Gura Module Reference - sdl

Updated: September 22, 2013

Official site: http://www.gura-lang.org/

Table of Contents

1. About This Document

This reference explains about specification of functions and classes defined in **Gura**'s `sdl` module. Official site of SDL is http://www.libsdl.org/.

2. Reference

2.1. General

sdl.Init(flags:number)

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

- `sdl.INIT_TIMER` .. Initializes the timer subsystem.
- `sdl.INIT_AUDIO` .. Initializes the audio subsystem.
- `sdl.INIT_VIDEO` .. Initializes the video subsystem.
- `sdl.INIT_CDROM` .. Initializes the cdrom subsystem.
- `sdl.INIT_JOYSTICK` .. Initializes the joystick subsystem.
- `sdl.INIT_EVERYTHING` .. Initialize all of the above.
- `sdl.INIT_NOPARACHUTE` .. Prevents SDL from catching fatal signals.
- `sdl.INIT_EVENTTHREAD`

Return Value: Returns -1 on an error or 0 on success.

sdl.InitSubSystem(flags:number)

After SDL has been initialized with sdl.Init you may initialize uninitialized subsystems with `sdl.InitSubSystem`. The flags parameter is the same as that used in `sdl.Init`.

Return Value: Returns -1 on an error or 0 on success.

sdl.QuitSubSystem(flags:number):void

`sdl.QuitSubSystem` allows you to shut down a subsystem that has been previously initialized by sdl.Init or `sdl.InitSubSystem`. The flags tells `sdl.QuitSubSystem` which subsystems to shut down, it uses the same values that are passed to `sdl.Init`.

sdl.Quit():void

`sdl.Quit` shuts down all SDL subsystems and frees the resources allocated to them. This should always be called before you exit.

sdl.WasInit(flags:number)

`sdl.WasInit` allows you to see which SDL subsytems have been initialized. flags is a bitwise OR'd combination of the subsystems you wish to check (see sdl.Init for a list of subsystem flags).

Return Value: `sdl.WasInit` returns a bitwised OR'd combination of the initialized subsystems.

`sdl.GetError()`

`sdl.GetError` returns a string containing information about the last internal SDL error.

Return Value: `sdl.GetError` returns a string containing the last error.

2.2. Video

`sdl.GetVideoSurface() {block?}`

This function returns a `sdl.Surface` instance of the current display surface. If SDL is doing format conversion on the display surface, this function returns the publicly visible surface, not the real video surface.

`sdl.GetVideoInfo()`

This function returns a `sdl.VideoInfo` instance that contains information about the video hardware. If this is called before `sdl.SetVideoMode`, the vfmt member of the returned structure will contain the pixel format of the "best" video mode.

`sdl.VideoDriverName()`

Returns a string of the name of the initialised video driver. The driver name is a simple one word identifier like "x11" or "windib".

Return Value: Returns `nil` if video has not been initialised with `sdl.Init` or a string of the driver name otherwise.

`sdl.ListModes(format:sdl.PixelFormat:nil, flags:number)`

Return a list of available screen dimensions for the given format and video flags, sorted largest to smallest. Returns nil if there are no dimensions available for a particular format or an empty list if any dimension is okay for the given format.

`sdl.VideoModeOK(width:number, height:number, bpp:number, flags:number)`

`sdl.VideoModeOK` returns 0 if the requested mode is not supported under any bit depth, or returns the bits-per-pixel of the closest available mode with the given width, height and requested surface flags (see `sdl.SetVideoMode`).

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

The arguments to `sdl.VideoModeOK` are the same ones you would pass to `sdl.SetVideoMode`.

`sdl.SetVideoMode(width:number, height:number, bpp:number, flags:number) {block?}`

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

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

The flags parameter is the same as the flags field of the `sdl.Surface` structure. OR'd combinations of the following values are valid.

- `sdl.SWSURFACE`
- `sdl.HWSURFACE`
- `sdl.ASYNCBLIT`
- `sdl.ANYFORMAT`
- `sdl.HWPALETTE`
- `sdl.DOUBLEBUF`
- `sdl.FULLSCREEN`
- `sdl.OPENGL`
- `sdl.OPENGLBLIT`
- `sdl.RESIZABLE`
- `sdl.NOFRAME`

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

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

Return Value: The framebuffer surface, or nil if it fails. The surface returned is freed by `sdl.Quit()` and should not be freed by the caller.

