h"
```

```
SDL_Surface *SDL_SetVideoMode(int width, int height, int bpp, Uint32 flags);
```



## Description

Set up a video mode with the specified width, height and bits-per-pixel.

If *bpp* is 0, it is treated as the current display bits per pixel.

The *flags* parameter is the same as the *flags* field of the [SDL\\_Surface](#) structure. OR'd combinations of the following values are valid.

SDL_SWSURFACE	Create the video surface in system memory
SDL_HWSURFACE	Create the video surface in video memory
SDL_ASYNCBLIT	Enables the use of asynchronous updates of the display surface. This will usually slow down blitting on single CPU machines, but may provide a speed increase on SMP systems.
SDL_ANYFORMAT	Normally, if a video surface of the requested bits-per-pixel ( <i>bpp</i> ) is not available, SDL will emulate one with a shadow surface. Passing SDL_ANYFORMAT prevents this and causes SDL to use the video surface, regardless of its pixel depth.
SDL_HWPALETTE	Give SDL exclusive palette access. Without this flag you may not always get the the colors you request with <a href="#">SDL_SetColors</a> or <a href="#">SDL_SetPalette</a> .
SDL_DOUBLEBUF	Enable hardware double buffering; only valid with SDL_HWSURFACE. Calling <a href="#">SDL_Flip</a> will flip the buffers and update the screen. All drawing will take place on the surface that is not displayed at the moment. If double buffering could not be enabled then SDL_Flip will just perform a <a href="#">SDL_UpdateRect</a> on the entire screen.
SDL_FULLSCREEN	SDL will attempt to use a fullscreen mode. If a hardware resolution change is not possible (for whatever reason), the next higher resolution will be used and the display window centered on a black background.
SDL_OPENGL	Create an OpenGL rendering context. You should have previously set OpenGL video attributes with <a href="#">SDL_GL_SetAttribute</a> .
	Create an OpenGL rendering context, like above, but allow

SDL_OPENGLBLIT	normal blitting operations. The screen (2D) surface may have an alpha channel, and <a href="#">SDL_UpdateRects</a> must be used for updating changes to the screen surface.
SDL_RESIZABLE	Create a resizable window. When the window is resized by the user a <a href="#">SDL_VIDEORESIZE</a> event is generated and <code>SDL_SetVideoMode</code> can be called again with the new size.
SDL_NOFRAME	If possible, <code>SDL_NOFRAME</code> causes SDL to create a window with no title bar or frame decoration. Fullscreen modes automatically have this flag set.

**Note:** Whatever *flags* `SDL_SetVideoMode` could satisfy are set in the *flags* member of the returned surface.

**Note:** The *bpp* parameter is the number of bits per pixel, so a *bpp* of 24 uses the packed representation of 3 bytes/pixel. For the more common 4 bytes/pixel mode, use a *bpp* of 32. Somewhat oddly, both 15 and 16 will request a 2 bytes/pixel mode, but different pixel formats.

## **Return Value**

The framebuffer surface, or NULL if it fails. The surface returned is freed by `SDL_Quit()` and should not be freed by the caller.

## See Also

[SDL\\_LockSurface](#), [SDL\\_SetColors](#), [SDL\\_Flip](#), [SDL\\_Surface](#)

---

[Prev](#)

SDL\_VideoModeOK

[Home](#)

[Up](#)

[Next](#)

SDL\_UpdateRect

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_UpdateRect**

## **Name**

SDL\_UpdateRect -- Makes sure the given area is updated on the given screen.

## Synopsis

```
#include "SDL.h"
```

```
void SDL_UpdateRect(SDL_Surface *screen, Sint32 x, Sint32 y, Sint32 w,  
Sint32 h);
```



## Description

Makes sure the given area is updated on the given screen. The rectangle must be confined within the screen boundaries (no clipping is done).

If 'x', 'y', 'w' and 'h' are all 0, `SDL_UpdateRect` will update the entire screen.

This function should not be called while '*screen*' is [locked](#).

## See Also

[SDL\\_UpdateRects](#), [SDL\\_Rect](#), [SDL\\_Surface](#), [SDL\\_LockSurface](#)

---

[Prev](#)

SDL\_SetVideoMode

[Home](#)

[Up](#)

[Next](#)

SDL\_UpdateRects

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_UpdateRects**

## **Name**

SDL\_UpdateRects -- Makes sure the given list of rectangles is updated on the given screen.

## Synopsis

```
#include "SDL.h"
```

```
void SDL_UpdateRects(SDL_Surface *screen, int numrects, SDL_Rect *rects);
```

## Description

Makes sure the given list of rectangles is updated on the given screen. The rectangles must all be confined within the screen boundaries (no clipping is done).

This function should not be called while *screen* is [locked](#).

**Note:** It is advised to call this function only once per frame, since each call has some processing overhead. This is no restriction since you can pass any number of rectangles each time.

The rectangles are not automatically merged or checked for overlap. In general, the programmer can use his knowledge about his particular rectangles to merge them in an efficient way, to avoid overdraw.

## See Also

[SDL\\_UpdateRect](#), [SDL\\_Rect](#), [SDL\\_Surface](#), [SDL\\_LockSurface](#)

---

[Prev](#)

SDL\_UpdateRect

[Home](#)

[Up](#)

[Next](#)

SDL\_Flip



## SDL Library Documentation

[Prev](#)

[Next](#)

---

# SDL\_Flip

## **Name**

SDL\_Flip -- Swaps screen buffers

## Synopsis

```
#include "SDL.h"
```

```
int SDL_Flip(SDL_Surface *screen);
```

## Description

On hardware that supports double-buffering, this function sets up a flip and returns. The hardware will wait for vertical retrace, and then swap video buffers before the next video surface blit or lock will return. On hardware that doesn't support double-buffering, this is equivalent to calling [SDL\\_UpdateRect](#)(*screen*, 0, 0, 0, 0)

The `SDL_DOUBLEBUF` flag must have been passed to [SDL\\_SetVideoMode](#), when setting the video mode for this function to perform hardware flipping.

## **Return Value**

This function returns 0 if successful, or -1 if there was an error.

## See Also

[SDL\\_SetVideoMode](#), [SDL\\_UpdateRect](#), [SDL\\_Surface](#)

---

[Prev](#)

SDL\_UpdateRects

[Home](#)

[Up](#)

[Next](#)

SDL\_SetColors

## SDL Library Documentation

[Prev](#)

[Next](#)

---



# **SDL\_SetColors**

## **Name**

SDL\_SetColors -- Sets a portion of the colormap for the given 8-bit surface.

## Synopsis

```
#include "SDL.h"
```

```
int SDL_SetColors(SDL_Surface *surface, SDL_Color *colors, int firstcolor,  
int ncolors);
```

## Description

Sets a portion of the colormap for the given 8-bit surface.

When *surface* is the surface associated with the current display, the display colormap will be updated with the requested colors. If `SDL_HWPALETTE` was set in [SDL\\_SetVideoMode](#) flags, `SDL_SetColors` will always return 1, and the palette is guaranteed to be set the way you desire, even if the window colormap has to be warped or run under emulation.

The color components of a [SDL\\_Color](#) structure are 8-bits in size, giving you a total of  $256^3 = 16777216$  colors.

Palettized (8-bit) screen surfaces with the `SDL_HWPALETTE` flag have two palettes, a logical palette that is used for mapping blits to/from the surface and a physical palette (that determines how the hardware will map the colors to the display). `SDL_SetColors` modifies both palettes (if present), and is equivalent to calling [SDL\\_SetPalette](#) with the *flags* set to `(SDL_LOGPAL | SDL_PHYSPAL)`.

## Return Value

If *surface* is not a palettized surface, this function does nothing, returning 0. If all of the colors were set as passed to `SDL_SetColors`, it will return 1. If not all the color entries were set exactly as given, it will return 0, and you should look at the surface palette to determine the actual color palette.

## Example

```
/* Create a display surface with a grayscale palette */
SDL_Surface *screen;
SDL_Color colors[256];
int i;
.
.
.
/* Fill colors with color information */
for(i=0;i<256;i++){
    colors[i].r=i;
    colors[i].g=i;
    colors[i].b=i;
}

/* Create display */
screen=SDL_SetVideoMode(640, 480, 8, SDL_HWPALETTE);
if(!screen){
    printf("Couldn't set video mode: %s\n", SDL_GetError());
    exit(-1);
}

/* Set palette */
SDL_SetColors(screen, colors, 0, 256);
.
.
.
.
```

## See Also

[SDL\\_Color](#) [SDL\\_Surface](#), [SDL\\_SetPalette](#), [SDL\\_SetVideoMode](#)

---

[Prev](#)

SDL\_Flip

[Home](#)

[Up](#)

[Next](#)

SDL\_SetPalette

## SDL Library Documentation

[Prev](#)

[Next](#)

---



# **SDL\_SetPalette**

## **Name**

SDL\_SetPalette -- Sets the colors in the palette of an 8-bit surface.

## Synopsis

```
#include "SDL.h"
```

```
int SDL_SetPalette(SDL_Surface *surface, int flags, SDL_Color *colors, int  
firstcolor, int ncolors);
```

## Description

Sets a portion of the palette for the given 8-bit surface.

Palettized (8-bit) screen surfaces with the `SDL_HWPALETTE` flag have two palettes, a logical palette that is used for mapping blits to/from the surface and a physical palette (that determines how the hardware will map the colors to the display).

[SDL\\_BlitSurface](#) always uses the logical palette when blitting surfaces (if it has to convert between surface pixel formats). Because of this, it is often useful to modify only one or the other palette to achieve various special color effects (e.g., screen fading, color flashes, screen dimming).

This function can modify either the logical or physical palette by specifying `SDL_LOGPAL` or `SDL_PHYSPAL` in the *flags* parameter.

When *surface* is the surface associated with the current display, the display colormap will be updated with the requested colors. If `SDL_HWPALETTE` was set in [SDL\\_SetVideoMode](#) flags, `SDL_SetPalette` will always return 1, and the palette is guaranteed to be set the way you desire, even if the window colormap has to be warped or run under emulation.

The color components of a [SDL\\_Color](#) structure are 8-bits in size, giving you a total of  $256^3=16777216$  colors.

## Return Value

If *surface* is not a palettized surface, this function does nothing, returning 0. If all of the colors were set as passed to `SDL_SetPalette`, it will return 1. If not all the color entries were set exactly as given, it will return 0, and you should look at the surface palette to determine the actual color palette.

## Example

```
/* Create a display surface with a grayscale palette */
SDL_Surface *screen;
SDL_Color colors[256];
int i;
.
.
.
/* Fill colors with color information */
for(i=0;i<256;i++){
    colors[i].r=i;
    colors[i].g=i;
    colors[i].b=i;
}

/* Create display */
screen=SDL_SetVideoMode(640, 480, 8, SDL_HWPALETTE);
if(!screen){
    printf("Couldn't set video mode: %s\n", SDL_GetError());
    exit(-1);
}

/* Set palette */
SDL_SetPalette(screen, SDL_LOGPAL|SDL_PHYSPAL, colors, 0, 25
.
.
.
.
```

## See Also

[SDL\\_SetColors](#), [SDL\\_SetVideoMode](#), [SDL\\_Surface](#), [SDL\\_Color](#)

---

[Prev](#)

SDL\_SetColors

[Home](#)

[Up](#)

[Next](#)

SDL\_SetGamma

## SDL Library Documentation

[Prev](#)

[Next](#)

---



# **SDL\_SetGamma**

## **Name**

SDL\_SetGamma -- Sets the color gamma function for the display

## Synopsis

```
#include "SDL.h"
```

```
int SDL_SetGamma(float redgamma, float greengamma, float bluegamma);
```

## Description

Sets the "gamma function" for the display of each color component. Gamma controls the brightness/contrast of colors displayed on the screen. A gamma value of 1.0 is identity (i.e., no adjustment is made).

This function adjusts the gamma based on the "gamma function" parameter, you can directly specify lookup tables for gamma adjustment with [SDL\\_SetGammaRamp](#).

Not all display hardware is able to change gamma.

## **Return Value**

Returns -1 on error (or if gamma adjustment is not supported).

## See Also

[SDL\\_GetGammaRamp](#) [SDL\\_SetGammaRamp](#)

---

[Prev](#)

SDL\_SetPalette

[Home](#)

[Up](#)

[Next](#)

SDL\_GetGammaRamp

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_GetGammaRamp**



## **Name**

SDL\_GetGammaRamp -- Gets the color gamma lookup tables for the display

## Synopsis

```
#include "SDL.h"
```

```
int SDL_GetGammaRamp(Uint16 *redtable, Uint16 *greentable, Uint16  
*bluetable);
```

## **Description**

Gets the gamma translation lookup tables currently used by the display. Each table is an array of 256 Uint16 values.

Not all display hardware is able to change gamma.

## **Return Value**

Returns -1 on error.

## See Also

[SDL\\_SetGamma](#) [SDL\\_SetGammaRamp](#)

---

[Prev](#)

SDL\_SetGamma

[Home](#)

[Up](#)

[Next](#)

SDL\_SetGammaRamp

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_SetGammaRamp**

## **Name**

SDL\_SetGammaRamp -- Sets the color gamma lookup tables for the display



## Synopsis

```
#include "SDL.h"
```

```
int SDL_SetGammaRamp(Uint16 *redtable, Uint16 *greentable, Uint16  
*bluetable);
```

## Description

Sets the gamma lookup tables for the display for each color component. Each table is an array of 256 Uint16 values, representing a mapping between the input and output for that channel. The input is the index into the array, and the output is the 16-bit gamma value at that index, scaled to the output color precision. You may pass NULL to any of the channels to leave them unchanged.

This function adjusts the gamma based on lookup tables, you can also have the gamma calculated based on a "gamma function" parameter with [SDL\\_SetGamma](#).

Not all display hardware is able to change gamma.

## **Return Value**

Returns -1 on error (or if gamma adjustment is not supported).

## See Also

[SDL\\_SetGamma](#) [SDL\\_GetGammaRamp](#)

---

[Prev](#)

SDL\_GetGammaRamp

[Home](#)

[Up](#)

[Next](#)

SDL\_MapRGB

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_MapRGB**

## **Name**

SDL\_MapRGB -- Map a RGB color value to a pixel format.

## Synopsis

```
#include "SDL.h"
```

```
Uint32 SDL_MapRGB(SDL_PixelFormat *fmt, Uint8 r, Uint8 g, Uint8 b);
```



## **Description**

Maps the RGB color value to the specified pixel format and returns the pixel value as a 32-bit int.

If the format has a palette (8-bit) the index of the closest matching color in the palette will be returned.

If the specified pixel format has an alpha component it will be returned as all 1 bits (fully opaque).

## **Return Value**

A pixel value best approximating the given RGB color value for a given pixel format. If the pixel format bpp (color depth) is less than 32-bpp then the unused upper bits of the return value can safely be ignored (e.g., with a 16-bpp format the return value can be assigned to a Uint16, and similarly a Uint8 for an 8-bpp format).

## See Also

[SDL\\_GetRGB](#), [SDL\\_GetRGBA](#), [SDL\\_MapRGBA](#), [SDL\\_PixelFormat](#)

---

[Prev](#)

SDL\_SetGammaRamp

[Home](#)

[Up](#)

[Next](#)

SDL\_MapRGBA

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_MapRGBA**

## **Name**

SDL\_MapRGBA -- Map a RGBA color value to a pixel format.

## Synopsis

```
#include "SDL.h"
```

```
Uint32 SDL_MapRGBA(SDL_PixelFormat *fmt, Uint8 r, Uint8 g, Uint8 b, Uint8  
a);
```

## **Description**

Maps the RGBA color value to the specified pixel format and returns the pixel value as a 32-bit int.

If the format has a palette (8-bit) the index of the closest matching color in the palette will be returned.

If the specified pixel format has no alpha component the alpha value will be ignored (as it will be in formats with a palette).



## **Return Value**

A pixel value best approximating the given RGBA color value for a given pixel format. If the pixel format bpp (color depth) is less than 32-bpp then the unused upper bits of the return value can safely be ignored (e.g., with a 16-bpp format the return value can be assigned to a Uint16, and similarly a Uint8 for an 8-bpp format).

## See Also

[SDL\\_GetRGB](#), [SDL\\_GetRGBA](#), [SDL\\_MapRGB](#), [SDL\\_PixelFormat](#)

---

[Prev](#)

SDL\_MapRGB

[Home](#)

[Up](#)

[Next](#)

SDL\_GetRGB

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_GetRGB**

## **Name**

SDL\_GetRGB -- Get RGB values from a pixel in the specified pixel format.

## Synopsis

```
#include "SDL.h"
```

```
void SDL_GetRGB(Uint32 pixel, SDL_PixelFormat *fmt, Uint8 *r, Uint8 *g,  
Uint8 *b);
```

## Description

Get RGB component values from a pixel stored in the specified pixel format.

This function uses the entire 8-bit [0..255] range when converting color components from pixel formats with less than 8-bits per RGB component (e.g., a completely white pixel in 16-bit RGB565 format would return [0xff, 0xff, 0xff] not [0xf8, 0xfc, 0xf8]).

## See Also

[SDL\\_GetRGBA](#), [SDL\\_MapRGB](#), [SDL\\_MapRGBA](#), [SDL\\_PixelFormat](#)

---

[Prev](#)

SDL\_MapRGBA

[Home](#)

[Up](#)

[Next](#)

SDL\_GetRGBA



## SDL Library Documentation

[Prev](#)

[Next](#)

---

**SDL\_GetRGBA**

## **Name**

SDL\_GetRGBA -- Get RGBA values from a pixel in the specified pixel format.

## Synopsis

```
#include "SDL.h"
```

```
void SDL_GetRGBA(Uint32 pixel, SDL_PixelFormat *fmt, Uint8 *r, Uint8 *g,  
Uint8 *b, Uint8 *a);
```

## Description

Get RGBA component values from a pixel stored in the specified pixel format.

This function uses the entire 8-bit [0..255] range when converting color components from pixel formats with less than 8-bits per RGB component (e.g., a completely white pixel in 16-bit RGB565 format would return [0xff, 0xff, 0xff] not [0xf8, 0xfc, 0xf8]).

If the surface has no alpha component, the alpha will be returned as 0xff (100% opaque).

## See Also

[SDL\\_GetRGB](#), [SDL\\_MapRGB](#), [SDL\\_MapRGBA](#), [SDL\\_PixelFormat](#)

---

[Prev](#)

SDL\_GetRGB

[Home](#)

[Up](#)

[Next](#)

SDL\_CreateRGBSurface

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_CreateRGBSurface**



## **Name**

SDL\_CreateRGBSurface -- Create an empty SDL\_Surface

## Synopsis

```
#include "SDL.h"
```

```
SDL_Surface *SDL_CreateRGBSurface(Uint32 flags, int width, int height, int  
depth, Uint32 Rmask, Uint32 Gmask, Uint32 Bmask, Uint32 Amask);
```

## Description

Allocate an empty surface (must be called after [SDL\\_SetVideoMode](#))

If *depth* is 8 bits an empty palette is allocated for the surface, otherwise a 'packed-pixel' [SDL\\_PixelFormat](#) is created using the *[RGBA]mask*'s provided (see [SDL\\_PixelFormat](#)). The *flags* specifies the type of surface that should be created, it is an OR'd combination of the following possible values.

SDL_SWSURFACE	SDL will create the surface in system memory. This improves the performance of pixel level access, however you may not be able to take advantage of some types of hardware blitting.
SDL_HWSURFACE	SDL will attempt to create the surface in video memory. This will allow SDL to take advantage of Video->Video blits (which are often accelerated).
SDL_SRCOLORKEY	This flag turns on colourkeying for blits from this surface. If SDL_HWSURFACE is also specified and colourkeyed blits are hardware-accelerated, then SDL will attempt to place the surface in video memory. Use <a href="#">SDL_SetColorKey</a> to set or clear this flag after surface creation.
SDL_SRCALPHA	This flag turns on alpha-blending for blits from this surface. If SDL_HWSURFACE is also specified and alpha-blending blits are hardware-accelerated, then the surface will be placed in video memory if possible. Use <a href="#">SDL_SetAlpha</a> to set or clear this flag after surface creation.

**Note:** If an alpha-channel is specified (that is, if *Amask* is nonzero), then the SDL\_SRCALPHA flag is automatically set. You may remove this flag by calling [SDL\\_SetAlpha](#) after surface creation.

## **Return Value**

Returns the created surface, or NULL upon error.

## Example

```
/* Create a 32-bit surface with the bytes of each pixel in R,G,B
   as expected by OpenGL for textures */
SDL_Surface *surface;
Uint32 rmask, gmask, bmask, amask;

/* SDL interprets each pixel as a 32-bit number, so our masks mu
   on the endianness (byte order) of the machine */
#if SDL_BYTEORDER == SDL_BIG_ENDIAN
    rmask = 0xff000000;
    gmask = 0x00ff0000;
    bmask = 0x0000ff00;
    amask = 0x000000ff;
#else
    rmask = 0x000000ff;
    gmask = 0x0000ff00;
    bmask = 0x00ff0000;
    amask = 0xff000000;
#endif

surface = SDL_CreateRGBSurface(SDL_SWSURFACE, width, height, 32,
                               rmask, gmask, bmask, amask);
if(surface == NULL) {
    fprintf(stderr, "CreateRGBSurface failed: %s\n", SDL_GetError());
    exit(1);
}
```

## See Also

[SDL\\_CreateRGBSurfaceFrom](#), [SDL\\_FreeSurface](#), [SDL\\_SetVideoMode](#),  
[SDL\\_LockSurface](#), [SDL\\_PixelFormat](#), [SDL\\_Surface](#) [SDL\\_SetAlpha](#)  
[SDL\\_SetColorKey](#)

---

[Prev](#)

SDL\_GetRGBA

[Home](#)

[Up](#)

[Next](#)

SDL\_CreateRGBSurfaceFrom

## SDL Library Documentation

[Prev](#)

[Next](#)

---

**SDL\_CreateRGBSurfaceFrom**



## **Name**

SDL\_CreateRGBSurfaceFrom -- Create an SDL\_Surface from pixel data

## Synopsis

```
#include "SDL.h"
```

```
SDL_Surface *SDL_CreateRGBSurfaceFrom(void *pixels, int width, int height,  
int depth, int pitch, Uint32 Rmask, Uint32 Gmask, Uint32 Bmask, Uint32  
Amask);
```

## Description

Creates an `SDL_Surface` from the provided pixel data.

The data stored in *pixels* is assumed to be of the *depth* specified in the parameter list. The pixel data is not copied into the `SDL_Surface` structure so it should not be freed until the surface has been freed with a call to [SDL\\_FreeSurface](#). *pitch* is the length of each scanline in bytes.

See [SDL\\_CreateRGBSurface](#) for a more detailed description of the other parameters.

## **Return Value**

Returns the created surface, or NULL upon error.

## See Also

[SDL\\_CreateRGBSurface](#), [SDL\\_FreeSurface](#)

---

[Prev](#)

SDL\_CreateRGBSurface

[Home](#)

[Up](#)

[Next](#)

SDL\_FreeSurface

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_FreeSurface**

## **Name**

SDL\_FreeSurface -- Frees (deletes) a SDL\_Surface



## Synopsis

```
#include "SDL.h"
```

```
void SDL_FreeSurface(SDL_Surface *surface);
```

## Description

Frees the resources used by a previously created `SDL_Surface`. If the surface was created using [SDL\\_CreateRGBSurfaceFrom](#) then the pixel data is not freed.

## See Also

[SDL\\_CreateRGBSurface](#) [SDL\\_CreateRGBSurfaceFrom](#)

---

[Prev](#)

SDL\_CreateRGBSurfaceFrom

[Home](#)

[Up](#)

[Next](#)

SDL\_LockSurface

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_LockSurface**

## **Name**

SDL\_LockSurface -- Lock a surface for directly access.

## Synopsis

```
#include "SDL.h"
```

```
int SDL_LockSurface(SDL_Surface *surface);
```

## Description

SDL\_LockSurface sets up a surface for directly accessing the pixels. Between calls to SDL\_LockSurface and SDL\_UnlockSurface, you can write to and read from *surface->pixels*, using the pixel format stored in *surface->format*. Once you are done accessing the surface, you should use SDL\_UnlockSurface to release it.

Not all surfaces require locking. If `SDL_MUSTLOCK(surface)` evaluates to 0, then you can read and write to the surface at any time, and the pixel format of the surface will not change.

No operating system or library calls should be made between lock/unlock pairs, as critical system locks may be held during this time.

It should be noted, that since SDL 1.1.8 surface locks are recursive. This means that you can lock a surface multiple times, but each lock must have a match unlock.

