pycsdl2 Documentation

Release 2.0.0.0.dev5

Paul Tan

December 14, 2016

pycsdl2 is a low-overhead, compiled SDL2 binding for CPython. It aims to provide the familiar C API of SDL2, while offering several Pythonic features such as automatic memory management, bounds checking and exceptions.

This is the documentation for pycsdl2 2.0.0.0.dev5, last generated December 14, 2016.

API Reference

1.1 Initialization and Shutdown

1.1.1 Subsystem Flags

csdl2.**SDL_INIT_TIMER**
Timer subsystem.

csdl2.**SDL_INIT_AUDIO**
Audio subsystem.

csdl2.**SDL_INIT_VIDEO**
Video subsystem. (Implies *SDL_INIT_EVENTS*)

csdl2.**SDL_INIT_JOYSTICK**
Joystick subsystem. (Implies *SDL_INIT_EVENTS*)

csdl2.**SDL_INIT_HAPTIC**
Haptic (force feedback) subsystem.

csdl2.**SDL_INIT_GAMECONTROLLER**
Controller subsystem. (Implies *SDL_INIT_JOYSTICK*)

csdl2.**SDL_INIT_EVENTS**
Events subsystem.

csdl2.**SDL_INIT_EVERYTHING**
Initialize all subsystems.

csdl2.**SDL_INIT_NOPARACHUTE**
This flag is provided for compatibility and is ignored.

1.1.2 Initialization

csdl2.**SDL_Init** (*flags*)
Initializes the SDL library. This must be called before using any other SDL function.

Parameters flags (*int*) – *Subsystem Flags* of subsystems to initialize, OR'd together.

csdl2.**SDL_InitSubSystem** (*flags*)
Initialize specific subsystems.

Subsystem initialization is ref-counted. You must call *SDL_QuitSubSystem()* for each *SDL_InitSubSystem()* to correctly shutdown a subsystem manually (or call *SDL_Quit()* to force a shutdown). If a subsystem is already loaded then this call will increase the refcount and return.

Parameters flags (*int*) – *Subsystem Flags* of subsystems to initialize, OR'd together.

csdl2.**SDL_WasInit**(*flags*) → int
Return a mask of the specified subsystems which have previously been initialized.

Parameters **flags** (*int*) – *Subsystem Flags* of subsystems to query, OR'd together.

Returns The initialization status of the specified subsystems, or a mask of all initialized subsystems if *flags* is 0.

1.1.3 Shutdown

csdl2.**SDL_QuitSubSystem**(*flags*)
Shut down specific subsystems.

Subsystem initialization is ref-counted. *SDL_QuitSubSystem()* will decrement the refcount for each of the specified subsystems, and if the refcount of a subsystem reached 0 that subsystem is shut down.

Parameters **flags** (*int*) – *Subsystem Flags* of subsystems to shut down, OR'd together.

Note: If you start a subsystem using a call to that subsystem's init function (e.g. `SDL_VideoInit()`) instead of *SDL_Init()* or *SDL_InitSubSystem()*, *SDL_QuitSubSystem()* will not work. You will need to use that subsystem's quit function (e.g. `SDL_VideoQuit()`) directly instead.

Note: You still need to call *SDL_Quit()* even if you close all open subsystems with *SDL_QuitSubSystem()*.

csdl2.**SDL_Quit**()
Clean up all initialized subsystems. This function should be called upon all exit conditions.

Note: This function should be called *even* if all initialized subsystems have been shut down with *SDL_QuitSubSystem()*.

Note: It is safe to call this function even in the case of errors in initialization.

Note: If a subsystem is started using a call to that subsystem's init function (e.g. `SDL_VideoInit()`) instead of *SDL_Init()* or *SDL_InitSubSystem()*, then the subsystem's quit function (e.g. `SDL_VideoQuit()`) must be called to shut the subsystem down before calling *SDL_Quit()*.

1.2 Display and Window Management

class csdl2.**SDL_Window**
A window.

It cannot be initialized directly. Instead, create one with *SDL_CreateWindow()*.

1.2.1 Window creation

csdl2.**SDL_CreateWindow**(*title: str*, *x: int*, *y: int*, *w: int*, *h: int*, *flags: int*) → SDL_Window
Creates a window with the specified title, position, dimensions and flags.

Parameters

- **title** (*str*) – Title of the window
- **x** (*int*) – X position of the window, *SDL_WINDOWPOS_CENTERED* or *SDL_WINDOWPOS_UNDEFINED*.
- **y** (*int*) – Y position of the window, *SDL_WINDOWPOS_CENTERED* or *SDL_WINDOWPOS_UNDEFINED*.
- **w** (*int*) – Width of the window.
- **h** (*int*) – Height of the window.
- **flags** (*int*) – 0, or one or more of the following flags OR'ed together: *SDL_WINDOW_FULLSCREEN*, *SDL_WINDOW_FULLSCREEN_DESKTOP*, *SDL_WINDOW_OPENGL*, *SDL_WINDOW_SHOWN*, *SDL_WINDOW_HIDDEN*, *SDL_WINDOW_BORDERLESS*, *SDL_WINDOW_RESIZABLE*, *SDL_WINDOW_MINIMIZED*, *SDL_WINDOW_MAXIMIZED*, *SDL_WINDOW_INPUT_GRABBED*.

Returns A new *SDL_Window*

csdl2.**SDL_WINDOWPOS_UNDEFINED**
Used to indicate that you don't care what the window position is in any display.

csdl2.**SDL_WINDOWPOS_CENTERED**
Used to indicate that the window position should be centered in any display.

1.2.2 Window flags

csdl2.**SDL_WINDOW_FULLSCREEN**
The window is fullscreen.

csdl2.**SDL_WINDOW_OPENGL**
The window is usable with an OpenGL context.

csdl2.**SDL_WINDOW_SHOWN**
The window is visible.

csdl2.**SDL_WINDOW_HIDDEN**
The window is hidden.

csdl2.**SDL_WINDOW_BORDERLESS**
The window has no window decoration.

csdl2.**SDL_WINDOW_RESIZABLE**
The window is resizable.

csdl2.**SDL_WINDOW_MINIMIZED**
The window is minimized.

csdl2.**SDL_WINDOW_MAXIMIZED**
The window is maximized.

csdl2.**SDL_WINDOW_INPUT_GRABBED**
The window has grabbed input focus.

csdl2.**SDL_WINDOW_INPUT_FOCUS**
The window has input focus.

csdl2.**SDL_WINDOW_MOUSE_FOCUS**
The window has mouse focus.

csdl2.**SDL_WINDOW_FULLSCREEN_DESKTOP**
The window is exclusively fullscreen – the screen display mode is set to match the window dimensions.

csdl2.**SDL_WINDOW_FOREIGN**
The window was not created by SDL.

1.2.3 Window destruction

csdl2.**SDL_DestroyWindow**(*window: SDL_Window*) → None
Destroys a window, freeing up its resources.

There is no need to manually call this function. *SDL_Window* will automatically call it as part of its destructor.

Parameters **window** (SDL_Window) – Window to be destroyed.

1.2.4 Window Properties

csdl2.**SDL_GetWindowTitle**(*window: SDL_Window*) → str
Returns the title of the window.

Parameters **window** (*SDL_Window*) – The window.

Returns The title of the window.

1.2.5 Window Events

csdl2.**SDL_WINDOWEVENT_NONE**
This constant is not used.

csdl2.**SDL_WINDOWEVENT_SHOWN**
The value of SDL_WindowEvent.type when the Window has been shown.

csdl2.**SDL_WINDOWEVENT_HIDDEN**
The value of SDL_WindowEvent.type when the window has been hidden.

csdl2.**SDL_WINDOWEVENT_EXPOSED**
The value of SDL_WindowEvent.type when the Window has been exposed and should been redrawn.

csdl2.**SDL_WINDOWEVENT_MOVED**
The value of SDL_WindowEvent.type when the window has been moved. SDL_WindowEvent.data1 is the new x position and SDL_WindowEvent.data2 is the new y position.

csdl2.**SDL_WINDOWEVENT_RESIZED**
The value of SDL_WindowEvent.type when the window has been resized. SDL_WindowEvent.data1 is the new width and SDL_WindowEvent.data2 is the new height.

csdl2.**SDL_WINDOWEVENT_SIZE_CHANGED**
The value of SDL_WindowEvent.type when the window size has changed, either as a result of an API call or through the system or user changing the window size. SDL_WindowEvent.data1 is the new width and SDL_WindowEvent.data2 is the new height.

csdl2.**SDL_WINDOWEVENT_MINIMIZED**
The value of SDL_WindowEvent.type when the window has been minimized.

csdl2.**SDL_WINDOWEVENT_MAXIMIZED**
The value of SDL_WindowEvent.type when the window has been maximized.

csdl2.**SDL_WINDOWEVENT_RESTORED**
The value of SDL_WindowEvent.type when the window has been restored to normal size and position.

csdl2.**SDL_WINDOWEVENT_ENTER**
The value of SDL_WindowEvent.type when the window has gained mouse focus.

csdl2.**SDL_WINDOWEVENT_LEAVE**
The value of SDL_WindowEvent.type when the window has lost mouse focus.

csdl2.**SDL_WINDOWEVENT_FOCUS_GAINED**
The value of SDL_WindowEvent.type when the window has gained keyboard focus.

csdl2.**SDL_WINDOWEVENT_FOCUS_LOST**
The value of SDL_WindowEvent.type when the window has lost keyboard focus.

csdl2.**SDL_WINDOWEVENT_CLOSE**
The value of SDL_WindowEvent.type when the window manager requests that the window be closed.

1.2.6 OpenGL configuration attributes

OpenGL configuration attributes control the properties of the OpenGL context that is created with SDL_GL_CreateContext. These attributes are set with SDL_GL_SetAttribute and read with SDL_GL_GetAttribute.

Note that the following attributes must be set *before* the window is created with *SDL_CreateWindow()*:

- *SDL_GL_RED_SIZE*
- *SDL_GL_GREEN_SIZE*
- *SDL_GL_BLUE_SIZE*
- *SDL_GL_ALPHA_SIZE*
- *SDL_GL_DOUBLEBUFFER*

csdl2.**SDL_GL_RED_SIZE**
OpenGL configuration attribute for the minimum number of bits for the red channel of the color buffer. Defaults to 3.

csdl2.**SDL_GL_GREEN_SIZE**
OpenGL configuration attribute for the minimum number of bits for the green channel of the color buffer. Defaults to 3.

csdl2.**SDL_GL_BLUE_SIZE**
OpenGL configuration attribute for the minimum number of bits for the blue channel of the color buffer. Defaults to 2.

csdl2.**SDL_GL_ALPHA_SIZE**
OpenGL configuration attribute for the minimum number of bits for the alpha channel of the color buffer. Defaults to 0.

csdl2.**SDL_GL_BUFFER_SIZE**
OpenGL configuration attribute for the minimum number of bits for frame buffer size. Defaults to 0.

csdl2.**SDL_GL_DOUBLEBUFFER**
OpenGL configuration attribute for whether the output is single or double buffered. Defaults to double buffering on.

csdl2.**SDL_GL_DEPTH_SIZE**
OpenGL configuration attribute for the minimum number of bits in the depth buffer. Defaults to 16.

csdl2.**SDL_GL_STENCIL_SIZE**
OpenGL configuration attribute for the minimum number of bits in the stencil buffer. Defaults to 0.

csdl2.**SDL_GL_ACCUM_RED_SIZE**
OpenGL configuration attribute for the minimum number of bits for the red channel of the accumulation buffer. Defaults to 0.

csdl2.**SDL_GL_ACCUM_GREEN_SIZE**
OpenGL configuration attribute for the minimum number of bits for the green channel of the accumulation buffer. Defaults to 0.

csdl2.**SDL_GL_ACCUM_BLUE_SIZE**
OpenGL configuration attribute for the the minimum number of bits for the blue channel of the accumulation buffer. Defaults to 0.

csdl2.**SDL_GL_ACCUM_ALPHA_SIZE**
OpenGL configuration attribute for the minimum number of bits for the alpha channel of the accumulation buffer. Defaults to 0.

csdl2.**SDL_GL_STEREO**
OpenGL configuration attribute for whether the output is stereo 3D. Defaults to off.

csdl2.**SDL_GL_MULTISAMPLEBUFFERS**
OpenGL configuration attribute for the number of buffers used for multisample anti-aliasing. Defaults to 0.

csdl2.**SDL_GL_MULTISAMPLESAMPLES**
OpenGL configuration attribute for the number of samples used around the current pixel used for multisample anti-aliasing. Defaults to 0.

csdl2.**SDL_GL_ACCELERATED_VISUAL**
Set this OpenGL configuration attribute to 1 to require hardware acceleration, set to 0 to force software rendering. Default is to allow either.

csdl2.**SDL_GL_CONTEXT_MAJOR_VERSION**
OpenGL configuration attribute for the OpenGL context major version.

csdl2.**SDL_GL_CONTEXT_MINOR_VERSION**
OpenGL configuration attribute for the OpenGL context minor version.

csdl2.**SDL_GL_CONTEXT_FLAGS**
OpenGL context creation flags. The value can be one or more of *SDL_GL_CONTEXT_DEBUG_FLAG*, *SDL_GL_CONTEXT_FORWARD_COMPATIBLE_FLAG*, *SDL_GL_CONTEXT_ROBUST_ACCESS_FLAG*, *SDL_GL_CONTEXT_RESET_ISOLATION_FLAG*. Default is 0 (no flags set).

csdl2.**SDL_GL_CONTEXT_PROFILE_MASK**
OpenGL context creation profile. The value must be one of *SDL_GL_CONTEXT_PROFILE_CORE*, *SDL_GL_CONTEXT_PROFILE_COMPATIBILITY*, *SDL_GL_CONTEXT_PROFILE_ES*. Default depends on the platform.

csdl2.**SDL_GL_SHARE_WITH_CURRENT_CONTEXT**
OpenGL configuration attribute to enable context sharing. Default is 0 (don't share contexts).

1.2.7 OpenGL flags

These flags are set through the *SDL_GL_CONTEXT_FLAGS* OpenGL configuration attribute.

csdl2.**SDL_GL_CONTEXT_DEBUG_FLAG**
This flag maps to `GLX_CONTEXT_DEBUG_BIT_ARB` in the GLX_ARB_create_context extension and `WGL_CONTEXT_DEBUG_BIT_ARB` in the WGL_ARB_create_context extension, and is ignored if these extensions are not available. This flag puts OpenGL into a "debug" mode which might assist with debugging, possibly at a loss of performance.

csdl2.**SDL_GL_CONTEXT_FORWARD_COMPATIBLE_FLAG**
This flag maps to `GLX_CONTEXT_FORWARD_COMPATIBLE_BIT_ARB` in the GLX_ARB_create_context extension and `WGL_CONTEXT_FORWARD_COMPATIBLE_BIT_ARB` in the WGL_ARB_create_context extension, and is ignored if these extensions are not available. This flag puts OpenGL into a "forward compatible" mode, where no deprecated functionality will be supported, possibly at a gain in performance. This only applies to OpenGL 3.0 and later contexts.

csdl2.**SDL_GL_CONTEXT_ROBUST_ACCESS_FLAG**
This flag maps to `GLX_CONTEXT_ROBUST_ACCESS_BIT_ARB` in the GLX_ARB_create_context_robustness extenstion and `WGL_CONTEXT_ROBUST_ACCESS_BIT_ARB` in the WGL_ARB_create_context_robustness extension, and is ignored if these extensions are not available. This flag creates an OpenGL context that supports the GL_ARB_robustness extension – a mode that offers a few APIs that are safer than the usual defaults.

csdl2.**SDL_GL_CONTEXT_RESET_ISOLATION_FLAG**
This flag maps to `GLX_CONTEXT_RESET_ISOLATION_BIT_ARB` in the

GLX_ARB_robustness_isolation extension and WGL_CONTEXT_RESET_ISOLATION_BIT_ARB in the WGL_ARB_create_context_robustness extension, and is ignored if these extensions are not available. This flag is intended to require OpenGL to make promises about what to do in the event of driver or hardware failure.