`Surface#UpdateRect(x:number => 0, y:number => 0, w:number => 0, h:number => 0):void`

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

If `x`, `y`, `w` and `h` are all 0, `sdl.Surface#UpdateRect` will update the entire screen.

This function should not be called while the sdl.Surface instance is locked.

`Surface#UpdateRects(rects[]:sdl.Rect):void`

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

This function should not be called while the `sdl.Surface` instance is locked.

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

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

`Surface#Flip()`

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

The `sdl.DOUBLEBUF` flag must have been passed to `sdl.SetVideoMode`, when setting the video mode for this function to perform hardware flipping.

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

`Surface#SetColors(colors[]:sdl.Color, firstcolor:number => 0)`

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

When surface is the surface associated with the current display, the display colormap will be updated with the requested colors. If `sdl.HWPALETTE` was set in `sdl.SetVideoMode` flags, `sdl.Surface#SetColors` will always return 1, and the palette is guaranteed to be set the way you desire, even if the window colormap has to be warped or run under emulation.

The color components of a sdl.Color structure are 8-bits in size, giving you a total of 2563 =16777216 colors.

Palettized (8-bit) screen surfaces with the `sdl.HWPALETTE` flag have two palettes, a logical palette that is used for mapping blits to/from the surface and a physical palette (that determines how the hardware will map the colors to the display). `sdl.Surface#SetColors` modifies both palettes (if present), and is equivalent to calling `sdl.Surface#SetPalette` with the flags set to `(sdl.LOGPAL | sdl.PHYSPAL)`.

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

`Surface#SetPalette(flags:number, colors[]:sdl.Color, firstcolor:number => 0)`

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

Palettized (8-bit) screen surfaces with the `sdl.HWPALETTE` flag have two palettes, a logical palette that is used for mapping blits to/from the surface and a physical palette (that determines how the hardware will map the colors to the display). `sdl.BlitSurface` always uses the logical palette when blitting surfaces (if it has to convert between surface pixel formats). Because of this, it is often useful to modify only one or the other palette to achieve various special color effects (e.g., screen fading, color flashes, screen dimming).

This function can modify either the logical or physical palette by specifing `sdl.LOGPAL` or `sdl.PHYSPAL` in the flags parameter.

When surface is the surface associated with the current display, the display colormap will be updated with the requested colors. If `sdl.HWPALETTE` was set in `sdl.SetVideoMode` flags, `sdl.Surface#SetPalette` will always return 1, and the palette is guaranteed to be set the way you desire, even if the window colormap has to be warped or run under emulation.

The color components of a sdl.Color structure are 8-bits in size, giving you a total of 2563=16777216 colors.

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

`sdl.SetGamma(redgamma:number, greengamma:number, bluegamma:number)`

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

This function adjusts the gamma based on the "gamma function" parameter, you can directly specify lookup tables for gamma adjustment with `sdl.SetGammaRamp`.

Not all display hardware is able to change gamma.

`sdl.GetGammaRamp()`

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

Not all display hardware is able to change gamma.

Return Value: Returns -1 on error.

`sdl.SetGammaRamp(redtable[]:number, greentable[]:number, bluetable[]:number)`

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

This function adjusts the gamma based on lookup tables, you can also have the gamma calculated based on a "gamma function" parameter with sdl.Surface#SetGamma.

Not all display hardware is able to change gamma.

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

`PixelFormat#MapRGB(r:number, g:number, b:number)`

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

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

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

Return Value: A pixel value best approximating the given RGB color value for a given pixel format. If the pixel format bpp (color depth) is less than 32-bpp then the unused upper bits of the

return value can safely be ignored (e.g., with a 16-bpp format the return value can be assigned to a Uint16, and similarly a Uint8 for an 8-bpp format).

`PixelFormat#MapRGBA(r:number, g:number, b:number, a:number)`

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

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

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

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

`PixelFormat#GetRGB(pixel:number)`

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

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

`PixelFormat#GetRGBA(pixel:number)`

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

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

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

`sdl.CreateRGBSurface(flags:number, width:number, height:number, depth:number, Rmask:number, Gmask:number, Bmask:number, Amask:number) {block?}`

Allocate an empty surface (must be called after `sdl.SetVideoMode`)