```
.
.
SDL_LockSurface( surface );
.
/* Surface is locked */
/* Direct pixel access on surface here */
.
SDL_LockSurface( surface );
.
/* More direct pixel access on surface */
.
SDL_UnlockSurface( surface );
/* Surface is still locked */
/* Note: In versions < 1.1.8, the surface would have been */
/* no longer locked at this stage */
.
SDL_UnlockSurface( surface );
/* Surface is now unlocked */
.
.
```



## **Return Value**

SDL\_LockSurface returns 0, or -1 if the surface couldn't be locked.

## See Also

[SDL\\_UnlockSurface](#)

---

[Prev](#)

SDL\_FreeSurface

[Home](#)

[Up](#)

[Next](#)

SDL\_UnlockSurface

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_UnlockSurface**

## **Name**

SDL\_UnlockSurface -- Unlocks a previously locked surface.

## Synopsis

```
#include "SDL.h"
```

```
void SDL_UnlockSurface(SDL_Surface *surface);
```

## Description

Surfaces that were previously locked using `SDL_LockSurface` must be unlocked with `SDL_UnlockSurface`. Surfaces should be unlocked as soon as possible.

It should be noted that since 1.1.8, surface locks are recursive. See [SDL\\_LockSurface](#).

## See Also

[SDL\\_LockSurface](#)

---

[Prev](#)

SDL\_LockSurface

[Home](#)

[Up](#)

[Next](#)

SDL\_LoadBMP



## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_LoadBMP**

## **Name**

SDL\_LoadBMP -- Load a Windows BMP file into an SDL\_Surface.

## Synopsis

```
#include "SDL.h"
```

```
SDL_Surface *SDL_LoadBMP(const char *file);
```

## **Description**

Loads a surface from a named Windows BMP file.

## **Return Value**

Returns the new surface, or NULL if there was an error.

## See Also

[SDL\\_SaveBMP](#)

---

[Prev](#)

SDL\_UnlockSurface

[Home](#)

[Up](#)

[Next](#)

SDL\_SaveBMP

## SDL Library Documentation

[Prev](#)

[Next](#)

---



# **SDL\_SaveBMP**

## **Name**

SDL\_SaveBMP -- Save an SDL\_Surface as a Windows BMP file.

## Synopsis

```
#include "SDL.h"
```

```
int SDL_SaveBMP(SDL_Surface *surface, const char *file);
```

## **Description**

Saves the SDL\_Surface *surface* as a Windows BMP file named *file*.

## **Return Value**

Returns 0 if successful or -1 if there was an error.

## See Also

[SDL\\_LoadBMP](#)

---

[Prev](#)

SDL\_LoadBMP

[Home](#)

[Up](#)

[Next](#)

SDL\_SetColorKey

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_SetColorKey**



## **Name**

SDL\_SetColorKey -- Sets the color key (transparent pixel) in a blittable surface and RLE acceleration.

## Synopsis

```
#include "SDL.h"
```

```
int SDL_SetColorKey(SDL_Surface *surface, Uint32 flag, Uint32 key);
```

## Description

Sets the color key (transparent pixel) in a blittable surface and enables or disables RLE blit acceleration.

RLE acceleration can substantially speed up blitting of images with large horizontal runs of transparent pixels (i.e., pixels that match the *key* value). The *key* must be of the same pixel format as the *surface*, [SDL\\_MapRGB](#) is often useful for obtaining an acceptable value.

If *flag* is `SDL_SRCCOLORKEY` then *key* is the transparent pixel value in the source image of a blit.

If *flag* is OR'd with `SDL_RLEACCEL` then the surface will be draw using RLE acceleration when drawn with [SDL\\_BlitSurface](#). The surface will actually be encoded for RLE acceleration the first time [SDL\\_BlitSurface](#) or [SDL\\_DisplayFormat](#) is called on the surface.

If *flag* is 0, this function clears any current color key.

## **Return Value**

This function returns 0, or -1 if there was an error.

## See Also

[SDL\\_BlitSurface](#), [SDL\\_DisplayFormat](#), [SDL\\_MapRGB](#), [SDL\\_SetAlpha](#)

---

[Prev](#)

SDL\_SaveBMP

[Home](#)

[Up](#)

[Next](#)

SDL\_SetAlpha

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_SetAlpha**

## **Name**

SDL\_SetAlpha -- Adjust the alpha properties of a surface



## Synopsis

```
#include "SDL.h"
```

```
int SDL_SetAlpha(SDL_Surface *surface, Uint32 flag, Uint8 alpha);
```

## Description

**Note:** This function and the semantics of SDL alpha blending have changed since version 1.1.4. Up until version 1.1.5, an alpha value of 0 was considered opaque and a value of 255 was considered transparent. This has now been inverted: 0 (SDL\_ALPHA\_TRANSPARENT) is now considered transparent and 255 (SDL\_ALPHA\_OPAQUE) is now considered opaque.

SDL\_SetAlpha is used for setting the per-surface alpha value and/or enabling and disabling alpha blending.

The *surface* parameter specifies which surface whose alpha attributes you wish to adjust. *flags* is used to specify whether alpha blending should be used (SDL\_SRCALPHA) and whether the surface should use RLE acceleration for blitting (SDL\_RLEACCEL). *flags* can be an OR'd combination of these two options, one of these options or 0. If SDL\_SRCALPHA is not passed as a flag then all alpha information is ignored when blitting the surface. The *alpha* parameter is the per-surface alpha value; a surface need not have an alpha channel to use per-surface alpha and blitting can still be accelerated with SDL\_RLEACCEL.

**Note:** The per-surface alpha value of 128 is considered a special case and is optimised, so it's much faster than other per-surface values.

Alpha effects surface blitting in the following ways:

RGBA->RGB  
with  
SDL\_SRCALPHA The source is alpha-blended with the destination, using the alpha channel. SDL\_SRCCOLORKEY and the per-surface alpha are ignored.

RGBA->RGB  
without  
SDL\_SRCALPHA The RGB data is copied from the source. The source alpha channel and the per-surface alpha value are ignored.

RGB->RGBA  
with  
SDL\_SRCALPHA The source is alpha-blended with the destination using the per-surface alpha value. If SDL\_SRCCOLORKEY is set, only the pixels not matching the colorkey value are copied. The alpha channel of the copied pixels is set to opaque.

RGB->RGBA The RGB data is copied from the source and the alpha value of  
without the copied pixels is set to opaque. If SDL\_SRCCOLORKEY is set,  
SDL\_SRCALPHA only the pixels not matching the colorkey value are copied.

RGBA->RGBA The source is alpha-blended with the destination using the source  
with alpha channel. The alpha channel in the destination surface is left  
SDL\_SRCALPHA untouched. SDL\_SRCCOLORKEY is ignored.

RGBA->RGBA The RGBA data is copied to the destination surface. If  
without SDL\_SRCCOLORKEY is set, only the pixels not matching the  
SDL\_SRCALPHA colorkey value are copied.

RGB->RGB The source is alpha-blended with the destination using the per-  
with surface alpha value. If SDL\_SRCCOLORKEY is set, only the pixels  
SDL\_SRCALPHA not matching the colorkey value are copied.

RGB->RGB The RGB data is copied from the source. If SDL\_SRCCOLORKEY is  
without set, only the pixels not matching the colorkey value are copied.  
SDL\_SRCALPHA

**Note:** Note that RGBA->RGBA blits (with SDL\_SRCALPHA set) keep the alpha of the destination surface. This means that you cannot compose two arbitrary RGBA surfaces this way and get the result you would expect from "overlying" them; the destination alpha will work as a mask.

Also note that per-pixel and per-surface alpha cannot be combined; the per-pixel alpha is always used if available

## **Return Value**

This function returns 0, or -1 if there was an error.

## See Also

[SDL\\_MapRGBA](#), [SDL\\_GetRGBA](#), [SDL\\_DisplayFormatAlpha](#), [SDL\\_BlitSurface](#)

---

[Prev](#)

SDL\_SetColorKey

[Home](#)

[Up](#)

[Next](#)

SDL\_SetClipRect

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_SetClipRect**

## **Name**

SDL\_SetClipRect -- Sets the clipping rectangle for a surface.



## Synopsis

```
#include "SDL.h"
```

```
void SDL_SetClipRect(SDL_Surface *surface, SDL_Rect *rect);
```

## Description

Sets the clipping rectangle for a surface. When this surface is the destination of a blit, only the area within the clip rectangle will be drawn into.

The rectangle pointed to by *rect* will be clipped to the edges of the surface so that the clip rectangle for a surface can never fall outside the edges of the surface.

If *rect* is NULL the clipping rectangle will be set to the full size of the surface.

## See Also

[SDL\\_GetClipRect](#), [SDL\\_BlitSurface](#), [SDL\\_Surface](#)

---

[Prev](#)

SDL\_SetAlpha

[Home](#)

[Up](#)

[Next](#)

SDL\_GetClipRect

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_GetClipRect**

## **Name**

SDL\_GetClipRect -- Gets the clipping rectangle for a surface.

## Synopsis

```
#include "SDL.h"
```

```
void SDL_GetClipRect(SDL_Surface *surface, SDL_Rect *rect);
```

## Description

Gets the clipping rectangle for a surface. When this surface is the destination of a blit, only the area within the clip rectangle is drawn into.

The rectangle pointed to by *rect* will be filled with the clipping rectangle of the surface.



## See Also

[SDL\\_SetClipRect](#), [SDL\\_BlitterSurface](#), [SDL\\_Surface](#)

---

[Prev](#)

SDL\_SetClipRect

[Home](#)

[Up](#)

[Next](#)

SDL\_ConvertSurface

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_ConvertSurface**

## **Name**

SDL\_ConvertSurface -- Converts a surface to the same format as another surface.

## Synopsis

```
#include "SDL/SDL.h"
```

```
SDL_Surface *SDL_ConvertSurface(SDL_Surface *src, SDL_PixelFormat  
*fmt, Uint32 flags);
```

## Description

Creates a new surface of the specified format, and then copies and maps the given surface to it. If this function fails, it returns NULL.

The *flags* parameter is passed to [SDL\\_CreateRGBSurface](#) and has those semantics.

This function is used internally by [SDL\\_DisplayFormat](#).

This function can only be called after `SDL_Init`.

## **Return Value**

Returns either a pointer to the new surface, or NULL on error.

## See Also

[SDL\\_CreateRGBSurface](#), [SDL\\_DisplayFormat](#), [SDL\\_PixelFormat](#), [SDL\\_Surface](#)

---

[Prev](#)

SDL\_GetClipRect

[Home](#)

[Up](#)

[Next](#)

SDL\_BlitSurface



## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_BlitSurface**

## **Name**

SDL\_BlitterSurface -- This performs a fast blit from the source surface to the destination surface.

## Synopsis

```
#include "SDL.h"
```

```
int SDL_BlitSurface(SDL_Surface *src, SDL_Rect *srcrect, SDL_Surface  
*dst, SDL_Rect *dstrect);
```

## Description

This performs a fast blit from the source surface to the destination surface.

Only the position is used in the *dstrect* (the width and height are ignored).

If either *srcrect* or *dstrect* are NULL, the entire surface (*src* or *dst*) is copied.

The final blit rectangle is saved in *dstrect* after all clipping is performed (*srcrect* is not modified).

The blit function should not be called on a locked surface.

The results of blitting operations vary greatly depending on whether `SDL_SRCALPHA` is set or not. See [SDL\\_SetAlpha](#) for an explanation of how this affects your results. Colorkeying and alpha attributes also interact with surface blitting, as the following pseudo-code should hopefully explain.

```
if (source surface has SDL_SRCALPHA set) {
    if (source surface has alpha channel (that is, format->Amask !=
        blit using per-pixel alpha, ignoring any colour key
    else {
        if (source surface has SDL_SRCCOLORKEY set)
            blit using the colour key AND the per-surface alpha valu
        else
            blit using the per-surface alpha value
    }
} else {
    if (source surface has SDL_SRCCOLORKEY set)
        blit using the colour key
    else
        ordinary opaque rectangular blit
}
```

## Return Value

If the blit is successful, it returns 0, otherwise it returns -1.

If either of the surfaces were in video memory, and the blit returns -2, the video memory was lost, so it should be reloaded with artwork and re-blitted:

```
while ( SDL_BlitSurface(image, imgrect, screen, dstrect) ==
        while ( SDL_LockSurface(image)) < 0 )
    Sleep(10);
    -- Write image pixels to image->pixels --
    SDL_UnlockSurface(image);
}
```

This happens under DirectX 5.0 when the system switches away from your fullscreen application. Locking the surface will also fail until you have access to the video memory again.

## See Also

[SDL\\_LockSurface](#), [SDL\\_FillRect](#), [SDL\\_Surface](#), [SDL\\_Rect](#)

---

[Prev](#)

SDL\_ConvertSurface

[Home](#)

[Up](#)

[Next](#)

SDL\_FillRect

## SDL Library Documentation

[Prev](#)

[Next](#)

---



# **SDL\_FillRect**

## **Name**

SDL\_FillRect -- This function performs a fast fill of the given rectangle with some color

## Synopsis

```
#include "SDL.h"
```

```
int SDL_FillRect(SDL_Surface *dst, SDL_Rect *dstrect, Uint32 color);
```

## Description

This function performs a fast fill of the given rectangle with *color*. If *dstrect* is `NULL`, the whole surface will be filled with *color*.

The color should be a pixel of the format used by the surface, and can be generated by the [SDL\\_MapRGB](#) or [SDL\\_MapRGBA](#) functions. If the color value contains an alpha value then the destination is simply "filled" with that alpha information, no blending takes place.

If there is a clip rectangle set on the destination (set via [SDL\\_SetClipRect](#)) then this function will clip based on the intersection of the clip rectangle and the *dstrect* rectangle and the *dstrect* rectangle will be modified to represent the area actually filled.

## **Return Value**

This function returns 0 on success, or -1 on error.

## See Also

[SDL\\_MapRGB](#), [SDL\\_MapRGBA](#), [SDL\\_BlitterSurface](#), [SDL\\_Rect](#)

---

[Prev](#)

SDL\_BlitterSurface

[Home](#)

[Up](#)

[Next](#)

SDL\_DisplayFormat

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_DisplayFormat**



## **Name**

SDL\_DisplayFormat -- Convert a surface to the display format

## Synopsis

```
#include "SDL.h"
```

```
SDL_Surface *SDL_DisplayFormat(SDL_Surface *surface);
```

## Description

This function takes a surface and copies it to a new surface of the pixel format and colors of the video framebuffer, suitable for fast blitting onto the display surface. It calls [SDL\\_ConvertSurface](#)

If you want to take advantage of hardware colorkey or alpha blit acceleration, you should set the colorkey and alpha value before calling this function.

If you want an alpha channel, see [SDL\\_DisplayFormatAlpha](#).

## **Return Value**

If the conversion fails or runs out of memory, it returns NULL

## See Also

[SDL\\_ConvertSurface](#), [SDL\\_DisplayFormatAlpha](#) [SDL\\_SetAlpha](#),  
[SDL\\_SetColorKey](#), [SDL\\_Surface](#)

---

[Prev](#)

SDL\_FillRect

[Home](#)

[Up](#)

[Next](#)

SDL\_DisplayFormatAlpha

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_DisplayFormatAlpha**

## **Name**

SDL\_DisplayFormatAlpha -- Convert a surface to the display format



## Synopsis

```
#include "SDL.h"
```

```
SDL_Surface *SDL_DisplayFormatAlpha(SDL_Surface *surface);
```

## Description

This function takes a surface and copies it to a new surface of the pixel format and colors of the video framebuffer plus an alpha channel, suitable for fast blitting onto the display surface. It calls [SDL\\_ConvertSurface](#)

If you want to take advantage of hardware colorkey or alpha blit acceleration, you should set the colorkey and alpha value before calling this function.

This function can be used to convert a colourkey to an alpha channel, if the `SDL_SRCCOLORKEY` flag is set on the surface. The generated surface will then be transparent (alpha=0) where the pixels match the colourkey, and opaque (alpha=255) elsewhere.

## **Return Value**

If the conversion fails or runs out of memory, it returns NULL

## See Also

[SDL\\_ConvertSurface](#), [SDL\\_SetAlpha](#), [SDL\\_SetColorKey](#),  
[SDL\\_DisplayFormat](#), [SDL\\_Surface](#)

---

[Prev](#)

SDL\_DisplayFormat

[Home](#)

[Up](#)

[Next](#)

SDL\_WarpMouse

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_WarpMouse**

## **Name**

SDL\_WarpMouse -- Set the position of the mouse cursor.

## Synopsis

```
#include "SDL.h"
```

```
void SDL_WarpMouse(Uint16 x, Uint16 y);
```



## **Description**

Set the position of the mouse cursor (generates a mouse motion event).

## See Also

[SDL\\_MouseMotionEvent](#)

---

[Prev](#)

SDL\_DisplayFormatAlpha

[Home](#)

[Up](#)

[Next](#)

SDL\_CreateCursor

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_CreateCursor**

## **Name**

SDL\_CreateCursor -- Creates a new mouse cursor.

## Synopsis

```
#include "SDL.h"
```

```
SDL_Cursor *SDL_CreateCursor(Uint8 *data, Uint8 *mask, int w, int h, int  
hot_x, int hot_y);
```

## Description

Create a cursor using the specified *data* and *mask* (in MSB format). The cursor width must be a multiple of 8 bits.

The cursor is created in black and white according to the following:

<b>Data / Mask</b>	<b>Resulting pixel on screen</b>
0 / 1	White
1 / 1	Black
0 / 0	Transparent
1 / 0	Inverted color if possible, black if not.

Cursors created with this function must be freed with [SDL\\_FreeCursor](#).





```

    "0,0"
};

static SDL_Cursor *init_system_cursor(const char *image[])
{
    int i, row, col;
    Uint8 data[4*32];
    Uint8 mask[4*32];
    int hot_x, hot_y;

    i = -1;
    for ( row=0; row<32; ++row ) {
        for ( col=0; col<32; ++col ) {
            if ( col % 8 ) {
                data[i] <<= 1;
                mask[i] <<= 1;
            } else {
                ++i;
                data[i] = mask[i] = 0;
            }
            switch (image[4+row][col]) {
                case 'X':
                    data[i] |= 0x01;
                    mask[i] |= 0x01;
                    break;
                case '.':
                    mask[i] |= 0x01;
                    break;
                case ' ':
                    break;
            }
        }
    }
    sscanf(image[4+row], "%d,%d", &hot_x, &hot_y);
    return SDL_CreateCursor(data, mask, 32, 32, hot_x, hot_y);
}

```

## See Also

[SDL\\_FreeCursor](#), [SDL\\_SetCursor](#), [SDL\\_ShowCursor](#)

---

[Prev](#)

SDL\_WarpMouse

[Home](#)

[Up](#)

[Next](#)

SDL\_FreeCursor

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_FreeCursor**

## **Name**

SDL\_FreeCursor -- Frees a cursor created with SDL\_CreateCursor.

## Synopsis

```
#include "SDL.h"
```

```
void SDL_FreeCursor(SDL_Cursor *cursor);
```

## **Description**

Frees a SDL\_Cursor that was created using [SDL\\_CreateCursor](#).

## See Also

[SDL\\_CreateCursor](#)

---

[Prev](#)

SDL\_CreateCursor

[Home](#)

[Up](#)

[Next](#)

SDL\_SetCursor



## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_SetCursor**

## **Name**

SDL\_SetCursor -- Set the currently active mouse cursor.

## Synopsis

```
#include "SDL.h"
```

```
void *SDL_SetCursor(SDL_Cursor *cursor);
```

## **Description**

Sets the currently active cursor to the specified one. If the cursor is currently visible, the change will be immediately represented on the display.

## See Also

[SDL\\_GetCursor](#), [SDL\\_CreateCursor](#), [SDL\\_ShowCursor](#)

---

[Prev](#)

SDL\_FreeCursor

[Home](#)

[Up](#)

[Next](#)

SDL\_GetCursor

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_GetCursor**



## **Name**

SDL\_GetCursor -- Get the currently active mouse cursor.

## Synopsis

```
#include "SDL.h"
```

```
SDL_Cursor *SDL_GetCursor(void);
```

## **Description**

Returns the currently active mouse cursor.

## See Also

[SDL\\_SetCursor](#), [SDL\\_CreateCursor](#), [SDL\\_ShowCursor](#)

---

[Prev](#)

SDL\_SetCursor

[Home](#)

[Up](#)

[Next](#)

SDL\_ShowCursor

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_ShowCursor**

## **Name**

SDL\_ShowCursor -- Toggle whether or not the cursor is shown on the screen.

## Synopsis

```
#include "SDL.h"
```

```
int SDL_ShowCursor(int toggle);
```



## **Description**

Toggle whether or not the cursor is shown on the screen. Passing `SDL_ENABLE` displays the cursor and passing `SDL_DISABLE` hides it. The current state of the mouse cursor can be queried by passing `SDL_QUERY`, either `SDL_DISABLE` or `SDL_ENABLE` will be returned.

The cursor starts off displayed, but can be turned off.

## **Return Value**

Returns the current state of the cursor.

## See Also

[SDL\\_CreateCursor](#), [SDL\\_SetCursor](#)

---

[Prev](#)

SDL\_GetCursor

[Home](#)

[Up](#)

[Next](#)

SDL\_GL\_LoadLibrary

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_GL\_LoadLibrary**

## **Name**

SDL\_GL\_LoadLibrary -- Specify an OpenGL library

## Synopsis

```
#include "SDL.h"
```

```
int SDL_GL_LoadLibrary(const char *path);
```

## Description

If you wish, you may load the OpenGL library at runtime, this must be done before [SDL\\_SetVideoMode](#) is called. The *path* of the GL library is passed to `SDL_GL_LoadLibrary` and it returns 0 on success, or -1 on an error. You must then use [SDL\\_GL\\_GetProcAddress](#) to retrieve function pointers to GL functions.



## See Also

[SDL\\_GL\\_GetProcAddress](#)

---

[Prev](#)

SDL\_ShowCursor

[Home](#)

[Up](#)

[Next](#)

SDL\_GL\_GetProcAddress

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_GL\_GetProcAddress**

## **Name**

SDL\_GL\_GetProcAddress -- Get the address of a GL function

## Synopsis

```
#include "SDL.h"
```

```
void *SDL_GL_GetProcAddress(const char* proc);
```

## Description

Returns the address of the GL function *proc*, or NULL if the function is not found. If the GL library is loaded at runtime, with [SDL\\_GL\\_LoadLibrary](#), then *all* GL functions must be retrieved this way. Usually this is used to retrieve function pointers to OpenGL extensions.

## Example

```
typedef void (*GL_ActiveTextureARB_Func)(unsigned int);
GL_ActiveTextureARB_Func glActiveTextureARB_ptr = 0;
int has_multitexture=1;
.
.
.
/* Get function pointer */
glActiveTextureARB_ptr=(GL_ActiveTextureARB_Func) SDL_GL_GetProcAddress

/* Check for a valid function ptr */
if(!glActiveTextureARB_ptr){
    fprintf(stderr, "Multitexture Extensions not present.\n");
    has_multitexture=0;
}
.
.
.
if(has_multitexture){
    glActiveTextureARB_ptr(GL_TEXTURE0_ARB);
    .
}
else{
    .
}
}
```

## See Also

[SDL\\_GL\\_LoadLibrary](#)

---

[Prev](#)

SDL\_GL\_LoadLibrary

[Home](#)

[Up](#)

[Next](#)

SDL\_GL\_GetAttribute



## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_GL\_GetAttribute**

## **Name**

SDL\_GL\_GetAttribute -- Get the value of a special SDL/OpenGL attribute

## Synopsis

```
#include "SDL.h"
```

```
int SDL_GL_GetAttribute(SDLGLAttr attr, int *value);
```

## Description

Places the value of the SDL/OpenGL [attribute](#) *attr* into *value*. This is useful after a call to [SDL\\_SetVideoMode](#) to check whether your attributes have been [set](#) as you expected.

## **Return Value**

Returns 0 on success, or -1 on an error.

## See Also

[SDL\\_GL\\_SetAttribute](#), [GL Attributes](#)

---

[Prev](#)

SDL\_GL\_GetProcAddress

[Home](#)

[Up](#)

[Next](#)

SDL\_GL\_SetAttribute

## SDL Library Documentation

[Prev](#)

[Next](#)

---



# **SDL\_GL\_SetAttribute**

## **Name**

SDL\_GL\_SetAttribute -- Set a special SDL/OpenGL attribute

## Synopsis

```
#include "SDL.h"
```

```
int SDL_GL_SetAttribute(SDL_GLattr attr, int value);
```

## Description

Sets the OpenGL [attribute](#) *attr* to *value*. The attributes you set don't take effect until after a call to [SDL\\_SetVideoMode](#). You should use [SDL\\_GL\\_GetAttribute](#) to check the values after a `SDL_SetVideoMode` call.

## **Return Value**

Returns 0 on success, or -1 on error.

## Example

```
SDL_GL_SetAttribute( SDL_GL_RED_SIZE, 5 );
SDL_GL_SetAttribute( SDL_GL_GREEN_SIZE, 5 );
SDL_GL_SetAttribute( SDL_GL_BLUE_SIZE, 5 );
SDL_GL_SetAttribute( SDL_GL_DEPTH_SIZE, 16 );
SDL_GL_SetAttribute( SDL_GL_DOUBLEBUFFER, 1 );
if ( (screen=SDL_SetVideoMode( 640, 480, 16, SDL_OPENGL )) == NULL )
    fprintf(stderr, "Couldn't set GL mode: %s\n", SDL_GetError());
    SDL_Quit();
    return;
}
```

**Note:** The `SDL_DOUBLEBUF` flag is not required to enable double buffering when setting an OpenGL video mode. Double buffering is enabled or disabled using the `SDL_GL_DOUBLEBUFFER` attribute.

## See Also

[SDL\\_GL\\_GetAttribute](#), [GL Attributes](#)

---

[Prev](#)

SDL\_GL\_GetAttribute

[Home](#)

[Up](#)

[Next](#)

SDL\_GL\_SwapBuffers

## SDL Library Documentation

[Prev](#)

[Next](#)

---



# **SDL\_GL\_SwapBuffers**

## **Name**

SDL\_GL\_SwapBuffers -- Swap OpenGL framebuffers/Update Display

## Synopsis

```
#include "SDL.h"
```

```
void SDL_GL_SwapBuffers(void );
```

## **Description**

Swap the OpenGL buffers, if double-buffering is supported.

## See Also

[SDL\\_SetVideoMode](#), [SDL\\_GL\\_SetAttribute](#)

---

[Prev](#)

SDL\_GL\_SetAttribute

[Home](#)

[Up](#)

[Next](#)

SDL\_CreateYUVOverlay

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_CreateYUVOverlay**

## **Name**

SDL\_CreateYUVOverlay -- Create a YUV video overlay



## Synopsis

```
#include "SDL.h"
```

```
SDL_Overlay *SDL_CreateYUVOverlay(int width, int height, Uint32 format,  
SDL_Surface *display);
```

## Description

`SDL_CreateYUVOverlay` creates a YUV overlay of the specified *width*, *height* and *format* (see [SDL Overlay](#) for a list of available formats), for the provided *display*. A [SDL Overlay](#) structure is returned.

The term 'overlay' is a misnomer since, unless the overlay is created in hardware, the contents for the display surface underneath the area where the overlay is shown will be overwritten when the overlay is displayed.

## See Also

[SDL\\_Overlay](#), [SDL\\_DisplayYUVOverlay](#), [SDL\\_FreeYUVOverlay](#)

---

[Prev](#)

SDL\_GL\_SwapBuffers

[Home](#)

[Up](#)

[Next](#)

SDL\_LockYUVOverlay

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_LockYUVOverlay**

## **Name**

SDL\_LockYUVOverlay -- Lock an overlay

## Synopsis