1.2.8 OpenGL Profiles

These profile constants are used with SDL_GL_SetAttribute and *SDL_GL_CONTEXT_PROFILE_MASK*. Note that these profiles are mutually exclusive and SDL_GL_SetAttribute accepts at most one of them. Setting *SDL_GL_CONTEXT_PROFILE_MASK* to 0 leaves the choice of profile up to SDL. Should be used in conjunction with *SDL_GL_CONTEXT_MAJOR_VERSION* and *SDL_GL_CONTEXT_MINOR_VERSION* as OpenGL profiles are defined relative to a particular version of OpenGL. There is no way to distinguish between the common and common lite profiles of OpenGL ES versions 1.0 and 1.1.

csdl2.**SDL_GL_CONTEXT_PROFILE_CORE**
Core profile. Deprecated functions are disabled.

csdl2.**SDL_GL_CONTEXT_PROFILE_COMPATIBILITY**
Compatibility profile. Deprecated functions are allowed.

csdl2.**SDL_GL_CONTEXT_PROFILE_ES**
OpenGL ES context. Only a subset of base OpenGL functionality is allowed.

1.3 Blend modes

The following constants are used in *SDL_SetTextureBlendMode()*, SDL_SetSurfaceBlendMode() and other drawing operations.

csdl2.**SDL_BLENDMODE_NONE**
No blending.

`dstRGBA = srcRGBA.`

csdl2.**SDL_BLENDMODE_BLEND**
Alpha blending.

`dstRGB = (srcRGB * srcA) + (dstRGB * (1 - srcA))`

`dstA = srcA + (dstA * (1 - srcA))`

csdl2.**SDL_BLENDMODE_ADD**
Additive blending.

`dstRGB = (srcRGB * srcA) + dstRGB`

`dstA = dstA`

csdl2.**SDL_BLENDMODE_MOD**
Color modulate.

`dstRGB = srcRGB * dstRGB`

`dstA = dstA`

1.4 Surface Creation and Simple Drawing

class csdl2.**SDL_Surface**
A structure that contains a collection of pixels used in software blitting.

This structure cannot be initiated directly. Use *SDL_CreateRGBSurface()*, *SDL_CreateRGBSurfaceFrom()*, *SDL_LoadBMP_RW()* or *SDL_LoadBMP()* to create a new instance.

flags
(readonly) A bitmask of the flags *SDL_PREALLOC*, *SDL_RLEACCEL* and/or *SDL_DONTFREE* for internal use.

format
(readonly) *SDL_PixelFormat* of the pixels stored in the surface.

w
(readonly) Width of the surface in pixels.

h
(readonly) Height of the surface in pixels.

pitch
(readonly) The length of a row of pixels in bytes.

pixels
(readonly) Buffer providing the actual pixel data.

userdata
An arbitrary object that an application can set for its own use.

locked
(readonly) True if the surface is locked.

clip_rect
(readonly) An *SDL_Rect* structure used to clip bits to the surface which can be set by `SDL_SetClipRect()`.

refcount
(readonly) SDL's reference count of the surface. For internal use.

csdl2.**SDL_PREALLOC**
Surface uses preallocated memory.

csdl2.**SDL_RLEACCEL**
Surface is RLE encoded.

csdl2.**SDL_DONTFREE**
Surface is referenced internally.

csdl2.**SDL_MUSTLOCK**(*surface: SDL_Surface*) → bool
Returns True if *surface* needs to be locked before its *SDL_Surface.pixels* can be accessed.

Parameters **surface** (`SDL_Surface`) – The surface to test

Returns True if the surface needs to be locked before its pixels can be accessed, False otherwise.

csdl2.**SDL_CreateRGBSurface**(*flags: int*, *width: int*, *height: int*, *depth: int*, *Rmask: int*, *Gmask: int*, *Bmask: int*, *Amask: int*) → SDL_Surface
Creates and returns a new blank *SDL_Surface* with the specified properties.

Parameters

- **flags** (*int*) – This argument is unused and should be set to 0.
- **width** (*int*) – The width of the surface in pixels.
- **height** (*int*) – The height of the surface in pixels.
- **depth** (*int*) – The depth of the surface in bits. If *depth* is 4 or 8 bits, an empty *SDL_Palette* is allocated for the surface. If *depth* is greater than 8 bits, the pixel format is set using the *Rmask*, *Gmask*, *Bmask* and *Amask* arguments.
- **Rmask** (*int*) – Bitmask used to extract the red component from a pixel. If 0, a default mask based on the depth is used.
- **Gmask** (*int*) – Bitmask used to extract the green component from a pixel. If 0, a default mask based on the depth is used.

- **Bmask** (*int*) – Bitmask used to extract the blue component from a pixel. If 0, a default mask based on the depth is used.
- **Amask** (*int*) – Bitmask used to extract the alpha component from a pixel. If 0, the surface has no alpha channel.

Returns A new blank *SDL_Surface* structure.

csdl2.**SDL_CreateRGBSurfaceFrom**(*pixels: buffer*, *width: int*, *height: int*, *depth: int*, *pitch: int*, *Rmask: int*, *Gmask: int*, *Bmask: int*, *Amask: int*) → SDL_Surface

Creates and returns a *SDL_Surface* with existing pixel data.

Parameters

- **pixels** (*buffer*) – Existing pixel data. This can be any object that supports the buffer protocol and exports a C-contiguous buffer of the correct size.
- **width** (*int*) – The width of the surface in pixels.
- **height** (*int*) – The height of the surface in pixels.
- **depth** (*int*) – The depth of the surface in bits. If *depth* is 4 or 8 bits, an empty *SDL_Palette* is allocated for the surface. If *depth* is greater than 8 bits, the pixel format is set using the *Rmask*, *Gmask*, *Bmask* and *Amask* arguments.
- **Rmask** (*int*) – Bitmask used to extract the red component from a pixel. If 0, a default mask based on the depth is used.
- **Gmask** (*int*) – Bitmask used to extract the green component from a pixel. If 0, a default mask based on the depth is used.
- **Bmask** (*int*) – Bitmask used to extract the blue component from a pixel. If 0, a default mask based on the depth is used.
- **Amask** (*int*) – Bitmask used to extract the alpha component from a pixel. If 0, the surface has no alpha channel.

Returns A *SDL_Surface* with its contents backed by the provided *pixels* buffer.

csdl2.**SDL_LoadBMP_RW**(*src*, *freesrc*) → SDL_Surface

Load a BMP image from a seekable SDL data stream. (memory or file).

Parameters

- **src** (*SDL_RWops*) – The data stream for the surface.
- **freesrc** (*bool*) – True to close the stream after being read.

Returns *SDL_Surface* with the image data.

csdl2.**SDL_LoadBMP**(*file*) → SDL_Surface

Load a surface from a BMP file on the filesystem.

Parameters **file** (*str*) – The path to the file containing a BMP image.

Returns *SDL_Surface* with the image data.

csdl2.**SDL_FreeSurface**(*surface: SDL_Surface*)

Frees the surface.

There is no need to manually call this function. *SDL_Surface* will automatically call this function as part of its destructor.

Parameters **surface** (SDL_Surface) – surface to free

1.5 2D Accelerated Rendering

1.5.1 Render drivers

A render driver is a set of code that handles rendering and texture management on a particular display.

class `csdl2.`**`SDL_RendererInfo`**(*name=None*, *flags=0*, *num_texture_formats=0*, *texture_formats=0*, *max_texture_width=0*, *max_texture_height=0*)

Information on the capabilities of a render driver or context.

`name`
Name of the renderer.

`flags`
A mask of supported *Renderer creation flags*.

`num_texture_formats`
The number of available texture formats.

`texture_formats`
The available texture formats as an array of *Pixel format constants* ints.

Note that the size of the array is always 16. However, only the first `num_texture_formats` values are valid.

`max_texture_width`
Maximum texture width.

`max_texture_height`
Maximum texture height

`csdl2.`**`SDL_GetNumRenderDrivers`**() → int

Get the number of 2D rendering drivers available for the current display.

`csdl2.`**`SDL_GetRenderDriverInfo`**(*index*) → SDL_RendererInfo

Gets information about a specific 2D rendering driver for the current display.

Parameters **`index`** (*`int`*) – The index of the driver to query information about. It must be in the range 0 to `SDL_GetNumRenderDrivers() - 1`.

Returns A new *`SDL_RendererInfo`* filled with information about the render driver.

1.5.2 Renderers

class `csdl2.`**`SDL_Renderer`**

A 2d rendering context.

This is an opaque handle that cannot be directly constructed. Instead, use *`SDL_CreateRenderer()`* or *`SDL_CreateSoftwareRenderer()`*.

`csdl2.`**`SDL_CreateWindowAndRenderer`**(*width*, *height*, *window_flags*) → tuple

Creates a window and a default renderer.

Parameters

- **`width`** (*`int`*) – The width of the window.
- **`height`** (*`int`*) – The height of the window.
- **`window_flags`** (*`int`*) – 0, or one or more of the *Window flags* OR'd together.

Returns A 2-tuple `(window, renderer)`, where *window* is the created *`SDL_Window`* and *renderer* is the created *`SDL_Renderer`*.

`csdl2.`**`SDL_CreateRenderer`**(*window: SDL_Window*, *index: int*, *flags: int*) → SDL_Renderer

Creates a *`SDL_Renderer`* for *window*.

Parameters

- **window** (SDL_Window) – *SDL_Window* to render to.
- **index** (*int*) – The index of the rendering driver to initialize, or -1 to initialize the first driver supporting *flags*.
- **flags** (*int*) – 0, or one or more *Renderer creation flags* OR'ed together.

Returns A new *SDL_Renderer* that renders to *window*.

csdl2.**SDL_CreateSoftwareRenderer**(*surface: SDL_Surface*) → SDL_Renderer
Creates a *SDL_Renderer* for *surface*.

Parameters **surface** (SDL_Surface) – *SDL_Surface* to render to.

Returns A new *SDL_Renderer* that renders to *surface*.

csdl2.**SDL_GetRenderer**(*window*) → SDL_Renderer
Returns the renderer associated with a window.

Parameters **window** (*SDL_Renderer*) – The window to query.

Returns The *SDL_Renderer* associated with the window, or None if there is no renderer associated with the window.

csdl2.**SDL_GetRendererInfo**(*renderer*) → SDL_RendererInfo
Get information about a rendering context.

Parameters **renderer** (*SDL_Renderer*) – The rendering context to query.

Returns A new *SDL_RendererInfo* filled with information about the renderer.

csdl2.**SDL_GetRendererOutputSize**(*renderer*) → tuple
Get the output size of a rendering context.

Parameters **renderer** (*SDL_Renderer*) – The rendering context to query.

Returns A 2-tuple `(width, height)` with the output width and height of the rendering context respectively.

csdl2.**SDL_DestroyRenderer**(*renderer: SDL_Renderer*) → None
Destroys *renderer*, freeing up its associated textures and resources.

There is no need to manually call this function. *SDL_Renderer* will automatically call this function as part of its destructor.

Parameters **renderer** (SDL_Renderer) – *SDL_Renderer* to destroy

1.5.3 Renderer creation flags

These flags can be passed to *SDL_CreateRenderer()* to request that the renderer support certain functions.

csdl2.**SDL_RENDERER_SOFTWARE**
The renderer is a software fallback.

csdl2.**SDL_RENDERER_ACCELERATED**
The renderer uses hardware acceleration.

csdl2.**SDL_RENDERER_PRESENTVSYNC**
SDL_RenderPresent() is synchronized with the refresh rate.

csdl2.**SDL_RENDERER_TARGETTEXTURE**
The renderer supports rendering to texture.

1.5.4 Textures

class `csdl2.`**`SDL_Texture`**

An efficient driver-specific representation of pixel data.

This is an opaque handle that cannot be directly constructed. Instead, use *`SDL_CreateTexture()`* or *`SDL_CreateTextureFromSurface()`*.

`csdl2.`**`SDL_CreateTexture`**(*renderer*, *format*, *access*, *w*, *h*) → SDL_Texture

Creates a texture for a rendering context with the specified properties.

Parameters

- **`renderer`** (*`SDL_Renderer`*) – The rendering context.
- **`format`** (*`int`*) – The texture pixel format. One of the *Pixel format constants*.
- **`access`** (*`int`*) – Specifies whether the texture data can be modified. One of *`SDL_TEXTUREACCESS_STATIC`*, *`SDL_TEXTUREACCESS_STREAMING`* or *`SDL_TEXTUREACCESS_TARGET`*.
- **`w`** (*`int`*) – Width of the texture in pixels.
- **`h`** (*`int`*) – Height of the texture in pixels.

Returns A new *`SDL_Texture`* for the rendering context.

`csdl2.`**`SDL_TEXTUREACCESS_STATIC`**

Texture changes rarely, not lockable.

`csdl2.`**`SDL_TEXTUREACCESS_STREAMING`**

Texture changes frequently, lockable.

`csdl2.`**`SDL_TEXTUREACCESS_TARGET`**

Texture can be used as a render target.

`csdl2.`**`SDL_CreateTextureFromSurface`**(*renderer*, *surface*) → SDL_Texture

Creates a texture for a rendering context with the pixel data of an existing surface.

The surface is not modified or freed by this function. The texture will be created with *`SDL_TEXTUREACCESS_STATIC`*.

Parameters

- **`renderer`** (*`SDL_Renderer`*) – The rendering context.
- **`surface`** (class:*SDL_Surface*) – The surface containing pixel data to fill the texture.

Returns A new *`SDL_Texture`* for the rendering context.

`csdl2.`**`SDL_QueryTexture`**(*texture*) → tuple

Query the attributes of a texture. Namely:

- The texture's raw pixel format, one of the *Pixel format constants*.
- The texture's access. One of *`SDL_TEXTUREACCESS_STATIC`*, *`SDL_TEXTUREACCESS_STREAMING`* or *`SDL_TEXTUREACCESS_TARGET`*.
- The texture's width and height, in pixels.

Parameters **`texture`** (*`SDL_Texture`*) – The texture to be queried.

Returns A tuple `(int, int, int, int)` with the texture's raw pixel format, access, width and height respectively.

`csdl2.`**`SDL_SetTextureColorMod`**(*texture*, *r*, *g*, *b*)

Sets an additional color value used in render copy operations.

When the texture is rendered, during the copy operation each source color channel is modulated by the appropriate color value according to the following formula:

```
srcC = srcC * (color / 255)
```

Parameters

- **texture** (*SDL_Texture*) – The texture to update.
- **r** (*int*) – The red color value multiplied into copy operations.
- **g** (*int*) – The green color value multiplied into copy operations.
- **b** (*int*) – The blue color value multiplied into copy operations.

csdl2.**SDL_GetTextureColorMod**(*texture*) → tuple

Returns the additional color value multiplied into render copy operations.

Parameters **texture** (*SDL_Texture*) – The texture to query.

Returns A tuple `(int, int, int)` with the red, green and blue components of the color respectively.

csdl2.**SDL_SetTextureAlphaMod**(*texture*, *alpha*)

Sets an additional alpha value multiplied into render copy operations.

When the texture is rendered, during the copy operation the source alpha value would be modulated by this alpha value according to the following formula:

```
srcA = srcA * (alpha / 255)
```

Parameters

- **texture** (*SDL_Texture*) – The texture to update.
- **alpha** (*int*) – The source alpha value multiplied into copy operations. It must be within the range 0-255.

csdl2.**SDL_GetTextureAlphaMod**(*texture*) → int

Returns the additional alpha value multiplied into render copy operations.

Parameters **texture** (*SDL_Texture*) – The texture to query.

Returns The current alpha value. It is within the range 0-255.

csdl2.**SDL_SetTextureBlendMode**(*texture*, *blendMode: int*)

Sets the blend mode for a texture.

Parameters

- **texture** (*SDL_Texture*) – The texture to update.
- **blendMode** (*int*) – The blend mode to use for texture blending. One of the *Blend modes*.

csdl2.**SDL_GetTextureBlendMode**(*texture*) → int

Returns the blend mode used for texture copy operations.

Parameters **texture** (*SDL_Texture*) – The texture to query.

Returns The texture's blend mode. One of the *Blend modes*.

csdl2.**SDL_UpdateTexture**(*texture*, *rect*, *pixels*, *pitch*)

Updates the given texture rectangle with new pixel data.

