1 Getting started guide

1.1 Introduction

Welcome to Allegro 4.9!

Yes, the documentation is still very patchy. But if you're interested anyway, this short guide should point you at the parts of the API that you'll want to know about first. It's not a tutorial, as there isn't much discussion, only links into the manual. The rest you'll have to discover for yourself. Read the examples, and ask questions at Allegro.cc.

There is an unofficial tutorial at the wiki. Be aware that, being on the wiki, it may be a little out of date, but the changes should be minor. Hopefully more will sprout when things stabilise, as they did for earlier versions of Allegro.

1.2 Structure of the library and its addons

Allegro 4.9 is divided into a core library and multiple addons. The addons are bundled together and built at the same time as the core, but they are distinct and kept in separate libraries. The core doesn't depend on anything in the addons, but addons may depend on the core and other addons and additional third party libraries.

Here are the addons and their dependencies:

```
allegro_main -> allegro

allegro_image -> allegro
allegro_primitives -> allegro
allegro_color -> allegro

allegro_font -> allegro_image -> allegro
allegro_ttf -> allegro_font -> allegro_image -> allegro

allegro_audio -> allegro
allegro_flac -> allegro_audio -> allegro
allegro_vorbis -> allegro_audio -> allegro

allegro_memfile -> allegro
allegro_physfs -> allegro

allegro_native_dialog -> allegro
```

The header file for the core library is `allegro5/allegro.h`. The header files for the addons are named `allegro5/allegro_image.h`, `allegro5/allegro_font.h`, etc. The allegro_main addon does not have a header file.

1.3 Initialisation

Before using Allegro you must call al_init (19.2). Some addons have their own initialisation, e.g. al_init_image_addon (29.1), al_init_font_addon (28.1.2), al_init_ttf_addon (28.3.1).

To receive input, you need to initialise some subsystems like al_install_keyboard (11.4), al_install_mouse (14.2), al_install_joystick (10.4).

1.4 Opening a window

al_create_display (4.1.2) will open a window and return an ALLEGRO_DISPLAY (4.1.1).

To clear the display, call al_clear_to_color (9.5.1). Use al_map_rgba (9.1.4) or al_map_rgba_f (9.1.5) to obtain an ALLEGRO_COLOR (9.1.1) parameter.

Drawing operations are performed on a backbuffer. To make the operations visible, call al_flip_display (4.2.2).

1.5 Display an image

To load an image from disk, you need to have initialised the image I/O addon with al_init_image_addon (29.1). Then use al_load_bitmap (29.7), which returns an ALLEGRO_BITMAP (9.3.1).

Use al_draw_bitmap (9.5.2), al_draw_scaled_bitmap (9.5.7) or al_draw_rotated_scaled_bitmap (9.5.6) to draw the image to the backbuffer. Remember to call al_flip_display (4.2.2).

1.6 Changing the drawing target

Notice that al_clear_to_color (9.5.1) and al_draw_bitmap (9.5.2) didn't take destination parameters: the destination is implicit. Allegro remembers the current "target bitmap" for the current thread. To change the target bitmap, call al_set_target_bitmap (9.5.11).

The backbuffer of the display is also a bitmap. You can get it with al_get_backbuffer (4.2.3) and then restore it as the target bitmap.

Other bitmaps can be created with al_create_bitmap (9.3.3), with options which can be adjusted with al_set_new_bitmap_flags (9.3.8) and al_set_new_bitmap_format (9.3.9).

1.7 Event queues and input

Input comes from multiple sources: keyboard, mouse, joystick, timers, etc. Event queues aggregate events from all these sources, then you can query the queue for events.

Create an event queue with al_create_event_queue (5.8), then tell input sources to place new events into that queue using al_register_event_source (5.19). The usual input event sources can be retrieved with al_get_keyboard_event_source (11.11), al_get_mouse_event_source (14.14) and al_get_joystick_event_source (10.19).

Events can be retrieved with al_wait_for_event (5.23) or al_get_next_event (5.17). Check the event type and other fields of ALLEGRO_EVENT (5.1) to react to the input.

Displays are also event sources, which emit events when they are resized. You'll need to set the ALLEGRO_RESIZABLE flag with al_set_new_display_flags (4.1.10) before creating the display, then register the display with an event queue. When you get a resize event, call al_acknowledge_resize (4.2.1).

Timers are event sources which "tick" periodically, causing an event to be inserted into the queues that the timer is registered with. Create some with al_install_timer (22.6).

al_current_time (21.2) and al_rest (21.4) are more direct ways to deal with time.

1.8 Displaying some text

To display some text, initialise the font addon with al_init_font_addon (28.1.2) then load a bitmap font with al_load_font (28.1.4). Use al_draw_text (28.1.10) or al_draw_textf (28.1.14).

For TrueType fonts, you'll need to initialise the TTF font addon with al_init_ttf_addon (28.3.1) and load a TTF font with al_load_ttf_font (28.3.2).

1.9 Drawing primitives

The primitives addon provides some handy routines to draw lines (al_draw_line (32.2.1)), rectangles (al_draw_rectangle (32.2.4)) circles (al_draw_circle (32.2.11)), etc.

1.10 Blending

To draw translucent or tinted images or primitives, change the blender state with al_set_blender (9.6.3).

As with al_set_target_bitmap (9.5.11), this changes Allegro's internal state (for the current thread). Often you'll want to save some part of the state and restore it later. The functions al_store_state (18.4) and al_restore_state (18.3) provide a convenient way to do that.

1.11 Sound

Use al_install_audio (25.2.1) to initialize sound. This will allow loading uncompressed .wav files, for other formats you need to initialize them first, for example with al_init_ogg_vorbis_addon (26.3.1) for the .ogg format.

After that, you can simply use al_reserve_samples (25.2.4) and pass the number of sound effects typically playing at the same time. Then load your sound effects with al_load_sample (25.8.7) and play them with al_play_sample (25.4.3). To stream large pieces of music from disk, you can use al_load_audio_stream (25.8.9) so the whole piece will not have to be pre-loaded into memory.