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

- `sdl.SWSURFACE` .. SDL will create the surface in system memory. This improves the performance of pixel level access, however you may not be able to take advantage of some types of hardware blitting.
- `sdl.HWSURFACE` .. SDL will attempt to create the surface in video memory. This will allow SDL to take advantage of Video->Video blits (which are often accelerated).
- `sdl.SRCCOLORKEY` .. This flag turns on colourkeying for blits from this surface. If `sdl.HWSURFACE` is also specified and colourkeyed blits are hardware-accelerated, then SDL will attempt to place the surface in video memory. Use sdl.SetColorKey to set or clear this flag after surface creation.
- `sdl.SRCALPHA` .. This flag turns on alpha-blending for blits from this surface. If `sdl.HWSURFACE` is also specified and alpha-blending blits are hardware-accelerated, then the surface will be placed in video memory if possible. Use `sdl.Surface#SetAlpha` to set or clear this flag after surface creation.

Note: If an alpha-channel is specified (that is, if `Amask` is nonzero), then the `sdl.SRCALPHA` flag is automatically set. You may remove this flag by calling `sdl.Surface#SetAlpha` after surface creation.

Return Value: Returns the created surface, or `nil` upon error.

`sdl.CreateRGBSurfaceFrom(image:image) {block?}`

Creates an `sdl.Surface` from the provided image instance. Reference to the image instance is kept in the created `sdl.Surface` instance.

See `sdl.CreateRGBSurface` for a more detailed description of the other parameters.

Return Value: Return the created surface, or `nil` upon error.

`Surface#LockSurface()`

`sdl.Surface#LockSurface` sets up a surface for directly accessing the pixels. Between calls to `sdl.Surface#LockSurface` and `sdl.Surface#UnlockSurface`, you can write to and read from surface.pixels, using the pixel format stored in surface.format. Once you are done accessing the surface, you should use `sdl.Surface#UnlockSurface` to release it.

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

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

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

`Surface#UnlockSurface():void`

Surfaces that were previously locked using `sdl.Surface#LockSurface` must be unlocked with `sdl.Surface#UnlockSurface`. Surfaces should be unlocked as soon as possible.

It should be noted that since 1.1.8, surface locks are recursive. See `sdl.Surface#LockSurface`.

`sdl.LoadBMP(file:string) {block?}`

Loads a surface from a named Windows BMP file.

Return Value: Returns the new surface, or `nil` if there was an error.

`Surface#SaveBMP(file:string):void`

Saves the sdl.Surface surface as a Windows BMP file named file.

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

`Surface#SetColorKey(flag:number, key:number)`

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

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

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

If flag is OR'd with `sdl.RLEACCEL` then the surface will be draw using RLE acceleration when drawn with `sdl.BlitSurface`. The surface will actually be encoded for RLE acceleration the first time sdl.BlitSurface or `sdl.Surface#DisplayFormat` is called on the surface.

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

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

Surface#SetAlpha(flag:number, alpha:number)

Surface#SetClipRect(rect:sdl.Rect:nil):map:void

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

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

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

Surface#GetClipRect()

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

This returns sdl.Rect instance filled with the clipping rectangle of the surface.

Surface#ConvertSurface(fmt:sdl.PixelFormat, flag:number) {block?}

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

The flags parameter is passed to `sdl.CreateRGBSurface` and has those semantics.

This function is used internally by `sdl.Surface#DisplayFormat`.

This function can only be called after `sdl.Init`.

Return Value: Returns either `sdl.Surface` instance of the new surface, or `nil` on error.

sdl.BlitSurface(src:sdl.Surface, srcrect:sdl.Rect:nil, dst:sdl.Surface, dstrect:sdl.Rect:nil)

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

The width and height in srcrect determine the size of the copied rectangle. Only the position is used in the dstrect (the width and height are ignored).

If srcrect is `nil`, the entire surface is copied. If dstrect is `nil`, then the destination position (upper left corner) is (0, 0).

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

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

The results of blitting operations vary greatly depending on whether `sdl.SRCAPLHA` is set or not. See `sdl.Surface#SetAlpha` for an explaination of how this affects your results. Colorkeying and alpha attributes also interact with surface blitting, as the following pseudo-code should hopefully explain.

```
if (source surface has SDL_SRCALPHA set) {
    if (source surface has alpha channel (that is, format->Amask != 0))
        blit using per-pixel alpha, ignoring any colour key
    else {
        if (source surface has SDL_SRCCOLORKEY set)
            blit using the colour key AND the per-surface alpha value
        else
            blit using the per-surface alpha value
    }
} else {
    if (source surface has SDL_SRCCOLORKEY set)
        blit using the colour key
    else
        ordinary opaque rectangular blit
}
```

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

`Surface#FillRect(rect:sdl.Rect:nil, color:sdl.Color):map:void`

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

The color should be a pixel of the format used by the surface, and can be generated by the `sdl.Surface#MapRGB` or `sdl.Surface#MapRGBA` functions. If the color value contains an alpha value then the destination is simply "filled" with that alpha information, no blending takes place.