Parameters

- **texture** (*SDL_Texture*) – The texture to update.
- **rect** (*SDL_Rect* buffer, or None) – The area to update, or None to update the entire texture.
- **pixels** (*buffer*) – The raw pixel data.

- **pitch** (*int*) – The number of bytes in a row of pixel data, including padding between lines.

Note: This is a fairly slow function, intended for use with static textures that do not change often. If the texture is intended to be updated often, it is preferred to create the texture as streaming and use the locking functions `SDL_LockTexture()` and `SDL_UnlockTexture()`. While this function will work with streaming textures, for optimization reasons you may not get the pixels back if you lock the texture afterward.

`csdl2.`**`SDL_LockTexture`**`(`*texture*, *rect*`)` → tuple

Locks a portion of the texture for write-only pixel access.

Parameters

- **texture** (*SDL_Texture*) – The texture to lock for access, which was created with *SDL_TEXTUREACCESS_STREAMING*.
- **rect** (*SDL_Rect* buffer or None) – The area to lock for access, or None to lock the entire texture.

Returns A tuple `(pixels, pitch)`. *pixels* is a buffer containing the locked pixels, and *pitch* is the integer length of one row in bytes.

Note: After modifying the pixels, you must use `SDL_UnlockTexture()` to unlock the pixels and apply any changes.

Note: This is a write-only operation. As an optimization, the pixels made available for editing don't necessarily contain the old texture data.

`csdl2.`**`SDL_UnlockTexture`**`(`*texture*`)`

Unlocks a texture, uploading any changes to video memory.

Parameters **texture** (*SDL_Texture*) – A texture locked by `SDL_LockTexture()`.

Note: The pixels buffer returned by `SDL_LockTexture()` may contain junk data. For consistent results, ensure that you have overwritten the pixel buffer fully before calling this function.

`csdl2.`**`SDL_DestroyTexture`**`(`*texture*`)`

Destroys the specified texture, freeing its resources.

There is no need to explictly call this function. *SDL_Texture* will automatically call it upon cleanup.

Parameters **texture** (*SDL_Texture*) – Texture to destroy.

1.5.5 Render targets

`csdl2.`**`SDL_RenderTargetSupported`**`(`*renderer*`)` → bool

Queries whether a renderer supports the use of render targets.

Parameters **renderer** (*SDL_Renderer*) – The rendering context.

Returns True if render targets are supported, False if not.

`csdl2.`**`SDL_SetRenderTarget`**`(`*renderer*, *texture*`)`

Sets a texture as the current rendering target.

Parameters

- **renderer** (*SDL_Renderer*) – The rendering context.

- **texture** (*SDL_Texture* or None) – The targeted texture, which must be created with the SDL_TEXTURTEACCESS_TARGET flag, or None for the default render target.

csdl2.**SDL_GetRenderTarget**(*renderer*) → SDL_Texture
Queries the renderer's current render target.

Parameters **renderer** (*SDL_Renderer*) – The rendering context.

Returns The current render target, or None for the default render target.

1.5.6 Device independent resolution

csdl2.**SDL_RenderSetLogicalSize**(*renderer*, *w*, *h*)
Sets a device independent resolution for rendering.

Parameters

- **renderer** (*SDL_Renderer*) – The renderer for which resolution should be set.
- **w** (*int*) – The width of the logical resolution.
- **h** (*int*) – The height of the logical resolution.

csdl2.**SDL_RenderGetLogicalSize**(*renderer*) → tuple
Queries the device independent resolution for rendering.

If the renderer did not have its logical size set by *SDL_RenderSetLogicalSize()*, the function returns (0, 0).

Parameters **renderer** (*SDL_Renderer*) – A rendering context.

Returns An (int, int) tuple with the width and height of the logical resolution respectively.

1.5.7 Viewport

csdl2.**SDL_RenderSetViewport**(*renderer*, *rect*)
Sets the drawing area for rendering on the current target.

When the window is resized, the current viewport is automatically centered within the new window size.

Parameters

- **renderer** (*SDL_Renderer*) – The rendering context.
- **rect** (*SDL_Rect* or None) – The drawing area, or None to set the viewport to the entire target.

csdl2.**SDL_RenderGetViewport**(*renderer*) → SDL_Rect
Queries the drawing area for the current target.

Parameters **renderer** (*SDL_Renderer*) – The rendering context.

Returns A *SDL_Rect* with the drawing area for the current target.

1.5.8 Clip Rectangle

csdl2.**SDL_RenderSetClipRect**(*renderer*, *rect*)
Sets the clip rectangle for the current target.

Parameters

- **renderer** (*SDL_Renderer*) – The renderer for which clip rectangle should be set.

- **rect** (*SDL_Rect* or None) – The rectangle to set as the clip rectangle, or None to disable clipping.

csdl2.**SDL_RenderGetClipRect**(*renderer*) → SDL_Rect
Gets the clip rectangle for the current target.

Parameters **renderer** (*SDL_Renderer*) – The renderer from which clip rectangle should be queried.

Returns A *SDL_Rect* with the current clip rectangle, or an empty rectangle if clipping is disabled.

1.5.9 Scaling

csdl2.**SDL_RenderSetScale**(*renderer*, *scaleX*, *scaleY*)
Sets the drawing scale for rendering on the current target.

The drawing coordinates are scaled by the `x/y` scaling factors before they are used by the renderer. This allows resolution independent drawing with a single coordinate system.

Parameters

- **renderer** (*SDL_Renderer*) – The renderer for which the drawing scale should be set.
- **scaleX** (*float*) – The horizontal scaling factor.
- **scaleY** (*float*) – The vertical scaling factor.

Note: If this results in scaling or subpixel drawing by the rendering backend, it will be hendled using the appropriate quality hints. For best results use integer scaling factors.

csdl2.**SDL_RenderGetScale**(*renderer*) → tuple
Gets the drawing scale for the current target.

Parameters **renderer** (*SDL_Renderer*) – The renderer from which drawing scale should be queried.

Returns A 2-tuple `(scaleX, scaleY)` with the float horizontal and vertical scaling factors respectively.

1.5.10 Drawing

csdl2.**SDL_SetRenderDrawColor**(*renderer: SDL_Renderer*, *r: int*, *g: int*, *b: int*, *a: int*) → None
Sets the color used for drawing primitives, and for `SDL_RenderClear()`.

Parameters

- **renderer** (*SDL_Renderer*) – The rendering context.
- **r** (*int*) – The red value used to draw on the rendering target, within the range 0-255.
- **g** (*int*) – The green value used to draw on the rendering target, within the range 0-255.
- **b** (*int*) – The blue value used to draw on the rendering target, within the range 0-255.
- **a** (*int*) – The alpha value used to draw on the rendering target, within the range 0-255. Use `SDL_SetRenderDrawBlendMode()` to specify how the alpha channel is used.

csdl2.**SDL_GetRenderDrawColor**(*renderer: SDL_Renderer*) → tuple
Returns the color used for drawing operations.

Parameters **renderer** (*SDL_Renderer*) – The rendering context.

Returns The (r, g, b, a) components of the drawing color.

Return type (int, int, int, int) tuple

`csdl2.`**`SDL_SetRenderDrawBlendMode`**(*renderer*, *blendMode*)

Sets the blend mode used for drawing operations (Fill and Line).

Parameters

- **`renderer`** (*`SDL_Renderer`*) – The renderer for which blend mode should be set.
- **`blendMode`** (*`int`*) – The blend mode to use for blending. One of the *Blend modes*.

Note: If the blend mode is not supported, the closest supported mode is chosen.

`csdl2.`**`SDL_GetRenderDrawBlendMode`**(*renderer*) → int

Gets the blend mode used for drawing operations.

Parameters **`renderer`** (*`SDL_Renderer`*) – The renderer from which blend mode should be queried.

Returns The current blend mode. One of the *Blend modes*.

`csdl2.`**`SDL_RenderClear`**(*renderer: SDL_Renderer*) → None

Clears the current rendering target with the current drawing color.

The entire rendering target will be cleared, ignoring the viewport.

Parameters **`renderer`** (*`SDL_Renderer`*) – The rendering context.

`csdl2.`**`SDL_RenderDrawPoint`**(*renderer*, *x*, *y*)

Draws a point on the current rendering target.

Parameters

- **`renderer`** (*`SDL_Renderer`*) – The renderer which should draw a point.
- **`x`** (*`int`*) – The x coordinate of the point.
- **`y`** (*`int`*) – The y coordinate of the point.

`csdl2.`**`SDL_RenderDrawPoints`**(*renderer*, *points*, *count*)

Draw multiple points on the current rendering target.

Parameters

- **`renderer`** (*`SDL_Renderer`*) – The rendering context.
- **`points`** (*`SDL_Point`* array) – The points to draw.
- **`count`** (*`int`*) – The number of points to draw.

`csdl2.`**`SDL_RenderDrawLine`**(*renderer*, *x1*, *y1*, *x2*, *y2*)

Draw a line on the current rendering target.

Parameters

- **`renderer`** (*`SDL_Renderer`*) – The rendering context.
- **`x1`** (*`int`*) – The x coordinate of the start point.
- **`y1`** (*`int`*) – The y coordinate of the start point.
- **`x2`** (*`int`*) – The x coordinate of the end point.
- **`y2`** (*`int`*) – The y coordinate of the end point.

`csdl2.`**`SDL_RenderDrawLines`**(*renderer*, *points*, *count*)

Draw a series of connected lines on the current rendering target.

Parameters

- **`renderer`** (*`SDL_Renderer`*) – The rendering context.

- **points** (*SDL_Point* array) – The points along the lines.
- **count** (*int*) – The number of points, drawing `count - 1` lines.

`csdl2.`**`SDL_RenderDrawRect`**(*renderer*, *rect*)

Draw a rectangle on the current rendering target.

Parameters

- **renderer** (*SDL_Renderer*) – The rendering context.
- **rect** (*SDL_Rect* or None) – The rectangle to draw, or None to outline the entire rendering target.

`csdl2.`**`SDL_RenderDrawRects`**(*renderer*, *rects*, *count*)

Draw some number of rectangles on the current rendering target.

Parameters

- **renderer** (*SDL_Renderer*) – The rendering context.
- **rects** (*SDL_Rect* array.) – The rectangles to be drawn.
- **count** (*int*) – The number of rectangles.

`csdl2.`**`SDL_RenderFillRect`**(*renderer: SDL_Renderer*, *rect: SDL_Rect*) → None

Fills a rectangle on the current rendering target with the current drawing color.

Parameters

- **renderer** (*SDL_Renderer*) – The rendering context.
- **rect** (*SDL_Rect* or None) – The *SDL_Rect* representing the rectangle to fill. If None, the entire rendering target will be filled.

`csdl2.`**`SDL_RenderFillRects`**(*renderer*, *rects*, *count*)

Fill some number of rectangles on the current rendering target with the current drawing color.

Parameters

- **renderer** (*SDL_Renderer*) – The rendering context.
- **rects** (*SDL_Rect* array) – The rectangles to be filled.
- **count** (*int*) – The number of rectangles.

`csdl2.`**`SDL_RenderCopy`**(*renderer*, *texture*, *srcrect*, *dstrect*)

Copies a portion of the texture to the current rendering target.

The texture is blended with the destination based on its blend mode set with `SDL_SetTextureBlendMode()`.

The texture color is affected based on its color modulation set by `SDL_SetTextureColorMod()`.

The texture alpha is affected based on its alpha modulation set by `SDL_SetTextureAlphaMod()`.

Parameters

- **renderer** (*SDL_Renderer*) – The rendering context.
- **texture** (*SDL_Texture*) – The source texture.
- **srcrect** (*SDL_Rect* buffer or None) – The source rectangle, or None for the entire texture.
- **dstrect** (*SDL_Rect* buffer or None) – The destination rectangle, or None for the entire rendering target. The texture will be stretched to fill the given rectangle.

`csdl2.`**`SDL_RenderCopyEx`**(*renderer*, *texture*, *srcrect*, *dstrect*, *angle*, *center*, *flip*)

Copies a portion of the texture to the current rendering target, optionally rotating it by an angle around the given center and also flipping it top-bottom and/or left-right.

The texture is blended with the destination based on its blend mode set with `SDL_SetTextureBlendMode()`.

The texture color is affected based on its color modulation set by `SDL_SetTextureColorMod()`.

The texture alpha is affected based on its alpha modulation set by `SDL_SetTextureAlphaMod()`.

Parameters

- **renderer** (`SDL_Renderer`) – The rendering context.
- **texture** (`SDL_Texture`) – The source texture.
- **srcrect** (`SDL_Rect` or None) – The source rectangle, or None for the entire texture.
- **dstrect** (`SDL_Rect` or None) – The destination rectangle, or None for the entire rendering target. The texture will be stretched to fill the given rectangle.
- **angle** (`float`) – An angle in degrees that indicates the rotation that will be applied to *dstrect*.
- **center** (`SDL_Point` or None) – The point around which *dstrect* will be rotated. If None, rotation will be done around `(dstrect.w/2, dstrect.h/2)`.
- **flip** (`int`) – Indicates which flipping actions should be performed on the texture. One or more of `SDL_FLIP_NONE`, `SDL_FLIP_HORIZONTAL` and/or `SDL_FLIP_VERTICAL` OR'd together.

`csdl2.`**`SDL_FLIP_NONE`**
Do not flip.

`csdl2.`**`SDL_FLIP_HORIZONTAL`**
Flip horizontally.

`csdl2.`**`SDL_FLIP_VERTICAL`**
Flip vertically.

1.5.11 Reading pixels

`csdl2.`**`SDL_RenderReadPixels`**(*renderer*, *rect*, *format*, *pixels*, *pitch*)
Read pixels from the current rendering target.

Parameters

- **renderer** (`SDL_Renderer`) – The rendering context.
- **rect** (`SDL_Rect` or None) – The area to read, or None for the entire render target.
- **format** (`int`) – The desired format of the pixel data (one of the *Pixel format constants*), or 0 to use the format of the rendering target.
- **pixels** (`buffer`) – The buffer to be filled in with the pixel data.
- **pitch** (`int`) – The pitch of the *pixels* buffer.

1.5.12 Updating the screen

SDL's rendering functions operate on a backbuffer. Calling a rendering function such as `SDL_RenderDrawLine()` does not directly draw a line on the screen, but rather updates the backbuffer. As such, after composing your entire scene with the drawing functions, you need to *present* the composed buffer to the screen as a complete picture. This is done with `SDL_RenderPresent()`.

`csdl2.`**`SDL_RenderPresent`**(*renderer: SDL_Renderer*) → None
Updates the screen with any rendering performed since the previous call.

If the renderer has VSync enabled, this function will block while waiting for the next vertical refresh, hence eliminating screen tearing.

Parameters **renderer** (*SDL_Renderer*) – The rendering context

---

Note: The backbuffer should be considered invalidated after each call to `SDL_RenderPresent()`. Do not assume that previous contents will exist between frames. You are strongly encouraged to call `SDL_RenderClear()` to initialize the backbuffer before drawing each frame.

1.5.13 OpenGL Support

`csdl2.`**`SDL_GL_BindTexture`**(*texture*) → tuple

Bind an OpenGL/ES/ES2 texture to the current context for use with when rendering OpenGL primitives directly.

Parameters **texture** (*SDL_Texture*) – The texture to bind to the current OpenGL/ES/ES2 context.

Returns A `(float, float)` tuple with the texture width and texture height respectively.

Note: In most cases, the texture height and width will be 1.0. However, on systems that support the GL_ARB_texture_rectangle extension, these values will actually be the pixel width and height used to create the texture, and so this factor needs to be taken into account when providing texture coordinates to OpenGL.

Note: SDL may upload RGB textures as BGR (or vice-versa), and re-order the color channels in the shader phase, so the uploaded texture may have swapped color channels.

`csdl2.`**`SDL_GL_UnbindTexture`**(*texture*)

Unbind an OpenGL/ES/ES2 texture from the current context.

Parameters **texture** (*SDL_Texture*) – The texture to unbind from the current OpenGL/ES/ES2 context.