```
#include "SDL.h"
```

```
int SDL_LockYUVOverlay(SDL_Overlay *overlay);
```

## Description

Much the same as [SDL\\_LockSurface](#), `SDL_LockYUVOverlay` locks the [overlay](#) for direct access to pixel data.



## **Return Value**

Returns 0 on success, or -1 on an error.

## See Also

[SDL\\_UnlockYUVOverlay](#), [SDL\\_CreateYUVOverlay](#), [SDL\\_Overlay](#)

---

[Prev](#)

SDL\_CreateYUVOverlay

[Home](#)

[Up](#)

[Next](#)

SDL\_UnlockYUVOverlay

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_UnlockYUVOverlay**

## **Name**

SDL\_UnlockYUVOverlay -- Unlock an overlay

## Synopsis

```
#include "SDL.h"
```

```
void SDL_UnlockYUVOverlay(SDL_Overlay *overlay);
```

## Description

The opposite to [SDL\\_LockYUVOverlay](#). Unlocks a previously locked overlay. An overlay must be unlocked before it can be displayed.

## See Also

[SDL\\_UnlockYUVOverlay](#), [SDL\\_CreateYUVOverlay](#), [SDL\\_Overlay](#)

---

[Prev](#)

SDL\_LockYUVOverlay

[Home](#)

[Up](#)

[Next](#)

SDL\_DisplayYUVOverlay



## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_DisplayYUVOverlay**

## **Name**

SDL\_DisplayYUVOverlay -- Blit the overlay to the display

## Synopsis

```
#include "SDL.h"
```

```
int SDL_DisplayYUVOverlay(SDL_Overlay *overlay, SDL_Rect *dstrect);
```

## Description

Blit the *overlay* to the surface specified when it was [created](#). The [SDL\\_Rect](#) structure, *dstrect*, specifies the position and size of the destination. If the *dstrect* is a larger or smaller than the overlay then the overlay will be scaled, this is optimized for 2x scaling.

## **Return Values**

Returns 0 on success

## See Also

[SDL\\_Overlay](#), [SDL\\_CreateYUVOverlay](#)

---

[Prev](#)

SDL\_UnlockYUVOverlay

[Home](#)

[Up](#)

[Next](#)

SDL\_FreeYUVOverlay

## SDL Library Documentation

[Prev](#)

[Next](#)

---



# **SDL\_FreeYUVOverlay**

## **Name**

SDL\_FreeYUVOverlay -- Free a YUV video overlay

## Synopsis

```
#include "SDL.h"
```

```
void SDL_FreeYUVOverlay(SDL_Overlay *overlay);
```

## Description

Frees and [overlay](#) created by [SDL\\_CreateYUVOverlay](#).

## See Also

[SDL\\_Overlay](#), [SDL\\_DisplayYUVOverlay](#), [SDL\\_FreeYUVOverlay](#)

---

[Prev](#)

SDL\_DisplayYUVOverlay

[Home](#)

[Up](#)

[Next](#)

SDL\_GLattr

## SDL Library Documentation

[Prev](#)

[Next](#)

---

**SDL\_GLAttr**

## **Name**

SDL\_GLattr -- SDL GL Attributes



## Attributes

SDL_GL_RED_SIZE	Size of the framebuffer red component, in bits
SDL_GL_GREEN_SIZE	Size of the framebuffer green component, in bits
SDL_GL_BLUE_SIZE	Size of the framebuffer blue component, in bits
SDL_GL_ALPHA_SIZE	Size of the framebuffer alpha component, in bits
SDL_GL_DOUBLEBUFFER	0 or 1, enable or disable double buffering
SDL_GL_BUFFER_SIZE	Size of the framebuffer, in bits
SDL_GL_DEPTH_SIZE	Size of the depth buffer, in bits
SDL_GL_STENCIL_SIZE	Size of the stencil buffer, in bits
SDL_GL_ACCUM_RED_SIZE	Size of the accumulation buffer red component, in bits
SDL_GL_ACCUM_GREEN_SIZE	Size of the accumulation buffer green component, in bits
SDL_GL_ACCUM_BLUE_SIZE	Size of the accumulation buffer blue component, in bits
SDL_GL_ACCUM_ALPHA_SIZE	Size of the accumulation buffer alpha component, in bits

## Description

While you can set most OpenGL attributes normally, the attributes list above must be known *before* SDL sets the video mode. These attributes are set and read with [SDL\\_GL\\_SetAttribute](#) and [SDL\\_GL\\_GetAttribute](#).

## See Also

[SDL\\_GL\\_SetAttribute](#), [SDL\\_GL\\_GetAttribute](#)

---

[Prev](#)

SDL\_FreeYUVOverlay

[Home](#)

[Up](#)

[Next](#)

SDL\_Rect

## SDL Library Documentation

[Prev](#)

[Next](#)

---

**SDL\_Rect**

## **Name**

SDL\_Rect -- Defines a rectangular area

## Structure Definition

```
typedef struct{  
    Sint16 x, y;  
    Uint16 w, h;  
} SDL_Rect;
```

## Structure Data

$x$ ,  $y$  Position of the upper-left corner of the rectangle

$w$ ,  $h$  The width and height of the rectangle



## Description

A `SDL_Rect` defines a rectangular area of pixels. It is used by [SDL\\_BlitSurface](#) to define blitting regions and by several other video functions.

## See Also

[SDL\\_BlitSurface](#), [SDL\\_UpdateRect](#)

---

[Prev](#)

SDL\_GLattr

[Home](#)

[Up](#)

[Next](#)

SDL\_Color

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_Color**

## **Name**

SDL\_Color -- Format independent color description

## Structure Definition

```
typedef struct{
    Uint8 r;
    Uint8 g;
    Uint8 b;
    Uint8 unused;
} SDL_Color;
```

## Structure Data

*r* Red intensity  
*g* Green intensity  
*b* Blue intensity  
*unused* Unused

## Description

SDL\_Color describes a color in a format independent way. You can convert a SDL\_Color to a pixel value for a certain pixel format using [SDL\\_MapRGB](#).



## See Also

[SDL\\_PixelFormat](#), [SDL\\_SetColors](#), [SDL\\_Palette](#)

---

[Prev](#)

SDL\_Rect

[Home](#)

[Up](#)

[Next](#)

SDL\_Palette

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# SDL\_Palette

## **Name**

SDL\_Palette -- Color palette for 8-bit pixel formats

## Structure Definition

```
typedef struct{  
    int ncolors;  
    SDL_Color *colors;  
} SDL_Palette;
```

## Structure Data

*ncolors* Number of colors used in this palette

*colors* Pointer to [SDL\\_Color](#) structures that make up the palette.

## Description

Each pixel in an 8-bit surface is an index into the *colors* field of the `SDL_Palette` structure store in [SDL\\_PixelFormat](#). A `SDL_Palette` should never need to be created manually. It is automatically created when SDL allocates a `SDL_PixelFormat` for a surface. The colors values of a [SDL\\_Surfaces](#) palette can be set with the [SDL\\_SetColors](#).

## See Also

[SDL\\_Color](#), [SDL\\_Surface](#), [SDL\\_SetColors](#) [SDL\\_SetPalette](#)

---

[Prev](#)

SDL\_Color

[Home](#)

[Up](#)

[Next](#)

SDL\_PixelFormat



## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_PixelFormat**

## **Name**

SDL\_PixelFormat -- Stores surface format information

## Structure Definition

```
typedef struct{
    SDL_Palette *palette;
    Uint8  BitsPerPixel;
    Uint8  BytesPerPixel;
    Uint32 Rmask, Gmask, Bmask, Amask;
    Uint8  Rshift, Gshift, Bshift, Ashift;
    Uint8  Rloss, Gloss, Bloss, Aloss;
    Uint32 colorkey;
    Uint8  alpha;
} SDL_PixelFormat;
```

## Structure Data

<i>palette</i>	Pointer to the <a href="#">palette</a> , or NULL if the <i>BitsPerPixel</i> >8
<i>BitsPerPixel</i>	The number of bits used to represent each pixel in a surface. Usually 8, 16, 24 or 32.
<i>BytesPerPixel</i>	The number of bytes used to represent each pixel in a surface. Usually one to four.
<i>[RGBA]mask</i>	Binary mask used to retrieve individual color values
<i>[RGBA]loss</i>	Precision loss of each color component ( $2^{[RGBA]loss}$ )
<i>[RGBA]shift</i>	Binary left shift of each color component in the pixel value
<i>colorkey</i>	Pixel value of transparent pixels
<i>alpha</i>	Overall surface alpha value

## Description

A `SDL_PixelFormat` describes the format of the pixel data stored at the `pixels` field of a [SDL\\_Surface](#). Every surface stores a `SDL_PixelFormat` in the `format` field.

If you wish to do pixel level modifications on a surface, then understanding how SDL stores its color information is essential.

8-bit pixel formats are the easiest to understand. Since its an 8-bit format, we have 8 *BitsPerPixel* and 1 *BytesPerPixel*. Since *BytesPerPixel* is 1, all pixels are represented by a `Uint8` which contains an index into `palette->colors`. So, to determine the color of a pixel in a 8-bit surface: we read the color index from `surface->pixels` and we use that index to read the [SDL\\_Color](#) structure from `surface->format->palette->colors`. Like so:

```
SDL_Surface *surface;
SDL_PixelFormat *fmt;
SDL_Color *color;
Uint8 index;

.
.

/* Create surface */
.
.
fmt=surface->format;

/* Check the bitdepth of the surface */
if(fmt->BitsPerPixel!=8){
    fprintf(stderr, "Not an 8-bit surface.\n");
    return(-1);
}

/* Lock the surface */
SDL_LockSurface(surface);

/* Get the topleft pixel */
index=(Uint8 *)surface->pixels;
color=fmt->palette->colors[index];

/* Unlock the surface */
SDL_UnlockSurface(surface);
```

```
printf("Pixel Color-> Red: %d, Green: %d, Blue: %d. Index: %d\n",
       color->r, color->g, color->b, index);
```

```
.  
.
```

Pixel formats above 8-bit are an entirely different experience. They are considered to be "TrueColor" formats and the color information is stored in the pixels themselves, not in a palette. The mask, shift and loss fields tell us how the color information is encoded. The mask fields allow us to isolate each color component, the shift fields tell us the number of bits to the right of each component in the pixel value and the loss fields tell us the number of bits lost from each component when packing 8-bit color component in a pixel.

```
/* Extracting color components from a 32-bit color value */
SDL_PixelFormat *fmt;
SDL_Surface *surface;
Uint32 temp, pixel;
Uint8 red, green, blue, alpha;
.
.
.
fmt=surface->format;
SDL_LockSurface(surface);
pixel=((Uint32*)surface->pixels);
SDL_UnlockSurface(surface);

/* Get Red component */
temp=pixel&fmt->Rmask; /* Isolate red component */
temp=temp>>fmt->Rshift; /* Shift it down to 8-bit */
temp=temp<<fmt->Rloss; /* Expand to a full 8-bit number */
red=(Uint8)temp;

/* Get Green component */
temp=pixel&fmt->Gmask; /* Isolate green component */
temp=temp>>fmt->Gshift; /* Shift it down to 8-bit */
temp=temp<<fmt->Gloss; /* Expand to a full 8-bit number */
green=(Uint8)temp;

/* Get Blue component */
temp=pixel&fmt->Bmask; /* Isolate blue component */
temp=temp>>fmt->Bshift; /* Shift it down to 8-bit */
temp=temp<<fmt->Bloss; /* Expand to a full 8-bit number */
blue=(Uint8)temp;

/* Get Alpha component */
temp=pixel&fmt->Amask; /* Isolate alpha component */
temp=temp>>fmt->Ashift; /* Shift it down to 8-bit */
```

```
temp=temp<<fmt->Aloss; /* Expand to a full 8-bit number */
alpha=(Uint8)temp;

printf("Pixel Color -> R: %d, G: %d, B: %d, A: %d\n", red, green,
.
.
.
```



## See Also

[SDL\\_Surface](#), [SDL\\_MapRGB](#)

---

[Prev](#)

SDL\_Palette

[Home](#)

[Up](#)

[Next](#)

SDL\_Surface

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# SDL\_Surface

## **Name**

SDL\_Surface -- Graphical Surface Structure

## Structure Definition

```
typedef struct SDL_Surface {
    Uint32 flags; /* Read-only */
    SDL_PixelFormat *format; /* Read-only */
    int w, h; /* Read-only */
    Uint16 pitch; /* Read-only */
    void *pixels; /* Read-write */

    /* clipping information */
    SDL_Rect clip_rect; /* Read-only */

    /* Reference count -- used when freeing surface */
    int refcount; /* Read-mostly */

    /* This structure also contains private fields not shown here */
} SDL_Surface;
```

## Structure Data

*flags*      Surface flags  
*format*     Pixel [format](#)  
*w, h*        Width and height of the surface  
*pitch*      Length of a surface scanline in bytes  
*pixels*     Pointer to the actual pixel data  
*clip\_rect* surface clip [rectangle](#)

## Description

SDL\_Surface's represent areas of "graphical" memory, memory that can be drawn to. The video framebuffer is returned as a SDL\_Surface by [SDL\\_SetVideoMode](#) and [SDL\\_GetVideoSurface](#). Most of the fields should be pretty obvious. *w* and *h* are the width and height of the surface in pixels. *pixels* is a pointer to the actual pixel data, the surface should be [locked](#) before accessing this field. The *clip\_rect* field is the clipping rectangle as set by [SDL\\_SetClipRect](#).

The following are supported in the *flags* field.

SDL_SWSURFACE	Surface is stored in system memory
SDL_HWSURFACE	Surface is stored in video memory
SDL_ASYNCBLIT	Surface uses asynchronous blits if possible
SDL_ANYFORMAT	Allows any pixel-format (Display surface)
SDL_HWPALETTE	Surface has exclusive palette
SDL_DOUBLEBUF	Surface is double buffered (Display surface)
SDL_FULLSCREEN	Surface is full screen (Display Surface)
SDL_OPENGL	Surface has an OpenGL context (Display Surface)
SDL_OPENGLBLIT	Surface supports OpenGL blitting (Display Surface)
SDL_RESIZABLE	Surface is resizable (Display Surface)
SDL_HWACCEL	Surface blit uses hardware acceleration
SDL_SRCCOLORKEY	Surface use colorkey blitting
SDL_RLEACCEL	Colorkey blitting is accelerated with RLE
SDL_SRCALPHA	Surface blit uses alpha blending
SDL_PREALLOC	Surface uses preallocated memory

## See Also

[SDL\\_PixelFormat](#)

---

[Prev](#)

SDL\_PixelFormat

[Home](#)

[Up](#)

[Next](#)

SDL\_VideoInfo



## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_VideoInfo**

## **Name**

SDL\_VideoInfo -- Video Target information

## Structure Definition

```
typedef struct{
    Uint32 hw_available:1;
    Uint32 wm_available:1;
    Uint32 blit_hw:1;
    Uint32 blit_hw_CC:1;
    Uint32 blit_hw_A:1;
    Uint32 blit_sw:1;
    Uint32 blit_sw_CC:1;
    Uint32 blit_sw_A:1;
    Uint32 blit_fill;
    Uint32 video_mem;
    SDL_PixelFormat *vfmt;
} SDL_VideoInfo;
```

## Structure Data

*hw\_available* Is it possible to create hardware surfaces?  
*wm\_available* Is there a window manager available  
*blit\_hw* Are hardware to hardware blits accelerated?  
*blit\_hw\_CC* Are hardware to hardware colorkey blits accelerated?  
*blit\_hw\_A* Are hardware to hardware alpha blits accelerated?  
*blit\_sw* Are software to hardware blits accelerated?  
*blit\_sw\_CC* Are software to hardware colorkey blits accelerated?  
*blit\_sw\_A* Are software to hardware alpha blits accelerated?  
*blit\_fill* Are color fills accelerated?  
*video\_mem* Total amount of video memory in Kilobytes  
*vfmt* [Pixel format](#) of the video device

## Description

This (read-only) structure is returned by [SDL\\_GetVideoInfo](#). It contains information on either the 'best' available mode (if called before [SDL\\_SetVideoMode](#)) or the current video mode.

## See Also

[SDL\\_PixelFormat](#), [SDL\\_GetVideoInfo](#)

---

[Prev](#)

SDL\_Surface

[Home](#)

[Up](#)

[Next](#)

SDL\_Overlay

## SDL Library Documentation

[Prev](#)

[Next](#)

---



# **SDL\_Overlay**

## **Name**

SDL\_Overlay -- YUV video overlay

## Structure Definition

```
typedef struct{
    Uint32 format;
    int w, h;
    int planes;
    Uint16 *pitches;
    Uint8 **pixels;
    Uint32 hw_overlay:1;
} SDL_Overlay;
```

## Structure Data

<i>format</i>	Overlay format (see below)
<i>w, h</i>	Width and height of overlay
<i>planes</i>	Number of planes in the overlay. Usually either 1 or 3
<i>itches</i>	An array of pitches, one for each plane. Pitch is the length of a row in bytes.
<i>ixels</i>	An array of pointers to the data of each plane. The overlay should be locked before these pointers are used.
<i>hw_overlay</i>	This will be set to 1 if the overlay is hardware accelerated.

## Description

A `SDL_Overlay` is similar to a [SDL\\_Surface](#) except it stores a YUV overlay. All the fields are read only, except for *pixels* which should be [locked](#) before use.

The *format* field stores the format of the overlay which is one of the following:

```
#define SDL_YV12_OVERLAY 0x32315659 /* Planar mode: Y + V + U */
#define SDL_IYUV_OVERLAY 0x56555949 /* Planar mode: Y + U + V */
#define SDL_YUY2_OVERLAY 0x32595559 /* Packed mode: Y0+U0+Y1+V0 */
#define SDL_UYVY_OVERLAY 0x59565955 /* Packed mode: U0+Y0+V0+Y1 */
#define SDL_YVYU_OVERLAY 0x55595659 /* Packed mode: Y0+V0+Y1+U0 */
```

More information on YUV formats can be found at <http://www.webartz.com/fourcc/indexyuv.htm>.

## See Also

[SDL\\_CreateYUVOverlay](#), [SDL\\_LockYUVOverlay](#), [SDL\\_UnlockYUVOverlay](#),  
[SDL\\_FreeYUVOverlay](#)

---

[Prev](#)

SDL\_VideoInfo

[Home](#)

[Up](#)

[Next](#)

Window Management

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# Chapter 7. Window Management

## Table of Contents

[SDL\\_WM\\_SetCaption](#) — Sets the window title and icon name.

[SDL\\_WM\\_GetCaption](#) — Gets the window title and icon name.

[SDL\\_WM\\_SetIcon](#) — Sets the icon for the display window.

[SDL\\_WM\\_IconifyWindow](#) — Iconify/Minimise the window

[SDL\\_WM\\_ToggleFullScreen](#) — Toggles fullscreen mode

[SDL\\_WM\\_GrabInput](#) — Grabs mouse and keyboard input.

SDL provides a small set of window management functions which allow applications to change their title and toggle from windowed mode to fullscreen (if available)

---

[Prev](#)

SDL\_Overlay

[Home](#)

[Up](#)

[Next](#)

SDL\_WM\_SetCaption



## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_WM\_GetCaption**

## **Name**

SDL\_WM\_GetCaption -- Gets the window title and icon name.

## Synopsis

```
#include "SDL.h"
```

```
void SDL_WM_GetCaption(char **title, char **icon);
```

## **Description**

Set pointers to the window *title* and *icon* name.

## See Also

[SDL\\_WM\\_SetCaption](#), [SDL\\_WM\\_SetIcon](#)

---

[Prev](#)

SDL\_WM\_SetCaption

[Home](#)

[Up](#)

[Next](#)

SDL\_WM\_SetIcon

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_WM\_SetCaption**



## **Name**

SDL\_WM\_SetCaption -- Sets the window title and icon name.

## Synopsis

```
#include "SDL.h"
```

```
void SDL_WM_SetCaption(const char *title, const char *icon);
```

## **Description**

Sets the title-bar and icon name of the display window.

## See Also

[SDL\\_WM\\_GetCaption](#), [SDL\\_WM\\_SetIcon](#)

---

[Prev](#)

Window Management

[Home](#)

[Up](#)

[Next](#)

SDL\_WM\_GetCaption

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_WM\_SetIcon**

## **Name**

SDL\_WM\_SetIcon -- Sets the icon for the display window.

## Synopsis

```
#include "SDL.h"
```

```
void SDL_WM_SetIcon(SDL_Surface *icon, Uint8 *mask);
```



## Description

Sets the icon for the display window. Win32 icons must be 32x32.

This function must be called before the first call to [SDL\\_SetVideoMode](#).

It takes an *icon* surface, and a *mask* in MSB format.

If *mask* is NULL, the entire icon surface will be used as the icon.

## Example

```
SDL_WM_SetIcon(SDL_LoadBMP("icon.bmp"), NULL);
```

## See Also

[SDL\\_SetVideoMode](#), [SDL\\_WM\\_SetCaption](#)

---

[Prev](#)

SDL\_WM\_GetCaption

[Home](#)

[Up](#)

[Next](#)

SDL\_WM\_IconifyWindow

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_WM\_IconifyWindow**

## **Name**

SDL\_WM\_IconifyWindow -- Iconify/Minimise the window

## Synopsis

```
#include "SDL.h"
```

```
int SDL_WM_IconifyWindow(void);
```

## Description

If the application is running in a window managed environment SDL attempts to iconify/minimise it. If `SDL_WM_IconifyWindow` is successful, the application will receive a [SDL\\_APPACTIVE](#) loss event.



## Return Value

Returns non-zero on success or 0 if iconification is not support or was refused by the window manager.

---

[Prev](#)

SDL\_WM\_SetIcon

[Home](#)

[Up](#)

[Next](#)

SDL\_WM\_ToggleFullScreen

## SDL Library Documentation

[Prev](#)

[Next](#)

---

**SDL\_WM\_ToggleFullScreen**

## **Name**

SDL\_WM\_ToggleFullScreen -- Toggles fullscreen mode

## Synopsis

```
#include "SDL.h"
```

```
int SDL_WM_ToggleFullScreen(SDL_Surface *surface);
```

## **Description**

Toggles the application between windowed and fullscreen mode, if supported. (X11 is the only target currently supported, BeOS support is experimental).

## Return Value

Returns 0 on failure or 1 on success.

---

[Prev](#)

SDL\_WM\_IconifyWindow

[Home](#)

[Up](#)

[Next](#)

SDL\_WM\_GrabInput

## SDL Library Documentation

[Prev](#)

[Next](#)

---



# **SDL\_WM\_GrabInput**

## **Name**

SDL\_WM\_GrabInput -- Grabs mouse and keyboard input.

## Synopsis

```
#include "SDL.h"
```

```
SDL_GrabMode SDL_WM_GrabInput(SDL_GrabMode mode);
```

## Description

Grabbing means that the mouse is confined to the application window, and nearly all keyboard input is passed directly to the application, and not interpreted by a window manager, if any.

When *mode* is `SDL_GRAB_QUERY` the grab mode is not changed, but the current grab mode is returned.