If the above sounds too simple and you can't help but think about clipping and latency issues, don't worry. Allegro gives you full control over how much or little you want its sound system to do. The al_reserve_samples (25.2.4) function mentioned above only sets up a default mixer and a number of sample instances but you don't need to use it.

Instead, to get a "direct connection" to the sound system you would use an ALLEGRO_VOICE (25.1.12) (but depending on the platform only one such voice is guaranteed to be available and it might require a specific format of audio data). Therefore all sound can be first routed through an ALLEGRO_MIXER (25.1.5) which is connected to such a voice (or another mixer) and will mix together all sample data fed to it.

You can then directly stream real-time sample data to a mixer or a voice using an ALLEGRO_AUDIO_STREAM (25.1.11) or play complete sounds using an ALLEGRO_SAMPLE_INSTANCE (25.1.10). The latter simply points to an ALLEGRO_SAMPLE (25.1.9) and will stream it for you.

1.12 Not the end

There's a heap of stuff we haven't even mentioned yet.

Enjoy!

2 Configuration files

These functions are declared in the main Allegro header file:

```
#include <allegro5/allegro.h>
```

2.1 ALLEGRO_CONFIG

```
typedef struct ALLEGRO_CONFIG ALLEGRO_CONFIG;
```

An abstract configuration structure.

2.2 al_create_config

```
ALLEGRO_CONFIG *al_create_config(void)
```

Create an empty configuration structure.

See also: al_load_config_file (2.4), al_destroy_config (2.3)

2.3 al_destroy_config

```
void al_destroy_config(ALLEGRO_CONFIG *config)
```

Free the resources used by a configuration structure. Does nothing if passed NULL.

See also: al_create_config (2.2), al_load_config_file (2.4)

2.4 al_load_config_file

```
ALLEGRO_CONFIG *al_load_config_file(const char *filename)
```

Read a configuration file from disk. Returns NULL on error. The configuration structure should be destroyed with al_destroy_config (2.3).

See also: al_save_config_file (2.5)

2.5 al_save_config_file

```
bool al_save_config_file(const ALLEGRO_CONFIG *config, const char *filename)
```

Write out a configuration file to disk. Returns true on success, false on error.

See also: al_load_config_file (2.4)

2.6 al_add_config_section

```
void al_add_config_section(ALLEGRO_CONFIG *config, const char *name)
```

Add a section to a configuration structure with the given name. If the section already exists then nothing happens.

2.7 al_add_config_comment

```
void al_add_config_comment(ALLEGRO_CONFIG *config,
   const char *section, const char *comment)
```

Add a comment in a section of a configuration. If the section doesn't yet exist, it will be created. The section can be NULL or "" for the global section.

The comment may or may not begin with a hash character. Any newlines in the comment string will be replaced by space characters.

See also: al_add_config_section (2.6)

2.8 al_get_config_value

```
const char *al_get_config_value(const ALLEGRO_CONFIG *config,
   const char *section, const char *key)
```

Gets a pointer to an internal character buffer that will only remain valid as long as the ALLEGRO_CONFIG structure is not destroyed. Copy the value if you need a copy. The section can be NULL or "" for the global section. Returns NULL if the section or key do not exist.

See also: al_set_config_value (2.9)

2.9 al_set_config_value

```
void al_set_config_value(ALLEGRO_CONFIG *config,
   const char *section, const char *key, const char *value)
```

Set a value in a section of a configuration. If the section doesn't yet exist, it will be created. If a value already existed for the given key, it will be overwritten. The section can be NULL or "" for the global section.

For consistency with the on-disk format of config files, any leading and trailing whitespace will be stripped from the value. If you have significant whitespace you wish to preserve, you should add your own quote characters and remove them when reading the values back in.

See also: al_get_config_value (2.8)

2.10 al_get_first_config_section

```
char const *al_get_first_config_section(ALLEGRO_CONFIG const *config,
   void **iterator)
```

Returns the name of the first section in the given config file. Usually this will return an empty string for the global section. The `iterator` parameter will receive an opaque iterator which is used by al_get_next_config_section (2.11) to iterate over the remaining sections.

The returned string and the iterator are only valid as long as no change is made to the passed ALLEGRO_CONFIG.

See also: al_get_next_config_section (2.11)

2.11 al_get_next_config_section

```
char const *al_get_next_config_section(void **iterator)
```

Returns the name of the next section in the given config file. The `iterator` must have been obtained with al_get_first_config_section (2.10) first.

See also: al_get_first_config_section (2.10)

2.12 al_get_first_config_entry

```
char const *al_get_first_config_entry(ALLEGRO_CONFIG const *config,
   char const *section, void **iterator)
```

Returns the name of the first key in the given section in the given config. The `iterator` works like the one for al_get_first_config_section (2.10).

The returned string and the iterator are only valid as long as no change is made to the passed ALLEGRO_CONFIG (2.1).

See also: al_get_next_config_entry (2.13)

2.13 al_get_next_config_entry

```
char const *al_get_next_config_entry(void **iterator)
```

Returns the next key for the iterator obtained by al_get_first_config_entry (2.12).

2.14 al_merge_config

```
ALLEGRO_CONFIG *al_merge_config(const ALLEGRO_CONFIG *cfg1,
    const ALLEGRO_CONFIG *cfg2)
```

Merge two configuration structures, and return the result as a new configuration. Values in configuration 'cfg2' override those in 'cfg1'. Neither of the input configuration structures are modified. Comments from 'cfg2' are not retained.

See also: al_merge_config_into (2.15)

2.15 al_merge_config_into

```
void al_merge_config_into(ALLEGRO_CONFIG *master, const ALLEGRO_CONFIG *add)
```

Merge one configuration structure into another. Values in configuration 'add' override those in 'master'. 'master' is modified. Comments from 'add' are not retained.