1.6 Pixel Formats and Conversion Routines

1.6.1 Pixel Types

The pixel type is one of the following values:

`csdl2.`**`SDL_PIXELTYPE_UNKNOWN`**

Unknown pixel type.

Indexed Pixel Types

`csdl2.`**`SDL_PIXELTYPE_INDEX1`**

`csdl2.`**`SDL_PIXELTYPE_INDEX4`**

`csdl2.`**`SDL_PIXELTYPE_INDEX8`**

Packed Pixel Types

`csdl2.`**`SDL_PIXELTYPE_PACKED8`**

`csdl2.`**`SDL_PIXELTYPE_PACKED16`**

csdl2.**SDL_PIXELTYPE_PACKED32**

Bitmap Pixel Types

csdl2.**SDL_PIXELTYPE_ARRAYU8**

csdl2.**SDL_PIXELTYPE_ARRAYU16**

csdl2.**SDL_PIXELTYPE_ARRAYU32**

csdl2.**SDL_PIXELTYPE_ARRAYF16**

csdl2.**SDL_PIXELTYPE_ARRAYF32**

1.6.2 Pixel Ordering

Depending on the pixel type there are three different types of orderings – bitmapped, packed or array.

Bitmap pixel order (high bit -> low bit)

csdl2.**SDL_BITMAPORDER_NONE**

csdl2.**SDL_BITMAPORDER_4321**

csdl2.**SDL_BITMAPORDER_1234**

Packed component order (high bit -> low bit)

csdl2.**SDL_PACKEDORDER_NONE**

csdl2.**SDL_PACKEDORDER_XRGB**

csdl2.**SDL_PACKEDORDER_RGBX**

csdl2.**SDL_PACKEDORDER_ARGB**

csdl2.**SDL_PACKEDORDER_RGBA**

csdl2.**SDL_PACKEDORDER_XBGR**

csdl2.**SDL_PACKEDORDER_BGRX**

csdl2.**SDL_PACKEDORDER_ABGR**

csdl2.**SDL_PACKEDORDER_BGRA**

Array component order (low byte -> high byte)

csdl2.**SDL_ARRAYORDER_NONE**

csdl2.**SDL_ARRAYORDER_RGB**

csdl2.**SDL_ARRAYORDER_RGBA**

csdl2.**SDL_ARRAYORDER_ARGB**

csdl2.**SDL_ARRAYORDER_BGR**

csdl2.**SDL_ARRAYORDER_BGRA**

csdl2.**SDL_ARRAYORDER_ABGR**

1.6.3 Pixel Formats

class csdl2.**SDL_PixelFormat**

Pixel format information.

This structure cannot be directly constructed. Use *SDL_AllocFormat()* instead.

format
(readonly) A constant specifying the pixel format. See *Pixel format constants* for possible values.

palette
(readonly) The *SDL_Palette* associated with this pixel format, or None if this format does not have a palette.

BitsPerPixel
(readonly) The number of significant bits in a pixel value. E.g. 8, 15, 16, 24, 32.

BytesPerPixel
(readonly) The number of bytes required to hold a pixel value. E.g. 1, 2, 3, 4.

Rmask
(readonly) A mask representing the location of the red component of a pixel.

Gmask
(readonly) A mask representing the location of the green component of a pixel.

Bmask
(readonly) A mask representing the location of the blue component of a pixel.

Rloss
(readonly) The red value of a pixel has this number of bits less compared to 8-bit values.

Gloss
(readonly) The green value of a pixel has this number of bits less compared to 8-bit values.

Bloss
(readonly) The blue value of a pixel has this number of bits less compared to 8-bit values.

Aloss
(readonly) The alpha value of a pixel has this number of bits less compared to 8-bit values.

Rshift
(readonly) The bit index of the red field of a pixel.

Gshift
(readonly) The bit index of the green value of a pixel.

Bshift
(readonly) The bit index of the blue value of a pixel.

Ashift
(readonly) The bit index of the alpha value of a pixel.

csdl2.**SDL_AllocFormat**(*pixel_format: int*) → SDL_PixelFormat

Creates a *SDL_PixelFormat* structure corresponding to the pixel format constant *pixel_format*.

Parameters **pixel_format** (*int*) – One of the *Pixel format constants*.

Returns A *SDL_PixelFormat*.

csdl2.**SDL_FreeFormat**(*format: SDL_PixelFormat*) → None

Frees the *SDL_PixelFormat* structure allocated by *SDL_AllocFormat()*.

There is no need to manually call this function. csdl2 will automatically call this function upon garbage collection.

Parameters **format** (SDL_PixelFormat) – *SDL_PixelFormat* structure to free.

Pixel format constants

csdl2.**SDL_PIXELFORMAT_UNKNOWN**

csdl2.**SDL_PIXELFORMAT_INDEX1LSB**

csdl2.**SDL_PIXELFORMAT_INDEX1MSB**

csdl2.**SDL_PIXELFORMAT_INDEX4LSB**

csdl2.**SDL_PIXELFORMAT_INDEX4MSB**

csdl2.**SDL_PIXELFORMAT_INDEX8**

csdl2.**SDL_PIXELFORMAT_RGB332**

csdl2.**SDL_PIXELFORMAT_RGB444**

csdl2.**SDL_PIXELFORMAT_RGB555**

csdl2.**SDL_PIXELFORMAT_BGR555**

csdl2.**SDL_PIXELFORMAT_ARGB4444**

csdl2.**SDL_PIXELFORMAT_RGBA4444**

csdl2.**SDL_PIXELFORMAT_ABGR4444**

csdl2.**SDL_PIXELFORMAT_BGRA4444**

csdl2.**SDL_PIXELFORMAT_ARGB1555**

csdl2.**SDL_PIXELFORMAT_RGBA5551**

csdl2.**SDL_PIXELFORMAT_ABGR1555**

csdl2.**SDL_PIXELFORMAT_BGRA5551**

csdl2.**SDL_PIXELFORMAT_RGB565**

csdl2.**SDL_PIXELFORMAT_BGR565**

csdl2.**SDL_PIXELFORMAT_RGB24**

csdl2.**SDL_PIXELFORMAT_BGR24**

csdl2.**SDL_PIXELFORMAT_RGB888**

csdl2.**SDL_PIXELFORMAT_RGBX8888**

csdl2.**SDL_PIXELFORMAT_BGR888**

csdl2.**SDL_PIXELFORMAT_BGRX8888**

csdl2.**SDL_PIXELFORMAT_ARGB8888**

csdl2.**SDL_PIXELFORMAT_RGBA8888**

csdl2.**SDL_PIXELFORMAT_ABGR8888**

csdl2.**SDL_PIXELFORMAT_BGRA8888**

csdl2.**SDL_PIXELFORMAT_ARGB2101010**

csdl2.**SDL_PIXELFORMAT_YV12**

csdl2.**SDL_PIXELFORMAT_IYUV**

csdl2.**SDL_PIXELFORMAT_YUY2**

csdl2.**SDL_PIXELFORMAT_UYVY**

csdl2.**SDL_PIXELFORMAT_YVYU**

1.6.4 Color Palette

class `csdl2.`**`SDL_Palette`**

A color palette.

Every pixel in an 8-bit surface is an index into the *colors* field of the `SDL_Palette` referenced by the `SDL_PixelFormat`.

This structure cannot be directly constructed. One will be automatically created as needed when SDL allocates a `SDL_PixelFormat`. It can also be created through `SDL_AllocPalette()`.

`ncolors`

(readonly) Number of colors in the palette.

`colors`

(readonly) An array of `SDL_Color` structures representing the palette. This array cannot be directly modified. Use `SDL_SetPaletteColors()` instead.

`csdl2.`**`SDL_AllocPalette`**(*ncolors: int*) → SDL_AllocPalette

Create a new `SDL_Palette` with *ncolors* number of color entries. The color entries are initialized to white.

Parameters **`ncolors`** (*`int`*) – Number of colors in the palette.

Returns A new `SDL_Palette`.

`csdl2.`**`SDL_FreePalette`**(*palette: SDL_Palette*) → None

Frees the specified palette.

There is no need to call this function as csdl2 will automatically call this function on garbage collection.

Parameters **`palette`** (`SDL_Palette`) – The `SDL_Palette` to be freed.

1.7 Rectangle Functions

class `csdl2.`**`SDL_Point`**(*x: int =0*, *y: int =0*)

A 2d point.

`x`

The x location of the point.

`y`

The y location of the point.

class `csdl2.`**`SDL_Rect`**(*x: int =0*, *y: int =0*, *w: int =0*, *h: int =0*)

A 2d rectangle with its origin at the upper left.

`x`

The x location of the rectangle's upper left corner.

`y`

The y location of the rectangle's upper left corner.

`w`

The width of the rectangle.

`h`

The height of the rectangle.

`csdl2.`**`SDL_HasIntersection`**(*A: SDL_Rect*, *B: SDL_Rect*) → bool

Determines if two rectangles intersect.

Parameters

- **`A`** (`SDL_Rect` or None) – First rectangle.
- **`B`** (`SDL_Rect` or None) – Second rectangle.

Returns True if there is an intersection, False otherwise. If *A* and/or *B* are None, the function will return False.

1.8 Event Handling

Event handling allows your application to receive input from the user. Event handling is initialized with a call to:

```
>>> from csdl2 import *
>>> SDL_Init(SDL_INIT_EVENTS)
```

SDL stores each event as a SDL_Event in an event queue. SDL_Event structures 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.

class `csdl2.`**`SDL_Event`**

A union that contains structures for the different event types.

`type`

An int specifying the event type. Use the event type's corresponding attribute to get/set information about the event:

Value of *type*	Attr of *SDL_Event*
SDL_CONTROLLERAXISMOTION	SDL_Event.caxis
SDL_CONTROLLERBUTTONDOWN, *SDL_CONTROLLERBUTTONUP*	SDL_Event.cbutton
SDL_CONTROLLERDEVICEADDED, *SDL_CONTROLLERDEVICEREMOVED*, *SDL_CONTROLLERDEVICEREMAPPED*	SDL_Event.cdevice
SDL_DOLLARGESTURE, *SDL_DOLLARRECORD*	SDL_Event.dgesture
SDL_DROPFILE	SDL_Event.drop
SDL_FINGERMOTION, *SDL_FINGERDOWN*, *SDL_FINGERUP*	SDL_Event.tfinger
SDL_KEYDOWN, *SDL_KEYUP*	SDL_Event.key
SDL_JOYAXISMOTION	SDL_Event.jaxis
SDL_JOYBALLMOTION	SDL_Event.jball
SDL_JOYHATMOTION	SDL_Event.jhat
SDL_JOYBUTTONDOWN, *SDL_JOYBUTTONUP*	SDL_Event.jbutton
SDL_JOYDEVICEADDED, *SDL_JOYDEVICEREMOVED*	SDL_Event.jdevice
SDL_MOUSEMOTION	*SDL_Event.motion*
SDL_MOUSEBUTTONDOWN, *SDL_MOUSEBUTTONUP*	SDL_Event.button
SDL_MOUSEWHEEL	SDL_Event.wheel
SDL_MULTIGESTURE	SDL_Event.mgesture
SDL_QUIT	SDL_Event.quit
SDL_SYSWMEVENT	SDL_Event.syswm
SDL_TEXTEDITING	SDL_Event.edit
SDL_TEXTINPUT	SDL_Event.text
SDL_USEREVENT	SDL_Event.user
SDL_WINDOWEVENT	SDL_Event.window

`motion`

(readonly) If *SDL_Event.type* is *SDL_MOUSEMOTION*, use this attribute to access the underlying *SDL_MouseMotionEvent* mouse motion event data.

`csdl2.`**`SDL_QUIT`**

User-requested quit.

`csdl2.`**`SDL_APP_TERMINATING`**

The application is being terminated by the OS. Called on iOS in applicationWillTerminate(). Called on Android in onDestroy().

`csdl2.`**`SDL_APP_LOWMEMORY`**

The application is low on memory, free memory if possible. Called on iOS in applicationDidReceiveMemoryWarning(). Called on Android in onLowMemory().

csdl2.**SDL_APP_WILLENTERBACKGROUND**
The application is about to enter the background. Called on iOS in applicationWillResignActive(). Called on Android in onPause().

csdl2.**SDL_APP_DIDENTERBACKGROUND**
The application did enter the background and may not get CPU for some time. Called on iOS in applicationDidEnterBackground(). Called on Android in onPause().

csdl2.**SDL_APP_WILLENTERFOREGROUND**
The application is about to enter the foreground. Called on iOS in applicationWillEnterForeground(). Called on Android in onResume().

csdl2.**SDL_APP_DIDENTERFOREGROUND**
The application is now interactive. Called on iOS in applicationDidBecomeActive(). Called on Android in onResume().

csdl2.**SDL_WINDOWEVENT**
Window state change.

csdl2.**SDL_SYSWMEVENT**
System specific event.

csdl2.**SDL_KEYDOWN**
Key pressed.

csdl2.**SDL_KEYUP**
Key released.

csdl2.**SDL_TEXTEDITING**
Keyboard text editing (composition).

csdl2.**SDL_TEXTINPUT**
Keyboard text input.

csdl2.**SDL_MOUSEMOTION**
Mouse moved.

csdl2.**SDL_MOUSEBUTTONDOWN**
Mouse button pressed.

csdl2.**SDL_MOUSEBUTTONUP**
Mouse button released.

csdl2.**SDL_MOUSEWHEEL**
Mouse wheel motion.

csdl2.**SDL_JOYAXISMOTION**
Joystick axis motion.

csdl2.**SDL_JOYBALLMOTION**
Joystick trackball motion.

csdl2.**SDL_JOYHATMOTION**
Joystick hat position change.

csdl2.**SDL_JOYBUTTONDOWN**
Joystick button pressed.

csdl2.**SDL_JOYBUTTONUP**
Joystick button released.

csdl2.**SDL_JOYDEVICEADDED**
A new joystick has been inserted into the system.

csdl2.**SDL_JOYDEVICEREMOVED**
An opened joystick has been removed.

`csdl2.`**`SDL_CONTROLLERAXISMOTION`**
Game controller axis motion.

`csdl2.`**`SDL_CONTROLLERBUTTONDOWN`**
Game controller button pressed.

`csdl2.`**`SDL_CONTROLLERBUTTONUP`**
Game controller button released.

`csdl2.`**`SDL_CONTROLLERDEVICEADDED`**
A new game controller has been inserted into the system.

`csdl2.`**`SDL_CONTROLLERDEVICEREMOVED`**
A opened game controller has been removed.

`csdl2.`**`SDL_CONTROLLERDEVICEREMAPPED`**
The controller mapping was updated.

`csdl2.`**`SDL_FINGERDOWN`**
User has touched input device.

`csdl2.`**`SDL_FINGERUP`**
User stopped touching input device.

`csdl2.`**`SDL_FINGERMOTION`**
User is dragging finger on input device.

`csdl2.`**`SDL_DOLLARGESTURE`**
User made a dollar gesture.

`csdl2.`**`SDL_DOLLARRECORD`**
When recording a gesture with SDL_RecordGesture, the user made a dollar gesture that was recorded.

`csdl2.`**`SDL_MULTIGESTURE`**
User made a gesture with multiple fingers.

`csdl2.`**`SDL_CLIPBOARDUPDATE`**
The clipboard changed.

`csdl2.`**`SDL_DROPFILE`**
The system requests a file open.

`csdl2.`**`SDL_USEREVENT`**
`csdl2.`**`SDL_LASTEVENT`**
Events *`SDL_USEREVENT`* through *`SDL_LASTEVENT`* are for your use, and should be allocated with SDL_RegisterEvents

`csdl2.`**`SDL_PumpEvents`**`()` → None
Pumps the event loop, gathering events from the input devices.

This function updates the event queue and internal input device state. Without calling this function, no input events will ever be placed on the queue.

`SDL_PollEvent()` and `SDL_WaitEvent()` implicitly call this function. If you are not polling or waiting for events using these functions, you must explicitly call *`SDL_PumpEvents()`* to force an event queue update.

This should only be run in the thread that sets the video mode.

`csdl2.`**`SDL_PeepEvents`**`(`*events*, *numevents: int*, *action: int*, *minType: int*, *maxType: int*`)` → int
If *action* is *`SDL_ADDEVENT`*, up to *numevents* events will be added to the back of the event queue. Returns the number of events added.