If there is a clip rectangle set on the destination (set via `sdl.Surface#SetClipRect`) then this function will clip based on the intersection of the clip rectangle and the dstrect rectangle and the dstrect rectangle will be modified to represent the area actually filled.

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

`Surface#DisplayFormat() {block?}`

This function takes a surface and copies it to a new surface of the pixel format and colors of the video framebuffer, suitable for fast blitting onto the display surface. It calls SDL*ConvertSurface*.

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

If you want an alpha channel, see `sdl.Surface#DisplayFormatAlpha`.

Return Value: It returns sdl.Surface instance on success. If the conversion fails or runs out of memory, it returns `nil`.

`Surface#DisplayFormatAlpha() {block?}`

This function takes a surface and copies it to a new surface of the pixel format and colors of the video framebuffer plus an alpha channel, suitable for fast blitting onto the display surface. It calls `SDL_ConvertSurface`.

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

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

Return Value: It returns `sdl.Surface` instance on success. If the conversion fails or runs out of memory, it returns `nil`.

`sdl.WarpMouse(x:number, y:number):void`

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

`sdl.CreateCursor(data:binary, mask:binary, w:number, h:number, hot_x:number, hot_y:number)`

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

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

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

`sdl.SetCursor(cursor:sdl.Cursor):void`

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

`sdl.GetCursor()`

Returns the currently active mouse cursor.

`sdl.ShowCursor(toggle:number)`

Toggle whether or not the cursor is shown on the screen. Passing `sdl.ENABLE` displays the cursor and passing `sdl.DISABLE` hides it. The current state of the mouse cursor can be queried by passing `sdl.QUERY`, either `sdl.DISABLE` or `sdl.ENABLE` will be returned. The cursor starts off displayed, but can be turned off.

Return Value: Returns the current state of the cursor.

`sdl.GL_GetAttribute(attr:number)`

Returns the value of the SDL/OpenGL attribute value. This is useful after a call to `sdl.SetVideoMode` to check whether your attributes have been set as you expected.

Return Value: Returns the attribute value on success, or `nil` on an error.

`sdl.GL_SetAttribute(attr:number, value:number)`

Sets the OpenGL attribute attr to value. The attributes you set don't take effect until after a call to `sdl.SetVideoMode`. You should use `sdl.GL_GetAttribute` to check the values after a `sdl.SetVideoMode` call.

Return Value: Returns 0 on success, or -1 on error.

sdl.GL_SwapBuffers():void

Swap the OpenGL buffers, if double-buffering is supported.

sdl.CreateYUVOverlay(width:number, height:number, format:number, display:sdl.Surface)

`sdl.CreateYUVOverlay` creates a YUV overlay of the specified width, height and format (see `sdl.Overlay` for a list of available formats), for the provided display. A sdl.Overlay structure is returned.

The term 'overlay' is a misnomer since, unless the overlay is created in hardware, the contents for the display surface underneath the area where the overlay is shown will be overwritten when the overlay is displayed.

Overlay#LockYUVOverlay()

Much the same as `sdl.Surface#LockSurface`, `sdl.Overlay#LockYUVOverlay` locks the overlay for direct access to pixel data.

Return Value: Returns 0 on success, or -1 on an error.

Overlay#UnlockYUVOverlay():void

The opposite to `sdl.Overlay#LockYUVOverlay`. Unlocks a previously locked overlay. An overlay must be unlocked before it can be displayed.

Overlay#DisplayYUVOverlay(dstrect:sdl.Rect)

Blit the overlay to the surface specified when it was created. The sdl.Rect structure, dstrect, specifies the position and size of the destination. If the dstrect is a larger or smaller than the overlay then the overlay will be scaled, this is optimized for 2x scaling.

Return Value: Returns 0 on success.

2.3. Window Management

sdl.WM_SetCaption(title:string, icon:string):void

Sets the title-bar and icon name of the display window.

sdl.WM_GetCaption()

Returns a list of strings of title-bar and icon name

sdl.WM_SetIcon(surface:sdl.Surface, mask?:binary)

Sets the icon for the display window. Win32 icons must be 32x32.