```
typedef enum {  
    SDL_GRAB_QUERY,  
    SDL_GRAB_OFF,  
    SDL_GRAB_ON  
} SDL_GrabMode;
```

## Return Value

The current/new SDL\_GrabMode.

---

[Prev](#)

SDL\_WM\_ToggleFullScreen

[Home](#)

[Up](#)

[Next](#)

Events

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# Chapter 8. Events

## Table of Contents

[Introduction](#)

[SDL Event Structures.](#)

[Event Functions.](#)

# Introduction

Event handling allows your application to receive input from the user. Event handling is initialised (along with video) with a call to:

```
SDL_Init(SDL_INIT_VIDEO);
```

Internally, SDL stores all the events waiting to be handled in an event queue. Using functions like [SDL\\_PollEvent](#) and [SDL\\_PeepEvents](#) you can observe and handle waiting input events.

The key to event handling in SDL is the [SDL\\_Event](#) union. The event queue itself is composed of a series of `SDL_Event` unions, one for each waiting event. `SDL_Event` unions are read from the queue with the `SDL_PollEvent` function and it is then up to the application to process the information stored with them.

---

[Prev](#)

SDL\_WM\_GrabInput

[Home](#)

[Up](#)

[Next](#)

SDL Event Structures.



# SDL Library Documentation

## Chapter 8. Events

[Prev](#)

[Next](#)

---

# SDL Event Structures.

## Table of Contents

[SDL\\_Event](#) — General event structure

[SDL\\_ActiveEvent](#) — Application visibility event structure

[SDL\\_KeyboardEvent](#) — Keyboard event structure

[SDL\\_MouseMotionEvent](#) — Mouse motion event structure

[SDL\\_MouseButtonEvent](#) — Mouse button event structure

[SDL\\_JoyAxisEvent](#) — Joystick axis motion event structure

[SDL\\_JoyButtonEvent](#) — Joystick button event structure

[SDL\\_JoyHatEvent](#) — Joystick hat position change event structure

[SDL\\_JoyBallEvent](#) — Joystick trackball motion event structure

[SDL\\_ResizeEvent](#) — Window resize event structure

[SDL\\_ExposeEvent](#) — Quit requested event

[SDL\\_SysWMEvent](#) — Platform-dependent window manager event.

[SDL\\_UserEvent](#) — A user-defined event type

[SDL\\_QuitEvent](#) — Quit requested event

[SDL\\_keysym](#) — Keysym structure

[SDLKey](#) — Keysym definitions.

---

[Prev](#)

Events

[Home](#)

[Up](#)

[Next](#)

SDL\_Event

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_Event**

## **Name**

SDL\_Event -- General event structure

## Structure Definition

```
typedef union{
    Uint8 type;
    SDL_ActiveEvent active;
    SDL_KeyboardEvent key;
    SDL_MouseMotionEvent motion;
    SDL_MouseButtonEvent button;
    SDL_JoyAxisEvent jaxis;
    SDL_JoyBallEvent jball;
    SDL_JoyHatEvent jhat;
    SDL_JoyButtonEvent jbutton;
    SDL_ResizeEvent resize;
    SDL_ExposeEvent expose;
    SDL_QuitEvent quit;
    SDL_UserEvent user;
    SDL_SywwMEvent syswm;
} SDL_Event;
```

## Structure Data

<i>type</i>	The type of event
<i>active</i>	<a href="#">Activation event</a>
<i>key</i>	<a href="#">Keyboard event</a>
<i>motion</i>	<a href="#">Mouse motion event</a>
<i>button</i>	<a href="#">Mouse button event</a>
<i>jaxis</i>	<a href="#">Joystick axis motion event</a>
<i>jball</i>	<a href="#">Joystick trackball motion event</a>
<i>jhat</i>	<a href="#">Joystick hat motion event</a>
<i>jbutton</i>	<a href="#">Joystick button event</a>
<i>resize</i>	<a href="#">Application window resize event</a>
<i>expose</i>	<a href="#">Application window expose event</a>
<i>quit</i>	<a href="#">Application quit request event</a>
<i>user</i>	<a href="#">User defined event</a>
<i>syswm</i>	<a href="#">Undefined window manager event</a>

## Description

The `SDL_Event` union is the core to all event handling in SDL, it's probably the most important structure after `SDL_Surface`. `SDL_Event` is a union of all event structures used in SDL, using it is a simple matter of knowing which union member relates to which event *type*.

<b>Event type</b>	<b>Event Structure</b>
SDL_ACTIVEEVENT	<a href="#">SDL_ActiveEvent</a>
SDL_KEYDOWN/UP	<a href="#">SDL_KeyboardEvent</a>
SDL_MOUSEMOTION	<a href="#">SDL_MouseMotionEvent</a>
SDL_MOUSEBUTTONDOWN/UP	<a href="#">SDL_MouseButtonEvent</a>
SDL_JOYAXISMOTION	<a href="#">SDL_JoyAxisEvent</a>
SDL_JOYBALLMOTION	<a href="#">SDL_JoyBallEvent</a>
SDL_JOYHATMOTION	<a href="#">SDL_JoyHatEvent</a>
SDL_JOYBUTTONDOWN/UP	<a href="#">SDL_JoyButtonEvent</a>
SDL_QUIT	<a href="#">SDL_QuitEvent</a>
SDL_SYSWMEVENT	<a href="#">SDL_SysWMEvent</a>
SDL_VIDEORESIZE	<a href="#">SDL_ResizeEvent</a>
SDL_VIDEOEXPOSE	<a href="#">SDL_ExposeEvent</a>
SDL_USEREVENT	<a href="#">SDL_UserEvent</a>



## Use

The `SDL_Event` structure has two uses

- Reading events on the event queue
- Placing events on the event queue

Reading events from the event queue is done with either [SDL\\_PollEvent](#) or [SDL\\_PeepEvents](#). We'll use `SDL_PollEvent` and step through an example.

First off, we create an empty `SDL_Event` structure.

```
SDL_Event test_event;
```

`SDL_PollEvent` removes the next event from the event queue, if there are no events on the queue it returns 0 otherwise it returns 1. We use a `while` loop to process each event in turn.

```
while(SDL_PollEvent(&test_event)) {
```

The `SDL_PollEvent` function take a pointer to an `SDL_Event` structure that is to be filled with event information. We know that if `SDL_PollEvent` removes an event from the queue then the event information will be placed in our `test_event` structure, but we also know that the *type* of event will be placed in the *type* member of `test_event`. So to handle each event *type* seperately we use a `switch` statement.

```
    switch(test_event.type) {
```

We need to know what kind of events we're looking for *and* the event *type*'s of those events. So lets assume we want to detect where the user is moving the mouse pointer within our application. We look through our event types and notice that `SDL_MOUSEMOTION` is, more than likely, the event we're looking for. A little [more](#) research tells use that `SDL_MOUSEMOTION` events are handled within the [SDL\\_MouseMotionEvent](#) structure which is the *motion* member of `SDL_Event`. We can check for the `SDL_MOUSEMOTION` event *type* within our `switch` statement like so:

```
        case SDL_MOUSEMOTION:
```

All we need do now is read the information out of the *motion* member of *test\_event*.

```
        printf("We got a motion event.\n");
        printf("Current mouse position is: (%d, %d)\n", test_event.mot
        break;
    default:
        printf("Unhandled Event!\n");
        break;
}
}
printf("Event queue empty.\n");
```

It is also possible to push events onto the event queue and so use it as a two-way communication path. Both [SDL\\_PushEvent](#) and [SDL\\_PeepEvents](#) allow you to place events onto the event queue. This is usually used to place a `SDL_USEREVENT` on the event queue, however you could use it to post fake input events if you wished. Creating your own events is a simple matter of choosing the event type you want, setting the *type* member and filling the appropriate member structure with information.

```
SDL_Event user_event;

user_event.type=SDL_USEREVENT;
user_event.user.code=2;
user_event.user.data1=NULL;
user_event.user.data2=NULL;
SDL_PushEvent(&user_event);
```

## See Also

[SDL\\_PollEvent](#), [SDL\\_PushEvent](#), [SDL\\_PeepEvents](#)

---

[Prev](#)

SDL Event Structures.

[Home](#)

[Up](#)

[Next](#)

SDL\_ActiveEvent

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_ActiveEvent**

## **Name**

SDL\_ActiveEvent -- Application visibility event structure

## Structure Definition

```
typedef struct{  
    Uint8 type;  
    Uint8 gain;  
    Uint8 state;  
} SDL_ActiveEvent;
```

## Structure Data

*type* SDL\_ACTIVEEVENT.

*gain* 0 if the event is a loss or 1 if it is a gain.

SDL\_APPMOUSEFOCUS if mouse focus was gained or lost,

*state* SDL\_APPINPUTFOCUS if input focus was gained or lost, or SDL\_APPACTIVE if the application was iconified (*gain*=0) or restored(*gain*=1).



## Description

SDL\_ActiveEvent is a member of the [SDL\\_Event](#) union and is used when an event of type SDL\_ACTIVEEVENT is reported.

When the mouse leaves or enters the window area a SDL\_APPMOUSEFOCUS type activation event occurs, if the mouse entered the window then *gain* will be 1, otherwise *gain* will be 0. A SDL\_APPINPUTFOCUS type activation event occurs when the application loses or gains keyboard focus. This usually occurs when another application is made active. Finally, a SDL\_APPACTIVE type event occurs when the application is either minimised/iconified (*gain*=0) or restored.

**Note:** This event does not occur when an application window is first created.

## See Also

[SDL\\_Event](#), [SDL\\_GetAppState](#)

---

[Prev](#)

SDL\_Event

[Home](#)

[Up](#)

[Next](#)

SDL\_KeyboardEvent

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_KeyboardEvent**

## **Name**

SDL\_KeyboardEvent -- Keyboard event structure

## Structure Definition

```
typedef struct{
    Uint8 type;
    Uint8 state;
    SDL_keysym keysym;
} SDL_KeyboardEvent;
```

## Structure Data

*type* SDL\_KEYDOWN or SDL\_KEYUP

*state* SDL\_PRESSED or SDL\_RELEASED

*keysym* Contains key press information

## Description

SDL\_KeyboardEvent is a member of the [SDL\\_Event](#) union and is used when an event of type SDL\_KEYDOWN or SDL\_KEYUP is reported.

The *type* and *state* actually report the same information, they just use different values to do it! A keyboard event occurs when a key is released (*type*=SDL\_KEYUP or *state*=SDL\_RELEASED) and when a key is pressed (*type*=SDL\_KEYDOWN or *state*=SDL\_PRESSED). The information on what key was pressed or released is in the [keysym](#) structure.

**Note:** Repeating SDL\_KEYDOWN events will occur if key repeat is enabled (see [SDL\\_EnableKeyRepeat](#)).



## See Also

[SDL\\_Event](#), [SDL\\_keysym](#), [SDL\\_EnableKeyRepeat](#), [SDL\\_EnableUNICODE](#)

---

[Prev](#)

SDL\_ActiveEvent

[Home](#)

[Up](#)

[Next](#)

SDL\_MouseMotionEvent

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_MouseMotionEvent**

## **Name**

SDL\_MouseMotionEvent -- Mouse motion event structure

## Structure Definition

```
typedef struct{
    Uint8 type;
    Uint8 state;
    Uint16 x, y;
    Sint16 xrel, yrel;
} SDL_MouseMotionEvent;
```

## Structure Data

*type*       SDL\_MOUSEMOTION  
*state*       The current button state  
*x, y*        The X/Y coordinates of the mouse  
*xrel, yrel* Relative motion in the X/Y direction

## Description

`SDL_MouseMotionEvent` is a member of the [SDL\\_Event](#) union and is used when an event of type `SDL_MOUSEMOTION` is reported.

Simply put, a `SDL_MOUSEMOTION` type event occurs when a user moves the mouse within the application window or when [SDL\\_WarpMouse](#) is called. Both the absolute ( $x$  and  $y$ ) and relative ( $xrel$  and  $yrel$ ) coordinates are reported along with the current button states (*state*). The button state can be interpreted using the `SDL_BUTTON` macro (see [SDL\\_GetMouseState](#)).

If the cursor is hidden ([SDL\\_ShowCursor\(0\)](#)) and the input is grabbed ([SDL\\_WM\\_GrabInput\(SDL\\_GRAB\\_ON\)](#)), then the mouse will give relative motion events even when the cursor reaches the edge of the screen. This is currently only implemented on Windows and Linux/Unix-a-likes.

## See Also

[SDL\\_Event](#), [SDL\\_MouseButtonEvent](#)

---

[Prev](#)

SDL\_KeyboardEvent

[Home](#)

[Up](#)

[Next](#)

SDL\_MouseButtonEvent



## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_MouseButtonEvent**

## **Name**

SDL\_MouseButtonEvent -- Mouse button event structure

## Structure Definition

```
typedef struct{
    Uint8 type;
    Uint8 button;
    Uint8 state;
    Uint16 x, y;
} SDL_MouseButtonEvent;
```

## Structure Data

*type* SDL\_MOUSEBUTTONDOWN or SDL\_MOUSEBUTTONUP  
*button* The mouse button index (SDL\_BUTTON\_LEFT,  
SDL\_BUTTON\_MIDDLE, SDL\_BUTTON\_RIGHT)  
*state* SDL\_PRESSED or SDL\_RELEASED  
*x, y* The X/Y coordinates of the mouse at press/release time

## Description

`SDL_MouseButtonEvent` is a member of the [SDL\\_Event](#) union and is used when an event of type `SDL_MOUSEBUTTONDOWN` or `SDL_MOUSEBUTTONUP` is reported.

When a mouse button press or release is detected then number of the button pressed (from 1 to 255, with 1 usually being the left button and 2 the right) is placed into *button*, the position of the mouse when this event occurred is stored in the *x* and the *y* fields. Like [SDL\\_KeyboardEvent](#), information on whether the event was a press or a release event is stored in both the *type* and *state* fields, but this should be obvious.

## See Also

[SDL\\_Event](#), [SDL\\_MouseMotionEvent](#)

---

[Prev](#)

SDL\_MouseMotionEvent

[Home](#)

[Up](#)

[Next](#)

SDL\_JoyAxisEvent

## SDL Library Documentation

[Prev](#)

[Next](#)

---



# **SDL\_JoyAxisEvent**

## **Name**

SDL\_JoyAxisEvent -- Joystick axis motion event structure

## Structure Definition

```
typedef struct{
    Uint8 type;
    Uint8 which;
    Uint8 axis;
    Sint16 value;
} SDL_JoyAxisEvent;
```

## Structure Data

*type* SDL\_JOYAXISMOTION

*which* Joystick device index

*axis* Joystick axis index

*value* Axis value (range: -32768 to 32767)

## Description

SDL\_JoyAxisEvent is a member of the [SDL\\_Event](#) union and is used when an event of type SDL\_JOYAXISMOTION is reported.

A SDL\_JOYAXISMOTION event occurs when ever a user moves an axis on the joystick. The field *which* is the index of the joystick that reported the event and *axis* is the index of the axis (for a more detailed explanation see the [Joystick section](#)). *value* is the current position of the axis.

## See Also

[SDL\\_Event](#), [Joystick Functions](#), [SDL\\_JoystickEventState](#),  
[SDL\\_JoystickGetAxis](#)

---

[Prev](#)

SDL\_MouseButtonEvent

[Home](#)

[Up](#)

[Next](#)

SDL\_JoyButtonEvent

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_JoyButtonEvent**



## **Name**

SDL\_JoyButtonEvent -- Joystick button event structure

## Structure Definition

```
typedef struct{
    Uint8 type;
    Uint8 which;
    Uint8 button;
    Uint8 state;
} SDL_JoyButtonEvent;
```

## Structure Data

*type* SDL\_JOYBUTTONDOWN or SDL\_JOYBUTTONUP

*which* Joystick device index

*button* Joystick button index

*state* SDL\_PRESSED or SDL\_RELEASED

## Description

SDL\_JoyButtonEvent is a member of the [SDL\\_Event](#) union and is used when an event of type SDL\_JOYBUTTONDOWN or SDL\_JOYBUTTONUP is reported.

A SDL\_JOYBUTTONDOWN or SDL\_JOYBUTTONUP event occurs when ever a user presses or releases a button on a joystick. The field *which* is the index of the joystick that reported the event and *button* is the index of the button (for a more detailed explanation see the [Joystick section](#)). *state* is the current state or the button which is either SDL\_PRESSED or SDL\_RELEASED.

## See Also

[SDL\\_Event](#), [Joystick Functions](#), [SDL\\_JoystickEventState](#),  
[SDL\\_JoystickGetButton](#)

---

[Prev](#)

SDL\_JoyAxisEvent

[Home](#)

[Up](#)

[Next](#)

SDL\_JoyHatEvent

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_JoyHatEvent**

## **Name**

SDL\_JoyHatEvent -- Joystick hat position change event structure



## Structure Definition

```
typedef struct{
    Uint8 type;
    Uint8 which;
    Uint8 hat;
    Uint8 value;
} SDL_JoyHatEvent;
```

## Structure Data

*type* SDL\_JOY

*which* Joystick device index

*hat* Joystick hat index

*value* Hat position

## Description

SDL\_JoyHatEvent is a member of the [SDL\\_Event](#) union and is used when an event of type SDL\_JOYHATMOTION is reported.

A SDL\_JOYHATMOTION event occurs when ever a user moves a hat on the joystick. The field *which* is the index of the joystick that reported the event and *hat* is the index of the hat (for a more detailed explanation see the [Joystick section](#)). *value* is the current position of the hat. It is a logically OR'd combination of the following values (whose meanings should be pretty obvious:) :

```
SDL_HAT_CENTERED
SDL_HAT_UP
SDL_HAT_RIGHT
SDL_HAT_DOWN
SDL_HAT_LEFT
```

The following defines are also provided:

```
SDL_HAT_RIGHTUP
SDL_HAT_RIGHTDOWN
SDL_HAT_LEFTUP
SDL_HAT_LEFTDOWN
```

## See Also

[SDL\\_Event](#), [Joystick Functions](#), [SDL\\_JoystickEventState](#),  
[SDL\\_JoystickGetHat](#)

---

[Prev](#)

SDL\_JoyButtonEvent

[Home](#)

[Up](#)

[Next](#)

SDL\_JoyBallEvent

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_JoyBallEvent**

## **Name**

SDL\_JoyBallEvent -- Joystick trackball motion event structure

## Structure Definition

```
typedef struct{
    Uint8 type;
    Uint8 which;
    Uint8 ball;
    Sint16 xrel, yrel;
} SDL_JoyBallEvent;
```



## Structure Data

*type*       SDL\_JOYBALLMOTION  
*which*      Joystick device index  
*ball*        Joystick trackball index  
*xrel, yrel* The relative motion in the X/Y direction

## Description

SDL\_JoyBallEvent is a member of the [SDL\\_Event](#) union and is used when an event of type SDL\_JOYBALLMOTION is reported.

A SDL\_JOYBALLMOTION event occurs when a user moves a trackball on the joystick. The field *which* is the index of the joystick that reported the event and *ball* is the index of the trackball (for a more detailed explanation see the [Joystick section](#)). Trackballs only return relative motion, this is the change in position on the ball since it was last polled (last cycle of the event loop) and it is stored in *xrel* and *yrel*.

## See Also

[SDL\\_Event](#), [Joystick Functions](#), [SDL\\_JoystickEventState](#),  
[SDL\\_JoystickGetBall](#)

---

[Prev](#)

SDL\_JoyHatEvent

[Home](#)

[Up](#)

[Next](#)

SDL\_ResizeEvent

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# SDL\_ResizeEvent

## **Name**

SDL\_ResizeEvent -- Window resize event structure

## Structure Definition

```
typedef struct{
    Uint8 type;
    int w, h;
} SDL_ResizeEvent;
```

## Structure Data

*type* SDL\_VIDEORESIZE

*w, h* New width and height of the window



## Description

SDL\_ResizeEvent is a member of the [SDL\\_Event](#) union and is used when an event of type SDL\_VIDEORESIZE is reported.

When SDL\_RESIZABLE is passed as a *flag* to [SDL\\_SetVideoMode](#) the user is allowed to resize the applications window. When the window is resized an SDL\_VIDEORESIZE is report, with the new window width and height values stored in  $w$  and  $h$ , respectively. When an SDL\_VIDEORESIZE is recieved the window should be resized to the new dimensions using [SDL\\_SetVideoMode](#).

## See Also

[SDL\\_Event](#), [SDL\\_SetVideoMode](#)

---

[Prev](#)

SDL\_JoyBallEvent

[Home](#)

[Up](#)

[Next](#)

SDL\_ExposeEvent

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_ExposeEvent**

## **Name**

SDL\_ExposeEvent -- Quit requested event

## Structure Definition

```
typedef struct{  
    Uint8 type  
} SDL_ExposeEvent;
```

# Structure Data

*type* SDL\_QUIT

## Description

SDL\_ExposeEvent is a member of the [SDL\\_Event](#) union and is used when an event of type SDL\_VIDEOEXPOSE is reported.

A VIDEOEXPOSE event is triggered when the screen has been modified outside of the application, usually by the window manager and needs to be redrawn.



## See Also

[SDL\\_Event](#), [SDL\\_SetEventFilter](#)

---

[Prev](#)

SDL\_ResizeEvent

[Home](#)

[Up](#)

[Next](#)

SDL\_SysWMEvent

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_SysWMEvent**

## **Name**

SDL\_SysWMEvent -- Platform-dependent window manager event.

## Description

The system window manager event contains a pointer to system-specific information about unknown window manager events. If you enable this event using [SDL\\_EventState\(\)](#), it will be generated whenever unhandled events are received from the window manager. This can be used, for example, to implement cut-and-paste in your application.

```
typedef struct {
    Uint8 type;    /* Always SDL_SysWM */
} SDL_SysWMEvent;
```

If you want to obtain system-specific information about the window manager, you can fill the version member of a `SDL_SysWMInfo` structure (details can be found in `SDL_syswm.h`, which must be included) using the `SDL_VERSION()` macro found in `SDL_version.h`, and pass it to the function:

```
int SDL_GetWMInfo(SDL_SysWMInfo *info);
```

## See Also

[SDL\\_EventState](#)

---

[Prev](#)

SDL\_ExposeEvent

[Home](#)

[Up](#)

[Next](#)

SDL\_UserEvent

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_UserEvent**



## **Name**

SDL\_UserEvent -- A user-defined event type

## Structure Definition

```
typedef struct{
    Uint8 type;
    int code;
    void *data1;
    void *data2;
} SDL_UserEvent;
```

## Structure Data

*type* SDL\_USEREVENT through to SDL\_NUMEVENTS-1

*code* User defined event code

*data1* User defined data pointer

*data2* User defined data pointer

## Description

SDL\_UserEvent is in the *user* member of the structure [SDL\\_Event](#). This event is unique, it is never created by SDL but only by the user. The event can be pushed onto the event queue using [SDL\\_PushEvent](#). The contents of the structure members or completely up to the programmer, the only requirement is that *type* is a value from SDL\_USEREVENT to SDL\_NUMEVENTS-1 (inclusive).

## Examples

```
SDL_Event event;  
  
event.type = SDL_USEREVENT;  
event.user.code = my_event_code;  
event.user.data1 = significant_data;  
event.user.data2 = 0;  
SDL_PushEvent(&event);
```

## See Also

[SDL\\_Event](#), [SDL\\_PushEvent](#)

---

[Prev](#)

SDL\_SysWMEvent

[Home](#)

[Up](#)

[Next](#)

SDL\_QuitEvent

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# SDL\_QuitEvent



## **Name**

SDL\_QuitEvent -- Quit requested event

## Structure Definition

```
typedef struct{  
    Uint8 type  
} SDL_QuitEvent;
```

# Structure Data

*type* SDL\_QUIT

## Description

SDL\_QuitEvent is a member of the [SDL\\_Event](#) union and is used when an event of type SDL\_QUIT is reported.

As can be seen, the SDL\_QuitEvent structure serves no useful purpose. The event itself, on the other hand, is very important. If you filter out or ignore a quit event then it is impossible for the user to close the window. On the other hand, if you do accept a quit event then the application window will be closed, and screen updates will still report success event though the application will no longer be visible.

**Note:** The macro `SDL_QuitRequested` will return non-zero if a quit event is pending

## See Also

[SDL\\_Event](#), [SDL\\_SetEventFilter](#)

---

[Prev](#)

SDL\_UserEvent

[Home](#)

[Up](#)

[Next](#)

SDL\_keysym

## SDL Library Documentation

[Prev](#)

[Next](#)

---

**SDL\_keysym**

## **Name**

SDL\_keysym -- Keysym structure



## Structure Definition

```
typedef struct{
    Uint8 scancode;
    SDLKey sym;
    SDLMod mod;
    Uint16 unicode;
} SDL_keysym;
```

## Structure Data

*scancode* Hardware specific scancode

*sym* SDL virtual keysym

*mod* Current key modifiers

*unicode* Translated character

## Description

The `SDL_keysym` structure is used by reporting key presses and releases since it is a part of the [SDL\\_KeyboardEvent](#).

The `scancode` field should generally be left alone, it is the hardware dependent scancode returned by the keyboard. The `sym` field is extremely useful. It is the SDL-defined value of the key (see [SDL Key Syms](#)). This field is very useful when you are checking for certain key presses, like so:

```
.
.
while(SDL_PollEvent(&event)){
    switch(event.type){
        case SDL_KEYDOWN:
            if(event.key.keysym.sym==SDLK_LEFT)
                move_left();
            break;
        .
        .
        .
    }
}
.
```

`mod` stores the current state of the keyboard modifiers as explained in [SDL\\_GetModState](#). The `unicode` is only used when UNICODE translation is enabled with [SDL\\_EnableUNICODE](#). If `unicode` is non-zero then this is the UNICODE character corresponding to the keypress. If the high 9 bits of the character are 0, then this maps to the equivalent ASCII character:

```
char ch;
if ( (keysym.unicode & 0xFF80) == 0 ) {
    ch = keysym.unicode & 0x7F;
}
else {
    printf("An International Character.\n");
}
```

UNICODE translation does have a slight overhead so don't enable it unless it's needed.

## See Also

[SDLKey](#)

---

[Prev](#)

SDL\_QuitEvent

[Home](#)

[Up](#)

[Next](#)

SDLKey

## SDL Library Documentation

[Prev](#)

[Next](#)

---

**SDLKey**

## **Name**

SDLKey -- Keysym definitions.

# Description

**Table 8-1. SDL Keysym definitions**

<b>SDLKey</b>	<b>ASCII value</b>	<b>Common name</b>
SDLK_BACKSPACE	'\b'	backspace
SDLK_TAB	'\t'	tab
SDLK_CLEAR		clear
SDLK_RETURN	'\r'	return
SDLK_PAUSE		pause
SDLK_ESCAPE	'\^['	escape
SDLK_SPACE	' '	space
SDLK_EXCLAIM	'!'	exclaim
SDLK_QUOTEDBL	'\"'	quotedbl
SDLK_HASH	'#'	hash
SDLK_DOLLAR	'\$'	dollar
SDLK_AMPERSAND	'&'	ampersand
SDLK_QUOTE	'\"'	quote
SDLK_LEFTPAREN	'('	left parenthesis
SDLK_RIGHTPAREN	')'	right parenthesis
SDLK_ASTERISK	'*'	asterisk
SDLK_PLUS	'+'	plus sign
SDLK_COMMA	','	comma
SDLK_MINUS	'-'	minus sign
SDLK_PERIOD	'.'	period
SDLK_SLASH	'/'	forward slash
SDLK_0	'0'	0
SDLK_1	'1'	1
SDLK_2	'2'	2
SDLK_3	'3'	3



SDLK_4	'4'	4
SDLK_5	'5'	5
SDLK_6	'6'	6
SDLK_7	'7'	7
SDLK_8	'8'	8
SDLK_9	'9'	9
SDLK_COLON	':'	colon
SDLK_SEMICOLON	','	semicolon
SDLK_LESS	'<'	less-than sign
SDLK_EQUALS	'='	equals sign
SDLK_GREATER	'>'	greater-than sign
SDLK_QUESTION	'?'	question mark
SDLK_AT	'@'	at
SDLK_LEFTBRACKET	'['	left bracket
SDLK_BACKSLASH	'\'	backslash
SDLK_RIGHTBRACKET	']'	right bracket
SDLK_CARET	'^'	caret
SDLK_UNDERSCORE	'_'	underscore
SDLK_BACKQUOTE	'`'	grave
SDLK_a	'a'	a
SDLK_b	'b'	b
SDLK_c	'c'	c
SDLK_d	'd'	d
SDLK_e	'e'	e
SDLK_f	'f'	f
SDLK_g	'g'	g
SDLK_h	'h'	h
SDLK_i	'i'	i
SDLK_j	'j'	j
SDLK_k	'k'	k
SDLK_l	'l'	l

SDLK_m	'm'	m
SDLK_n	'n'	n
SDLK_o	'o'	o
SDLK_p	'p'	p
SDLK_q	'q'	q
SDLK_r	'r'	r
SDLK_s	's'	s
SDLK_t	't'	t
SDLK_u	'u'	u
SDLK_v	'v'	v
SDLK_w	'w'	w
SDLK_x	'x'	x
SDLK_y	'y'	y
SDLK_z	'z'	z
SDLK_DELETE	'^?'	delete
SDLK_KP0		keypad 0
SDLK_KP1		keypad 1
SDLK_KP2		keypad 2
SDLK_KP3		keypad 3
SDLK_KP4		keypad 4
SDLK_KP5		keypad 5
SDLK_KP6		keypad 6
SDLK_KP7		keypad 7
SDLK_KP8		keypad 8
SDLK_KP9		keypad 9
SDLK_KP_PERIOD	'.'	keypad period
SDLK_KP_DIVIDE	'/'	keypad divide
SDLK_KP_MULTIPLY	'*'	keypad multiply
SDLK_KP_MINUS	'-'	keypad minus
SDLK_KP_PLUS	'+'	keypad plus

SDLK_KP_ENTER	'\r'	keypad enter
SDLK_KP_EQUALS	'='	keypad equals
SDLK_UP		up arrow
SDLK_DOWN		down arrow
SDLK_RIGHT		right arrow
SDLK_LEFT		left arrow
SDLK_INSERT		insert
SDLK_HOME		home
SDLK_END		end
SDLK_PAGEUP		page up
SDLK_PAGEDOWN		page down
SDLK_F1		F1
SDLK_F2		F2
SDLK_F3		F3
SDLK_F4		F4
SDLK_F5		F5
SDLK_F6		F6
SDLK_F7		F7
SDLK_F8		F8
SDLK_F9		F9
SDLK_F10		F10
SDLK_F11		F11
SDLK_F12		F12
SDLK_F13		F13
SDLK_F14		F14
SDLK_F15		F15
SDLK_NUMLOCK		numlock
SDLK_CAPSLOCK		capslock
SDLK_SCROLLLOCK		scrolllock
SDLK_RSHIFT		right shift

SDLK_LSHIFT		left shift
SDLK_RCTRL		right ctrl
SDLK_LCTRL		left ctrl
SDLK_RALT		right alt
SDLK_LALT		left alt
SDLK_RMETA		right meta
SDLK_LMETA		left meta
SDLK_LSUPER		left windows key
SDLK_RSUPER		right windows key
SDLK_MODE		mode shift
SDLK_HELP		help
SDLK_PRINT		print-screen
SDLK_SYSREQ		SysRq
SDLK_BREAK		break
SDLK_MENU		menu
SDLK_POWER		power
SDLK_EURO		euro

**Table 8-2. SDL modifier definitions**

<b>SDL Modifier</b>	<b>Meaning</b>
KMOD_NONE	No modifiers applicable
KMOD_NUM	Numlock is down
KMOD_CAPS	Capslock is down
KMOD_LCTRL	Left Control is down
KMOD_RCTRL	Right Control is down
KMOD_RSHIFT	Right Shift is down
KMOD_LSHIFT	Left Shift is down
KMOD_RALT	Right Alt is down
KMOD_LALT	Left Alt is down
KMOD_CTRL	A Control key is down

KMOD_SHIFT	A Shift key is down
KMOD_ALT	An Alt key is down

---

[Prev](#)

SDL\_keysym

[Home](#)

[Up](#)

[Next](#)

Event Functions.

# SDL Library Documentation

## Chapter 8. Events

[Prev](#)

[Next](#)

---

# Event Functions.

## Table of Contents

[SDL\\_PumpEvents](#) — Pumps the event loop, gathering events from the input devices.

[SDL\\_PeepEvents](#) — Checks the event queue for messages and optionally returns them.

[SDL\\_PollEvent](#) — Polls for currently pending events.

[SDL\\_WaitEvent](#) — Waits indefinitely for the next available event.

[SDL\\_PushEvent](#) — Pushes an event onto the event queue

[SDL\\_SetEventFilter](#) — Sets up a filter to process all events before they are posted to the event queue.

[SDL\\_GetEventFilter](#) — Retrieves a pointer to the event filter

[SDL\\_EventState](#) — This function allows you to set the state of processing certain events.

[SDL\\_GetKeyState](#) — Get a snapshot of the current keyboard state

[SDL\\_GetModState](#) — Get the state of modifier keys.

[SDL\\_SetModState](#) — Set the current key modifier state

[SDL\\_GetKeyName](#) — Get the name of an SDL virtual keysym

[SDL\\_EnableUNICODE](#) — Enable UNICODE translation

[SDL\\_EnableKeyRepeat](#) — Set keyboard repeat rate.

[SDL\\_GetMouseState](#) — Retrieve the current state of the mouse

[SDL\\_GetRelativeMouseState](#) — Retrieve the current state of the mouse

[SDL\\_GetAppState](#) — Get the state of the application

[SDL\\_JoystickEventState](#) — Enable/disable joystick event polling

[SDL\\_PumpEvents](#)

Pumps the event loop, gathering events from the input devices

[SDL\\_PeepEvents](#)

Checks the event queue for messages and optionally returns them

[SDL\\_PollEvent](#)

Polls for currently pending events

[SDL\\_WaitEvent](#)

Waits indefinitely for the next available event

[SDL\\_PushEvent](#)

Pushes an event onto the event queue

[SDL\\_SetEventFilter](#)

Sets up a filter to process all events

Allows you to set the state of processing certain

<a href="#">SDL_EventState</a>	events
<a href="#">SDL_GetKeyState</a>	Get a snapshot of the current keyboard state
<a href="#">SDL_GetModState</a>	Get the state of modifier keys
<a href="#">SDL_SetModState</a>	Set the state of modifier keys
<a href="#">SDL_GetKeyName</a>	Get the name of an SDL virtual keysym
<a href="#">SDL_EnableUNICODE</a>	Enable UNICODE translation
<a href="#">SDL_EnableKeyRepeat</a>	Set keyboard repeat rate
<a href="#">SDL_GetMouseState</a>	Retrieve the current state of the mouse
<a href="#">SDL_GetRelativeMouseState</a>	Retrieve the current state of the mouse
<a href="#">SDL_GetAppState</a>	Get the state of the application
<a href="#">SDL_JoystickEventState</a>	Enable/disable joystick event polling

---

[Prev](#)

SDLKey

[Home](#)

[Up](#)

[Next](#)

SDL\_PumpEvents



## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_PumpEvents**

## **Name**

SDL\_PumpEvents -- Pumps the event loop, gathering events from the input devices.

## Synopsis

```
#include "SDL.h"
```

```
void SDL_PumpEvents(void);
```

## Description

Pumps the event loop, gathering events from the input devices.

`SDL_PumpEvents` gathers all the pending input information from devices and places it on the event queue. Without calls to `SDL_PumpEvents` no events would ever be placed on the queue. Often calls the need for `SDL_PumpEvents` is hidden from the user since [SDL\\_PollEvent](#) and [SDL\\_WaitEvent](#) implicitly call `SDL_PumpEvents`. However, if you are not polling or waiting for events (e.g. your filtering them), then you must call `SDL_PumpEvents` to force an event queue update.

**Note:** You can only call this function in the thread that set the video mode.

## See Also

[SDL\\_PollEvent](#)

---

[Prev](#)

Event Functions.

[Home](#)

[Up](#)

[Next](#)

SDL\_PeepEvents

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_WaitEvent**



## **Name**

SDL\_WaitEvent -- Waits indefinitely for the next available event.

## Synopsis

```
#include "SDL.h"
```

```
int SDL_waitEvent(SDL_Event *event);
```

## Description

Waits indefinitely for the next available event, returning 1, or 0 if there was an error while waiting for events.

If *event* is not NULL, the next event is removed from the queue and stored in that area.

## See Also

[SDL\\_Event](#), [SDL\\_PollEvent](#)

---

[Prev](#)

SDL\_PollEvent

[Home](#)

[Up](#)

[Next](#)

SDL\_PushEvent

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_PeepEvents**

## **Name**

SDL\_PeepEvents -- Checks the event queue for messages and optionally returns them.

## Synopsis

```
#include "SDL.h"
```

```
int SDL_PeepEvents(SDL_Event *events, int numevents, SDL_eventaction  
action, Uint32 mask);
```



## Description

Checks the event queue for messages and optionally returns them.

If *action* is `SDL_ADDEVENT`, up to *numevents* events will be added to the back of the event queue.

If *action* is `SDL_PEEKEVENT`, up to *numevents* events at the front of the event queue, matching [\*mask\*](#), will be returned and will not be removed from the queue.

If *action* is `SDL_GETEVENT`, up to *numevents* events at the front of the event queue, matching [\*mask\*](#), will be returned and will be removed from the queue.

This function is thread-safe.

## **Return Value**

This function returns the number of events actually stored, or -1 if there was an error.

## See Also

[SDL\\_Event](#), [SDL\\_PollEvent](#), [SDL\\_PushEvent](#)

---

[Prev](#)

SDL\_PumpEvents

[Home](#)

[Up](#)

[Next](#)

SDL\_PollEvent

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_PollEvent**

## **Name**

SDL\_PollEvent -- Polls for currently pending events.

## Synopsis

```
#include "SDL.h"
```

```
int SDL_PollEvent(SDL_Event *event);
```

## Description

Polls for currently pending events, and returns 1 if there are any pending events, or 0 if there are none available.

If *event* is not NULL, the next event is removed from the queue and stored in that area.



## Examples

```
SDL_Event event; /* Event structure */

.
.
.
/* Check for events */
while(SDL_PollEvent(&event)){ /* Loop until there are no events left */
    switch(event.type){ /* Process the appropriate event type */
        case SDL_KEYDOWN: /* Handle a KEYDOWN event */
            printf("Oh! Key press\n");
            break;
        case SDL_MOUSEMOTION:
            .
            .
            .
        default: /* Report an unhandled event */
            printf("I don't know what this event is!\n");
    }
}
```

## See Also

[SDL\\_Event](#), [SDL\\_WaitEvent](#), [SDL\\_PeepEvents](#)

---

[Prev](#)

SDL\_PeepEvents

[Home](#)

[Up](#)

[Next](#)

SDL\_WaitEvent

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_PushEvent**

## **Name**

SDL\_PushEvent -- Pushes an event onto the event queue

## Synopsis

```
#include "SDL.h"
```

```
int SDL_PushEvent(SDL_Event *event);
```

## Description

The event queue can actually be used as a two way communication channel. Not only can events be read from the queue, but the user can also push their own events onto it. *event* is a pointer to the event structure you wish to push onto the queue.

**Note:** Pushing device input events onto the queue doesn't modify the state of the device within SDL.

## **Return Value**

Returns 0 on success or -1 if the event couldn't be pushed.



# Examples

See [SDL\\_Event](#).

## See Also

[SDL\\_PollEvent](#), [SDL\\_PeepEvents](#), [SDL\\_Event](#)

---

[Prev](#)

SDL\_WaitEvent

[Home](#)

[Up](#)

[Next](#)

SDL\_SetEventFilter

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_SetEventFilter**

## **Name**

SDL\_SetEventFilter -- Sets up a filter to process all events before they are posted to the event queue.

## Synopsis

```
#include "SDL.h"
```

```
void SDL_SetEventFilter(SDL_EventFilter filter);
```

## Description

This function sets up a filter to process all events before they are posted to the event queue. This is a very powerful and flexible feature. The filter is prototyped as:

```
typedef int (*SDL_EventFilter)(const SDL_Event *event);
```

If the filter returns 1, then the event will be added to the internal queue. If it returns 0, then the event will be dropped from the queue. This allows selective filtering of dynamically.

There is one caveat when dealing with the `SDL_QUIT` event type. The event filter is only called when the window manager desires to close the application window. If the event filter returns 1, then the window will be closed, otherwise the window will remain open if possible. If the quit event is generated by an interrupt signal, it will bypass the internal queue and be delivered to the application at the next event poll.

**Note:** Events pushed onto the queue with [SDL\\_PushEvent](#) or [SDL\\_PeepEvents](#) do not get passed through the event filter.

**Note:** *Be Careful!* The event filter function may run in a different thread so be careful what you do within it.

## See Also

[SDL\\_Event](#), [SDL\\_GetEventFilter](#), [SDL\\_PushEvent](#)

---

[Prev](#)

SDL\_PushEvent

[Home](#)

[Up](#)

[Next](#)

SDL\_GetEventFilter



## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_GetEventFilter**

## **Name**

SDL\_GetEventFilter -- Retrieves a pointer to the event filter

## Synopsis

```
#include "SDL.h"
```

```
SDL_EventFilter SDL_GetEventFilter(void);
```

## Description

This function retrieves a pointer to the event filter that was previously set using [SDL\\_SetEventFilter](#). An `SDL_EventFilter` function is defined as:

```
typedef int (*SDL_EventFilter)(const SDL_Event *event);
```

## **Return Value**

Returns a pointer to the event filter or NULL if no filter has been set.

## See Also

[SDL\\_Event](#), [SDL\\_SetEventFilter](#)

---

[Prev](#)

SDL\_SetEventFilter

[Home](#)

[Up](#)

[Next](#)

SDL\_EventState

## SDL Library Documentation

[Prev](#)

[Next](#)

---



# **SDL\_EventState**

## **Name**

SDL\_EventState -- This function allows you to set the state of processing certain events.

## Synopsis

```
#include "SDL.h"
```

```
Uint8 SDL_EventState(Uint8 type, int state);
```

## Description

This function allows you to set the state of processing certain event *type*'s.

If *state* is set to `SDL_IGNORE`, that event *type* will be automatically dropped from the event queue and will not be filtered.

If *state* is set to `SDL_ENABLE`, that event *type* will be processed normally.

If *state* is set to `SDL_QUERY`, `SDL_EventState` will return the current processing state of the specified event *type*.

A list of event *type*'s can be found in the [SDL\\_Event](#) section.

## See Also

[SDL\\_Event](#)

---

[Prev](#)

SDL\_GetEventFilter

[Home](#)

[Up](#)

[Next](#)

SDL\_GetKeyState

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# SDL\_GetKeyState

## **Name**

SDL\_GetKeyState -- Get a snapshot of the current keyboard state



## Synopsis

```
#include "SDL.h"
```

```
Uint8 *SDL_GetKeyState(int *numkeys);
```

## Description

Gets a snapshot of the current keyboard state. The current state is return as a pointer to an array, the size of this array is stored in *numkeys*. The array is indexed by the [SDLK\\_\\*](#) symbols. A value of 1 means the key is pressed and a value of 0 means its not. The pointer returned is a pointer to an internal SDL array and should not be freed by the caller.

**Note:** Use [SDL\\_PumpEvents](#) to update the state array.

## Example

```
Uint8 *keystate = SDL_GetKeyState(NULL);  
if ( keystate[SDLK_RETURN] ) printf("Return Key Pressed.\n");
```

## See Also

[SDL\\_Key\\_Symbols](#), [SDL\\_PumpEvents](#)

---

[Prev](#)

SDL\_EventState

[Home](#)

[Up](#)

[Next](#)

SDL\_GetModState

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_GetModState**

## **Name**

SDL\_GetModState -- Get the state of modifier keys.

## Synopsis

```
#include "SDL.h"
```

```
SDLMod SDL_GetModState(void);
```



## **Description**

Returns the current state of the modifier keys (CTRL, ALT, etc.).

## Return Value

The return value can be an OR'd combination of the SDLMod enum.

### SDLMod

```
typedef enum {
    KMOD_NONE = 0x0000,
    KMOD_LSHIFT= 0x0001,
    KMOD_RSHIFT= 0x0002,
    KMOD_LCTRL = 0x0040,
    KMOD_RCTRL = 0x0080,
    KMOD_LALT = 0x0100,
    KMOD_RALT = 0x0200,
    KMOD_LMETA = 0x0400,
    KMOD_RMETA = 0x0800,
    KMOD_NUM = 0x1000,
    KMOD_CAPS = 0x2000,
    KMOD_MODE = 0x4000,
} SDLMod;
```

SDL also defines the following symbols for convenience:

```
#define KMOD_CTRL (KMOD_LCTRL|KMOD_RCTRL)
#define KMOD_SHIFT (KMOD_LSHIFT|KMOD_RSHIFT)
#define KMOD_ALT (KMOD_LALT|KMOD_RALT)
#define KMOD_META (KMOD_LMETA|KMOD_RMETA)
```

## See Also

[SDL\\_GetKeyState](#)

---

[Prev](#)

SDL\_GetKeyState

[Home](#)

[Up](#)

[Next](#)

SDL\_SetModState

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_SetModState**

## **Name**

SDL\_SetModState -- Set the current key modifier state

## Synopsis

```
#include "SDL.h"
```

```
void SDL_SetModState(SDLMod modstate);
```

## Description

The inverse of [SDL\\_GetModState](#), `SDL_SetModState` allows you to impose modifier key states on your application.

Simply pass your desired modifier states into *modstate*. This value may be a logical OR'd combination of the following:

```
typedef enum {
    KMOD_NONE = 0x0000,
    KMOD_LSHIFT= 0x0001,
    KMOD_RSHIFT= 0x0002,
    KMOD_LCTRL = 0x0040,
    KMOD_RCTRL = 0x0080,
    KMOD_LALT = 0x0100,
    KMOD_RALT = 0x0200,
    KMOD_LMETA = 0x0400,
    KMOD_RMETA = 0x0800,
    KMOD_NUM = 0x1000,
    KMOD_CAPS = 0x2000,
    KMOD_MODE = 0x4000,
} SDLMod;
```



## See Also

[SDL\\_GetModState](#)

---

[Prev](#)

SDL\_GetModState

[Home](#)

[Up](#)

[Next](#)

SDL\_GetKeyName

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_GetKeyName**

## **Name**

SDL\_GetKeyName -- Get the name of an SDL virtual keysym

## Synopsis

```
#include "SDL.h"
```

```
char *SDL_GetKeyName(SDLKey key);
```

## Description

Returns the SDL-defined name of the [SDLKey](#) key.

## See Also

[SDLKey](#)

---

[Prev](#)

SDL\_SetModState

[Home](#)

[Up](#)

[Next](#)

SDL\_EnableUNICODE

## SDL Library Documentation

[Prev](#)

[Next](#)

---



**SDL\_EnableUNICODE**

## **Name**

SDL\_EnableUNICODE -- Enable UNICODE translation

## Synopsis

```
#include "SDL.h"
```

```
int SDL_EnableUNICODE(int enable);
```

## Description

Enables/Disables Unicode keyboard translation.

To obtain the character codes corresponding to received keyboard events, Unicode translation must first be turned on using this function. The translation incurs a slight overhead for each keyboard event and is therefore disabled by default. For each subsequently received key down event, the *unicode* member of the [SDL\\_keysym](#) structure will then contain the corresponding character code, or zero for keysyms that do not correspond to any character code.

A value of 1 for *enable* enables Unicode translation; 0 disables it, and -1 leaves it unchanged (useful for querying the current translation mode).

Note that only key press events will be translated, not release events.

## **Return Value**

Returns the previous translation mode (0 or 1).

## See Also

[SDL\\_keysym](#)

---

[Prev](#)

SDL\_GetKeyName

[Home](#)

[Up](#)

[Next](#)

SDL\_EnableKeyRepeat

## SDL Library Documentation

[Prev](#)

[Next](#)

---

**SDL\_EnableKeyRepeat**



## **Name**

SDL\_EnableKeyRepeat -- Set keyboard repeat rate.

## Synopsis

```
#include "SDL.h"
```

```
int SDL_EnableKeyRepeat(int delay, int interval);
```

## Description

Enables or disables the keyboard repeat rate. *delay* specifies how long the key must be pressed before it begins repeating, it then repeats at the speed specified by *interval*. Both *delay* and *interval* are expressed in milliseconds.

Setting *delay* to 0 disables key repeating completely. Good default values are `SDL_DEFAULT_REPEAT_DELAY` and `SDL_DEFAULT_REPEAT_INTERVAL`.

## Return Value

Returns 0 on success and -1 on failure.

---

[Prev](#)

SDL\_EnableUNICODE

[Home](#)

[Up](#)

[Next](#)

SDL\_GetMouseState

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_GetMouseState**

## **Name**

SDL\_GetMouseState -- Retrieve the current state of the mouse

## Synopsis

```
#include "SDL.h"
```

```
Uint8 SDL_GetMouseState(int *x, int *y);
```



## Description

The current button state is returned as a button bitmask, which can be tested using the `SDL_BUTTON(X)` macros, and `x` and `y` are set to the current mouse cursor position. You can pass `NULL` for either `x` or `y`.

## Example

```
SDL_PumpEvents();  
if(SDL_GetMouseState(NULL, NULL)&SDL_BUTTON(1))  
    printf("Mouse Button 1(left) is pressed.\n");
```

## See Also

[SDL\\_GetRelativeMouseState](#), [SDL\\_PumpEvents](#)

---

[Prev](#)

SDL\_EnableKeyRepeat

[Home](#)

[Up](#)

[Next](#)

SDL\_GetRelativeMouseState

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_GetRelativeMouseState**

## **Name**

SDL\_GetRelativeMouseState -- Retrieve the current state of the mouse

## Synopsis

```
#include "SDL.h"
```

```
Uint8 SDL_GetRelativeMouseState(int *x, int *y);
```

## Description

The current button state is returned as a button bitmask, which can be tested using the `SDL_BUTTON(X)` macros, and `x` and `y` are set to the change in the mouse position since the last call to `SDL_GetRelativeMouseState` or since event initialization. You can pass `NULL` for either `x` or `y`.



## See Also

[SDL\\_GetMouseState](#)

---

[Prev](#)

SDL\_GetMouseState

[Home](#)

[Up](#)

[Next](#)

SDL\_GetAppState

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_GetAppState**

## **Name**

SDL\_GetAppState -- Get the state of the application

## Synopsis

```
#include "SDL.h"
```

```
Uint8 SDL_GetAppState(void);
```

## Description

This function returns the current state of the application. The value returned is a bitwise combination of:

SDL\_APPMOUSEFOCUS The application has mouse focus.  
SDL\_APPINPUTFOCUS The application has keyboard focus  
SDL\_APPACTIVE The application is visible

## See Also

[SDL\\_ActiveEvent](#)

---

[Prev](#)

SDL\_GetRelativeMouseState

[Home](#)

[Up](#)

[Next](#)

SDL\_JoystickEventState

## SDL Library Documentation

[Prev](#)

[Next](#)

---



# **SDL\_JoystickEventState**

## **Name**

SDL\_JoystickEventState -- Enable/disable joystick event polling

## Synopsis

```
#include "SDL.h"
```

```
int SDL_JoystickEventState(int state);
```

## Description

This function is used to enable or disable joystick event processing. With joystick event processing disabled you will have to update joystick states with [SDL\\_JoystickUpdate](#) and read the joystick information manually. *state* is either `SDL_QUERY`, `SDL_ENABLE` or `SDL_IGNORE`.

**Note:** Joystick event handling is preferred

## Return Value

If *state* is `SDL_QUERY` then the current state is returned, otherwise the new processing *state* is returned.

## See Also

[SDL Joystick Functions](#), [SDL\\_JoystickUpdate](#), [SDL\\_JoyAxisEvent](#),  
[SDL\\_JoyBallEvent](#), [SDL\\_JoyButtonEvent](#), [SDL\\_JoyHatEvent](#)

---

[Prev](#)

SDL\_GetAppState

[Home](#)

[Up](#)

[Next](#)

Joystick

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# Chapter 9. Joystick

## Table of Contents

- [SDL\\_NumJoysticks](#) — Count available joysticks.
- [SDL\\_JoystickName](#) — Get joystick name.
- [SDL\\_JoystickOpen](#) — Opens a joystick for use.
- [SDL\\_JoystickOpened](#) — Determine if a joystick has been opened
- [SDL\\_JoystickIndex](#) — Get the index of an SDL\_Joystick.
- [SDL\\_JoystickNumAxes](#) — Get the number of joystick axes
- [SDL\\_JoystickNumBalls](#) — Get the number of joystick trackballs
- [SDL\\_JoystickNumHats](#) — Get the number of joystick hats
- [SDL\\_JoystickNumButtons](#) — Get the number of joystick buttons
- [SDL\\_JoystickUpdate](#) — Updates the state of all joysticks
- [SDL\\_JoystickGetAxis](#) — Get the current state of an axis
- [SDL\\_JoystickGetHat](#) — Get the current state of a joystick hat
- [SDL\\_JoystickGetButton](#) — Get the current state of a given button on a given joystick
- [SDL\\_JoystickGetBall](#) — Get relative trackball motion
- [SDL\\_JoystickClose](#) — Closes a previously opened joystick

Joysticks, and other similar input devices, have a very strong role in game playing and SDL provides comprehensive support for them. Axes, Buttons, POV Hats and trackballs are all supported.

Joystick support is initialized by passing the `SDL_INIT_JOYSTICK` flag to [SDL\\_Init](#). Once initialized joysticks must be opened using [SDL\\_JoystickOpen](#).

While using the functions describe in this section may seem like the best way to access and read from joysticks, in most cases they aren't. Ideally joysticks should be read using the [event](#) system. To enable this, you must set the joystick event processing state with [SDL\\_JoystickEventState](#). Joysticks must be [opened](#) before they can be used of course.

**Note:** If you are *not* handling the joystick via the event queue then you must explicitly request a joystick update by calling [SDL\\_JoystickUpdate](#).

**Note:** Force Feedback is not yet support. Sam (slouken@libsdl.org) is



soliciting suggestions from people with force-feedback experience on the best way to design the API.

---

[Prev](#)

SDL\_JoystickEventState

[Home](#)

[Up](#)

[Next](#)

SDL\_NumJoysticks

## SDL Library Documentation

[Prev](#)

[Next](#)

---

**SDL\_NumJoysticks**

## **Name**

SDL\_NumJoysticks -- Count available joysticks.

## Synopsis