If *action* is *`SDL_PEEKEVENT`*, up to *numevents* events from the front of the event queue, within *minType* and *maxType*, will be returned in *events*, but will not be removed from the queue. Returns number of events peeked.

If *action* is *SDL_GETEVENT*, up to *numevents* events from the front of the event queue, within *minType* and *maxType*, will be returned in *events*, and will be removed from the queue. Returns number of events retrieved.

Parameters

- **events** (SDL_Event) – Either a *SDL_Event* object, or a buffer of equivalent size.
- **numevents** (*int*) – If *action* is *SDL_ADDEVENT*, the number of events to add to the event queue. If *action* is *SDL_PEEKEVENT* or *SDL_GETEVENT*, the maximum number of events to retrieve.
- **action** (*int*) – One of *SDL_ADDEVENT*, *SDL_PEEKEVENT* or *SDL_GETEVENT*.
- **minType** (*int*) – minimum value of the event type to be considered. SDL_FIRSTEVENT is a safe choice.
- **maxType** (*int*) – maximum value of the event type to be considered. *SDL_LASTEVENT* is a safe choice.

Returns Number of events added to the event queue for *SDL_ADDEVENT*, number of events retrieved from the event queue for *SDL_PEEKEVENT* and *SDL_GETEVENT*.

csdl2.**SDL_ADDEVENT**
csdl2.**SDL_PEEKEVENT**
csdl2.**SDL_GETEVENT**
Possible actions for *SDL_PeepEvents()*.

csdl2.**SDL_FlushEvents**(*minType: int*, *maxType: int*) → None
Removes all events from the event queue within the specified *minType* and *maxType*.

To clear all events, set *minType* to SDL_FIRSTEVENT and *maxType* to *SDL_LASTEVENT*. To clear all user events, set *minType* to *SDL_USEREVENT* and *maxType* to *SDL_LASTEVENT*.

This function only affects currently queued events. If you wish to make sure that all pending OS events are flushed, you can call *SDL_PumpEvents()* on the main thread immediately before *SDL_FlushEvents()*.

Parameters

- **minType** (*int*) – minimum event type to be cleared.
- **maxType** (*int*) – maximum event type to be cleared.

csdl2.**SDL_PollEvent**(*event*) → bool
Polls for currently pending events.

Parameters **event** (*SDL_Event or None*) – If not None, the next event is removed from the queue and stored in it. If None, no event will be removed from the queue.

Returns True if there are events in the queue, False otherwise.

csdl2.**SDL_PushEvent**(*event*) → bool
Copies *event* into the event queue.

Parameters **event** (SDL_Event) – Event to be copied into the event queue. Either a *SDL_Event* instance, or a buffer of equivalent size.

Returns True on success, False if the event was filtered.

Note: For pushing application-specific events, please use SDL_RegisterEvents() to get an event type that does not conflict with other code that also wants its own custom event types.

1.8.1 Mouse motion events

A *SDL_MOUSEMOTION* event occurs whenever a user moves the mouse within any window or when SDL_WarpMouseInWindow() is called.

class csdl2.**SDL_MouseMotionEvent**
A structure that contains mouse motion event information.

SDL_MouseMotionEvent is a member of the *SDL_Event* union and is used when an event of type *SDL_MOUSEMOTION* is reported. You would access it through the *SDL_Event.motion* attribute.

type
The event type. This should be *SDL_MOUSEMOTION*.

timestamp
Timestamp of the event.

windowID
The window with mouse focus, if any.

which
The mouse instance ID. This may be SDL_TOUCH_MOUSEID, for events that were generated by a touch input device, and not a real mouse. You might want to ignore such events, if your application already handles the *SDL_FINGERMOTION* event.

state
A 32-bit bitmask of the current button state and is the same as that returned by SDL_GetMouseState(). You can test different buttons by using the masks SDL_BUTTON_LMASK, SDL_BUTTON_MMASK, SDL_BUTTON_RMASK, SDL_BUTTON_X1MASK and SDL_BUTTON_X2MASK.

x
X coordinate, relative to window.

y
Y coordinate, relative to window.

xrel
Motion in the X direction, relative to the last *SDL_MOUSEMOTION* event. If relative mouse mode is enabled with SDL_SetRelativeMouseMode(), relative movement will still be reported even when the cursor reached the edge of the screen.

yrel
Motion in the Y direction, relative to the last *SDL_MOUSEMOTION* event. If relative mouse mode is enabled with SDL_SetRelativeMouseMode(), relative movement will still be reported even when the cursor reached the edge of the screen.

1.9 Scancode Constants

These constants are used to represent the physical location of a keyboard key on the keybord. They are used in many places, such as the SDL_KeyboardEvent.keysym attribute.

csdl2.**SDL_SCANCODE_UNKNOWN**

csdl2.**SDL_SCANCODE_A**

csdl2.**SDL_SCANCODE_B**

csdl2.**SDL_SCANCODE_C**

csdl2.**SDL_SCANCODE_D**

csdl2.**SDL_SCANCODE_E**

csdl2.**SDL_SCANCODE_F**

csdl2.**SDL_SCANCODE_G**

csdl2.**SDL_SCANCODE_H**

csdl2.**SDL_SCANCODE_I**

csdl2.**SDL_SCANCODE_J**

csdl2.**SDL_SCANCODE_K**

csdl2.**SDL_SCANCODE_L**

csdl2.**SDL_SCANCODE_M**

csdl2.**SDL_SCANCODE_N**

csdl2.**SDL_SCANCODE_O**

csdl2.**SDL_SCANCODE_P**

csdl2.**SDL_SCANCODE_Q**

csdl2.**SDL_SCANCODE_R**

csdl2.**SDL_SCANCODE_S**

csdl2.**SDL_SCANCODE_T**

csdl2.**SDL_SCANCODE_U**

csdl2.**SDL_SCANCODE_V**

csdl2.**SDL_SCANCODE_W**

csdl2.**SDL_SCANCODE_X**

csdl2.**SDL_SCANCODE_Y**

csdl2.**SDL_SCANCODE_Z**

csdl2.**SDL_SCANCODE_1**

csdl2.**SDL_SCANCODE_2**

csdl2.**SDL_SCANCODE_3**

csdl2.**SDL_SCANCODE_4**

csdl2.**SDL_SCANCODE_5**

csdl2.**SDL_SCANCODE_6**

csdl2.**SDL_SCANCODE_7**

csdl2.**SDL_SCANCODE_8**

csdl2.**SDL_SCANCODE_9**

csdl2.**SDL_SCANCODE_0**

csdl2.**SDL_SCANCODE_RETURN**

csdl2.**SDL_SCANCODE_ESCAPE**

csdl2.**SDL_SCANCODE_BACKSPACE**

csdl2.**SDL_SCANCODE_TAB**

csdl2.**SDL_SCANCODE_SPACE**

csdl2.**SDL_SCANCODE_MINUS**

csdl2.**SDL_SCANCODE_EQUALS**

csdl2.**SDL_SCANCODE_LEFTBRACKET**

csdl2.**SDL_SCANCODE_RIGHTBRACKET**

csdl2.**SDL_SCANCODE_BACKSLASH**
Located at the lower left of the return key on ISO keyboards and at the right end of the QWERTY row on ANSI keyboards. Produces REVERSE SOLIDUS (backslash) and VERTICAL LINE in a US layout, REVERSE SOLIDUS and VERTICAL LINE in a UK Mac layout, NUMBER SIGN and TILDE in a UK Windows layout, DOLLAR SIGN and POUND SIGN in a Swiss German layout, NUMBER SIGN and APOSTROPHE in a German layout, GRAVE ACCENT and POUND SIGN in a French Mac layout, and ASTERISK and MICRO SIGN in a French Windows layout.

csdl2.**SDL_SCANCODE_NONUSHASH**
ISO USB keyboards actually use this code instead of 49 for the same key, but all OSes I've seen treat the two codes identically. So, as an implementor, unless your keyboard generates both of those codes and your OS treats them differently, you should generate SDL_SCANCODE_BACKSLASH instead of this code. As a user, you should not rely on this code because SDL will never generate it with most (all?) keyboards.

csdl2.**SDL_SCANCODE_SEMICOLON**

csdl2.**SDL_SCANCODE_APOSTROPHE**

csdl2.**SDL_SCANCODE_GRAVE**
Located in the top left corner (on both ANSI and ISO keyboards). Produces GRAVE ACCENT and TILDE in a US Windows layout and in US and UK Mac layouts on ANSI keyboards, GRAVE ACCENT and NOT SIGN in a UK Windows layout, SECTION SIGN and PLUS-MINUS SIGN in US and UK Mac layouts on ISO keyboards, SECTION SIGN and DEGREE SIGN in a Swiss German layout (Mac: only on ISO keyboards), CIRCUMFLEX ACCENT and DEGREE SIGN in a German layout (Mac: only on ISO keyboards), SUPERSCRIPT TWO and TILDE in a French Windows layout, COMMERCIAL AT and NUMBER SIGN in a French Mac layout on ISO keyboards, and LESS-THAN SIGN and GREATER-THAN SIGN in a Swiss German, German, or French Mac layout on ANSI keyboards.

csdl2.**SDL_SCANCODE_COMMA**

csdl2.**SDL_SCANCODE_PERIOD**

csdl2.**SDL_SCANCODE_SLASH**

csdl2.**SDL_SCANCODE_CAPSLOCK**

csdl2.**SDL_SCANCODE_F1**

csdl2.**SDL_SCANCODE_F2**

csdl2.**SDL_SCANCODE_F3**

csdl2.**SDL_SCANCODE_F4**

csdl2.**SDL_SCANCODE_F5**

csdl2.**SDL_SCANCODE_F6**

csdl2.**SDL_SCANCODE_F7**

csdl2.**SDL_SCANCODE_F8**

csdl2.**SDL_SCANCODE_F9**

csdl2.**SDL_SCANCODE_F10**

csdl2.**SDL_SCANCODE_F11**

csdl2.**SDL_SCANCODE_F12**

csdl2.**SDL_SCANCODE_PRINTSCREEN**

csdl2.**SDL_SCANCODE_SCROLLLOCK**

csdl2.**SDL_SCANCODE_PAUSE**

csdl2.**SDL_SCANCODE_INSERT**
Insert on PC, help on some Mac keyboards (but does send code 73, not 117)

csdl2.**SDL_SCANCODE_HOME**

csdl2.**SDL_SCANCODE_PAGEUP**

csdl2.**SDL_SCANCODE_DELETE**

csdl2.**SDL_SCANCODE_END**

csdl2.**SDL_SCANCODE_PAGEDOWN**

csdl2.**SDL_SCANCODE_RIGHT**

csdl2.**SDL_SCANCODE_LEFT**

csdl2.**SDL_SCANCODE_DOWN**

csdl2.**SDL_SCANCODE_UP**

csdl2.**SDL_SCANCODE_NUMLOCKCLEAR**
Num lock on PC, clear on Mac keyboards

csdl2.**SDL_SCANCODE_KP_DIVIDE**

csdl2.**SDL_SCANCODE_KP_MULTIPLY**

csdl2.**SDL_SCANCODE_KP_MINUS**

csdl2.**SDL_SCANCODE_KP_PLUS**

csdl2.**SDL_SCANCODE_KP_ENTER**

csdl2.**SDL_SCANCODE_KP_1**

csdl2.**SDL_SCANCODE_KP_2**

csdl2.**SDL_SCANCODE_KP_3**

csdl2.**SDL_SCANCODE_KP_4**

csdl2.**SDL_SCANCODE_KP_5**

csdl2.**SDL_SCANCODE_KP_6**

csdl2.**SDL_SCANCODE_KP_7**

csdl2.**SDL_SCANCODE_KP_8**

csdl2.**SDL_SCANCODE_KP_9**

csdl2.**SDL_SCANCODE_KP_0**

csdl2.**SDL_SCANCODE_KP_PERIOD**

csdl2.**SDL_SCANCODE_NONUSBACKSLASH**
This is the additional key that ISO keyboards have over ANSI ones, located between left shift and Y. Produces GRAVE ACCENT and TILDE in a US or UK Mac layout, REVERSE SOLIDUS (backslash) and VERTICAL LINE in a US or UK Windows layout, and LESS-THAN SIGN and GREATER-THAN SIGN in a Swiss German, German, or French layout.

csdl2.**SDL_SCANCODE_APPLICATION**
Windows contextual menu, compose.

csdl2.**SDL_SCANCODE_POWER**
The USB document says this is a status flag, not a physical key - but some Mac keyboards do have a power key.

csdl2.**SDL_SCANCODE_KP_EQUALS**

csdl2.**SDL_SCANCODE_F13**

csdl2.**SDL_SCANCODE_F14**

csdl2.**SDL_SCANCODE_F15**

csdl2.**SDL_SCANCODE_F16**

csdl2.**SDL_SCANCODE_F17**

csdl2.**SDL_SCANCODE_F18**

csdl2.**SDL_SCANCODE_F19**

csdl2.**SDL_SCANCODE_F20**

csdl2.**SDL_SCANCODE_F21**

csdl2.**SDL_SCANCODE_F22**

csdl2.**SDL_SCANCODE_F23**

csdl2.**SDL_SCANCODE_F24**

csdl2.**SDL_SCANCODE_EXECUTE**

csdl2.**SDL_SCANCODE_HELP**

csdl2.**SDL_SCANCODE_MENU**

csdl2.**SDL_SCANCODE_SELECT**

csdl2.**SDL_SCANCODE_STOP**

csdl2.**SDL_SCANCODE_AGAIN**
Redo.

csdl2.**SDL_SCANCODE_UNDO**

csdl2.**SDL_SCANCODE_CUT**

csdl2.**SDL_SCANCODE_COPY**

csdl2.**SDL_SCANCODE_PASTE**

csdl2.**SDL_SCANCODE_FIND**

csdl2.**SDL_SCANCODE_MUTE**

csdl2.**SDL_SCANCODE_VOLUMEUP**

csdl2.**SDL_SCANCODE_VOLUMEDOWN**

csdl2.**SDL_SCANCODE_KP_COMMA**

csdl2.**SDL_SCANCODE_KP_EQUALSAS400**

csdl2.**SDL_SCANCODE_INTERNATIONAL1**
Used on Asian keyboards, see footnotes in USB doc.

csdl2.**SDL_SCANCODE_INTERNATIONAL2**

csdl2.**SDL_SCANCODE_INTERNATIONAL3**
Yen.

csdl2.**SDL_SCANCODE_INTERNATIONAL4**

csdl2.**SDL_SCANCODE_INTERNATIONAL5**

csdl2.**SDL_SCANCODE_INTERNATIONAL6**

csdl2.**SDL_SCANCODE_INTERNATIONAL7**

csdl2.**SDL_SCANCODE_INTERNATIONAL8**

csdl2.**SDL_SCANCODE_INTERNATIONAL9**

csdl2.**SDL_SCANCODE_LANG1**
Hangul/English toggle.

csdl2.**SDL_SCANCODE_LANG2**
Hanja conversion.

csdl2.**SDL_SCANCODE_LANG3**
Katakana.

csdl2.**SDL_SCANCODE_LANG4**
Hiragana.