This function must be called before the first call to `sdl.SetVideoMode`.

The mask is a bitmask that describes the shape of the icon. If mask is omitted, then the shape is determined by the colorkey of icon, if any, or makes the icon rectangular (no transparency) otherwise.

If mask is specified, it points to a bitmap with bits set where the corresponding pixel should be visible. The format of the bitmap is as follows: Scanlines come in the usual top-down order. Each scanline consists of (width / 8) bytes, rounded up. The most significant bit of each byte represents the leftmost pixel.

sdl.WM_IconifyWindow()

If the application is running in a window managed environment SDL attempts to iconify/minimise it. If `sdl.WM_IconifyWindow` is successful, the application will receive a `sdl.APPACTIVE` loss event.

Return Value: Returns non-zero on success or 0 if iconification is not support or was refused by the window manager.

sdl.WM_ToggleFullScreen(surface:sdl.Surface)

Toggles the application between windowed and fullscreen mode, if supported. (X11 is the only target currently supported, BeOS support is experimental).

Return Value: Returns 0 on failure or 1 on success.

sdl.WM_GrabInput(mode:number)

Grabbing means that the mouse is confined to the application window, and nearly all keyboard input is passed directly to the application, and not interpreted by a window manager, if any.

When mode is `sdl.GRAB_QUERY` the grab mode is not changed, but the current grab mode is returned.

Available values for mode are:

- `sdl.GRAB_QUERY`
- `sdl.GRAB_OFF`
- `sdl.GRAB_ON`

Return Value: The current/new mode value.

2.4. Events

`sdl.PumpEvents():void`

Pumps the event loop, gathering events from the input devices.

`sdl.PumpEvents` gathers all the pending input information from devices and places it on the event queue. Without calls to `sdl.PumpEvents` no events would ever be placed on the queue. Often calls the need for `sdl.PumpEvents` is hidden from the user since `sdl.PollEvent` and `sdl.WaitEvent` implicitly call `sdl.PumpEvents`. However, if you are not polling or waiting for events (e.g. you are filtering them), then you must call `sdl.PumpEvents` to force an event queue update.

Note: You can only call this function in the thread that set the video mode.

`sdl.AddEvents(events[]:sdl.Event, mask:number)`

This calles a function `SDL_PeepEvents` with `SDL_ADDEVENT`.

`sdl.Event` instances, events, will be added to the back of the event queue.

This function is thread-safe.

`sdl.PeekEvents(numevents:number, mask:number)`

This calles a function `SDL_PeepEvents` with `SDL_PEEKEVENT`.

Up to numevents events at the front of the event queue, matching mask, will be returned and will not be removed from the queue.

The mask parameter is an bitwise OR of `sdl.EVENTMASK(event_type)`, for all event types you are interested in.

This function is thread-safe.

`sdl.GetEvents(numevents:number, mask:number)`

This calles a function `SDL_PeepEvents` with `SDL_GETEVENT`.

Up to numevents events at the front of the event queue, matching mask, will be returned and will be removed from the queue.

The mask parameter is an bitwise OR of `sdl.EVENTMASK(event_type)`, for all event types you are interested in.

This function is thread-safe.

sdl.PollEvent()

Polls for currently pending events, and returns `sdl.Event` instance if there are any pending events, or nil if there are none available.

sdl.WaitEvent()

Waits indefinitely for the next available event, returning `sdl.Event` instance, or nil if there was an error while waiting for events.

sdl.PushEvent(event:sdl.Event)

The event queue can actually be used as a two way communication channel. Not only can events be read from the queue, but the user can also push their own events onto it. event is an instance of sdl.Event you wish to push onto the queue.

Note: Pushing device input events onto the queue doesn't modify the state of the device within SDL.

Return Value: Returns 0 on success or -1 if the event couldn't be pushed.

sdl.SetEventFilter(filter:function)

This function sets up a filter to process all events before they are posted to the event queue. This is a very powerful and flexible feature. The filter is prototyped as:

```
filter(event:sdl.Event)
```

If the filter returns true, then the event will be added to the internal queue. If it returns false, then the event will be dropped from the queue. This allows selective filtering of dynamically.