```
#include "SDL.h"
```

```
int SDL_NumJoysticks(void);
```

## **Description**

Counts the number of joysticks attached to the system.

## **Return Value**

Returns the number of attached joysticks

## See Also

[SDL\\_JoystickName](#), [SDL\\_JoystickOpen](#)

---

[Prev](#)

Joystick

[Home](#)

[Up](#)

[Next](#)

SDL\_JoystickName



## SDL Library Documentation

[Prev](#)

[Next](#)

---

**SDL\_JoystickName**

## **Name**

SDL\_JoystickName -- Get joystick name.

## Synopsis

```
#include "SDL.h"
```

```
const char *SDL_JoystickName(int index);
```

## Description

Get the implementation dependent name of joystick. The *index* parameter refers to the N'th joystick on the system.

## **Return Value**

Returns a char pointer to the joystick name.

## Examples

```
/* Print the names of all attached joysticks */
int num_joy, i;
num_joy=SDL_NumJoysticks();
printf("%d joysticks found\n", num_joy);
for(i=0;i<num_joy;i++)
    printf("%s\n", SDL_JoystickName(i));
```

## See Also

[SDL\\_JoystickOpen](#)

---

[Prev](#)

SDL\_NumJoysticks

[Home](#)

[Up](#)

[Next](#)

SDL\_JoystickOpen



## SDL Library Documentation

[Prev](#)

[Next](#)

---

# SDL\_JoystickOpen

## **Name**

SDL\_JoystickOpen -- Opens a joystick for use.

## Synopsis

```
#include "SDL.h"
```

```
SDL_Joystick *SDL_JoystickOpen(int index);
```

## Description

Opens a joystick for use within SDL. The *index* refers to the N'th joystick in the system. A joystick must be opened before it game be used.

## **Return Value**

Returns a `SDL_Joystick` structure on success. `NULL` on failure.

## Examples

```
SDL_Joystick *joy;
// Check for joystick
if(SDL_NumJoysticks(>0)){
    // Open joystick
    joy=SDL_JoystickOpen(0);

    if(joy)
    {
        printf("Opened Joystick 0\n");
        printf("Name: %s\n", SDL_JoystickName(0));
        printf("Number of Axes: %d\n", SDL_JoystickNumAxes(joy));
        printf("Number of Buttons: %d\n", SDL_JoystickNumButtons(joy));
        printf("Number of Balls: %d\n", SDL_JoystickNumBalls(joy));
    }
    else
        printf("Couldn't open Joystick 0\n");

    // Close if opened
    if(SDL_JoystickOpened(0))
        SDL_JoystickClose(joy);
}
```

## See Also

[SDL\\_JoystickClose](#)

---

[Prev](#)

SDL\_JoystickName

[Home](#)

[Up](#)

[Next](#)

SDL\_JoystickOpened



## SDL Library Documentation

[Prev](#)

[Next](#)

---

**SDL\_JoystickOpened**

## **Name**

SDL\_JoystickOpened -- Determine if a joystick has been opened

## Synopsis

```
#include "SDL.h"
```

```
int SDL_JoystickOpened(int index);
```

## **Description**

Determines whether a joystick has already been opened within the application. *index* refers to the N'th joystick on the system.

## **Return Value**

Returns 1 if the joystick has been opened, or 0 if it has not.

## See Also

[SDL\\_JoystickOpen](#), [SDL\\_JoystickClose](#)

---

[Prev](#)

SDL\_JoystickOpen

[Home](#)

[Up](#)

[Next](#)

SDL\_JoystickIndex

## SDL Library Documentation

[Prev](#)

[Next](#)

---



# **SDL\_JoystickIndex**

## **Name**

SDL\_JoystickIndex -- Get the index of an SDL\_Joystick.

## Synopsis

```
#include "SDL.h"
```

```
int SDL_JoystickIndex(SDL_Joystick *joystick);
```

## **Description**

Returns the index of a given SDL\_Joystick structure.

## **Return Value**

Index number of the joystick.

## See Also

[SDL\\_JoystickOpen](#)

---

[Prev](#)

SDL\_JoystickOpened

[Home](#)

[Up](#)

[Next](#)

SDL\_JoystickNumAxes

## SDL Library Documentation

[Prev](#)

[Next](#)

---

**SDL\_JoystickNumAxes**



## **Name**

SDL\_JoystickNumAxes -- Get the number of joystick axes

## Synopsis

```
#include "SDL.h"
```

```
int SDL_JoystickNumAxes(SDL_Joystick *joystick);
```

## **Description**

Return the number of axes available from a previously opened `SDL_Joystick`.

## **Return Value**

Number of axes.

## See Also

[SDL\\_JoystickGetAxis](#), [SDL\\_JoystickOpen](#)

---

[Prev](#)

SDL\_JoystickIndex

[Home](#)

[Up](#)

[Next](#)

SDL\_JoystickNumBalls

## SDL Library Documentation

[Prev](#)

[Next](#)

---

**SDL\_JoystickNumBalls**

## **Name**

SDL\_JoystickNumBalls -- Get the number of joystick trackballs



## Synopsis

```
#include "SDL.h"
```

```
int SDL_JoystickNumBalls(SDL_Joystick *joystick);
```

## **Description**

Return the number of trackballs available from a previously opened `SDL_Joystick`.

## **Return Value**

Number of trackballs.

## See Also

[SDL\\_JoystickGetBall](#), [SDL\\_JoystickOpen](#)

---

[Prev](#)

SDL\_JoystickNumAxes

[Home](#)

[Up](#)

[Next](#)

SDL\_JoystickNumHats

## SDL Library Documentation

[Prev](#)

[Next](#)

---

**SDL\_JoystickNumHats**

## **Name**

SDL\_JoystickNumHats -- Get the number of joystick hats

## Synopsis

```
#include "SDL.h"
```

```
int SDL_JoystickNumHats(SDL_Joystick *joystick);
```



## **Description**

Return the number of hats available from a previously opened SDL\_Joystick.

## **Return Value**

Number of hats.

## See Also

[SDL\\_JoystickGetHat](#), [SDL\\_JoystickOpen](#)

---

[Prev](#)

SDL\_JoystickNumBalls

[Home](#)

[Up](#)

[Next](#)

SDL\_JoystickNumButtons

## SDL Library Documentation

[Prev](#)

[Next](#)

---

**SDL\_JoystickNumButtons**

## **Name**

SDL\_JoystickNumButtons -- Get the number of joystick buttons

## Synopsis

```
#include "SDL.h"
```

```
int SDL_JoystickNumButtons(SDL_Joystick *joystick);
```

## **Description**

Return the number of buttons available from a previously opened `SDL_Joystick`.



## **Return Value**

Number of buttons.

## See Also

[SDL\\_JoystickGetButton](#), [SDL\\_JoystickOpen](#)

---

[Prev](#)

SDL\_JoystickNumHats

[Home](#)

[Up](#)

[Next](#)

SDL\_JoystickUpdate

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_JoystickUpdate**

## **Name**

SDL\_JoystickUpdate -- Updates the state of all joysticks

## Synopsis

```
#include "SDL.h"
```

```
void SDL_JoystickUpdate(void);
```

## Description

Updates the state(position, buttons, etc.) of all open joysticks. If joystick events have been enabled with [SDL\\_JoystickEventState](#) then this is called automatically in the event loop.

## See Also

[SDL\\_JoystickEventState](#)

---

[Prev](#)

SDL\_JoystickNumButtons

[Home](#)

[Up](#)

[Next](#)

SDL\_JoystickGetAxis



## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_JoystickGetAxis**

## **Name**

SDL\_JoystickGetAxis -- Get the current state of an axis

## Synopsis

```
#include "SDL.h"
```

```
Sint16 SDL_JoystickGetAxis(SDL_Joystick *joystick, int axis);
```

## Description

`SDL_JoystickGetAxis` returns the current state of the given *axis* on the given *joystick*.

On most modern joysticks the X axis is usually represented by *axis* 0 and the Y axis by *axis* 1. The value returned by `SDL_JoystickGetAxis` is a signed integer (-32768 to 32768) representing the current position of the *axis*, it may be necessary to impose certain tolerances on these values to account for jitter. It is worth noting that some joysticks use axes 2 and 3 for extra buttons.

## **Return Value**

Returns a 16-bit signed integer representing the current position of the *axis*.

## Examples

```
Sint16 x_move, y_move;  
SDL_Joystick *joy1;  
.  
.  
x_move=SDL_JoystickGetAxis(joy1, 0);  
y_move=SDL_JoystickGetAxis(joy1, 1);
```

## See Also

[SDL\\_JoystickNumAxes](#)

---

[Prev](#)

SDL\_JoystickUpdate

[Home](#)

[Up](#)

[Next](#)

SDL\_JoystickGetHat



## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_JoystickGetHat**

## **Name**

SDL\_JoystickGetHat -- Get the current state of a joystick hat

## Synopsis

```
#include "SDL.h"
```

```
Uint8 SDL_JoystickGetHat(SDL_Joystick *joystick, int hat);
```

## Description

SDL\_JoystickGetHat returns the current state of the given *hat* on the given *joystick*.

## Return Value

The current state is returned as a Uint8 which is defined as an OR'd combination of one or more of the following

- SDL\_HAT\_CENTERED
- SDL\_HAT\_UP
- SDL\_HAT\_RIGHT
- SDL\_HAT\_DOWN
- SDL\_HAT\_LEFT
- SDL\_HAT\_RIGHTUP
- SDL\_HAT\_RIGHTDOWN
- SDL\_HAT\_LEFTUP
- SDL\_HAT\_LEFTDOWN

## See Also

[SDL\\_JoystickNumHats](#)

---

[Prev](#)

SDL\_JoystickGetAxis

[Home](#)

[Up](#)

[Next](#)

SDL\_JoystickGetButton

## SDL Library Documentation

[Prev](#)

[Next](#)

---



# **SDL\_JoystickGetButton**

## **Name**

SDL\_JoystickGetButton -- Get the current state of a given button on a given joystick

## Synopsis

```
#include "SDL.h"
```

```
Uint8 SDL_JoystickGetButton(SDL_Joystick *joystick, int button);
```

## Description

SDL\_JoystickGetButton returns the current state of the given *button* on the given *joystick*.

## **Return Value**

1 if the button is pressed. Otherwise, 0.

## See Also

[SDL\\_JoystickNumButtons](#)

---

[Prev](#)

SDL\_JoystickGetHat

[Home](#)

[Up](#)

[Next](#)

SDL\_JoystickGetBall

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_JoystickGetBall**



## **Name**

SDL\_JoystickGetBall -- Get relative trackball motion

## Synopsis

```
#include "SDL.h"
```

```
int SDL_JoystickGetBall(SDL_Joystick *joystick, int ball, int *dx, int *dy);
```

## Description

Get the *ball* axis change.

Trackballs can only return relative motion since the last call to `SDL_JoystickGetBall`, these motion deltas are placed into *dx* and *dy*.

## **Return Value**

Returns 0 on success or -1 on failure

## Examples

```
int delta_x, delta_y;
SDL_Joystick *joy;
.
.
.
SDL_JoystickUpdate();
if(SDL_JoystickGetBall(joy, 0, &delta_x, &delta_y)==-1)
    printf("TrackBall Read Error!\n");
printf("Trackball Delta- X:%d, Y:%d\n", delta_x, delta_y);
```

## See Also

[SDL\\_JoystickNumBalls](#)

---

[Prev](#)

SDL\_JoystickGetButton

[Home](#)

[Up](#)

[Next](#)

SDL\_JoystickClose

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_JoystickClose**



## **Name**

SDL\_JoystickClose -- Closes a previously opened joystick

## Synopsis

```
#include "SDL.h"
```

```
void SDL_JoystickClose(SDL_Joystick *joystick);
```

## Description

Close a *joystick* that was previously opened with [SDL\\_JoystickOpen](#).

## See Also

[SDL\\_JoystickOpen](#), [SDL\\_JoystickOpened](#)

---

[Prev](#)

SDL\_JoystickGetBall

[Home](#)

[Up](#)

[Next](#)

Audio

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# Chapter 10. Audio

## Table of Contents

- [SDL\\_AudioSpec](#) — Audio Specification Structure
- [SDL\\_OpenAudio](#) — Opens the audio device with the desired parameters.
- [SDL\\_PauseAudio](#) — Pauses and unpauses the audio callback processing
- [SDL\\_GetAudioStatus](#) — Get the current audio state
- [SDL\\_LoadWAV](#) — Load a WAVE file
- [SDL\\_FreeWAV](#) — Frees previously opened WAV data
- [SDL\\_AudioCVT](#) — Audio Conversion Structure
- [SDL\\_BuildAudioCVT](#) — Initializes a SDL\_AudioCVT structure for conversion
- [SDL\\_ConvertAudio](#) — Convert audio data to a desired audio format.
- [SDL\\_MixAudio](#) — Mix audio data
- [SDL\\_LockAudio](#) — Lock out the callback function
- [SDL\\_UnlockAudio](#) — Unlock the callback function
- [SDL\\_CloseAudio](#) — Shuts down audio processing and closes the audio device.

Sound on the computer is translated from waves that you hear into a series of values, or samples, each representing the amplitude of the wave. When these samples are sent in a stream to a sound card, an approximation of the original wave can be recreated. The more bits used to represent the amplitude, and the greater frequency these samples are gathered, the closer the approximated sound is to the original, and the better the quality of sound.

This library supports both 8 and 16 bit signed and unsigned sound samples, at frequencies ranging from 11025 Hz to 44100 Hz, depending on the underlying hardware. If the hardware doesn't support the desired audio format or frequency, it can be emulated if desired (See [SDL\\_OpenAudio\(\)](#))

A commonly supported audio format is 16 bits per sample at 22050 Hz.

---

[Prev](#)

SDL\_JoystickClose

[Home](#)

[Up](#)

[Next](#)

SDL\_AudioSpec

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_AudioSpec**



## **Name**

SDL\_AudioSpec -- Audio Specification Structure

## Structure Definition

```
typedef struct{
    int freq;
    Uint16 format;
    Uint8 channels;
    Uint8 silence;
    Uint16 samples;
    Uint32 size;
    void (*callback)(void *userdata, Uint8 *stream, int len);
    void *userdata;
} SDL_AudioSpec;
```

## Structure Data

<i>freq</i>	Audio frequency in samples per second
<i>format</i>	Audio data format
<i>channels</i>	Number of channels: 1 mono, 2 stereo
<i>silence</i>	Audio buffer silence value (calculated)
<i>samples</i>	Audio buffer size in samples
<i>size</i>	Audio buffer size in bytes (calculated)
<i>callback(...)</i>	Callback function for filling the audio buffer
<i>userdata</i>	Pointer the user data which is passed to the callback function

## Description

The `SDL_AudioSpec` structure is used to describe the format of some audio data. This structure is used by [SDL\\_OpenAudio](#) and [SDL\\_LoadWAV](#). While all fields are used by `SDL_OpenAudio` only *freq*, *format*, *samples* and *channels* are used by `SDL_LoadWAV`. We will detail these common members here.

*freq*      The number of samples sent to the sound device every second.  
Common values are 11025, 22050 and 44100. The higher the better.

Specifies the size and type of each sample element

AUDIO\_U8

Unsigned 8-bit samples

AUDIO\_S8

Signed 8-bit samples

AUDIO\_U16 or AUDIO\_U16LSB

Unsigned 16-bit little-endian samples

AUDIO\_S16 or AUDIO\_S16LSB

Signed 16-bit little-endian samples

*format*

AUDIO\_U16MSB

Unsigned 16-bit big-endian samples

AUDIO\_S16MSB

Signed 16-bit big-endian samples

AUDIO\_U16SYS

Either AUDIO\_U16LSB or AUDIO\_U16MSB depending on you systems endianness

AUDIO\_S16SYS

Either AUDIO\_S16LSB or AUDIO\_S16MSB depending on you systems endianness

*channels* The number of seperate sound channels. 1 is mono (single channel), 2 is stereo (dual channel).

*samples* When used with [SDL\\_OpenAudio](#) this refers to the size of the audio buffer in samples. A sample a chunk of audio data of the size specified in *format* mulitplied by the number of channels. When the SDL\_AudioSpec is used with [SDL\\_LoadWAV](#) *samples* is set to 4096.

## See Also

[SDL\\_OpenAudio](#), [SDL\\_LoadWAV](#)

---

[Prev](#)  
Audio

[Home](#)  
[Up](#)

[Next](#)  
SDL\_OpenAudio

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_OpenAudio**



## **Name**

SDL\_OpenAudio -- Opens the audio device with the desired parameters.

## Synopsis

```
#include "SDL.h"
```

```
int SDL_OpenAudio(SDL_AudioSpec *desired, SDL_AudioSpec *obtained);
```

## Description

This function opens the audio device with the *desired* parameters, and returns 0 if successful, placing the actual hardware parameters in the structure pointed to by *obtained*. If *obtained* is NULL, the audio data passed to the callback function will be guaranteed to be in the requested format, and will be automatically converted to the hardware audio format if necessary. This function returns -1 if it failed to open the audio device, or couldn't set up the audio thread.

To open the audio device a *desired* [SDL\\_AudioSpec](#) must be created.

```
SDL_AudioSpec *desired;  
.  
.  
desired=(SDL_AudioSpec *)malloc(sizeof(SDL_AudioSpec));
```

You must then fill this structure with your desired audio specifications.

*desired->freq*

The desired audio frequency in samples-per-second.

*desired->format*

The desired audio format (see [SDL\\_AudioSpec](#))

*desired->samples*

The desired size of the audio buffer in samples. This number should be a power of two, and may be adjusted by the audio driver to a value more suitable for the hardware. Good values seem to range between 512 and 8192 inclusive, depending on the application and CPU speed. Smaller values yield faster response time, but can lead to underflow if the application is doing heavy processing and cannot fill the audio buffer in time. A stereo sample consists of both right and left channels in LR ordering. Note that the number of samples is directly related to time by the following formula:  $ms = (samples * 1000) / freq$

*desired->callback*

This should be set to a function that will be called when the audio device is ready for more data. It is passed a pointer to the audio buffer, and the length in bytes of the audio buffer. This function usually runs in a separate thread, and so you should protect data structures that it accesses by calling [SDL\\_LockAudio](#) and [SDL\\_UnlockAudio](#) in your code. The callback prototype is:

```
void callback(void *userdata, Uint8 *stream, int len);
```

*userdata* is the pointer stored in *userdata* field of the `SDL_AudioSpec`.  
*stream* is a pointer to the audio buffer you want to fill with information and  
*len* is the length of the audio buffer in bytes.

desired->*userdata*

This pointer is passed as the first parameter to the `callback` function.

`SDL_OpenAudio` reads these fields from the *desired* `SDL_AudioSpec` structure pass to the function and attempts to find an audio configuration matching your *desired*. As mentioned above, if the *obtained* parameter is `NULL` then `SDL` will convert from your *desired* audio settings to the hardware settings as it plays.

If *obtained* is `NULL` then the *desired* `SDL_AudioSpec` is your working specification, otherwise the *obtained* `SDL_AudioSpec` becomes the working specification and the *desired* specification can be deleted. The data in the working specification is used when building `SDL_AudioCVT`'s for converting loaded data to the hardware format.

`SDL_OpenAudio` calculates the *size* and *silence* fields for both the *desired* and *obtained* specifications. The *size* field stores the total size of the audio buffer in bytes, while the *silence* stores the value used to represent silence in the audio buffer

The audio device starts out playing *silence* when it's opened, and should be enabled for playing by calling [SDL\\_PauseAudio\(0\)](#) when you are ready for your audio *callback* function to be called. Since the audio driver may modify the requested *size* of the audio buffer, you should allocate any local mixing buffers after you open the audio device.

## Examples

```
/* Prototype of our callback function */
void my_audio_callback(void *userdata, Uint8 *stream, int len);

/* Open the audio device */
SDL_AudioSpec *desired, *obtained;
SDL_AudioSpec *hardware_spec;

/* Allocate a desired SDL_AudioSpec */
desired=(SDL_AudioSpec *)malloc(sizeof(SDL_AudioSpec));

/* Allocate space for the obtained SDL_AudioSpec */
obtained=(SDL_AudioSpec *)malloc(sizeof(SDL_AudioSpec));

/* 22050Hz - FM Radio quality */
desired->freq=22050;

/* 16-bit signed audio */
desired->format=AUDIO_S16LSB;

/* Mono */
desired->channels=0;

/* Large audio buffer reduces risk of dropouts but increases respons
desired->samples=8192;

/* Our callback function */
desired->callback=my_audio_callback;

desired->userdata=NULL;

/* Open the audio device */
if ( SDL_OpenAudio(desired, obtained) < 0 ){
    fprintf(stderr, "Couldn't open audio: %s\n", SDL_GetError());
    exit(-1);
}
/* desired spec is no longer needed */
free(desired);
hardware_spec=obtained;
.
.
/* Prepare callback for playing */
.
.
.
/* Start playing */
```

```
SDL_PauseAudio(0);
```

## See Also

[SDL\\_AudioSpec](#), [SDL\\_LockAudio](#), [SDL\\_UnlockAudio](#), [SDL\\_PauseAudio](#)

---

[Prev](#)

SDL\_AudioSpec

[Home](#)

[Up](#)

[Next](#)

SDL\_PauseAudio

## SDL Library Documentation

[Prev](#)

[Next](#)

---



# **SDL\_PauseAudio**

## **Name**

SDL\_PauseAudio -- Pauses and unpauses the audio callback processing

## Synopsis

```
#include "SDL.h"
```

```
void SDL_PauseAudio(int pause_on);
```

## Description

This function pauses and unpauses the audio callback processing. It should be called with *pause\_on*=0 after opening the audio device to start playing sound. This is so you can safely initialize data for your callback function after opening the audio device. Silence will be written to the audio device during the pause.

## See Also

[SDL\\_GetAudioStatus](#), [SDL\\_OpenAudio](#)

---

[Prev](#)

SDL\_OpenAudio

[Home](#)

[Up](#)

[Next](#)

SDL\_GetAudioStatus

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_GetAudioStatus**

## **Name**

SDL\_GetAudioStatus -- Get the current audio state



## Synopsis

```
#include "SDL.h"
```

```
SDL_audiostatus SDL_GetAudioStatus(void);
```

## Description

```
typedef enum{  
    SDL_AUDIO_STOPPED,  
    SDL_AUDIO_PAUSED,  
    SDL_AUDIO_PLAYING  
} SDL_audiostatus;
```

Returns either `SDL_AUDIO_STOPPED`, `SDL_AUDIO_PAUSED` or `SDL_AUDIO_PLAYING` depending on the current audio state.

## See Also

[SDL\\_PauseAudio](#)

---

[Prev](#)

SDL\_PauseAudio

[Home](#)

[Up](#)

[Next](#)

SDL\_LoadWAV

## SDL Library Documentation

[Prev](#)

[Next](#)

---

**SDL\_LoadWAV**

## **Name**

SDL\_LoadWAV -- Load a WAVE file

## Synopsis

```
#include "SDL.h"
```

```
SDL_AudioSpec *SDL_LoadWAV(const char *file, SDL_AudioSpec *spec, Uint8  
**audio_buf, Uint32 *audio_len);
```

## Description

`SDL_LoadWAV` This function loads a WAVE *file* into memory.

If this function succeeds, it returns the given [SDL\\_AudioSpec](#), filled with the audio data format of the wave data, and sets *audio\_buf* to a malloc'd buffer containing the audio data, and sets *audio\_len* to the length of that audio buffer, in bytes. You need to free the audio buffer with [SDL\\_FreeWAV](#) when you are done with it.

This function returns `NULL` and sets the SDL error message if the wave file cannot be opened, uses an unknown data format, or is corrupt. Currently raw, MS-ADPCM and IMA-ADPCM WAVE files are supported.



## Example

```
SDL_AudioSpec wav_spec;
Uint32 wav_length;
Uint8 *wav_buffer;

/* Load the WAV */
if( SDL_LoadWAV("test.wav", &wav_spec, &wav_buffer, &wav_length) ==
    fprintf(stderr, "Could not open test.wav: %s\n", SDL_GetError());
    exit(-1);
}
.
.
.
/* Do stuff with the WAV */
.
.
/* Free It */
SDL_FreeWAV(wav_buffer);
```

## See Also

[SDL\\_AudioSpec](#), [SDL\\_OpenAudio](#), [SDL\\_FreeWAV](#)

---

[Prev](#)

SDL\_GetAudioStatus

[Home](#)

[Up](#)

[Next](#)

SDL\_FreeWAV

## SDL Library Documentation

[Prev](#)

[Next](#)

---

**SDL\_FreeWAV**

## **Name**

SDL\_FreeWAV -- Frees previously opened WAV data

## Synopsis

```
#include "SDL.h"
```

```
void SDL_FreeWAV(Uint8 *audio_buf);
```

## Description

After a WAVE file has been opened with [SDL\\_LoadWAV](#) its data can eventually be freed with `SDL_FreeWAV`. *audio\_buf* is a pointer to the buffer created by `SDL_LoadWAV`.

## See Also

[SDL\\_LoadWAV](#)

---

[Prev](#)

SDL\_LoadWAV

[Home](#)

[Up](#)

[Next](#)

SDL\_AudioCVT



## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_AudioCVT**

## **Name**

SDL\_AudioCVT -- Audio Conversion Structure

## Structure Definition