csdl2.**SDL_SCANCODE_LANG5**
Zenkaku/Hankaku

csdl2.**SDL_SCANCODE_LANG6**

csdl2.**SDL_SCANCODE_LANG7**

csdl2.**SDL_SCANCODE_LANG8**

csdl2.**SDL_SCANCODE_LANG9**

csdl2.**SDL_SCANCODE_ALTERASE**
Erase-Eaze

csdl2.**SDL_SCANCODE_SYSREQ**

csdl2.**SDL_SCANCODE_CANCEL**

csdl2.**SDL_SCANCODE_CLEAR**

csdl2.**SDL_SCANCODE_PRIOR**

csdl2.**SDL_SCANCODE_RETURN2**

csdl2.**SDL_SCANCODE_SEPARATOR**

csdl2.**SDL_SCANCODE_OUT**

csdl2.**SDL_SCANCODE_OPER**

csdl2.**SDL_SCANCODE_CLEARAGAIN**

csdl2.**SDL_SCANCODE_CRSEL**

csdl2.**SDL_SCANCODE_EXSEL**

csdl2.**SDL_SCANCODE_KP_00**

csdl2.**SDL_SCANCODE_KP_000**

csdl2.**SDL_SCANCODE_THOUSANDSSEPARATOR**

csdl2.**SDL_SCANCODE_DECIMALSEPARATOR**

csdl2.**SDL_SCANCODE_CURRENCYUNIT**

csdl2.**SDL_SCANCODE_CURRENCYSUBUNIT**

csdl2.**SDL_SCANCODE_KP_LEFTPAREN**

csdl2.**SDL_SCANCODE_KP_RIGHTPAREN**

csdl2.**SDL_SCANCODE_KP_LEFTBRACE**

csdl2.**SDL_SCANCODE_KP_RIGHTBRACE**

csdl2.**SDL_SCANCODE_KP_TAB**

csdl2.**SDL_SCANCODE_KP_BACKSPACE**

csdl2.**SDL_SCANCODE_KP_A**

csdl2.**SDL_SCANCODE_KP_B**

csdl2.**SDL_SCANCODE_KP_C**

csdl2.**SDL_SCANCODE_KP_D**

csdl2.**SDL_SCANCODE_KP_E**

csdl2.**SDL_SCANCODE_KP_F**

csdl2.**SDL_SCANCODE_KP_XOR**

csdl2.**SDL_SCANCODE_KP_POWER**

csdl2.**SDL_SCANCODE_KP_PERCENT**

csdl2.**SDL_SCANCODE_KP_LESS**

csdl2.**SDL_SCANCODE_KP_GREATER**

csdl2.**SDL_SCANCODE_KP_AMPERSAND**

csdl2.**SDL_SCANCODE_KP_DBLAMPERSAND**

csdl2.**SDL_SCANCODE_KP_VERTICALBAR**

csdl2.**SDL_SCANCODE_KP_DBLVERTICALBAR**

csdl2.**SDL_SCANCODE_KP_COLON**

csdl2.**SDL_SCANCODE_KP_HASH**

csdl2.**SDL_SCANCODE_KP_SPACE**

csdl2.**SDL_SCANCODE_KP_AT**

csdl2.**SDL_SCANCODE_KP_EXCLAM**

csdl2.**SDL_SCANCODE_KP_MEMSTORE**

csdl2.**SDL_SCANCODE_KP_MEMRECALL**

csdl2.**SDL_SCANCODE_KP_MEMCLEAR**

csdl2.**SDL_SCANCODE_KP_MEMADD**

csdl2.**SDL_SCANCODE_KP_MEMSUBTRACT**

csdl2.**SDL_SCANCODE_KP_MEMMULTIPLY**

csdl2.**SDL_SCANCODE_KP_MEMDIVIDE**

csdl2.**SDL_SCANCODE_KP_PLUSMINUS**

csdl2.**SDL_SCANCODE_KP_CLEAR**

csdl2.**SDL_SCANCODE_KP_CLEARENTRY**

csdl2.**SDL_SCANCODE_KP_BINARY**

csdl2.**SDL_SCANCODE_KP_OCTAL**

csdl2.**SDL_SCANCODE_KP_DECIMAL**

csdl2.**SDL_SCANCODE_KP_HEXADECIMAL**

csdl2.**SDL_SCANCODE_LCTRL**

csdl2.**SDL_SCANCODE_LSHIFT**

csdl2.**SDL_SCANCODE_LALT**
Alt, option.

csdl2.**SDL_SCANCODE_LGUI**
Windows, command (apple), meta

csdl2.**SDL_SCANCODE_RCTRL**

csdl2.**SDL_SCANCODE_RSHIFT**

csdl2.**SDL_SCANCODE_RALT**
Alt gr, option.

csdl2.**SDL_SCANCODE_RGUI**
Windows, command (apple), meta.

csdl2.**SDL_SCANCODE_MODE**

csdl2.**SDL_SCANCODE_AUDIONEXT**

csdl2.**SDL_SCANCODE_AUDIOPREV**

csdl2.**SDL_SCANCODE_AUDIOSTOP**

csdl2.**SDL_SCANCODE_AUDIOPLAY**

csdl2.**SDL_SCANCODE_AUDIOMUTE**

csdl2.**SDL_SCANCODE_MEDIASELECT**

csdl2.**SDL_SCANCODE_WWW**

csdl2.**SDL_SCANCODE_MAIL**

csdl2.**SDL_SCANCODE_CALCULATOR**

csdl2.**SDL_SCANCODE_COMPUTER**

csdl2.**SDL_SCANCODE_AC_SEARCH**

csdl2.**SDL_SCANCODE_AC_HOME**

csdl2.**SDL_SCANCODE_AC_BACK**

csdl2.**SDL_SCANCODE_AC_FORWARD**

csdl2.**SDL_SCANCODE_AC_STOP**

csdl2.**SDL_SCANCODE_AC_REFRESH**

csdl2.**SDL_SCANCODE_AC_BOOKMARKS**

csdl2.**SDL_SCANCODE_BRIGHTNESSDOWN**

csdl2.**SDL_SCANCODE_BRIGHTNESSUP**

csdl2.**SDL_SCANCODE_DISPLAYSWITCH**
Display mirroring/dual display switch, video mode switch.

csdl2.**SDL_SCANCODE_KBDILLUMTOGGLE**

csdl2.**SDL_SCANCODE_KBDILLUMDOWN**

csdl2.**SDL_SCANCODE_KBDILLUMUP**

csdl2.**SDL_SCANCODE_EJECT**

csdl2.**SDL_SCANCODE_SLEEP**

csdl2.**SDL_SCANCODE_APP1**

csdl2.**SDL_SCANCODE_APP2**

csdl2.**SDL_NUM_SCANCODES**
Not a key, just marks the number of scancodes for array bounds.

1.10 Keycode and Key Modifier Constants

1.10.1 Keycode Constants

These constants are mapped to the current layout of the keyboard and correlate to one of the *Scancode Constants*. The scancode identifies the location of a key press and the corresponding keycode gives that key press meaning in the context of the current keyboard layout.

csdl2.**SDLK_UNKNOWN**

csdl2.**SDLK_RETURN**

csdl2.**SDLK_ESCAPE**

csdl2.**SDLK_BACKSPACE**

csdl2.**SDLK_TAB**

csdl2.**SDLK_SPACE**

csdl2.**SDLK_EXCLAIM**

csdl2.**SDLK_QUOTEDBL**

csdl2.**SDLK_HASH**

csdl2.**SDLK_PERCENT**

csdl2.**SDLK_DOLLAR**

csdl2.**SDLK_AMPERSAND**

csdl2.**SDLK_QUOTE**

csdl2.**SDLK_LEFTPAREN**

csdl2.**SDLK_RIGHTPAREN**

csdl2.**SDLK_ASTERISK**

csdl2.**SDLK_PLUS**

csdl2.**SDLK_COMMA**

csdl2.**SDLK_MINUS**

csdl2.**SDLK_PERIOD**

csdl2.**SDLK_SLASH**

csdl2.**SDLK_0**

csdl2.**SDLK_1**

csdl2.**SDLK_2**

csdl2.**SDLK_3**

csdl2.**SDLK_4**

csdl2.**SDLK_5**

csdl2.**SDLK_6**

csdl2.**SDLK_7**

csdl2.**SDLK_8**

csdl2.**SDLK_9**

csdl2.**SDLK_COLON**

csdl2.**SDLK_SEMICOLON**

csdl2.**SDLK_LESS**

csdl2.**SDLK_EQUALS**

csdl2.**SDLK_GREATER**

csdl2.**SDLK_QUESTION**

csdl2.**SDLK_AT**

csdl2.**SDLK_LEFTBRACKET**

csdl2.**SDLK_BACKSLASH**

csdl2.**SDLK_RIGHTBRACKET**

csdl2.**SDLK_CARET**

csdl2.**SDLK_UNDERSCORE**

csdl2.**SDLK_BACKQUOTE**

csdl2.**SDLK_a**

csdl2.**SDLK_b**

csdl2.**SDLK_c**

csdl2.**SDLK_d**

csdl2.**SDLK_e**

csdl2.**SDLK_f**

csdl2.**SDLK_g**

csdl2.**SDLK_h**

csdl2.**SDLK_i**

csdl2.**SDLK_j**

csdl2.**SDLK_k**

csdl2.**SDLK_l**

csdl2.**SDLK_m**

csdl2.**SDLK_n**

csdl2.**SDLK_o**

csdl2.**SDLK_p**

csdl2.**SDLK_q**

csdl2.**SDLK_r**

csdl2.**SDLK_s**

csdl2.**SDLK_t**

csdl2.**SDLK_u**

csdl2.**SDLK_v**

csdl2.**SDLK_w**

csdl2.**SDLK_x**

csdl2.**SDLK_y**

csdl2.**SDLK_z**

csdl2.**SDLK_CAPSLOCK**

csdl2.**SDLK_F1**

csdl2.**SDLK_F2**

csdl2.**SDLK_F3**

csdl2.**SDLK_F4**

csdl2.**SDLK_F5**

csdl2.**SDLK_F6**

csdl2.**SDLK_F7**

csdl2.**SDLK_F8**

csdl2.**SDLK_F9**

csdl2.**SDLK_F10**

csdl2.**SDLK_F11**

csdl2.**SDLK_F12**

csdl2.**SDLK_PRINTSCREEN**

csdl2.**SDLK_SCROLLLOCK**

csdl2.**SDLK_PAUSE**

csdl2.**SDLK_INSERT**

csdl2.**SDLK_HOME**

csdl2.**SDLK_PAGEUP**

csdl2.**SDLK_DELETE**

csdl2.**SDLK_END**

csdl2.**SDLK_PAGEDOWN**

csdl2.**SDLK_RIGHT**

csdl2.**SDLK_LEFT**

csdl2.**SDLK_DOWN**

csdl2.**SDLK_UP**

csdl2.**SDLK_NUMLOCKCLEAR**

csdl2.**SDLK_KP_DIVIDE**

csdl2.**SDLK_KP_MULTIPLY**

csdl2.**SDLK_KP_MINUS**

csdl2.**SDLK_KP_PLUS**

csdl2.**SDLK_KP_ENTER**

csdl2.**SDLK_KP_1**

csdl2.**SDLK_KP_2**

csdl2.**SDLK_KP_3**

csdl2.**SDLK_KP_4**

csdl2.**SDLK_KP_5**

csdl2.**SDLK_KP_6**

csdl2.**SDLK_KP_7**

csdl2.**SDLK_KP_8**

csdl2.**SDLK_KP_9**

csdl2.**SDLK_KP_0**

csdl2.**SDLK_KP_PERIOD**

csdl2.**SDLK_APPLICATION**

csdl2.**SDLK_POWER**

csdl2.**SDLK_KP_EQUALS**

csdl2.**SDLK_F13**

csdl2.**SDLK_F14**

csdl2.**SDLK_F15**

csdl2.**SDLK_F16**

csdl2.**SDLK_F17**

csdl2.**SDLK_F18**

csdl2.**SDLK_F19**

csdl2.**SDLK_F20**

csdl2.**SDLK_F21**

csdl2.**SDLK_F22**

csdl2.**SDLK_F23**

csdl2.**SDLK_F24**

csdl2.**SDLK_EXECUTE**

csdl2.**SDLK_HELP**

csdl2.**SDLK_MENU**

csdl2.**SDLK_SELECT**

csdl2.**SDLK_STOP**

csdl2.**SDLK_AGAIN**

csdl2.**SDLK_UNDO**

csdl2.**SDLK_CUT**

csdl2.**SDLK_COPY**

csdl2.**SDLK_PASTE**

csdl2.**SDLK_FIND**

csdl2.**SDLK_MUTE**

csdl2.**SDLK_VOLUMEUP**

csdl2.**SDLK_VOLUMEDOWN**

csdl2.**SDLK_KP_COMMA**

csdl2.**SDLK_KP_EQUALSAS400**

csdl2.**SDLK_ALTERASE**

csdl2.**SDLK_SYSREQ**

csdl2.**SDLK_CANCEL**

csdl2.**SDLK_CLEAR**

csdl2.**SDLK_PRIOR**

csdl2.**SDLK_RETURN2**

csdl2.**SDLK_SEPARATOR**

csdl2.**SDLK_OUT**

csdl2.**SDLK_OPER**

csdl2.**SDLK_CLEARAGAIN**

csdl2.**SDLK_CRSEL**

csdl2.**SDLK_EXSEL**

csdl2.**SDLK_KP_00**

csdl2.**SDLK_KP_000**

csdl2.**SDLK_THOUSANDSSEPARATOR**

csdl2.**SDLK_DECIMALSEPARATOR**

csdl2.**SDLK_CURRENCYUNIT**

csdl2.**SDLK_CURRENCYSUBUNIT**

csdl2.**SDLK_KP_LEFTPAREN**

csdl2.**SDLK_KP_RIGHTPAREN**

csdl2.**SDLK_KP_LEFTBRACE**

csdl2.**SDLK_KP_RIGHTBRACE**

csdl2.**SDLK_KP_TAB**

csdl2.**SDLK_KP_BACKSPACE**

csdl2.**SDLK_KP_A**

csdl2.**SDLK_KP_B**

csdl2.**SDLK_KP_C**

csdl2.**SDLK_KP_D**

csdl2.**SDLK_KP_E**

csdl2.**SDLK_KP_F**

csdl2.**SDLK_KP_XOR**

csdl2.**SDLK_KP_POWER**

csdl2.**SDLK_KP_PERCENT**

csdl2.**SDLK_KP_LESS**

csdl2.**SDLK_KP_GREATER**

csdl2.**SDLK_KP_AMPERSAND**

csdl2.**SDLK_KP_DBLAMPERSAND**

csdl2.**SDLK_KP_VERTICALBAR**

csdl2.**SDLK_KP_DBLVERTICALBAR**

csdl2.**SDLK_KP_COLON**

csdl2.**SDLK_KP_HASH**

csdl2.**SDLK_KP_SPACE**

csdl2.**SDLK_KP_AT**

csdl2.**SDLK_KP_EXCLAM**

csdl2.**SDLK_KP_MEMSTORE**

csdl2.**SDLK_KP_MEMRECALL**

csdl2.**SDLK_KP_MEMCLEAR**

csdl2.**SDLK_KP_MEMADD**

csdl2.**SDLK_KP_MEMSUBTRACT**

csdl2.**SDLK_KP_MEMMULTIPLY**

csdl2.**SDLK_KP_MEMDIVIDE**

csdl2.**SDLK_KP_PLUSMINUS**

csdl2.**SDLK_KP_CLEAR**

csdl2.**SDLK_KP_CLEARENTRY**

csdl2.**SDLK_KP_BINARY**

csdl2.**SDLK_KP_OCTAL**

csdl2.**SDLK_KP_DECIMAL**

csdl2.**SDLK_KP_HEXADECIMAL**

csdl2.**SDLK_LCTRL**

csdl2.**SDLK_LSHIFT**

csdl2.**SDLK_LALT**

csdl2.**SDLK_LGUI**

csdl2.**SDLK_RCTRL**

csdl2.**SDLK_RSHIFT**

csdl2.**SDLK_RALT**

csdl2.**SDLK_RGUI**

csdl2.**SDLK_MODE**

csdl2.**SDLK_AUDIONEXT**

csdl2.**SDLK_AUDIOPREV**

csdl2.**SDLK_AUDIOSTOP**

csdl2.**SDLK_AUDIOPLAY**

csdl2.**SDLK_AUDIOMUTE**

csdl2.**SDLK_MEDIASELECT**

csdl2.**SDLK_WWW**

csdl2.**SDLK_MAIL**

csdl2.**SDLK_CALCULATOR**

csdl2.**SDLK_COMPUTER**

csdl2.**SDLK_AC_SEARCH**

csdl2.**SDLK_AC_HOME**

csdl2.**SDLK_AC_BACK**

csdl2.**SDLK_AC_FORWARD**

csdl2.**SDLK_AC_STOP**

csdl2.**SDLK_AC_REFRESH**

csdl2.**SDLK_AC_BOOKMARKS**

csdl2.**SDLK_BRIGHTNESSDOWN**

csdl2.**SDLK_BRIGHTNESSUP**

csdl2.**SDLK_DISPLAYSWITCH**

csdl2.**SDLK_KBDILLUMTOGGLE**

csdl2.**SDLK_KBDILLUMDOWN**

csdl2.**SDLK_KBDILLUMUP**

csdl2.**SDLK_EJECT**

csdl2.**SDLK_SLEEP**

1.10.2 Key Modifier Constants

Key Modifier masks. These constants may be OR'd together.