See also: al_merge_config (2.14)

3 Direct3D

These functions are declared in the following header file:

```
#include <allegro5/allegro_direct3d.h>
```

3.1 al_get_d3d_device

```
LPDIRECT3DDEVICE9 al_get_d3d_device(ALLEGRO_DISPLAY *display)
```

Returns the Direct3D device of the current display. The return value is undefined if the display was not created with the Direct3D flag.

Returns: A pointer to the Direct3D device.

3.2 al_get_d3d_system_texture

```
LPDIRECT3DTEXTURE9 al_get_d3d_system_texture(ALLEGRO_BITMAP *bitmap)
```

Returns the system texture (stored with the D3DPOOL_SYSTEMMEM flags). This texture is used for the render-to-texture feature set.

Returns: A pointer to the Direct3D system texture.

3.3 al_get_d3d_video_texture

```
LPDIRECT3DTEXTURE9 al_get_d3d_video_texture(ALLEGRO_BITMAP *bitmap)
```

Returns the video texture (stored with the D3DPOOL_DEFAULT or D3DPOOL_MANAGED flags depending on whether render-to-texture is enabled or disabled respectively).

Returns: A pointer to the Direct3D video texture.

3.4 al_have_d3d_non_pow2_texture_support

```
bool al_have_d3d_non_pow2_texture_support(void)
```

Returns whether the Direct3D device supports textures whose dimensions are not powers of two.

Returns: True if device suports NPOT textures, false otherwise.

3.5 al_have_d3d_non_square_texture_support

```
bool al_have_d3d_non_square_texture_support(void)
```

Returns whether the Direct3D device supports textures that are not square.

Returns: True if the Direct3D device suports non-square textures, false otherwise.

3.6 al_get_d3d_texture_position

```
void al_get_d3d_texture_position(ALLEGRO_BITMAP *bitmap, int *u, int *v)
```

Returns the u/v coordinates for the top/left corner of the bitmap within the used texture, in pixels.

Parameters:

- bitmap - ALLEGRO_BITMAP to examine
- u - Will hold the returned u coordinate
- v - Will hold the returned v coordinate

4 Display

These functions are declared in the main Allegro header file:

```
#include <allegro5/allegro.h>
```

4.1 Display creation

4.1.1 ALLEGRO_DISPLAY

```
typedef struct ALLEGRO_DISPLAY ALLEGRO_DISPLAY;
```

An opaque type representing an open display or window.

4.1.2 al_create_display

```
ALLEGRO_DISPLAY *al_create_display(int w, int h)
```

Create a display, or window, with the specified dimensions. The parameters of the display are determined by the last calls to al_set_new_display_*. Default parameters are used if none are set explicitly. Creating a new display will automatically make it the active one, with the backbuffer selected for drawing.

Returns NULL on error.

See also: al_set_new_display_flags (4.1.10), al_set_new_display_option (4.1.7), al_set_new_display_refresh_rate (4.1.11)

4.1.3 al_destroy_display

```
void al_destroy_display(ALLEGRO_DISPLAY *display)
```

Destroy a display.

4.1.4 al_get_new_display_flags

```
int al_get_new_display_flags(void)
```

Gets the current flags used for newly created displays.

See also: al_set_new_display_flags (4.1.10), al_toggle_display_flag (4.2.19)

4.1.5 al_get_new_display_refresh_rate

```
int al_get_new_display_refresh_rate(void)
```

Gets the current refresh rate used for newly created displays.

See also: al_set_new_display_refresh_rate (4.1.11)

4.1.6 al_get_new_window_position

```
void al_get_new_window_position(int *x, int *y)
```

Gets the position where newly created non-fullscreen displays will be placed.

See also: al_set_new_window_position (4.1.12)

4.1.7 al_set_new_display_option

```
void al_set_new_display_option(int option, int value, int importance)
```

Sets an extra display option. Allegro itself will not care about those options itself, but you may want to specify them, for example if you want to use multisampling.

The 'importance' parameter can be either:

- ALLEGRO_REQUIRE - The display will not be created if the setting can not be met.
- ALLEGRO_SUGGEST - If the setting is not available, the display will be created anyway. FIXME: We need a way to query the settings back from a created display.
- ALLEGRO_DONTCARE - If you added a display option with one of the above two settings before, it will be removed again. Else this does nothing.

The supported options are:

- ALLEGRO_RED_SIZE

- ALLEGRO_GREEN_SIZE
- ALLEGRO_BLUE_SIZE
- ALLEGRO_ALPHA_SIZE
- ALLEGRO_COLOR_SIZE
- ALLEGRO_RED_SHIFT
- ALLEGRO_GREEN_SHIFT
- ALLEGRO_BLUE_SHIFT
- ALLEGRO_ALPHA_SHIFT: These settings can be used to specify the pixel layout the display should use.
- ALLEGRO_ACC_RED_SIZE
- ALLEGRO_ACC_GREEN_SIZE
- ALLEGRO_ACC_BLUE_SIZE
- ALLEGRO_ACC_ALPHA_SIZE: This and the preceding three settings can be used to define the required accumulation buffer size.
- ALLEGRO_STEREO: Whether the display is a stereo display.
- ALLEGRO_AUX_BUFFERS: Number of auxiliary buffers the display should have.
- ALLEGRO_DEPTH_SIZE: How many depth buffer (z-buffer) bits to use.
- ALLEGRO_STENCIL_SIZE: How many bits to use for the stencil buffer.
- ALLEGRO_SAMPLE_BUFFERS: Whether to use multisampling (1) or not (0).
- ALLEGRO_SAMPLES: If the above is 1, the number of samples to use per pixel. Else 0.
- ALLEGRO_RENDER_METHOD: 0 if hardware acceleration is not used with this display.
- ALLEGRO_FLOAT_COLOR: Whether to use floating point color components.
- ALLEGRO_FLOAT_DEPTH: Whether to use a floating point depth buffer.
- ALLEGRO_SINGLE_BUFFER: Whether the display uses a single buffer (1) or another update method (0).
- ALLEGRO_SWAP_METHOD: If the above is 0, this is set to 1 to indicate the display is using a copying method to make the next buffer in the flip chain available, or to 2 to indicate a flipping or other method.
- ALLEGRO_COMPATIBLE_DISPLAY: Indicates if Allegro's graphics functions can use this display. If you request a display not useable by Allegro, you can still use for example OpenGL to draw graphics.
- ALLEGRO_UPDATE_DISPLAY_REGION: Set to 1 if the current display is capable of updating just a region, and 0 if calling al_update_display_region (4.2.20) is equivalent to al_flip_display (4.2.2).
- ALLEGRO_VSYNC: Set to 1 to tell the driver to wait for vsync in al_flip_display (4.2.2), or to 2 to force vsync off.