There is one caveat when dealing with the `sdl.QUITEVENT` event type. The event filter is only called when the window manager desires to close the application window. If the event filter returns

true, then the window will be closed, otherwise the window will remain open if possible. If the quit event is generated by an interrupt signal, it will bypass the internal queue and be delivered to the application at the next event poll.

Note: Events pushed onto the queue with sdl.PushEvent or sdl.PeepEvents do not get passed through the event filter.

Note: Be Careful! The event filter function may run in a different thread so be careful what you do within it.

`sdl.GetEventFilter()`

This function retrieces a pointer to the event filter that was previously set using `sdl.SetEventFilter`. A filter function is defined as:

```
filter(event:sdl.Event)
```

Return Value: Returns a pointer to the event filter or nil if no filter has been set.

`sdl.EventState(type:number, state:number)`

This function allows you to set the state of processing certain event type's.

If state is set to `sdl.IGNORE`, that event type will be automatically dropped from the event queue and will not be filtered.

If state is set to `sdl.ENABLE`, that event type will be processed normally.

If state is set to `sdl.QUERY`, sdl.EventState will return the current processing state of the specified event type.

A list of event type's can be found in the `SDL_Event` section.

`sdl.CheckKeyState(key:number):map`

Check if the specified key is being pushed down. key is one of `sdl.K_*` value. It returns true if the key is down, or false otherwise.

This functions calls `SDL_GetKeyState` internally.

`sdl.GetModState()`

Returns the current state of the modifier keys (CTRL, ALT, etc.).

Return Value: The return value can be an OR'd combination of the following value.

- `sdl.KMOD_NONE`
- `sdl.KMOD_LSHIFT`
- `sdl.KMOD_RSHIFT`
- `sdl.KMOD_LCTRL`
- `sdl.KMOD_RCTRL`
- `sdl.KMOD_LALT`
- `sdl.KMOD_RALT`
- `sdl.KMOD_LMETA`
- `sdl.KMOD_RMETA`
- `sdl.KMOD_NUM`
- `sdl.KMOD_CAPS`
- `sdl.KMOD_MODE`

SDL also defines the following symbols for convenience:

- `sdl.KMOD_CTRL` (= `sdl.KMOD_LCTRL | sdl.KMOD_RCTRL`)
- `sdl.KMOD_SHIFT` (= `sdl.KMOD_LSHIFT | sdl.KMOD_RSHIFT`)
- `sdl.KMOD_ALT` (= `sdl.KMOD_LALT | sdl.KMOD_RALT`)
- `sdl.KMOD_META` (= `sdl.KMOD_LMETA | sdl.KMOD_RMETA`)

`sdl.SetModState(modstate:number):void`

The inverse of sdl.GetModState, sdl.SetModState allows you to impose modifier key states on your application.

Simply pass your desired modifier states into modstate. This value my be a logical OR'd combination of the following:

- `sdl.KMOD_NONE`
- `sdl.KMOD_LSHIFT`
- `sdl.KMOD_RSHIFT`
- `sdl.KMOD_LCTRL`
- `sdl.KMOD_RCTRL`
- `sdl.KMOD_LALT`
- `sdl.KMOD_RALT`
- `sdl.KMOD_LMETA`
- `sdl.KMOD_RMETA`
- `sdl.KMOD_NUM`
- `sdl.KMOD_CAPS`

- `sdl.KMOD_MODE`

`sdl.GetKeyName(key:number)`

Returns the SDL-defined name of the key.

`sdl.EnableUNICODE(enable:number)`

Enables/Disables Unicode keyboard translation.

To obtain the character codes corresponding to received keyboard events, Unicode translation must first be turned on using this function. The translation incurs a slight overhead for each keyboard event and is therefore disabled by default. For each subsequently received key down event, the unicode member of the `SDL_keysym` structure will then contain the corresponding character code, or zero for keysyms that do not correspond to any character code.

A value of 1 for enable enables Unicode translation; 0 disables it, and -1 leaves it unchanged (useful for querying the current translation mode).

Note that only key press events will be translated, not release events.

Return Value: Returns the previous translation mode (0 or 1).

`sdl.EnableKeyRepeat(delay:number, interval:number)`

Enables or disables the keyboard repeat rate. delay specifies how long the key must be pressed before it begins repeating, it then repeats at the speed specified by interval. Both delay and interval are expressed in milliseconds.

Setting delay to 0 disables key repeating completely. Good default values are `sdl.DEFAULT_REPEAT_DELAY` and `sdl.DEFAULT_REPEAT_INTERVAL`.