csdl2.**KMOD_NONE**
: 0 (no modifier is applicable)

csdl2.**KMOD_LSHIFT**
: The left Shift key is down.

csdl2.**KMOD_RSHIFT**
: The right Shift key is down.

csdl2.**KMOD_LCTRL**
: The left Ctrl (Control) key is down.

csdl2.**KMOD_RCTRL**
: The right Ctrl (Control) key is down.

csdl2.**KMOD_LALT**
: The left Alt key is down.

csdl2.**KMOD_RALT**
: The right Alt key is down.

csdl2.**KMOD_LGUI**
: The left GUI key (often the Windows key) is down.

csdl2.**KMOD_RGUI**
: The right GUI key (often the Windows key) is down.

csdl2.**KMOD_NUM**
: The Num lock key (may be located on an extended keypad) is down.

csdl2.**KMOD_CAPS**
: The Caps Lock key is down.

csdl2.**KMOD_MODE**
: The AltGr key is down.

csdl2.**KMOD_CTRL**
: `(KMOD_LCTRL | KMOD_RCTRL)`

csdl2.**KMOD_SHIFT**
: `(KMOD_LSHIFT | KMOD_RSHIFT)`

csdl2.**KMOD_ALT**
: `(KMOD_LALT | KMOD_RALT)`

csdl2.**KMOD_GUI**
: `(KMOD_LGUI | KMOD_RGUI)`

1.11 Audio Device Management, Playing and Recording

1.11.1 Audio output format

class csdl2.**SDL_AudioSpec**(*freq=0*, *format=0*, *channels=0*, *silence=0*, *samples=0*, *size=0*, *callback=None*, *userdata=None*)
: Specifies an audio output format. This is used in functions like *SDL_OpenAudioDevice()*, *SDL_OpenAudio()* for specifying the audio callback, desired and obtained audio output, and by functions like *SDL_LoadWAV()* for returning the audio data format of the wave source data.

freq
: Specifies the number of sample frames sent to the audio device per second. Common values are

11025, 22050, 44100 and 48000. Larger values produce cleaner audio, in much the same way that larger resolutions produce cleaner graphics.

format
Specifies the size and type of each sample element. One of the *Audio data format values*. See *Audio data format* for more info.

channels
Specifies the number of output channels. Supported values are 1 (mono), 2 (stereo), 4 (quad) and 6 (5.1).

silence
Calculated by `SDL_OpenAudioDevice()`. The audio device silence value.

samples
The audio buffer size in samples. A sample is a chunk of audio data of the size specified in *format* multiplied by the number of channels. Must be a power of two.

size
Calculated by `SDL_OpenAudioDevice()`. The audio buffer size in bytes.

callback
When used with `SDL_OpenAudioDevice()`, it specifies the callable to call when the audio device needs more data. The callable must have the function signature:

```
callback(userdata, stream, len) -> None
```

where *stream* is a pointer to the audio data buffer which must be filled in by the callback. *len* is the length of that buffer in bytes.

Stereo samples are stored in a LRLR ordering.

userdata
Object that is passed as the *userdata* argument to the audio callback.

1.11.2 Audio data format

The audio format is a 16-bit integer, with its bits used as follows:

Bits	Value
0-7	Sample bit size
8	Sample is float if set
12	Sample is big-endian if set
15	Sample is signed if set

Unspecified bits are always zero, but may be used in later versions of SDL.

csdl2.**SDL_AUDIO_MASK_BITSIZE**
Bitmask of the bits storing the sample bit size (bits 0-7).

csdl2.**SDL_AUDIO_MASK_DATATYPE**
Bitmask of the bit storing the sample data type flag (bit 8).

csdl2.**SDL_AUDIO_MASK_ENDIAN**
Bitmask of the bit storing the sample endianness flag (bit 12).

csdl2.**SDL_AUDIO_MASK_SIGNED**
Bitmask of the bit storing the sample sign flag (bit 15).

csdl2.**SDL_AUDIO_BITSIZE**(*x*) → int
Query the sample bit size of the audio format.

This is equivalent to the value of:

```
x & SDL_AUDIO_MASK_BITSIZE
```

Parameters `x` (*int*) – The audio format integer.

Returns The sample bit size of the audio format.

`csdl2.`**`SDL_AUDIO_ISFLOAT`**(*x*) → bool

Query whether the audio format is a floating point format.

This is equivalent to the value of:

```
bool(x & SDL_AUDIO_MASK_DATATYPE)
```

Parameters `x` (*int*) – The audio format integer.

Returns True if the audio format is a floating point format, False otherwise.

`csdl2.`**`SDL_AUDIO_ISBIGENDIAN`**(*x*) → bool

Query whether the audio format is a big endian format.

This is equivalent to the value of:

```
bool(x & SDL_AUDIO_MASK_ENDIAN)
```

Parameters `x` (*int*) – The audio format integer.

Returns True if the audio format is a big endian format, False otherwise.

`csdl2.`**`SDL_AUDIO_ISSIGNED`**(*x*) → bool

Query whether the audio format is a signed format.

This is equivalent to the value of:

```
bool(x & SDL_AUDIO_MASK_SIGNED)
```

Parameters `x` (*int*) – The audio format integer.

Returns True if the audio format is a signed format, False otherwise.

`csdl2.`**`SDL_AUDIO_ISINT`**(*x*) → bool

Query whether the audio format is an integer format.

This is equivalent to the value of:

```
not SDL_AUDIO_ISFLOAT(x)
```

Parameters `x` (*int*) – The audio format integer.

Returns True if the audio format is an integer format, False otherwise.

`csdl2.`**`SDL_AUDIO_ISLITTLEENDIAN`**(*x*) → bool

Query whether the audio format is a little endian fornmat.

This is equivalent to the value of:

```
not SDL_AUDIO_ISBIGENDIAN(x)
```

Parameters `x` (*int*) – The audio format integer.

Returns True if the audio format is a little endian format, False otherwise.

`csdl2.`**`SDL_AUDIO_ISUNSIGNED`**(*x*) → bool

Query whether the audio format is an unsigned format.

This is equivalent to the value of:

```
not SDL_AUDIO_ISSIGNED(x)
```

Parameters **x** (*int*) – The audio format integer.

Returns True if the audio format is an unsigned format, False otherwise.

Audio data format values

The following are thus the possible audio data format values:

csdl2.**AUDIO_U8**
Unsigned 8-bit samples.

csdl2.**AUDIO_S8**
Signed 8-bit samples.

csdl2.**AUDIO_S16LSB**
Signed 16-bit samples in little-endian byte order.

csdl2.**AUDIO_S16MSB**
Signed 16-bit samples in big-endian byte order.

csdl2.**AUDIO_S16SYS**
Signed 16-bit samples in native byte order.

csdl2.**AUDIO_S16**
Aliased to *AUDIO_S16LSB*.

csdl2.**AUDIO_U16LSB**
Unsigned 16-bit samples in little-endian byte order.

csdl2.**AUDIO_U16MSB**
Unsigned 16-bit samples in big-endian byte order.

csdl2.**AUDIO_U16SYS**
Unsigned 16-bit samples in native byte order.

csdl2.**AUDIO_U16**
Aliased to *AUDIO_U16LSB*.

csdl2.**AUDIO_S32LSB**
32-bit integer samples in little-endian byte order.

csdl2.**AUDIO_S32MSB**
32-bit integer samples in big-endian byte order.

csdl2.**AUDIO_S32SYS**
32-bit integer samples in native byte order.

csdl2.**AUDIO_S32**
Aliased to *AUDIO_S32LSB*.

csdl2.**AUDIO_F32LSB**
32-bit floating point samples in little-endian byte order.

csdl2.**AUDIO_F32MSB**
32-bit floating point samples in big-endian byte order.

csdl2.**AUDIO_F32SYS**
32-bit floating point samples in native byte order.

csdl2.**AUDIO_F32**
Aliased to *AUDIO_F32LSB*.

1.11.3 Audio Driver Discovery

`csdl2.SDL_GetNumAudioDrivers()` → int

Returns the number of audio drivers that SDL supports.

Returns The number of builtin audio drivers.

Return type int

`csdl2.SDL_GetAudioDriver(index: int)` → str

Use this function to get the name of a builtin audio driver. The presence of a driver in this list does not mean that it will function, it just means SDL is capable of interacting with that interface.

Parameters **index** (*int*) – Index of the audio driver. The value ranges from 0 to `SDL_GetNumAudioDrivers() - 1`.

Returns The name of the audio driver at the requested index.

Return type str

`csdl2.SDL_GetCurrentAudioDriver()`

Returns the name of the current audio driver.

Returns The name of the current audio driver, or None if no driver has been initialized.

Return type str or None

1.11.4 Initialization and Cleanup

These functions are used internally, and should not be used unless you have a specific need to specify the audio driver.

`csdl2.SDL_AudioInit(driver_name)`

Initializes a particular audio driver.

Parameters **driver_name** (*str or None*) – The name of the desired audio driver.

`csdl2.SDL_AudioQuit()`

Use this function to shut down audio if you initialized it with `SDL_AudioInit()`.

1.11.5 Audio Device Discovery

`csdl2.SDL_GetNumAudioDevices(iscapture)` → int

Query the number of audio devices.

This function may trigger a complete redetection of available hardware, which is an expensive operation.

Parameters **iscapture** (*bool*) – False to request playback devices, True to request recording devices.

Returns The number of available devices exposed by the current driver, or -1 if an explicit list of devices can't be determined.

Note: The *iscapture* parameter is for future expansion and should always be False for now.

`csdl2.SDL_GetAudioDeviceName(index, iscapture)` → str

Query the name of an audio device.

Parameters

- **index** (*int*) – The index of the audio device. The value ranges from 0 to `SDL_GetNumAudioDevices() - 1`
- **iscapture** (*bool*) – True to specify a device that has recording capability.

Returns The name of the audio device at the requested index.

Note: This function is only valid after successfully initializing the audio subsystem. The values returned by this function reflect the latest call to *SDL_GetNumAudioDevices()*. Re-call that function to re-detect available hardware.

1.11.6 Opening and Closing an Audio Device

SDL provides 2 methods for accessing audio devices. The recommended way is to open the audio device with *SDL_OpenAudioDevice()* and then control it with *SDL_PauseAudioDevice()*, *SDL_LockAudioDevice()* etc. The legacy way is to open the audio device with *SDL_OpenAudio()*, and then control it with *SDL_PauseAudio()*, *SDL_LockAudio()* etc.

Audio data is passed to the audio device through an audio callback, which is specified through the *SDL_AudioSpec.callback* attribute. Once the audio device has been opened, and the audio device unpaused, SDL will call the audio callback to fill the audio buffer with audio data as needed.

class `csdl2.`**`SDL_AudioDevice`**

An opaque handle returned by *SDL_OpenAudioDevice()* representing an opened audio device. It's destructor will call *SDL_CloseAudioDevice()*.

`csdl2.`**`SDL_OpenAudioDevice`**(*device*, *iscapture*, *desired*, *obtained*, *allowed_changes*) → SDL_AudioDevice

Opens a specific audio device.

An opened audio device starts out paused, and should be enabled for playing by calling *SDL_PauseAudioDevice()* when the audio callback function is ready to be called.

The audio callback runs in a separate thread in most cases. You can prevent race conditions between your callback and other threads without fully pausing playback with *SDL_LockAudioDevice()*.

Parameters

- **`device`** (*str or None*) – The name of the device to open as reported by *SDL_GetAudioDeviceName()*. If None, the default device is opened.
- **`iscapture`** (*bool*) – True to specify the device should be obtained for recording, not playback.
- **`desired`** (*SDL_AudioSpec*) – A *SDL_AudioSpec* specifying the audio callback and desired output format.
- **`obtained`** (*SDL_AudioSpec* or None) – If a *SDL_AudioSpec* is provided, it will be filled with the actual output format. Depending on the value of *allowed_changes*, this can differ from the *desired* *SDL_AudioSpec*.
- **`allowed_changes`** – If set to 0, SDL will transparently handle all differences between the *desired* audio output format and the actual hardware. This handling can be selectively disabled by specifying zero or more of the following flags OR'd together:
  - *SDL_AUDIO_ALLOW_FREQUENCY_CHANGE*
  - *SDL_AUDIO_ALLOW_FORMAT_CHANGE*
  - *SDL_AUDIO_ALLOW_CHANNELS_CHANGE*
  - *SDL_AUDIO_ALLOW_ANY_CHANGE*

  If these flags are set, the corresponding fields in the *obtained* *SDL_AudioSpec* will be set to the values of the actual hardware audio output format.

Returns An *SDL_AudioDevice* object representing the opened audio device.

`csdl2.`**`SDL_AUDIO_ALLOW_FREQUENCY_CHANGE`**

Allow the actual audio output frequency to differ from the desired frequency.

csdl2.**SDL_AUDIO_ALLOW_FORMAT_CHANGE**
Allow the actual audio output format to differ from the desired format.

csdl2.**SDL_AUDIO_ALLOW_CHANNELS_CHANGE**
Allow the actual number of channels to differ from the desired number of channels.

csdl2.**SDL_AUDIO_ALLOW_ANY_CHANGE**
Allow all of the above changes.

csdl2.**SDL_CloseAudioDevice**(*dev*) → None
Shuts down audio processing and closes the specified device.

There is no need to explictly call this function. *SDL_AudioDevice* will automatically call this function as part of its destructor.

Parameters **dev** (*SDL_AudioDevice*) – Audio device to close

csdl2.**SDL_OpenAudio**(*desired*, *obtained*)
Opens the audio device with the *desired* output format.

This function is a legacy means of opening the audio device. Use *SDL_OpenAudioDevice()* instead.

Parameters

- **desired** (*SDL_AudioSpec*) – Specifies the desired output format and audio callback
- **obtained** (*SDL_AudioSpec* or None) – A *SDL_AudioSpec* that will be filled in with the hardware parameters. If None, the the output format of the audio device is guaranteed to match the desired output format. SDL will convert the audio data to the actual hardware audio format if necessary. The *desired* *SDL_AudioSpec* will have its fields modified as well.

csdl2.**SDL_CloseAudio**()
Shuts down audio processing and closes the audio device.

1.11.7 Querying Playback Status

An audio device can be in any one of these 3 states:

csdl2.**SDL_AUDIO_STOPPED**
Audio device is stopped.

csdl2.**SDL_AUDIO_PLAYING**
Audio device is playing.

csdl2.**SDL_AUDIO_PAUSED**
Audio device is paused.

SDL_GetAudioStatus() and *SDL_GetAudioDeviceStatus()* can be used to query the playback status of an audio device.

csdl2.**SDL_GetAudioDeviceStatus**(*dev*) → int
Query the playback status of the specified audio device.

Parameters **dev** (*SDL_AudioDevice*) – Audio device to query.

Returns The playback status of the specified audio device, which is one of *SDL_AUDIO_STOPPED*, *SDL_AUDIO_PLAYING* or *SDL_AUDIO_PAUSED*.

csdl2.**SDL_GetAudioStatus**() → int
Query the playback status of the audio device.

This function is a legacy means of querying the audio device. Use *SDL_GetAudioDeviceStatus()* instead.

Returns The playback status of the audio device, which is one of *SDL_AUDIO_STOPPED*, *SDL_AUDIO_PLAYING* or *SDL_AUDIO_PAUSED*.

1.11.8 Controlling Playback

csdl2.**SDL_PauseAudioDevice**(*dev*, *pause_on*) → None

Pause or unpause audio playback on the specified device. When the device is paused, silence will be written to the audio device and the audio callback is guaranteed to not be called.

Pausing state does not stack. Even if the device is paused several times, a single unpause will start the device playing again, and vice versa.