```
typedef struct{
    int needed;
    Uint16 src_format;
    Uint16 dest_format;
    double rate_incr;
    Uint8 *buf;
    int len;
    int len_cvt;
    int len_mult;
    double len_ratio;
    void (*filters[10])(struct SDL_AudioCVT *cvt, Uint16 format);
    int filter_index;
} SDL_AudioCVT;
```

## Structure Data

<i>needed</i>	Set to one if the conversion is possible
<i>src_format</i>	Audio format of the source
<i>dest_format</i>	Audio format of the destination
<i>rate_incr</i>	Rate conversion increment
<i>buf</i>	Audio buffer
<i>len</i>	Length of the original audio buffer in bytes
<i>len_cvt</i>	Length of converted audio buffer in bytes (calculated)
<i>len_mult</i>	<i>buf</i> must be $len * len\_mult$ bytes in size(calculated)
<i>len_ratio</i>	Final audio size is $len * len\_ratio$
<i>filters[10](..)</i>	Pointers to functions needed for this conversion
<i>filter_index</i>	Current conversion function

## Description

The `SDL_AudioCVT` is used to convert audio data between different formats. A `SDL_AudioCVT` structure is created with the [SDL\\_BuildAudioCVT](#) function, while the actual conversion is done by the [SDL\\_ConvertAudio](#) function.

Many of the fields in the `SDL_AudioCVT` structure should be considered private and their function will not be discussed here.

`Uint8 *buf`

This points to the audio data that will be used in the conversion. It is both the source and the destination, which means the converted audio data overwrites the original data. It also means that the converted data may be larger than the original data (if you were converting from 8-bit to 16-bit, for instance), so you must ensure *buf* is large enough. See below.

`int len`

This is the length of the original audio data in bytes.

`int len_mult`

As explained above, the audio buffer needs to be big enough to store the converted data, which may be bigger than the original audio data. The length of *buf* should be  $len * len\_mult$ .

`double len_ratio`

When you have finished converting your audio data, you need to know how much of your audio buffer is valid.  $len * len\_ratio$  is the size of the converted audio data in bytes. This is very similar to *len\_mult*, however when the converted audio data is shorter than the original *len\_mult* would be 1. *len\_ratio*, on the other hand, would be a fractional number between 0 and 1.

## See Also

[SDL\\_BuildAudioCVT](#), [SDL\\_ConvertAudio](#), [SDL\\_AudioSpec](#)

---

[Prev](#)

SDL\_FreeWAV

[Home](#)

[Up](#)

[Next](#)

SDL\_BuildAudioCVT

## SDL Library Documentation

[Prev](#)

[Next](#)

---



# **SDL\_BuildAudioCVT**

## **Name**

SDL\_BuildAudioCVT -- Initializes a SDL\_AudioCVT structure for conversion

## Synopsis

```
#include "SDL.h"
```

```
int SDL_BuildAudioCVT(SDL_AudioCVT *cvt, Uint16 src_format, Uint8  
src_channels, int src_rate, Uint16 dst_format, Uint8 dst_channels, int dst_rate);
```

## Description

Before an [SDL\\_AudioCVT](#) structure can be used to convert audio data it must be initialized with source and destination information.

*src\_format* and *dst\_format* are the source and destination format of the conversion. (For information on audio formats see [SDL\\_AudioSpec](#)).  
*src\_channels* and *dst\_channels* are the number of channels in the source and destination formats. Finally, *src\_rate* and *dst\_rate* are the frequency or samples-per-second of the source and destination formats. Once again, see [SDL\\_AudioSpec](#).

## **Return Values**

Returns -1 if the filter could not be built or 1 if it could.

# Examples

See [SDL\\_ConvertAudio](#).

## See Also

[SDL\\_ConvertAudio](#), [SDL\\_AudioCVT](#)

---

[Prev](#)

SDL\_AudioCVT

[Home](#)

[Up](#)

[Next](#)

SDL\_ConvertAudio

## SDL Library Documentation

[Prev](#)

[Next](#)

---



# **SDL\_ConvertAudio**

## **Name**

SDL\_ConvertAudio -- Convert audio data to a desired audio format.

## Synopsis

```
#include "SDL.h"
```

```
int SDL_ConvertAudio(SDL_AudioCVT *cvt);
```

## Description

`SDL_ConvertAudio` takes one parameter, `cvt`, which was previously initialized. Initializing a [SDL\\_AudioCVT](#) is a two step process. First of all, the structure must be passed to [SDL\\_BuildAudioCVT](#) along with source and destination format parameters. Secondly, the `cvt->buf` and `cvt->len` fields must be setup. `cvt->buf` should point to the audio data and `cvt->len` should be set to the length of the audio data in bytes. Remember, the length of the buffer pointed to by `buf` should be  $len * len\_mult$  bytes in length.

Once the `SDL_AudioCVT` structure is initialized then we can pass it to `SDL_ConvertAudio`, which will convert the audio data pointer to by `cvt->buf`. If `SDL_ConvertAudio` returned 0 then the conversion was completed successfully, otherwise -1 is returned.

If the conversion completed successfully then the converted audio data can be read from `cvt->buf`. The amount of valid, converted, audio data in the buffer is equal to  $cvt->len * cvt->len\_ratio$ .

## Examples

```
/* Converting some WAV data to hardware format */
void my_audio_callback(void *userdata, Uint8 *stream, int len);

SDL_AudioSpec *desired, *obtained;
SDL_AudioSpec wav_spec;
SDL_AudioCVT wav_cvt;
Uint32 wav_len;
Uint8 *wav_buf;
int ret;

/* Allocated audio specs */
desired=(SDL_AudioSpec *)malloc(sizeof(SDL_AudioSpec));
obtained=(SDL_AudioSpec *)malloc(sizeof(SDL_AudioSpec));

/* Set desired format */
desired->freq=22050;
desired->format=AUDIO_S16LSB;
desired->samples=8192;
desired->callback=my_audio_callback;
desired->userdata=NULL;

/* Open the audio device */
if ( SDL_OpenAudio(desired, obtained) < 0 ){
    fprintf(stderr, "Couldn't open audio: %s\n", SDL_GetError());
    exit(-1);
}

free(desired);

/* Load the test.wav */
if( SDL_LoadWAV("test.wav", &wav_spec, &wav_buf, &wav_len) == NULL )
    fprintf(stderr, "Could not open test.wav: %s\n", SDL_GetError());
    SDL_CloseAudio();
    free(obtained);
    exit(-1);
}

/* Build AudioCVT */
ret = SDL_BuildAudioCVT(&wav_cvt,
                        wav_spec.format, wav_spec.channels, wav_spec
                        obtained->format, obtained->channels, obtain

/* Check that the convert was built */
if(ret==-1){
    fprintf(stderr, "Couldn't build converter!\n");
}
```

```
    SDL_CloseAudio();
    free(obtained);
    SDL_FreeWAV(wav_buf);
}

/* Setup for conversion */
wav_cvt.buf=(Uint8 *)malloc(wav_len*wav_cvt.len_mult);
wav_cvt.len=wav_len;
memcpy(wav_cvt.buf, wav_buf, wav_len);

/* We can delete to original WAV data now */
SDL_FreeWAV(wav_buf);

/* And now we're ready to convert */
SDL_ConvertAudio(&wav_cvt);

/* do whatever */
.
.
.
.
```

## See Also

[SDL\\_BuildAudioCVT](#), [SDL\\_AudioCVT](#)

---

[Prev](#)

SDL\_BuildAudioCVT

[Home](#)

[Up](#)

[Next](#)

SDL\_MixAudio

## SDL Library Documentation

[Prev](#)

[Next](#)

---



# **SDL\_MixAudio**

## **Name**

SDL\_MixAudio -- Mix audio data

## Synopsis

```
#include "SDL.h"
```

```
void SDL_MixAudio(Uint8 *dst, Uint8 *src, Uint32 len, int volume);
```

## Description

This function takes two audio buffers of *len* bytes each of the playing audio format and mixes them, performing addition, volume adjustment, and overflow clipping. The *volume* ranges from 0 to `SDL_MIX_MAXVOLUME` and should be set to the maximum value for full audio volume. Note this does not change hardware volume. This is provided for convenience -- you can mix your own audio data.

**Note:** Do not use this function for mixing together more than two streams of sample data. The output from repeated application of this function may be distorted by clipping, because there is no accumulator with greater range than the input (not to mention this being an inefficient way of doing it). Use mixing functions from `SDL_mixer`, `OpenAL`, or write your own mixer instead.

## See Also

[SDL\\_OpenAudio](#)

---

[Prev](#)

SDL\_ConvertAudio

[Home](#)

[Up](#)

[Next](#)

SDL\_LockAudio

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_LockAudio**

## **Name**

SDL\_LockAudio -- Lock out the callback function



## Synopsis

```
#include "SDL.h"
```

```
void SDL_LockAudio(void);
```

## **Description**

The lock manipulated by these functions protects the callback function. During a LockAudio period, you can be guaranteed that the callback function is not running. Do not call these from the callback function or you will cause deadlock.

## See Also

[SDL\\_OpenAudio](#)

---

[Prev](#)

SDL\_MixAudio

[Home](#)

[Up](#)

[Next](#)

SDL\_UnlockAudio

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_UnlockAudio**

## **Name**

SDL\_UnlockAudio -- Unlock the callback function

## Synopsis

```
#include "SDL.h"
```

```
void SDL_UnlockAudio(void);
```

## Description

Unlocks a previous [SDL\\_LockAudio](#) call.



## See Also

[SDL\\_OpenAudio](#)

---

[Prev](#)

SDL\_LockAudio

[Home](#)

[Up](#)

[Next](#)

SDL\_CloseAudio

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_CloseAudio**

## **Name**

SDL\_CloseAudio -- Shuts down audio processing and closes the audio device.

## Synopsis

```
#include "SDL.h"
```

```
void SDL_CloseAudio(void);
```

## **Description**

This function shuts down audio processing and closes the audio device.

## See Also

[SDL\\_OpenAudio](#)

---

[Prev](#)

SDL\_UnlockAudio

[Home](#)

[Up](#)

[Next](#)

CD-ROM

## SDL Library Documentation

[Prev](#)

[Next](#)

---



# Chapter 11. CD-ROM

## Table of Contents

[SDL\\_CDNumDrives](#) — Returns the number of CD-ROM drives on the system.

[SDL\\_CDName](#) — Returns a human-readable, system-dependent identifier for the CD-ROM.

[SDL\\_CDOpen](#) — Opens a CD-ROM drive for access.

[SDL\\_CDStatus](#) — Returns the current status of the given drive.

[SDL\\_CDPlay](#) — Play a CD

[SDL\\_CDPlayTracks](#) — Play the given CD track(s)

[SDL\\_CDPause](#) — Pauses a CDROM

[SDL\\_CDResume](#) — Resumes a CDROM

[SDL\\_CDStop](#) — Stops a CDROM

[SDL\\_CDEject](#) — Ejects a CDROM

[SDL\\_CDClose](#) — Closes a SDL\_CD handle

[SDL\\_CD](#) — CDROM Drive Information

[SDL\\_CDtrack](#) — CD Track Information Structure

SDL supports audio control of up to 32 local CD-ROM drives at once.

You use this API to perform all the basic functions of a CD player, including listing the tracks, playing, stopping, and ejecting the CD-ROM. (Currently, multi-changer CD drives are not supported.)

Before you call any of the SDL CD-ROM functions, you must first call "SDL\_Init(SDL\_INIT\_CDROM)", which scans the system for CD-ROM drives, and sets the program up for audio control. Check the return code, which should be 0, to see if there were any errors in starting up.

After you have initialized the library, you can find out how many drives are available using the `SDL_CDNumDrives()` function. The first drive listed is the system default CD-ROM drive. After you have chosen a drive, and have opened it with `SDL_CDOpen()`, you can check the status and start playing if there's a CD in the drive.

A CD-ROM is organized into one or more tracks, each consisting of a certain number of "frames". Each frame is ~2K in size, and at normal playing speed, a

CD plays 75 frames per second. SDL works with the number of frames on a CD, but this can easily be converted to the more familiar minutes/seconds format by using the `FRAMES_TO_MSF()` macro.

---

[Prev](#)

SDL\_CloseAudio

[Home](#)

[Up](#)

[Next](#)

SDL\_CDNumDrives

## SDL Library Documentation

[Prev](#)

[Next](#)

---

**SDL\_CDNumDrives**

## **Name**

SDL\_CDNumDrives -- Returns the number of CD-ROM drives on the system.

## Synopsis

```
#include "SDL.h"
```

```
int SDL_CDNumDrives(void);
```

## **Description**

Returns the number of CD-ROM drives on the system.

## See Also

[SDL\\_CDOpen](#)

---

[Prev](#)

CD-ROM

[Home](#)

[Up](#)

[Next](#)

SDL\_CDName



## SDL Library Documentation

[Prev](#)

[Next](#)

---

**SDL\_CDName**

## **Name**

SDL\_CDName -- Returns a human-readable, system-dependent identifier for the CD-ROM.

## Synopsis

```
#include "SDL.h"
```

```
const char *SDL_CDName(int drive);
```

## Description

Returns a human-readable, system-dependent identifier for the CD-ROM. *drive* is the index of the drive. Drive indices start to 0 and end at `SDL_CDNumDrives()-1`.

## Examples

- `"/dev/cdrom"`
- `"E:"`
- `"/dev/disk/ide/1/master"`

## See Also

[SDL\\_CDNumDrives](#)

---

[Prev](#)

SDL\_CDNumDrives

[Home](#)

[Up](#)

[Next](#)

SDL\_CDOpen

## SDL Library Documentation

[Prev](#)

[Next](#)

---



**SDL\_CDOpen**

## **Name**

SDL\_CDOpen -- Opens a CD-ROM drive for access.

## Synopsis

```
#include "SDL.h"
```

```
SDL_CD *SDL_CDOpen(int drive);
```

## Description

Opens a CD-ROM drive for access. It returns a [SDL\\_CD](#) structure on success, or NULL if the drive was invalid or busy. This newly opened CD-ROM becomes the default CD used when other CD functions are passed a NULL CD-ROM handle.

Drives are numbered starting with 0. Drive 0 is the system default CD-ROM.

## Examples

```
SDL_CD *cdrom;
int cur_track;
int min, sec, frame;
SDL_Init(SDL_INIT_CDROM);
atexit(SDL_Quit);

/* Check for CD drives */
if(!SDL_CDNumDrives()){
    /* None found */
    fprintf(stderr, "No CDROM devices available\n");
    exit(-1);
}

/* Open the default drive */
cdrom=SDL_CDOpen(0);

/* Did it open? Check if cdrom is NULL */
if(!cdrom){
    fprintf(stderr, "Couldn't open drive: %s\n", SDL_GetError());
    exit(-1);
}

/* Print Volume info */
printf("Name: %s\n", SDL_CDName(0));
printf("Tracks: %d\n", cdrom->numtracks);
for(cur_track=0;cur_track < cdrom->numtracks; cur_track++){
    FRAMES_TO_MSFCdrom->track[cur_track].length, &min, &sec, &frame);
    printf("\tTrack %d: Length %d:%d\n", cur_track, min, sec);
}

SDL_CDClose(cdrom);
```

## See Also

[SDL\\_CD](#), [SDL\\_CDtrack](#), [SDL\\_CDClose](#)

---

[Prev](#)

SDL\_CDName

[Home](#)

[Up](#)

[Next](#)

SDL\_CDStatus

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_CDStatus**



## **Name**

SDL\_CDStatus -- Returns the current status of the given drive.

## Synopsis

```
#include "SDL.h"
```

```
CDstatus SDL_CDStatus(SDL_CD *cdrom);
```

```
/* Given a status, returns true if there's a disk in the drive */  
#define CD_INDRIVE(status)      ((int)status > 0)
```

## Description

This function returns the current status of the given drive. Status is described like so:

```
typedef enum {  
    CD_TRAYEMPTY,  
    CD_STOPPED,  
    CD_PLAYING,  
    CD_PAUSED,  
    CD_ERROR = -1  
} CDstatus;
```

If the drive has a CD in it, the table of contents of the CD and current play position of the CD will be stored in the `SDL_CD` structure.

The macro `CD_INDRIVE` is provided for convenience, and given a status returns true if there's a disk in the drive.

**Note:** `SDL_CDstatus` also updates the [SDL\\_CD](#) structure passed to it.

## Example

```
int playTrack(int track)
{
    int playing = 0;

    if ( CD_INDRIVE(SDL_CDStatus(cdrom)) ) {
        /* clamp to the actual number of tracks on the CD */
        if (track >= cdrom->numtracks) {
            track = cdrom->numtracks-1;
        }

        if ( SDL_CDPlayTracks(cdrom, track, 0, 1, 0) == 0 ) {
            playing = 1;
        }
    }
    return playing;
}
```

## See Also

[SDL\\_CD](#)

---

[Prev](#)

SDL\_CDOpen

[Home](#)

[Up](#)

[Next](#)

SDL\_CDPlay

## SDL Library Documentation

[Prev](#)

[Next](#)

---

**SDL\_CDPlay**

## **Name**

SDL\_CDPlay -- Play a CD



## Synopsis

```
#include "SDL.h"
```

```
int SDL_CDP1ay(SDL_CD *cdrom, int start, int length);
```

## **Description**

Plays the given *cdrom*, starting a frame *start* for *length* frames.

## **Return Values**

Returns 0 on success, or -1 on an error.

## See Also

[SDL\\_CDPlayTracks](#), [SDL\\_CDStop](#)

---

[Prev](#)

SDL\_CDStatus

[Home](#)

[Up](#)

[Next](#)

SDL\_CDPlayTracks

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_CDPlayTracks**

## **Name**

SDL\_CDPlayTracks -- Play the given CD track(s)

## Synopsis

```
#include "SDL.h"
```

```
int SDL_CDPlayTracks(SDL_CD *cdrom, int start_track, int start_frame, int  
ntracks, int nframes);
```



## Description

`SDL_CDPlayTracks` plays the given CD starting at track *start\_track*, for *ntracks* tracks.

*start\_frame* is the frame offset, from the beginning of the *start\_track*, at which to start. *nframes* is the frame offset, from the beginning of the last track (*start\_track+ntracks*), at which to end playing.

`SDL_CDPlayTracks` should only be called after calling [SDL\\_CDStatus](#) to get track information about the CD.

**Note:** Data tracks are ignored.

## **Return Value**

Returns 0, or -1 if there was an error.

## Examples

```
/* assuming cdrom is a previously opened device */
/* Play the entire CD */
if(CD_INDRIVE(SDL_CDStatus(cdrom)))
    SDL_CDPlayTracks(cdrom, 0, 0, 0, 0);

/* Play the first track */
if(CD_INDRIVE(SDL_CDStatus(cdrom)))
    SDL_CDPlayTracks(cdrom, 0, 0, 1, 0);

/* Play first 15 seconds of the 2nd track */
if(CD_INDRIVE(SDL_CDStatus(cdrom)))
    SDL_CDPlayTracks(cdrom, 1, 0, 0, CD_FPS*15);
```

## See Also

[SDL\\_CDPlay](#), [SDL\\_CDStatus](#), [SDL\\_CD](#)

---

[Prev](#)

SDL\_CDPlay

[Home](#)

[Up](#)

[Next](#)

SDL\_CDPause

## SDL Library Documentation

[Prev](#)

[Next](#)

---

**SDL\_CDPause**

## **Name**

SDL\_CDPause -- Pauses a CDROM

## Synopsis

```
#include "SDL.h"
```

```
int SDL_CDPause(SDL_CD *cdrom);
```



## **Description**

Pauses play on the given *cdrom*.

## **Return Value**

Returns 0 on success, or -1 on an error.

## See Also

[SDL\\_CDPlay](#), [SDL\\_CDResume](#)

---

[Prev](#)

SDL\_CDPlayTracks

[Home](#)

[Up](#)

[Next](#)

SDL\_CDResume

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_CDResume**

## **Name**

SDL\_CDResume -- Resumes a CDROM

## Synopsis

```
#include "SDL.h"
```

```
int SDL_CDResume(SDL_CD *cdrom);
```

## **Description**

Resumes play on the given *cdrom*.



## **Return Value**

Returns 0 on success, or -1 on an error.

## See Also

[SDL\\_CDPlay](#), [SDL\\_CDPause](#)

---

[Prev](#)

SDL\_CDPause

[Home](#)

[Up](#)

[Next](#)

SDL\_CDStop

## SDL Library Documentation

[Prev](#)

[Next](#)

---

**SDL\_CDStop**

## **Name**

SDL\_CDStop -- Stops a CDROM

## Synopsis

```
#include "SDL.h"
```

```
int SDL_CDStop(SDL_CD *cdrom);
```

## **Description**

Stops play on the given *cdrom*.

## **Return Value**

Returns 0 on success, or -1 on an error.



## See Also

[SDL\\_CDPlay](#),

---

[Prev](#)

SDL\_CDResume

[Home](#)

[Up](#)

[Next](#)

SDL\_CDEject

## SDL Library Documentation

[Prev](#)

[Next](#)

---

**SDL\_CDEject**

## **Name**

SDL\_CDEject -- Ejects a CDROM

## Synopsis

```
#include "SDL.h"
```

```
int SDL_CDEject(SDL_CD *cdrom);
```

## **Description**

Ejects the given *cdrom*.

## **Return Value**

Returns 0 on success, or -1 on an error.

## See Also

[SDL\\_CD](#)

---

[Prev](#)

SDL\_CDStop

[Home](#)

[Up](#)

[Next](#)

SDL\_CDClose



## SDL Library Documentation

[Prev](#)

[Next](#)

---

**SDL\_CDClose**

## **Name**

SDL\_CDclose -- Closes a SDL\_CD handle

## Synopsis

```
#include "SDL.h"
```

```
void SDL_CDClose(SDL_CD *cdrom);
```

## **Description**

Closes the given *cdrom* handle.

## See Also

[SDL\\_CDOpen](#), [SDL\\_CD](#)

---

[Prev](#)

SDL\_CDEject

[Home](#)

[Up](#)

[Next](#)

SDL\_CD

## SDL Library Documentation

[Prev](#)

[Next](#)

---

**SDL\_CD**



## **Name**

SDL\_CD -- CDROM Drive Information

## Structure Definition

```
typedef struct{
    int id;
    CDstatus status;
    int numtracks;
    int cur_track;
    int cur_frame;
    SDL_CDtrack track[SDL_MAX_TRACKS+1];
} SDL_CD;
```

## Structure Data

<i>id</i>	Private drive identifier
<i>status</i>	Drive <a href="#">status</a>
<i>numtracks</i>	Number of tracks on the CD
<i>cur_track</i>	Current track
<i>cur_frame</i>	Current frame offset within the track
<i>track</i> [SDL_MAX_TRACKS+1]	Array of track descriptions. (see <a href="#">SDL_CDtrack</a> )

## Description

An `SDL_CD` structure is returned by [SDL\\_CDOpen](#). It represents an opened CDROM device and stores information on the layout of the tracks on the disc.

A frame is the base data unit of a CD. `CD_FPS` frames is equal to 1 second of music. SDL provides two macros for converting between time and frames: `FRAMES_TO_MSF(f, M, S, F)` and `MSF_TO_FRAMES`.

## Examples

```
int min, sec, frame;  
int frame_offset;
```

```
FRAMES_TO_MSF(cdrom->cur_frame, &min, &sec, &frame);  
printf("Current Position: %d minutes, %d seconds, %d frames\n", min,  
frame_offset=MSF_TO_FRAMES(min, sec, frame));
```

## See Also

[SDL\\_CDopen](#), [SDL\\_CDtrack](#)

---

[Prev](#)

SDL\_CDclose

[Home](#)

[Up](#)

[Next](#)

SDL\_CDtrack

## SDL Library Documentation

[Prev](#)

[Next](#)

---

**SDL\_CDtrack**



## **Name**

SDL\_CDtrack -- CD Track Information Structure

## Structure Definition

```
typedef struct{
    Uint8 id;
    Uint8 type;
    Uint32 length;
    Uint32 offset;
} SDL_CDtrack;
```

## Structure Data

*id* Track number (0-99)

*type* SDL\_AUDIO\_TRACK or SDL\_DATA\_TRACK

*length* Length, in frames, of this track

*offset* Frame offset to the beginning of this track

## Description

SDL\_CDtrack stores data on each track on a CD, its fields should be pretty self explanatory. It is a member a the [SDL\\_CD](#) structure.

**Note:** Frames can be converted to standard timings. There are CD\_FPS frames per second, so  $SDL\_CDtrack.length/CD\_FPS=length\_in\_seconds$ .

## See Also

[SDL\\_CD](#)

---

[Prev](#)

SDL\_CD

[Home](#)

[Up](#)

[Next](#)

Multi-threaded  
Programming

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# Chapter 12. Multi-threaded Programming

## Table of Contents

[SDL\\_CreateThread](#) — Creates a new thread of execution that shares its parent's properties.

[SDL\\_ThreadID](#) — Get the 32-bit thread identifier for the current thread.

[SDL\\_GetThreadID](#) — Get the SDL thread ID of a `SDL_Thread`

[SDL\\_WaitThread](#) — Wait for a thread to finish.

[SDL\\_KillThread](#) — Gracelessly terminates the thread.

[SDL\\_CreateMutex](#) — Create a mutex

[SDL\\_DestroyMutex](#) — Destroy a mutex

[SDL\\_mutexP](#) — Lock a mutex

[SDL\\_mutexV](#) — Unlock a mutex

[SDL\\_CreateSemaphore](#) — Creates a new semaphore and assigns an initial value to it.

[SDL\\_DestroySemaphore](#) — Destroys a semaphore that was created by

[SDL\\_CreateSemaphore](#).

[SDL\\_SemWait](#) — Lock a semaphore and suspend the thread if the semaphore value is zero.

[SDL\\_SemTryWait](#) — Attempt to lock a semaphore but don't suspend the thread.

[SDL\\_SemWaitTimeout](#) — Lock a semaphore, but only wait up to a specified maximum time.

[SDL\\_SemPost](#) — Unlock a semaphore.

[SDL\\_SemValue](#) — Return the current value of a semaphore.

[SDL\\_CreateCond](#) — Create a condition variable

[SDL\\_DestroyCond](#) — Destroy a condition variable

[SDL\\_CondSignal](#) — Restart a thread wait on a condition variable

[SDL\\_CondBroadcast](#) — Restart all threads waiting on a condition variable

[SDL\\_CondWait](#) — Wait on a condition variable

[SDL\\_CondWaitTimeout](#) — Wait on a condition variable, with timeout

SDL provides functions for creating threads, mutexes, semaphores and condition variables.

In general, you must be very aware of concurrency and data integrity issues when writing multi-threaded programs. Some good guidelines include:

- Don't call SDL video/event functions from separate threads
- Don't use any library functions in separate threads
- Don't perform any memory management in separate threads
- Lock global variables which may be accessed by multiple threads
- Never terminate threads, always set a flag and wait for them to quit
- Think very carefully about all possible ways your code may interact

**Note:** SDL's threading is not implemented on MacOS, due to that lack of preemptive thread support (MacOS X does not suffer from this problem)

---

[Prev](#)

SDL\_CDtrack

[Home](#)

[Up](#)

[Next](#)

SDL\_CreateThread



## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_CreateThread**

## **Name**

SDL\_CreateThread -- Creates a new thread of execution that shares its parent's properties.

## Synopsis