FIXME: document them all in detail

See also: al_set_new_display_flags (4.1.10)

4.1.8 al_get_new_display_option

```
int al_get_new_display_option(int option, int *importance)
```

Retrieve an extra display setting which was previously set with al_set_new_display_option (4.1.7).

4.1.9 al_reset_new_display_options

```
void al_reset_new_display_options(void)
```

This undoes any previous calls to al_set_new_display_option (4.1.7).

4.1.10 al_set_new_display_flags

```
void al_set_new_display_flags(int flags)
```

Sets various flags for display creation. flags is a bitfield containing any reasonable combination of the following:

- ALLEGRO_WINDOWED - prefer a windowed mode
- ALLEGRO_FULLSCREEN - prefer a fullscreen mode
- ALLEGRO_FULLSCREEN_WINDOW

Make the window span the entire screen. Unlike ALLEGRO_FULLSCREEN this will never attempt to modify the screen resolution. Instead the pixel dimensions of the created display will be the same as the desktop.

The passed width and height are only used if the window is switched out of fullscreen mode later but will be ignored initially.

Under Windows and X11 a fullscreen display created with this flag will behave differently from one created with the ALLEGRO_FULLSCREEN flag - even if the ALLEGRO_FULLSCREEN display is passed the desktop dimensions. The exact difference is platform dependent, but some things which may be different is how alt-tab works, how fast you can toggle between fullscreen/windowed mode or how additional monitors behave while your display is in fullscreen mode.

- ALLEGRO_RESIZABLE - the display is resizable (only applicable if combined with ALLEGRO_WINDOWED)
- ALLEGRO_OPENGL - require the driver to provide an initialized opengl context after returning successfully
- ALLEGRO_OPENGL_3_0 - require the driver to provide an initialized opengl context compatible with OpenGL version 3.0
- ALLEGRO_OPENGL_FORWARD_COMPATIBLE - the opengl context created with ALLEGRO_OPENGL_3_0 will be forwad compatible, meaning that all of the OpenGL API declared deprecated in OpenGL 3.0 will not be supported. For such displays, option ALLEGRO_COMPATIBLE_DISPLAY will be set to false.

- ALLEGRO_DIRECT3D - require the driver to do rendering with Direct3D and provide a Direct3D device
- ALLEGRO_NOFRAME - Try to create a window without a frame (i.e. no border or titlebar). This usualy does nothing for fullscreen modes, and even in windowed moded it depends on the underlying platform whether it is supported or not.
- ALLEGRO_GENERATE_EXPOSE_EVENTS - Let the display generate expose events.

0 can be used for default values.

See also: al_set_new_display_option (4.1.7), al_get_display_option (4.2.16)

4.1.11 al_set_new_display_refresh_rate

```
void al_set_new_display_refresh_rate(int refresh_rate)
```

Sets the refresh rate to use for newly created displays. If the refresh rate is not available, al_create_display (4.1.2) will fail. A list of modes with refresh rates can be found with al_get_num_display_modes (4.3.3) and al_get_display_mode (4.3.2).

4.1.12 al_set_new_window_position

```
void al_set_new_window_position(int x, int y)
```

Sets where the top left pixel of the client area of newly created windows (non-fullscreen) will be on screen. Negative values allowed on some multihead systems.

See also: al_get_new_window_position (4.1.6)

4.2 Display operations

4.2.1 al_acknowledge_resize

```
bool al_acknowledge_resize(ALLEGRO_DISPLAY *display)
```

When the user receives a resize event from a resizable display, if they wish the display to be resized they must call this function to let the graphics driver know that it can now resize the display. Returns true on success.

Adjusts the clipping rectangle to the full size of the backbuffer.

Note that a resize event may be outdated by the time you acknowledge it; there could be further resize events generated in the meantime.

See also: al_resize_display (4.2.13), ALLEGRO_EVENT (5.1)

4.2.2 al_flip_display

```
void al_flip_display(void)
```

Copies or updates the front and back buffers so that what has been drawn previously on the currently selected display becomes visible on screen. Pointers to the special back and front buffer bitmaps remain valid and retain their semantics as back and front buffers respectively, although their contents may have changed.

Several display options change how this function behaves:

With ALLEGRO_SINGLE_BUFFER, no flipping is done. You still have to call this function to display graphics, depending on how the used graphics system works.

The ALLEGRO_SWAP_METHOD option may have additional information about what kind of operation is used internally to flip the front and back buffers.

If ALLEGRO_VSYNC is 1, this function will force waiting for vsync. If ALLEGRO_VSYNC is 2, this function will not wait for vsync. With many drivers the vsync behavior is controlled by the user and not the application, and ALLEGRO_VSYNC will not be set; in this case al_flip_display (4.2.2) will wait for vsync depending on the settings set in the system's graphics preferences.