Return Value: Returns 0 on success and -1 on failure.

`sdl.GetMouseState()`

It returns a list `[button, x, y]`. `button` is a current button state as a bitmask, which can be tested using the `sdl.BUTTON(button)` function, and `x` and `y` are set to the current mouse cursor position.

`sdl.GetRelativeMouseState()`

It returns a list `[button, x, y].button` is a current button state as a bitmask, which can be tested using the `sdl.BUTTON(button)` function, and `x` and `y` are set to the change in the mouse position since the last call to `sdl.GetRelativeMouseState` or since event initialization.

sdl.GetAppState()

This function returns the current state of the application. The value returned is a bitwise combination of:

- `sdl.APPMOUSEFOCUS` .. The application has mouse focus.
- `sdl.APPINPUTFOCUS` .. The application has keyboard focus
- `sdl.APPACTIVE` .. The application is visible

sdl.JoystickEventState(state:number)

This function is used to enable or disable joystick event processing. With joystick event processing disabled you will have to update joystick states with `sdl.JoystickUpdate` and read the joystick information manually. state is either `sdl.QUERY`, `sdl.ENABLE` or `sdl.IGNORE`.

Note: Joystick event handling is prefered

Return Value: If state is `sdl.QUERY` then the current state is returned, otherwise the new processing state is returned.

2.5. Joystick

sdl.NumJoysticks()

Counts the number of joysticks attached to the system.

Return Value: Returns the number of attached joysticks.

sdl.JoystickName(index:number):map

Get the implementation dependent name of joystick. The index parameter refers to the N'th joystick on the system.

Return Value: Returns a string of the joystick name.

sdl.JoystickOpen(index:number):map

Opens a joystick for use within SDL. The index refers to the N'th joystick in the system. A joystick must be opened before it game be used.

Return Value: Returns a `sdl.Joystick` instance on success. `nil` on failure.

`sdl.JoystickOpened(index:number):map`

Determines whether a joystick has already been opened within the application. index refers to the N'th joystick on the system.

Return Value: Returns `true` if the joystick has been opened, or `false` if it has not.

`Joystick#JoystickIndex()`

Returns the index of a given `sdl.Joystick` instance.

Return Value: Index number of the joystick.

`Joystick#JoystickNumAxes()`

Return the number of axes available from a previously opened `sdl.Joystick`.

Return Value: Number of axes.

`Joystick#JoystickNumBalls()`

Return the number of trackballs available from a previously opened `sdl.Joystick`.

Return Value: Number of trackballs.

`Joystick#JoystickNumHats()`

Return the number of hats available from a previously opened `sdl.Joystick`.

Return Value: Number of hats.

`Joystick#JoystickNumButtons()`

Return the number of buttons available from a previously opened `sdl.Joystick`.

Return Value: Number of buttons.

`sdl.JoystickUpdate():void`

Updates the state(position, buttons, etc.) of all open joysticks. If joystick events have been enabled with `sdl.JoystickEventState` then this is called automatically in the event loop.

Joystick#JoystickGetAxis(axis:number)

`sdl.Joystick#JoystickGetAxis` returns the current state of the given axis on the given joystick.

On most modern joysticks the X axis is usually represented by axis 0 and the Y axis by axis 1. The value returned by sdl.Joystick#JoystickGetAxis is a signed integer (-32768 to 32768) representing the current position of the axis, it maybe necessary to impose certain tolerances on these values to account for jitter. It is worth noting that some joysticks use axes 2 and 3 for extra buttons.

Return Value: Returns a 16-bit signed integer representing the current position of the axis.

Joystick#JoystickGetHat(hat:number)

`sdl.Joystick#JoystickGetHat` returns the current state of the given hat on the given joystick.

Return Value: The current state is returned as a Uint8 which is defined as an OR'd combination of one or more of the following

- `SDL_HAT_CENTERED`
- `SDL_HAT_UP`
- `SDL_HAT_RIGHT`
- `SDL_HAT_DOWN`
- `SDL_HAT_LEFT`
- `SDL_HAT_RIGHTUP`
- `SDL_HAT_RIGHTDOWN`
- `SDL_HAT_LEFTUP`
- `SDL_HAT_LEFTDOWN`

Joystick#JoystickGetButton(button:number)

`sdl.Joystick#JoystickGetButton` returns the current state of the given button on the given joystick.

Return Value: true if the button is pressed. Otherwise, false.