If you need to protect a few variables from race conditions with the audio callback, you should not pause the audio device as it will lead to dropouts in audio playback. Instead, use `SDL_LockAudioDevice()`.

Parameters

- **dev** (`SDL_AudioDevice`) – Audio device to pause or unpause
- **pause_on** (`bool`) – If True, the audio device will be paused, otherwise the audio device will be unpaused.

csdl2.**SDL_PauseAudio**(*pause_on*)

Pause or unpause audio playback on the audio device. When the device is paused, silence will be written to the audio device and the audio callback is guaranteed to not be called.

Pausing state does not stack. Even if the device is paused several times, a single unpause will start the device playing again, and vice versa.

If you need to protect a few variables from race conditions with the audio callback, you should not pause the audio device as it will lead to dropouts in audio playback. Instead, use `SDL_LockAudio()`.

This function is a legacy means of pausing the audio device. Use `SDL_PauseAudioDevice()` instead.

Parameters **pause_on** (`bool`) – If True, the audio device will be paused, otherwise audio device will be unpaused.

1.11.9 WAVE file format support

SDL supports loading a Waveform Audio File Format (WAVE) file from a data stream.

csdl2.**SDL_LoadWAV_RW**(*src: SDL_RWops*, *freesrc: bool*)

Loads a WAVE from the data source.

Parameters

- **src** (`SDL_RWops`) – Data source for the wave file.
- **freesrc** (`bool`) – If True, the data source will be freed with `SDL_RWclose()`.

Returns

A 3-tuple (`SDL_AudioSpec`, buffer, int):

- A `SDL_AudioSpec` specifying the audio format of the wave file.
- A byte buffer containing the audio data.
- An int specifying the size of the audio data buffer in bytes.

csdl2.**SDL_LoadWAV**(*file: str*)

Loads a WAVE from a file.

Parameters **file** (`str`) – Name of the file to load

Returns

A 3-tuple (`SDL_AudioSpec`, buffer, int):

- A `SDL_AudioSpec` specifying the audio format of the wave file.
- A byte buffer containing the audio data.

- An int specifying the size of the audio data buffer in bytes.

csdl2.**SDL_FreeWAV**(*audio_buf*) → None

Frees the buffer previously allocated with *SDL_LoadWAV()* or *SDL_LoadWAV_RW()*.

There is no need to explictly call this function. The buffer returned by *SDL_LoadWAV()* or *SDL_LoadWAV_RW()* will automatically call this function as part of its destructor.

Parameters **audio_buf** (*buffer*) – Buffer created by *SDL_LoadWAV()* or *SDL_LoadWAV_RW()*.

1.11.10 Audio Data Conversion

Audio data conversion is done in 3 steps:

1. An *SDL_AudioCVT* structure is initialized with *SDL_BuildAudioCVT()*.
2. The application sets up an appropriately-sized buffer containing the source data, assigning it to *SDL_AudioCVT.buf*. The application must also set *SDL_AudioCVT.len* to the source data size in bytes. The actual size of the buffer must be at least `len * len_mult` bytes large, as the conversion will be done using this buffer.
3. The actual audio data conversion is done by calling *SDL_ConvertAudio()* with the *SDL_AudioCVT* struct. The converted audio data will be written to the provided audio buffer.

class csdl2.**SDL_AudioCVT**

A structure that contains audio data conversion information.

It is initialized with *SDL_BuildAudioCVT()*, and passed to *SDL_ConvertAudio()* to do the actual conversion once the application has set up appropriately-sized buffers between these two function calls.

conversion is done by *SDL_ConvertAudio()*

needed

(readonly) True if conversion is needed.

src_format

(readonly) Source audio format.

dst_format

(readonly) Target audio format

rate_incr

(readonly) Rate conversion increment.

buf

This attribute should point to the audio data that will be used in the conversion.

The buffer is both the source and the destination, which means the converted audio data overwrites the original data. It also means that 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 *SDL_AudioCVT.buf* is larger enough for any stage of the conversion, regardless of the final converted data's size.

The buffer must have a size of at least `len * len_mult`.

len

Length of original audio buffer in bytes.

len_cvt

(readonly) Length of converted audio buffer.

len_mult

(readonly) The length multiplier for determining the size of the converted data.

The audio buffer may need to be larger than either the original data or the converted data. The allocated size of *SDL_AudioCVT.buf* must have a size of at least `len * len_mult` bytes.

len_ratio
(readonly) The length ratio of the converted data to the original data.

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 similar to *SDL_AudioCVT.len_mult*. However, when the converted audio data is shorter than the original, *SDL_AudioCVT.len_mult* will be 1. *SDL_AudioCVT.len_ratio* on the other hand will be a fractional number between 0 and 1.

csdl2.**SDL_BuildAudioCVT**(*cvt*, *src_format*, *src_channels*, *src_rate*, *dst_format*, *dst_channels*, *dst_rate*) → bool
Initialize a *SDL_AudioCVT* structure for conversion.

Parameters

- **cvt** (*SDL_AudioCVT*) – An *SDL_AudioCVT* structure to be filled in with audio conversion information.
- **src_format** (*int*) – The source format of the audio data. One of the *Audio data format values*.
- **src_channels** (*int*) – The number of channels in the source.
- **src_rate** (*int*) – The frequency (sample-frames-per-second) of the source.
- **dst_format** (*int*) – The destination format of the audio data. One of the *Audio data format values*.
- **dst_channels** (*int*) – The number of channels in the destination.
- **dst_rate** (*int*) – The frequency (sample-frames-per-second) of the destination.

Returns True if conversion is needed, False otherwise.

Note: This function will zero out every field of the *SDL_AudioCVT*, so it must be called before the application fills in the final buffer information.

csdl2.**SDL_ConvertAudio**(*cvt*)
Convert the audio data as specified by the *SDL_AudioCVT* structure.

Parameters cvt – An *SDL_AudioCVT* structure with the information required for audio conversion.

Note: The *SDL_AudioCVT* structure must first be initialized with *SDL_BuildAudioCVT()*.

The application then needs to set the *SDL_AudioCVT* structure's *SDL_AudioCVT.buf* attribute to the audio buffer containing the source audio data, and *SDL_AudioCVT.len* attribute to the size, in bytes, of the source data.

This same buffer is used for data conversion, and will contain the converted audio data after calling this function. The converted audio data, or any of the intermediate conversion data, may be larger than the source data, and thus the actual size of the buffer must be at least `len * len_mult` bytes long.

This function will write the converted audio data to the buffer, and will set *SDL_AudioCVT.len_cvt* to the size in bytes of the converted audio data.

1.11.11 Audio Mixing

csdl2.**SDL_MixAudioFormat**(*dst*, *src*, *len*, *volume*)
Mix audio data in a specified format.

This takes a source audio buffer, and mixes it into the destination audio buffer, performing addition, volume adjustment, and overflow clipping.

This is provided for convenience – you can mix your own audio data.

Parameters

- **dst** (*buffer*) – The destination for the mixed audio.
- **src** (*buffer*) – The source audio data to be mixed in.
- **format** (*int*) – The audio format. One of the *Audio data format values*.
- **len** (*int*) – The length of the source and destination buffers in bytes.
- **volume** (*int*) – Ranges from 0 to 128, and should be set to *SDL_MIX_MAXVOLUME* for full audio volume.

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. Use mixing functions from SDL_mixer, OpenAL or write your own mixer instead.

csdl2.**SDL_MixAudio**(*dst*, *src*, *len*, *volume*)

This function is a legacy means of mixing audio, and is equivalent to calling:

```
SDL_MixAudioFormat(dst, src, format, len, volume)
```

where *format* is the obtained format of the audio device from the legacy *SDL_OpenAudio()* function.

Parameters

- **dst** (*buffer*) – The destination buffer for the mixed audio.
- **src** (*buffer*) – The source audio buffer to be mixed in.
- **len** (*int*) – The length of the source and destination buffers in bytes.
- **volume** (*int*) – Ranges from 0 to 128, and should be set to *SDL_MIX_MAXVOLUME* for full audio volume.

Note: This function requires the audio device to be open with *SDL_OpenAudio()*, and will silently fail if the audio device is not open.

csdl2.**SDL_MIX_MAXVOLUME**

The maximum volume for mixing.

1.11.12 Audio Locking

The lock manipulated by these functions protects the callback function. During a *SDL_LockAudio()*/*SDL_UnlockAudio()* or *SDL_LockAudioDevice()*/*SDL_UnlockAudioDevice()* pair, you can be guaranteed the callback function is not running. Do not call these from the callback function or you will cause deadlock.

It is safe to lock the audio device multiple times, as long as you unlock it an equivalent number of times. The audio callback will not run until the device has been unlocked completely.

csdl2.**SDL_LockAudioDevice**(*dev*)

Lock out the audio callback function for a specified audio device.

Parameters **dev** (*SDL_AudioDevice*) – The audio device to be locked.

csdl2.**SDL_UnlockAudioDevice**(*dev*)

Unlock the audio callback function for a specified audio device.

Parameters **dev** (*SDL_AudioDevice*) – The audio device to be unlocked.

`csdl2.SDL_LockAudio()`

This function is a legacy means of locking the audio device. Use *SDL_LockAudioDevice()* instead.

`csdl2.SDL_UnlockAudio()`

This function is a legacy means of unlocking the audio device. Use *SDL_UnlockAudioDevice()* instead.

1.12 File I/O Abstraction

SDL provides the RWops interface for reading from and writing to various types of data streams.

class `csdl2.SDL_RWops`

Provides an abstract interface to stream I/O.

This structure cannot be initiated directly. Instead, use *SDL_AllocRW()*, *SDL_RWFromFile()*, `SDL_RWFromFP()`, `SDL_RWFromMem()` or `SDL_RWFromConstMem()` to create an instance of this structure.

Applications shouldn't have to care about the specifics of this structure. They should treat this as an opaque object and use the *SDL_RWsize()*, *SDL_RWseek()*, `SDL_RWtell()`, *SDL_RWread()*, *SDL_RWwrite()* and *SDL_RWclose()* functions on them.

type

The type of stream. It is currently one of these values, though applications can usually ignore this information:

Identifier	Description
SDL_RWOPS_UNKNOWN	Unknown stream type or application defined.
SDL_RWOPS_WINFILE	Win32 file handle.
SDL_RWOPS_STDFILE	stdio `FILE*`
SDL_RWOPS_JNIFILE	Android asset
SDL_RWOPS_MEMORY	Memory stream (read/write)
SDL_RWOPS_MEMORY_RO	Memory stream (read-only)

Applications and libraries rolling their own implementations should use *SDL_RWOPS_UNKNOWN*. All other values are currently reserved for SDL's internal use.

size

A function that reports the stream's total size in bytes. It must the same function signature as *SDL_RWsize()*.

seek

A function that positions the next read/write operation in the stream. It must have the same function signature as *SDL_RWseek()*.

read

A function that reads from the stream. It must have the same function signature as *SDL_RWread()*.

write

A function that writes to the stream. It must have the same function signature as *SDL_RWwrite()*.

close

A function that cleans up the stream. It must release any resources used by the stream and free the *SDL_RWops* itself with *SDL_FreeRW()*. It must have the same function signature as *SDL_RWclose()*.

`csdl2.SDL_RWOPS_UNKNOWN`

Unknown stream type.

`csdl2.SDL_RWOPS_WINFILE`

Win32 file.

csdl2.**SDL_RWOPS_STDFILE**
Stdio file.

csdl2.**SDL_RWOPS_JNIFILE**
Android asset.

csdl2.**SDL_RWOPS_MEMORY**
Memory stream.

csdl2.**SDL_RWOPS_MEMORY_RO**
Read-only memory stream.

csdl2.**SDL_RWFromFile**(*file: str*, *mode: str*) → SDL_RWops
Creates and returns a *SDL_RWops* structure for reading from and/or writing to the file with name *file*.

mode is one of the following:

mode	Behavior
r	Open a file for reading. The file must exist.
w	Create an empty file for writing. If a file with the same name already exists, its contents are erased and the file is treated as a new empty file.
a	Append to a file. Writing operations append data at the end of the file. The file is created if it does not exist.
r+	Open a file for both reading and writing. The file must exist.
w+	Create an empty file for both reading and writing. If a file with the same name already exists its contents are erased and the file is treated as a new empty file.
a+	Open a file for reading and appending. All writing operations are performed at the end of the file. You can seek the internal pointer to anywhere in the file for reading, but writing operations will move it back to the end of the file. The file is created if it does not exist.

Parameters

- **file** (*str*) – File path
- **mode** (*str*) – File open mode

Returns A new *SDL_RWops* structure

csdl2.**SDL_AllocRW**() → SDL_RWops
Allocates a new *SDL_RWops* structure and returns it.

Applications do not need to use this function unless they are providing their own RWops implementation. You should use the built-in implementations in SDL, like *SDL_RWFromFile()*, SDL_RWFromMem() etc.

Returns A new *SDL_RWops* structure

csdl2.**SDL_FreeRW**(*area: SDL_RWops*) → None
Frees the *SDL_RWops* structure allocated by *SDL_AllocRW()*.

Applications do not need to use this function unless they are providing their own *SDL_RWops.close* implementation. When using the built-in implementations of *SDL_RWops* (e.g. through *SDL_RWFromFile()*, SDL_RWFromMem() etc.), you just need to call *SDL_RWclose()* with the *SDL_RWops* object, as the built-in implementations of *SDL_RWops.close* will call *SDL_FreeRW()* internally.

Parameters **area** (SDL_RWops) – The *SDL_RWops* structure allocated with *SDL_AllocRW()*.

csdl2.**SDL_RWsize**(*context: SDL_RWops*) → int
Returns the size of the data stream in the *SDL_RWops*.

Parameters **context** (SDL_RWops) – The *SDL_RWops* stream to get the size of.

Returns Size of the data stream in bytes.

`csdl2.`**`SDL_RWseek`**(*context: SDL_RWops*, *offset: int*, *whence: int*) → int

Seeks to *offset* relative to *whence*.

Parameters

- **`context`** (`SDL_RWops`) – Stream to seek in.
- **`offset`** (*int*) – New position in stream, measured in bytes, relative to *whence*.
- **`whence`** (*int*) – *RW_SEEK_SET*, *RW_SEEK_CUR* or *RW_SEEK_END*.

Returns New offset, measured in bytes, from the start of the stream.

`csdl2.`**`RW_SEEK_SET`**

Seek from the beginning of data.

`csdl2.`**`RW_SEEK_CUR`**

Seek relative to current read point.

`csdl2.`**`RW_SEEK_END`**

Seek relative to the end of data.

`csdl2.`**`SDL_RWread`**(*context: SDL_RWops*, *ptr: buffer*, *size: int*, *maxnum: int*) → int

Reads up to *maxnum* objects, each of size *size* bytes, from the data source to the buffer *ptr*.

Parameters

- **`context`** (`SDL_RWops`) – Data stream to read from.
- **`ptr`** (*buffer*) – Buffer to read data into. It must be exactly `size * maxnum` bytes.
- **`size`** (*int*) – The size of each object to read, in bytes.
- **`maxnum`** (*int*) – The maximum number of objects to be read.

Returns The number of objects read. This function may read less objects than requested.

`csdl2.`**`SDL_RWwrite`**(*context: SDL_RWops*, *ptr: buffer*, *size: int*, *num: int*) → int

Writes exactly *num* objects, each *size* bytes, from the buffer *ptr* to the stream.

Parameters

- **`context`** (`SDL_RWops`) – Data stream to write to.
- **`ptr`** (*buffer*) – Buffer containing the data to write to the stream. It must be exactly `size * num` bytes.
- **`size`** (*int*) – The size of each object to write, in bytes.
- **`maxnum`** (*int*) – The number of objects to write.

Returns The number of objects written, which can be less than *num* on error or when the end of file has been reached.

`csdl2.`**`SDL_RWclose`**(*context: SDL_RWops*) → None

Closes and cleans up the data stream. The *SDL_RWops* object will be freed.

Parameters **`context`** (`SDL_RWops`) – Data stream to close.

Note: The *SDL_RWops* object will still be freed even when an exception occurs while closing the stream.

CHAPTER 2

Indices and tables

- genindex
- search

A

B

C

D

F

G

H

K

L

M

N

P

R

S

T

U

W

X

Y