See also: al_set_new_display_flags (4.1.10), al_set_new_display_option (4.1.7)

4.2.3 al_get_backbuffer

```
ALLEGRO_BITMAP *al_get_backbuffer(void)
```

Return a special bitmap representing the back-buffer of the current display.

Care should be taken when using the backbuffer bitmap (and its sub-bitmaps) as the source bitmap (e.g as the bitmap argument to al_draw_bitmap (9.5.2)). Only untransformed operations are hardware accelerated. This consists of al_draw_bitmap (9.5.2) and al_draw_bitmap_region (9.5.3) when the current transformation is the identity. If the tranformation is not the identity, or some other drawing operation is used, the call will be routed through the memory bitmap routines, which are slow. If you need those operations to be accelerated, then first copy a region of the backbuffer into a temporary bitmap (via the al_draw_bitmap (9.5.2) and al_draw_bitmap_region (9.5.3)), and then use that temporary bitmap as the source bitmap.

See also: al_get_frontbuffer (4.2.10)

4.2.4 al_get_current_display

```
ALLEGRO_DISPLAY *al_get_current_display(void)
```

Query for the current display in the calling thread. Returns NULL if there is none.

See also: al_set_current_display (4.2.14)

4.2.5 al_get_display_flags

```
int al_get_display_flags(void)
```

Gets the flags of the current display.

See also: al_set_new_display_flags (4.1.10)

4.2.6 al_get_display_format

```
int al_get_display_format(void)
```

Gets the pixel format of the current display.

See also: ALLEGRO_PIXEL_FORMAT (9.2.2)

4.2.7 al_get_display_height

```
int al_get_display_height(void)
```

Gets the height of the current display. This is like SCREEN_H in Allegro 4.x.

See also: al_get_display_width (4.2.9)

4.2.8 al_get_display_refresh_rate

```
int al_get_display_refresh_rate(void)
```

Gets the refresh rate of the current display.

See also: al_set_new_display_refresh_rate (4.1.11)

4.2.9 al_get_display_width

```
int al_get_display_width(void)
```

Gets the width of the current display. This is like SCREEN_W in Allegro 4.x.

See also: al_get_display_height (4.2.7)

4.2.10 al_get_frontbuffer

```
ALLEGRO_BITMAP *al_get_frontbuffer(void)
```

Return a special bitmap representing the front-buffer of the current display. This may not be supported by the driver; returns NULL in that case.

See also: al_get_backbuffer (4.2.3)

4.2.11 al_get_window_position

```
void al_get_window_position(ALLEGRO_DISPLAY *display, int *x, int *y)
```

Gets the position of a non-fullscreen display.

See also: al_set_window_position (4.2.17)

4.2.12 al_inhibit_screensaver

```
bool al_inhibit_screensaver(bool inhibit)
```

This function allows the user to stop the system screensaver from starting up if true is passed, or resets the system back to the default state (the state at program start) if false is passed. It returns true if the state was set successfully, otherwise false.

4.2.13 al_resize_display

```
bool al_resize_display(int width, int height)
```

Resize the current display. Returns true on success, or false on error. This works on both fullscreen and windowed displays, regardless of the ALLEGRO_RESIZABLE flag.

Adjusts the clipping rectangle to the full size of the backbuffer.

See also: al_acknowledge_resize (4.2.1)

4.2.14 al_set_current_display

```
bool al_set_current_display(ALLEGRO_DISPLAY *display)
```

Change the current display for the calling thread. Also sets the target bitmap to the display's backbuffer.

A display may not be "current" for multiple threads simultaneously. To stop a display being current for the calling thread, call `al_set_current_display(NULL)`. Then the display may be made current by another thread.

Returns true on success.

See also: al_get_current_display (4.2.4)

4.2.15 al_set_display_icon

```
void al_set_display_icon(ALLEGRO_BITMAP *icon)
```

Changes the icon associated with the current display (window).

Note: If the underlying OS can not use an icon with the size of the provided bitmap, it will be scaled.

TODO: Describe best practice for the size? TODO: Allow providing multiple icons in differet sizes?

4.2.16 al_get_display_option

```
int al_get_display_option(int option)
```

Return an extra display setting of the current display.

See also: al_set_new_display_option (4.1.7)

4.2.17 al_set_window_position

```
void al_set_window_position(ALLEGRO_DISPLAY *display, int x, int y)
```

Sets the position on screen of a non-fullscreen display.

See also: al_get_window_position (4.2.11)

4.2.18 al_set_window_title

```
void al_set_window_title(const char *title)
```

Set the title on a display.

4.2.19 al_toggle_display_flag

```
bool al_toggle_display_flag(int flag, bool onoff)
```

Enable or disable one of the display flags. The flags are the same as for al_set_new_display_flags (4.1.10). Note however that some of the flags cannot be changed after the display has been created.

Returns true if the driver supports toggling the specified flag else false. You can use al_get_display_flags (4.2.5) to query whether the given display property actually changed.

See also: al_set_new_display_flags (4.1.10), al_get_display_flags (4.2.5)

4.2.20 al_update_display_region

```
void al_update_display_region(int x, int y, int width, int height)
```

Does the same as al_flip_display (4.2.2), but tries to update only the specified region. With many drivers this is not possible, but for some it can improve performance.

The ALLEGRO_UPDATE_DISPLAY_REGION option (see al_get_display_option (4.2.16)) will specify the behavior of this function in the current display.

See also: al_flip_display (4.2.2), al_get_display_option (4.2.16)

4.2.21 al_wait_for_vsync

```
bool al_wait_for_vsync(void)
```

Wait for the beginning of a vertical retrace. Some driver/card/monitor combinations may not be capable of this.

Note how al_flip_display (4.2.2) usually already waits for the vertical retrace, so unless you are doing something special, there is no reason to call this function.

Returns false if not possible, true if successful.

See also: al_flip_display (4.2.2)

4.2.22 al_get_display_event_source

```
ALLEGRO_EVENT_SOURCE *al_get_display_event_source(ALLEGRO_DISPLAY *display)
```

Retrieve the associated event source.