Joystick#JoystickGetBall(ball:number)

Get the ball axis change.

Trackballs can only return relative motion since the last call to `sdl.Joystick#JoystickGetBall`, these motion deltas a placed into dx and dy.

Return Value: Returns `[dx,dy]` on success or `nil` on failure.

Joystick#JoystickClose():void

Close a joystick that was previously opened with `sdl.JoystickOpen`.

2.6. Audio

sdl.OpenAudio(desired:sdl.AudioSpec)

sdl.PauseAudio(pause_on:number):void

sdl.GetAudioStatus()

sdl.LoadWAV(file:string)

sdl.AudioCVT(src_format:number, src_channels:number, src_rate:number, dst_format:number, dst_channels:number, dst_rate:number)

sdl.BuildAudioCVT(src_format:number, src_channels:number, src_rate:number, dst_format:number, dst_channels:number, dst_rate:number)

sdl.LockAudio():void

sdl.UnlockAudio():void

sdl.CloseAudio():void

2.7. CD-ROM

sdl.CDNumDrives()

sdl.CDName(drive:number):map

sdl.CDOpen(drive:number)

CD#CDStatus()

This function returns the current status of the given drive. Status is described like so:

- `sdl.CD_TRAYEMPTY`
- `sdl.CD_STOPPED`
- `sdl.CD_PLAYING`

- `sdl.CD_PAUSED`
- `sdl.CD_ERROR`

`CD#CDPlay(start:number, length:number)`

Plays the given cdrom, starting a frame start for length frames.

Return Value: Returns 0 on success, or -1 on an error.

`CD#CDPlayTracks(start_track:number, start_frame:number, ntracks:number, nframes:number)`

sdl.CD#CDPlayTracks plays the given CD starting at track `start_track`, for `ntracks` tracks.

`start_frame` is the frame offset, from the beginning of the `start_track`, at which to start. `nframes` is the frame offset, from the beginning of the last track (`start_track`+`ntracks`), at which to end playing.

`sdl.CD#CDPlayTracks` should only be called after calling `sdl.CD#CDStatus` to get track information about the CD.

Note: Data tracks are ignored.

Return Value: Returns 0, or -1 if there was an error.

`CD#CDPause()`

Pauses play on the given cdrom.

Return Value: Returns 0 on success, or -1 on an error.

`CD#CDResume()`

Resumes play on the given cdrom.

Return Value: Returns 0 on success, or -1 on an error.

`CD#CDStop()`

Stops play on the given cdrom.

Return Value: Returns 0 on success, or -1 on an error.

CD#CDEject()

Ejects the given cdrom.

Return Value: Returns 0 on success, or -1 on an error.

CD#CDClose():void

Closes the given cdrom handle.

2.8. Time

sdl.GetTicks()

Get the number of milliseconds since the SDL library initialization. Note that this value wraps if the program runs for more than ~49 days.

sdl.Delay(ms:number):void

Wait a specified number of milliseconds before returning. `sdl.Delay` will wait at least the specified time, but possible longer due to OS scheduling.

Note: Count on a delay granularity of at least 10 ms. Some platforms have shorter clock ticks but this is the most common.

sdl.AddTimer(interval:number, callback?:function):[thread] {block?}

Adds a callback function to be run after the specified number of milliseconds has elapsed. The callback function is passed the current timer interval and the user supplied parameter from the `sdl.AddTimer` call and returns the next timer interval. If the returned value from the callback is the same as the one passed in, the periodic alarm continues, otherwise a new alarm is scheduled.

To cancel a currently running timer call `sdl.Timer#RemoveTimer` with the `sdl.Timer` instance returned from `sdl.AddTimer`.

The granularity of the timer is platform-dependent, but you should count on it being at least 10 ms as this is the most common number. This means that if you request a 16 ms timer, your callback will run approximately 20 ms later on an unloaded system. If you wanted to set a flag signaling a frame update at 30 frames per second (every 33 ms), you might set a timer for 30 ms (see example below). If you use this function, you need to pass `sdl.INIT_TIMER` to `sdl.Init`.

Gura: You can register the timer callback function by specifying callback function in the argument or declaring block It will be called in the same thread of event dispatching loop while you can also run it in a different thread by specying `:thread` attribute.

Return Value: Returns a `sdl.Timer` instance for the added timer.

`Timer#RemoveTimer()`

Removes a timer callback previously added with `sdl.AddTimer`.

Return Value: Returns a boolean value indicating success.