```
#include "SDL.h"  
#include "SDL_thread.h"
```

```
SDL_Thread *SDL_CreateThread(int (*fn)(void *), void *data);
```

## Description

`SDL_CreateThread` creates a new thread of execution that shares all of its parent's global memory, signal handlers, file descriptors, etc, and runs the function *fn* passed the void pointer *data*. The thread quits when this function returns.

## See Also

[SDL\\_KillThread](#)

---

[Prev](#)

Multi-threaded  
Programming

[Home](#)

[Up](#)

[Next](#)

SDL\_ThreadID

## SDL Library Documentation

[Prev](#)

[Next](#)

---

**SDL\_ThreadID**



## **Name**

SDL\_ThreadID -- Get the 32-bit thread identifier for the current thread.

## Synopsis

```
#include "SDL.h"  
#include "SDL_thread.h"  
  
Uint32 SDL_ThreadID(void);
```

## Description

Get the 32-bit thread identifier for the current thread.

---

[Prev](#)

SDL\_CreateThread

[Home](#)

[Up](#)

[Next](#)

SDL\_GetThreadID

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_GetThreadID**

## **Name**

SDL\_GetThreadID -- Get the SDL thread ID of a SDL\_Thread

## Synopsis

```
#include "SDL.h"  
#include "SDL_thread.h"
```

```
Uint32 SDL_GetThreadID(SDL_Thread *thread);
```

## Description

Returns the ID of a SDL\_Thread created by [SDL\\_CreateThread](#).



## See Also

[SDL\\_CreateThread](#)

---

[Prev](#)

SDL\_ThreadID

[Home](#)

[Up](#)

[Next](#)

SDL\_WaitThread

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_WaitThread**

## **Name**

SDL\_WaitThread -- Wait for a thread to finish.

## Synopsis

```
#include "SDL.h"  
#include "SDL_thread.h"
```

```
void SDL_WaitThread(SDL_Thread *thread, int *status);
```

## **Description**

Wait for a thread to finish (timeouts are not supported).

## **Return Value**

The return code for the thread function is placed in the area pointed to by *status*, if *status* is not NULL.

## See Also

[SDL\\_CreateThread](#)

---

[Prev](#)

SDL\_GetThreadID

[Home](#)

[Up](#)

[Next](#)

SDL\_KillThread



## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_KillThread**

## **Name**

SDL\_KillThread -- Gracelessly terminates the thread.

## Synopsis

```
#include "SDL.h"  
#include "SDL_thread.h"
```

```
void SDL_KillThread(SDL_Thread *thread);
```

## Description

`SDL_KillThread` gracelessly terminates the thread associated with *thread*. If possible, you should use some other form of IPC to signal the thread to quit.

## See Also

[SDL\\_CreateThread](#), [SDL\\_WaitThread](#)

---

[Prev](#)

SDL\_WaitThread

[Home](#)

[Up](#)

[Next](#)

SDL\_CreateMutex

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_CreateMutex**



## **Name**

SDL\_CreateMutex -- Create a mutex

## Synopsis

```
#include "SDL.h"  
#include "SDL_thread.h"
```

```
SDL_mutex *SDL_CreateMutex(void);
```

## **Description**

Create a new, unlocked mutex.

## Examples

```
SDL_mutex *mut;

mut=SDL_CreateMutex();
.
.
if(SDL_mutexP(mut)==-1){
    fprintf(stderr, "Couldn't lock mutex\n");
    exit(-1);
}
.
/* Do stuff while mutex is locked */
.
.
if(SDL_mutexV(mut)==-1){
    fprintf(stderr, "Couldn't unlock mutex\n");
    exit(-1);
}

SDL_DestroyMutex(mut);
```

## See Also

[SDL\\_mutexP](#), [SDL\\_mutexV](#), [SDL\\_DestroyMutex](#)

---

[Prev](#)

SDL\_KillThread

[Home](#)

[Up](#)

[Next](#)

SDL\_DestroyMutex

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_DestroyMutex**

## **Name**

SDL\_DestroyMutex -- Destroy a mutex



## Synopsis

```
#include "SDL.h"  
#include "SDL_thread.h"
```

```
void SDL_DestroyMutex(SDL_mutex *mutex);
```

## Description

Destroy a previously [created](#) mutex.

## See Also

[SDL\\_CreateMutex](#)

---

[Prev](#)

SDL\_CreateMutex

[Home](#)

[Up](#)

[Next](#)

SDL\_mutexP

## SDL Library Documentation

[Prev](#)

[Next](#)

---

**SDL\_mutexP**

## **Name**

SDL\_mutexP -- Lock a mutex

## Synopsis

```
#include "SDL.h"  
#include "SDL_thread.h"
```

```
int SDL_mutexP(SDL_mutex *mutex);
```

## Description

Locks the *mutex*, which was previously created with [SDL\\_CreateMutex](#). If the mutex is already locked then `SDL_mutexP` will not return until it is [unlocked](#). Returns 0 on success, or -1 on an error.

SDL also defines a macro `#define SDL_LockMutex(m) SDL_mutexP(m)`.



## See Also

[SDL\\_CreateMutex](#), [SDL\\_mutexV](#)

---

[Prev](#)

SDL\_DestroyMutex

[Home](#)

[Up](#)

[Next](#)

SDL\_mutexV

## SDL Library Documentation

[Prev](#)

[Next](#)

---

**SDL\_mutexV**

## **Name**

SDL\_mutexV -- Unlock a mutex

## Synopsis

```
#include "SDL.h"  
#include "SDL_thread.h"
```

```
int SDL_mutexV(SDL_mutex *mutex);
```

## Description

Unlocks the *mutex*, which was previously created with [SDL\\_CreateMutex](#). Returns 0 on success, or -1 on an error.

SDL also defines a macro `#define SDL_UnlockMutex(m) SDL_mutexV(m)`.

## See Also

[SDL\\_CreateMutex](#), [SDL\\_mutexP](#)

---

[Prev](#)

SDL\_mutexP

[Home](#)

[Up](#)

[Next](#)

SDL\_CreateSemaphore

## SDL Library Documentation

[Prev](#)

[Next](#)

---



# **SDL\_CreateSemaphore**

## **Name**

SDL\_CreateSemaphore -- Creates a new semaphore and assigns an initial value to it.

## Synopsis

```
#include "SDL.h"  
#include "SDL_thread.h"
```

```
SDL_sem *SDL_CreateSemaphore(Uint32 initial_value);
```

## Description

`SDL_CreateSemaphore()` creates a new semaphore and initializes it with the value *initial\_value*. Each locking operation on the semaphore by [SDL\\_SemWait](#), [SDL\\_SemTryWait](#) or [SDL\\_SemWaitTimeout](#) will atomically decrement the semaphore value. The locking operation will be blocked if the semaphore value is not positive (greater than zero). Each unlock operation by [SDL\\_SemPost](#) will atomically increment the semaphore value.

## **Return Value**

Returns a pointer to an initialized semaphore or NULL if there was an error.

## Examples

```
SDL_sem *my_sem;  
  
my_sem = SDL_CreateSemaphore(INITIAL_SEM_VALUE);  
  
if (my_sem == NULL) {  
    return CREATE_SEM_FAILED;  
}
```

## See Also

[SDL\\_DestroySemaphore](#), [SDL\\_SemWait](#), [SDL\\_SemTryWait](#), [SDL\\_SemWaitTimeout](#),  
[SDL\\_SemPost](#), [SDL\\_SemValue](#)

---

[Prev](#)

SDL\_mutexV

[Home](#)

[Up](#)

[Next](#)

SDL\_DestroySemaphore

## SDL Library Documentation

[Prev](#)

[Next](#)

---



# **SDL\_DestroySemaphore**

## **Name**

SDL\_DestroySemaphore -- Destroys a semaphore that was created by [SDL\\_CreateSemaphore](#).

## Synopsis

```
#include "SDL.h"  
#include "SDL_thread.h"
```

```
void SDL_DestroySemaphore(SDL_sem *sem);
```

## Description

`SDL_DestroySemaphore` destroys the semaphore pointed to by *sem* that was created by [SDL\\_CreateSemaphore](#). It is not safe to destroy a semaphore if there are threads currently blocked waiting on it.

## Examples

```
if (my_sem != NULL) {  
    SDL_DestroySemaphore(my_sem);  
    my_sem = NULL;  
}
```

## See Also

[SDL\\_CreateSemaphore](#), [SDL\\_SemWait](#), [SDL\\_SemTryWait](#), [SDL\\_SemWaitTimeout](#),  
[SDL\\_SemPost](#), [SDL\\_SemValue](#)

---

[Prev](#)

SDL\_CreateSemaphore

[Home](#)

[Up](#)

[Next](#)

SDL\_SemWait

## SDL Library Documentation

[Prev](#)

[Next](#)

---

**SDL\_SemWait**



## **Name**

SDL\_SemWait -- Lock a semaphore and suspend the thread if the semaphore value is zero.

## Synopsis

```
#include "SDL.h"  
#include "SDL_thread.h"
```

```
int SDL_SemWait(SDL_sem *sem);
```

## Description

`SDL_Semwait()` suspends the calling thread until either the semaphore pointed to by *sem* has a positive value, the call is interrupted by a signal or error. If the call is successful it will atomically decrement the semaphore value.

After `SDL_Semwait()` is successful, the semaphore can be released and its count atomically incremented by a successful call to [SDL\\_SemPost](#).

## **Return Value**

Returns 0 if successful or -1 if there was an error (leaving the semaphore unchanged).

## Examples

```
if (SDL_SemWait(my_sem) == -1) {  
    return WAIT_FAILED;  
}  
  
...  
  
SDL_SemPost(my_sem);
```

## See Also

[SDL\\_CreateSemaphore](#), [SDL\\_DestroySemaphore](#), [SDL\\_SemTryWait](#),  
[SDL\\_SemWaitTimeout](#), [SDL\\_SemPost](#), [SDL\\_SemValue](#)

---

[Prev](#)

SDL\_DestroySemaphore

[Home](#)

[Up](#)

[Next](#)

SDL\_SemTryWait

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# SDL\_SemTryWait



## **Name**

SDL\_SemTryWait -- Attempt to lock a semaphore but don't suspend the thread.

## Synopsis

```
#include "SDL.h"  
#include "SDL_thread.h"
```

```
int SDL_SemTrywait(SDL_sem *sem);
```

## Description

`SDL_SemTryWait` is a non-blocking variant of [SDL\\_SemWait](#). If the value of the semaphore pointed to by *sem* is positive it will atomically decrement the semaphore value and return 0, otherwise it will return `SDL_MUTEX_TIMEOUT` instead of suspending the thread.

After `SDL_SemTryWait` is successful, the semaphore can be released and its count atomically incremented by a successful call to [SDL\\_SemPost](#).

## **Return Value**

Returns 0 if the semaphore was successfully locked or either `SDL_MUTEX_TIMEOUT` or -1 if the thread would have suspended or there was an error, respectively.

If the semaphore was not successfully locked, the semaphore will be unchanged.

## Examples

```
res = SDL_SemTryWait(my_sem);

if (res == SDL_MUTEX_TIMEOUT) {
    return TRY_AGAIN;
}
if (res == -1) {
    return WAIT_ERROR;
}

...

SDL_SemPost(my_sem);
```

## See Also

[SDL\\_CreateSemaphore](#), [SDL\\_DestroySemaphore](#), [SDL\\_SemWait](#),  
[SDL\\_SemWaitTimeout](#), [SDL\\_SemPost](#), [SDL\\_SemValue](#)

---

[Prev](#)

SDL\_SemWait

[Home](#)

[Up](#)

[Next](#)

SDL\_SemWaitTimeout

## SDL Library Documentation

[Prev](#)

[Next](#)

---

**SDL\_SemWaitTimeout**



## **Name**

SDL\_SemWaitTimeout -- Lock a semaphore, but only wait up to a specified maximum time.

## Synopsis

```
#include "SDL.h"  
#include "SDL_thread.h"
```

```
int SDL_SemWaitTimeout(SDL_sem *sem, Uint32 timeout);
```

## Description

`SDL_SemWaitTimeout()` is a variant of [SDL\\_SemWait](#) with a maximum timeout value. If the value of the semaphore pointed to by *sem* is positive (greater than zero) it will atomically decrement the semaphore value and return 0, otherwise it will wait up to *timeout* milliseconds trying to lock the semaphore. This function is to be avoided if possible since on some platforms it is implemented by polling the semaphore every millisecond in a busy loop.

After `SDL_SemWaitTimeout()` is successful, the semaphore can be released and its count atomically incremented by a successful call to [SDL\\_SemPost](#).

## **Return Value**

Returns 0 if the semaphore was successfully locked or either `SDL_MUTEX_TIMEOUT` or -1 if the timeout period was exceeded or there was an error, respectively.

If the semaphore was not successfully locked, the semaphore will be unchanged.

## Examples

```
res = SDL_SemWaitTimeout(my_sem, WAIT_TIMEOUT_MILLISEC);

if (res == SDL_MUTEX_TIMEOUT) {
    return TRY_AGAIN;
}
if (res == -1) {
    return WAIT_ERROR;
}

...

SDL_SemPost(my_sem);
```

## See Also

[SDL\\_CreateSemaphore](#), [SDL\\_DestroySemaphore](#), [SDL\\_SemWait](#), [SDL\\_SemTryWait](#), [SDL\\_SemPost](#), [SDL\\_SemValue](#)

---

[Prev](#)

SDL\_SemTryWait

[Home](#)

[Up](#)

[Next](#)

SDL\_SemPost

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_SemPost**



## **Name**

SDL\_SemPost -- Unlock a semaphore.

## Synopsis

```
#include "SDL.h"  
#include "SDL_thread.h"
```

```
int SDL_SemPost(SDL_sem *sem);
```

## Description

SDL\_SemPost unlocks the semaphore pointed to by *sem* and atomically increments the semaphores value. Threads that were blocking on the semaphore may be scheduled after this call succeeds.

SDL\_SemPost should be called after a semaphore is locked by a successful call to [SDL\\_SemWait](#), [SDL\\_SemTryWait](#) or [SDL\\_SemWaitTimeout](#).

## **Return Value**

Returns 0 if successful or -1 if there was an error (leaving the semaphore unchanged).

# Examples

```
SDL_SemPost(my_sem);
```

## See Also

[SDL\\_CreateSemaphore](#), [SDL\\_DestroySemaphore](#), [SDL\\_SemWait](#), [SDL\\_SemTryWait](#),  
[SDL\\_SemWaitTimeout](#), [SDL\\_SemValue](#)

---

[Prev](#)

SDL\_SemWaitTimeout

[Home](#)

[Up](#)

[Next](#)

SDL\_SemValue

## SDL Library Documentation

[Prev](#)

[Next](#)

---

**SDL\_SemValue**



## **Name**

SDL\_SemValue -- Return the current value of a semaphore.

## Synopsis

```
#include "SDL.h"  
#include "SDL/SDL_thread.h"  
  
Uint32 SDL_SemValue(SDL_sem *sem);
```

## Description

`SDL_SemValue()` returns the current semaphore value from the semaphore pointed to by *sem*.

## **Return Value**

Returns current value of the semaphore.

## Examples

```
sem_value = SDL_SemValue(my_sem);
```

## See Also

[SDL\\_CreateSemaphore](#), [SDL\\_DestroySemaphore](#), [SDL\\_SemWait](#), [SDL\\_SemTryWait](#),  
[SDL\\_SemWaitTimeout](#), [SDL\\_SemPost](#)

---

[Prev](#)

SDL\_SemPost

[Home](#)

[Up](#)

[Next](#)

SDL\_CreateCond

## SDL Library Documentation

[Prev](#)

[Next](#)

---

**SDL\_CreateCond**



## **Name**

SDL\_CreateCond -- Create a condition variable

## Synopsis

```
#include "SDL.h"  
#include "SDL_thread.h"
```

```
SDL_cond *SDL_CreateCond(void);
```

## **Description**

Creates a condition variable.

## Examples

```
SDL_cond *cond;

cond=SDL_CreateCond();
.
.
/* Do stuff */

.
.
SDL_DestroyCond(cond);
```

## See Also

[SDL\\_DestroyCond](#), [SDL\\_CondWait](#), [SDL\\_CondSignal](#)

---

[Prev](#)

SDL\_SemValue

[Home](#)

[Up](#)

[Next](#)

SDL\_DestroyCond

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_DestroyCond**

## **Name**

SDL\_DestroyCond -- Destroy a condition variable



## Synopsis

```
#include "SDL.h"  
#include "SDL_thread.h"
```

```
void SDL_DestroyCond(SDL_cond *cond);
```

## **Description**

Destroys a condition variable.

## See Also

[SDL\\_CreateCond](#)

---

[Prev](#)

SDL\_CreateCond

[Home](#)

[Up](#)

[Next](#)

SDL\_CondSignal

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_CondSignal**

## **Name**

SDL\_CondSignal -- Restart a thread wait on a condition variable

## Synopsis

```
#include "SDL.h"  
#include "SDL_thread.h"
```

```
int SDL_CondSignal(SDL_cond *cond);
```

## **Description**

Restart one of the threads that are waiting on the condition variable, *cond*.  
Returns 0 on success of -1 on an error.



## See Also

[SDL\\_CondWait](#), [SDL\\_CondBroadcast](#)

---

[Prev](#)

SDL\_DestroyCond

[Home](#)

[Up](#)

[Next](#)

SDL\_CondBroadcast

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_CondBroadcast**

## **Name**

SDL\_CondBroadcast -- Restart all threads waiting on a condition variable

## Synopsis

```
#include "SDL.h"  
#include "SDL_thread.h"
```

```
int SDL_CondBroadcast(SDL_cond *cond);
```

## **Description**

Restarts all threads that are waiting on the condition variable, *cond*. Returns 0 on success, or -1 on an error.

## See Also

[SDL\\_CondSignal](#), [SDL\\_CondWait](#)

---

[Prev](#)

SDL\_CondSignal

[Home](#)

[Up](#)

[Next](#)

SDL\_CondWait

## SDL Library Documentation

[Prev](#)

[Next](#)

---



# **SDL\_CondWait**

## **Name**

SDL\_CondWait -- Wait on a condition variable

## Synopsis

```
#include "SDL.h"  
#include "SDL_thread.h"
```

```
int SDL_CondWait(SDL_cond *cond, SDL_mutex *mut);
```

## Description

Wait on the condition variable *cond* and unlock the provided mutex. The mutex must be locked before entering this function. Returns 0 when it is signalled, or -1 on an error.

## See Also

[SDL\\_CondWaitTimeout](#), [SDL\\_CondSignal](#), [SDL\\_mutexP](#)

---

[Prev](#)

SDL\_CondBroadcast

[Home](#)

[Up](#)

[Next](#)

SDL\_CondWaitTimeout

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_CondWaitTimeout**

## **Name**

SDL\_CondWaitTimeout -- Wait on a condition variable, with timeout



## Synopsis

```
#include "SDL.h"  
#include "SDL_thread.h"
```

```
int SDL_CondWaitTimeout(SDL_cond *cond, SDL_mutex *mutex, Uint32 ms);
```

## Description

Wait on the condition variable *cond* for, at most, *ms* milliseconds. *mut* is unlocked so it must be locked when the function is called. Returns `SDL_MUTEX_TIMEDOUT` if the condition is not signalled in the allotted time, 0 if it was signalled or -1 on an error.

## See Also

[SDL\\_CondWait](#)

---

[Prev](#)

SDL\_CondWait

[Home](#)

[Up](#)

[Next](#)

Time

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# Chapter 13. Time

## Table of Contents

[SDL\\_GetTicks](#) — Get the number of milliseconds since the SDL library initialization.

[SDL\\_Delay](#) — Wait a specified number of milliseconds before returning.

[SDL\\_AddTimer](#) — Add a timer which will call a callback after the specified number of milliseconds has elapsed.

[SDL\\_RemoveTimer](#) — Remove a timer which was added with [SDL\\_AddTimer](#).

[SDL\\_SetTimer](#) — Set a callback to run after the specified number of milliseconds has elapsed.

SDL provides several cross-platform functions for dealing with time. It provides a way to get the current time, a way to wait a little while, and a simple timer mechanism. These functions give you two ways of moving an object every x milliseconds:

- Use a timer callback function. This may have the bad effect that it runs in a separate thread or uses alarm signals, but it's easier to implement.
- Or you can get the number of milliseconds passed, and move the object if, for example, 30 ms passed.

---

[Prev](#)

SDL\_CondWaitTimeout

[Home](#)

[Up](#)

[Next](#)

SDL\_GetTicks

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_GetTicks**

## **Name**

SDL\_GetTicks -- Get the number of milliseconds since the SDL library initialization.



## Synopsis

```
#include "SDL.h"
```

```
Uint32 SDL_GetTicks(void);
```

## **Description**

Get the number of milliseconds since the SDL library initialization. Note that this value wraps if the program runs for more than ~49 days.

## See Also

[SDL\\_Delay](#)

---

[Prev](#)  
Time

[Home](#)  
[Up](#)

[Next](#)  
SDL\_Delay

## SDL Library Documentation

[Prev](#)

[Next](#)

---

**SDL\_Delay**

## **Name**

SDL\_Delay -- Wait a specified number of milliseconds before returning.

## Synopsis

```
#include "SDL.h"
```

```
void SDL_Delay(Uint32 ms);
```

## Description

Wait a specified number of milliseconds before returning. `SDL_Delay` will wait at *least* the specified time, but possible longer due to OS scheduling.

**Note:** Count on a delay granularity of *at least* 10 ms. Some platforms have shorter clock ticks but this is the most common.



## See Also

[SDL\\_AddTimer](#)

---

[Prev](#)

SDL\_GetTicks

[Home](#)

[Up](#)

[Next](#)

SDL\_AddTimer

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_AddTimer**

## **Name**

SDL\_AddTimer -- Add a timer which will call a callback after the specified number of milliseconds has elapsed.

## Synopsis

```
#include "SDL.h"
```

```
SDL_TimerID SDL_AddTimer(Uint32 interval, SDL_NewTimerCallback callback,  
void *param);
```

## Callback

```
/* type definition for the "new" timer callback function */  
typedef Uint32 (*SDL_NewTimerCallback)(Uint32 interval, void *param)
```

## Description

Adds a callback function to be run after the specified number of milliseconds has elapsed. The callback function is passed the current timer interval and the user supplied parameter from the `SDL_AddTimer` call and returns the next timer interval. If the returned value from the callback is the same as the one passed in, the periodic alarm continues, otherwise a new alarm is scheduled.

To cancel a currently running timer call [SDL\\_RemoveTimer](#) with the timer ID returned from `SDL_AddTimer`.

The timer callback function may run in a different thread than your main program, and so shouldn't call any functions from within itself. You may always call [SDL\\_PushEvent](#), however.

The granularity of the timer is platform-dependent, but you should count on it being at least 10 ms as this is the most common number. This means that if you request a 16 ms timer, your callback will run approximately 20 ms later on an unloaded system. If you wanted to set a flag signaling a frame update at 30 frames per second (every 33 ms), you might set a timer for 30 ms (see example below). If you use this function, you need to pass `SDL_INIT_TIMER` to [SDL\\_Init](#).

## **Return Value**

Returns an ID value for the added timer or NULL if there was an error.



## Examples

```
my_timer_id = SDL_AddTimer((33/10)*10, my_callbackfunc, my_callback_
```

## See Also

[SDL\\_RemoveTimer](#), [SDL\\_PushEvent](#)

---

[Prev](#)

SDL\_Delay

[Home](#)

[Up](#)

[Next](#)

SDL\_RemoveTimer

## SDL Library Documentation

[Prev](#)

[Next](#)

---

# **SDL\_RemoveTimer**

## **Name**

SDL\_RemoveTimer -- Remove a timer which was added with [SDL\\_AddTimer](#).

## Synopsis

```
#include "SDL.h"
```

```
SDL_bool SDL_RemoveTimer(SDL_TimerID id);
```

## **Description**

Removes a timer callback previously added with [SDL\\_AddTimer](#).

## **Return Value**

Returns a boolean value indicating success.



## Examples

```
SDL_RemoveTimer(my_timer_id);
```

## See Also

[SDL\\_AddTimer](#)

---

[Prev](#)

SDL\_AddTimer

[Home](#)

[Up](#)

[Next](#)

SDL\_SetTimer

# SDL Library Documentation

[Prev](#)

---

# SDL\_SetTimer

## **Name**

SDL\_SetTimer -- Set a callback to run after the specified number of milliseconds has elapsed.

## Synopsis

```
#include "SDL.h"
```

```
int SDL_SetTimer(Uint32 interval, SDL_TimerCallback callback);
```

## Callback

```
/* Function prototype for the timer callback function */ typedef Uint32  
(*SDL_TimerCallback)(Uint32 interval);
```

## Description

Set a callback to run after the specified number of milliseconds has elapsed. The callback function is passed the current timer interval and returns the next timer interval. If the returned value is the same as the one passed in, the periodic alarm continues, otherwise a new alarm is scheduled.

To cancel a currently running timer, call `SDL_SetTimer(0, NULL);`

The timer callback function may run in a different thread than your main constant, and so shouldn't call any functions from within itself.

The maximum resolution of this timer is 10 ms, which means that if you request a 16 ms timer, your callback will run approximately 20 ms later on an unloaded system. If you wanted to set a flag signaling a frame update at 30 frames per second (every 33 ms), you might set a timer for 30 ms (see example below).

If you use this function, you need to pass `SDL_INIT_TIMER` to `SDL_Init()`.

**Note:** This function is kept for compatibility but has been superseded by the new timer functions [SDL\\_AddTimer](#) and [SDL\\_RemoveTimer](#) which support multiple timers.



## Examples

```
SDL_SetTimer((33/10)*10, my_callback);
```

## See Also

[SDL\\_AddTimer](#)

---

[Prev](#)

SDL\_RemoveTimer

[Home](#)

[Up](#)