4.3 Fullscreen display modes

4.3.1 ALLEGRO_DISPLAY_MODE

```
typedef struct ALLEGRO_DISPLAY_MODE
```

Used for display mode queries. Contains information about a supported fullscreen display mode.

```
typedef struct ALLEGRO_DISPLAY_MODE {
   int width;          // Screen width
   int height;         // Screen height
   int format;         // The pixel format of the mode
   int refresh_rate;   // The refresh rate of the mode
} ALLEGRO_DISPLAY_MODE;
```

See also: al_get_display_mode (4.3.2)

4.3.2 al_get_display_mode

```
ALLEGRO_DISPLAY_MODE *al_get_display_mode(int index, ALLEGRO_DISPLAY_MODE *mode)
```

Retrieves a display mode. Display parameters should not be changed between a call of al_get_num_display_modes (4.3.3) and al_get_display_mode (4.3.2). index must be between 0 and the number returned from al_get_num_display_modes–1. mode must be an allocated ALLEGRO_DISPLAY_MODE structure. This function will return NULL on failure, and the mode parameter that was passed in on success.

See also: ALLEGRO_DISPLAY_MODE (4.3.1), al_get_num_display_modes (4.3.3)

4.3.3 al_get_num_display_modes

```
int al_get_num_display_modes(void)
```

Get the number of available fullscreen display modes for the current set of display parameters. This will use the values set with al_set_new_display_refresh_rate (4.1.11), and al_set_new_display_flags (4.1.10) to find the number of modes that match. Settings the new display parameters to zero will give a list of all modes for the default driver.

See also: al_get_display_mode (4.3.2)

4.4 Monitors

4.4.1 ALLEGRO_MONITOR_INFO

```
typedef struct ALLEGRO_MONITOR_INFO
```

Describes a monitors size and position relative to other monitors. x1, y1 will be 0, 0 on the primary display. Other monitors can have negative values if they are to the left or above the primary display.

```
typedef struct ALLEGRO_MONITOR_INFO
{
   int x1;
   int y1;
   int x2;
   int y2;
} ALLEGRO_MONITOR_INFO;
```

See also: al_get_monitor_info (4.4.4)

4.4.2 al_get_current_video_adapter

```
int al_get_current_video_adapter(void)
```

Gets the video adapter index where new displays will be created.

See also: al_set_current_video_adapter (4.4.3)

4.4.3 al_set_current_video_adapter

```
void al_set_current_video_adapter(int adapter)
```

Sets the adapter to use for newly created displays. The adapter has a monitor attached to it. Information about the monitor can be gotten using al_get_num_video_adapters (4.4.5) and al_get_monitor_info (4.4.4).

See also: al_get_num_video_adapters (4.4.5), al_get_monitor_info (4.4.4)

4.4.4 al_get_monitor_info

```
void al_get_monitor_info(int adapter, ALLEGRO_MONITOR_INFO *info)
```

Get information about a monitor's position on the desktop. adapter is a number from 0 to al_get_num_video_adapters()–1.

See also: ALLEGRO_MONITOR_INFO (4.4.1), al_get_num_video_adapters (4.4.5)

4.4.5 al_get_num_video_adapters

```
int al_get_num_video_adapters(void)
```

Get the number of video “adapters” attached to the computer. Each video card attached to the computer counts as one or more adapters. An adapter is thus really a video port that can have a monitor connected to it.

See also: al_get_monitor_info (4.4.4)

5 Events

These functions are declared in the main Allegro header file:

```
#include <allegro5/allegro.h>
```

5.1 ALLEGRO_EVENT

```
typedef union ALLEGRO_EVENT ALLEGRO_EVENT;
```

An ALLEGRO_EVENT is a union of all builtin event structures, i.e. it is an object large enough to hold the data of any event type. All events have the following fields in common:

```
ALLEGRO_EVENT_TYPE          type;
ALLEGRO_EVENT_SOURCE *      any.source;
double                      any.timestamp;
```

By examining the type field you can then access type-specific fields. The any.source field tells you which event source generated that particular event. The any.timestamp field tells you when the event was generated. The time is referenced to the same starting point as al_current_time().

Each event is of one of the following types:

- ALLEGRO_EVENT_JOYSTICK_AXIS - a joystick axis value changed.

  Fields are:

  - joystick.stick,
  - joystick.axis,
  - joystick.pos (–1.0 to 1.0).

- ALLEGRO_EVENT_JOYSTICK_BUTTON_DOWN - a joystick button was pressed.

  Fields are:

  - joystick.button.

- ALLEGRO_EVENT_JOYSTICK_BUTTON_UP - a joystick button was released.

  Fields are:

  - joystick.button.

- ALLEGRO_EVENT_KEY_DOWN - a keyboard key was pressed.

  Fields:

  - keyboard.keycode,
  - keyboard.unichar,
  - keyboard.modifiers,
  - keyboard.display.

- ALLEGRO_EVENT_KEY_REPEAT - a typed character auto-repeated.

  Fields:

  - keyboard.keycode (ALLEGRO_KEY_*),
  - keyboard.unichar (unicode character),
  - keyboard.modifiers (ALLEGRO_KEYMOD_*),
  - keyboard.display.

- ALLEGRO_EVENT_KEY_UP - a keyboard key was released.

  Fields:

  - keyboard.keycode,
  - keyboard.display.

- ALLEGRO_EVENT_MOUSE_AXES - one or more mouse axis values changed.

  Fields:

  - mouse.x,
  - mouse.y,
  - mouse.z,
  - mouse.dx,
  - mouse.dy,
  - mouse.dz,
  - mouse.display.

  Note: Calling al_set_mouse_xy (14.11) also will result in a change of axis values, but such a change is reported with ALLEGRO_EVENT_MOUSE_WARPED events instead.

  Note: currently mouse.display may be NULL if an event is generated in response to al_set_mouse_axis (14.10).

- ALLEGRO_EVENT_MOUSE_BUTTON_DOWN - a mouse button was pressed.

  Fields:

  - mouse.x,
  - mouse.y,
  - mouse.z,
  - mouse.button,
  - mouse.display.

- ALLEGRO_EVENT_MOUSE_BUTTON_UP - a mouse button was released.

  Fields:

  - mouse.x,
  - mouse.y,
  - mouse.z,
  - mouse.button,
  - mouse.display.

- ALLEGRO_EVENT_MOUSE_WARPED - al_set_mouse_xy (14.11) was called to move the mouse. This event is identical to ALLEGRO_EVENT_MOUSE_AXES otherwise.

- ALLEGRO_EVENT_MOUSE_ENTER_DISPLAY - the mouse cursor entered a window opened by the program.

  Fields:

  - mouse.x,
  - mouse.y,
  - mouse.z,
  - mouse.display.

- ALLEGRO_EVENT_MOUSE_LEAVE_DISPLAY - the mouse cursor leave the boundaries of a window opened by the program.

  Fields:

  - mouse.x,
  - mouse.y,
  - mouse.z,
  - mouse.display.

- ALLEGRO_EVENT_TIMER - a timer counter incremented.

  Fields:

  - timer.count.

- ALLEGRO_EVENT_DISPLAY_EXPOSE - The display (or a portion thereof) has become visible. Note: The display needs to be created with ALLEGRO_GENERATE_EXPOSE_EVENTS flag.

  Fields:

  - display.x,
  - display.y,
  - display.width,
  - display.height

- ALLEGRO_EVENT_DISPLAY_RESIZE - The window has been resized.

  Fields:

  - display.x,
  - display.y,
  - display.width,
  - display.height

Note that further resize events may be generated by the time you process the event, so these fields may hold outdated information.

- ALLEGRO_EVENT_DISPLAY_CLOSE - The close button of the window has been pressed.
- ALLEGRO_EVENT_DISPLAY_LOST - Displays can be lost with some drivers (just Direct3D?). This means that rendering is impossible. The device will be restored as soon as it is possible. The program should be able to ignore this event and continue rendering however it will have no effect.
- ALLEGRO_EVENT_DISPLAY_FOUND - Generated when a lost device is regained. Drawing will no longer be a no-op.
- ALLEGRO_EVENT_DISPLAY_SWITCH_OUT - The window is no longer active, that is the user might have clicked into another window or "tabbed" away.
- ALLEGRO_EVENT_DISPLAY_SWITCH_IN - The window is the active one again.

See also: ALLEGRO_EVENT_SOURCE (5.4), ALLEGRO_EVENT_TYPE (5.5), ALLEGRO_USER_EVENT (5.2)

5.2 ALLEGRO_USER_EVENT

```
typedef struct ALLEGRO_USER_EVENT ALLEGRO_USER_EVENT;
```

An event structure that can be emitted by user event sources. These are the public fields:

```
ALLEGRO_EVENT_SOURCE *source;
intptr_t data1;
intptr_t data2;
intptr_t data3;
intptr_t data4;
```

See also: al_emit_user_event (5.13)

5.3 ALLEGRO_EVENT_QUEUE

```
typedef struct ALLEGRO_EVENT_QUEUE ALLEGRO_EVENT_QUEUE;
```

An event queue holds events that have been generated by event sources that are registered with the queue. Events are stored in the order they are generated. Access is in a strictly FIFO (first-in-first-out) order.

See also: al_create_event_queue (5.8), al_destroy_event_queue (5.10)

5.4 ALLEGRO_EVENT_SOURCE

```
typedef struct ALLEGRO_EVENT_SOURCE ALLEGRO_EVENT_SOURCE;
```

An event source is any object which can generate events. For example, an ALLEGRO_DISPLAY can generate events, and you can get the ALLEGRO_EVENT_SOURCE pointer from an ALLEGRO_DISPLAY with al_get_display_event_source (4.2.22).

You may create your own "user" event sources that emit custom events.

See also: ALLEGRO_EVENT (5.1), al_init_user_event_source (5.9), al_emit_user_event (5.13)

5.5 ALLEGRO_EVENT_TYPE

```
typedef unsigned int ALLEGRO_EVENT_TYPE;
```

An integer used to distinguish between different types of events.

See also: ALLEGRO_EVENT (5.1), ALLEGRO_GET_EVENT_TYPE (5.6), ALLEGRO_EVENT_TYPE_IS_USER (5.7)

5.6 ALLEGRO_GET_EVENT_TYPE

```
#define ALLEGRO_GET_EVENT_TYPE(a, b, c, d)   AL_ID(a, b, c, d)
```

Make an event type identifier, which is a 32-bit integer. Usually this will be made from four 8-bit character codes, for example:

```
#define MY_EVENT_TYPE   ALLEGRO_GET_EVENT_TYPE('M','I','N','E')
```

You should try to make your IDs unique so they don't clash with any 3rd party code you may be using.

IDs less than 1024 are reserved for Allegro or its addons.

See also: ALLEGRO_EVENT (5.1), ALLEGRO_EVENT_TYPE_IS_USER (5.7)

5.7 ALLEGRO_EVENT_TYPE_IS_USER

```
#define ALLEGRO_EVENT_TYPE_IS_USER(t)        ((t) >= 512)
```

A macro which evaluates to true if the event type is not a builtin event type, i.e. one of those described in ALLEGRO_EVENT_TYPE (5.5).

5.8 al_create_event_queue

```
ALLEGRO_EVENT_QUEUE *al_create_event_queue(void)
```

Create a new, empty event queue, returning a pointer to object if successful. Returns NULL on error.

See also: ALLEGRO_EVENT_QUEUE (5.3)

5.9 al_init_user_event_source

```
void al_init_user_event_source(ALLEGRO_EVENT_SOURCE *src)
```

Initialise an event source for emitting user events. The space for the event source must already have been allocated.

One possible way of creating custom event sources is to derive other structures with ALLEGRO_EVENT_SOURCE at the head, e.g.

```
typedef struct THING THING;

struct THING {
    ALLEGRO_EVENT_SOURCE event_source;
    int field1;
    int field2;
    /* etc. */
};

THING *create_thing(void)
{
    THING *thing = malloc(sizeof(THING));

    if (thing) {
        al_init_user_event_source(&thing->event_source);
        thing->field1 = 0;
        thing->field2 = 0;
    }

    return thing;
}
```

The advantage here is that the THING pointer will be the same as the ALLEGRO_EVENT_SOURCE pointer. Events emitted by the event source will have the event source pointer as the `source` field, from which you can get a pointer to a THING by a simple cast (after ensuring checking the event is of the correct type).

However, it is only one technique and you are not obliged to use it.

See also: ALLEGRO_EVENT_SOURCE (5.4), al_emit_user_event (5.13), al_destroy_user_event_source (5.11)

5.10 al_destroy_event_queue

```
void al_destroy_event_queue(ALLEGRO_EVENT_QUEUE *queue)
```

Destroy the event queue specified. All event sources currently registered with the queue will be automatically unregistered before the queue is destroyed.

See also: ALLEGRO_EVENT_QUEUE (5.3)

5.11 al_destroy_user_event_source

```
void al_destroy_user_event_source(ALLEGRO_EVENT_SOURCE *src)
```

Destroy an event source initialised with al_init_user_event_source (5.9).

See also: ALLEGRO_EVENT_SOURCE (5.4)

5.12 al_drop_next_event

```
bool al_drop_next_event(ALLEGRO_EVENT_QUEUE *queue)
```

Drop the next event from the queue. If the queue is empty, nothing happens. Returns true iff an event was dropped.

5.13 al_emit_user_event

```
bool al_emit_user_event(ALLEGRO_EVENT_SOURCE *src,
   ALLEGRO_EVENT *event, void (*dtor)(ALLEGRO_USER_EVENT *))
```

Emit a user event. The event source must have been initialised with al_init_user_event_source (5.9). Some fields of the event being passed in may be modified. Returns `false` if the event source isn't registered with any queues, hence the event wouldn't have been delivered into any queues.

Reference counting will be performed on the event if `dtor` is non-NULL. When the reference count drops to zero `dtor` will be called with a copy of the event as an argument. It should free the resources associated with the event. If `dtor` is NULL then reference counting will not be performed.

You need to call al_unref_user_event (5.21) when you are done with a reference counted user event that you have gotten from al_get_next_event (5.17), al_peek_next_event (5.18), al_wait_for_event (5.23), etc. You may, but do not need to, call al_unref_user_event (5.21) on non-reference counted user events.

See also: ALLEGRO_USER_EVENT (5.2)

5.14 al_event_queue_is_empty

```
bool al_event_queue_is_empty(ALLEGRO_EVENT_QUEUE *queue)
```

Return true if the event queue specified is currently empty.

5.15 al_flush_event_queue

```
void al_flush_event_queue(ALLEGRO_EVENT_QUEUE *queue)
```

Drops all events, if any, from the queue.

5.16 al_get_event_source_data

```
intptr_t al_get_event_source_data(const ALLEGRO_EVENT_SOURCE *source)
```

Returns the abstract user data associated with the event source. If no data was previously set, returns NULL.

See also: al_set_event_source_data (5.20)

5.17 al_get_next_event

```
bool al_get_next_event(ALLEGRO_EVENT_QUEUE *queue, ALLEGRO_EVENT *ret_event)
```

Take the next event out of the event queue specified, and copy the contents into `ret_event`, returning true. The original event will be removed from the queue. If the event queue is empty, return false and the contents of `ret_event` are unspecified.

See also: ALLEGRO_EVENT (5.1)

5.18 al_peek_next_event

```
bool al_peek_next_event(ALLEGRO_EVENT_QUEUE *queue, ALLEGRO_EVENT *ret_event)
```

Copy the contents of the next event in the event queue specified into `ret_event` and return true. The original event packet will remain at the head of the queue. If the event queue is actually empty, this function returns false and the contents of `ret_event` are unspecified.

See also: ALLEGRO_EVENT (5.1)

5.19 al_register_event_source

```
void al_register_event_source(ALLEGRO_EVENT_QUEUE *queue,
   ALLEGRO_EVENT_SOURCE *source)
```

Register the event source with the event queue specified. An event source may be registered with any number of event queues simultaneously, or none. Trying to register an event source with the same event queue more than once does nothing.

See also: ALLEGRO_EVENT_QUEUE (5.3), ALLEGRO_EVENT_SOURCE (5.4)

5.20 al_set_event_source_data

```
void al_set_event_source_data(ALLEGRO_EVENT_SOURCE *source, intptr_t data)
```

Assign the abstract user data to the event source. Allegro does not use the data internally for anything; it is simply meant as a convenient way to associate your own data or objects with events.

See also: al_get_event_source_data (5.16)

5.21 al_unref_user_event

```
void al_unref_user_event(ALLEGRO_USER_EVENT *event)
```

Unreference a user-defined event. This must be called on any user event that you get from al_get_next_event (5.17), al_peek_next_event (5.18), al_wait_for_event (5.23), etc. which is reference counted. This function does nothing if the event is not reference counted.

See also: al_emit_user_event (5.13).

5.22 al_unregister_event_source

```
void al_unregister_event_source(ALLEGRO_EVENT_QUEUE *queue,
   ALLEGRO_EVENT_SOURCE *source)
```

Unregister an event source with an event queue. If the event source is not actually registered with the event queue, nothing happens.

If the queue had any events in it which originated from the event source, they will no longer be in the queue after this call.

5.23 al_wait_for_event

```
void al_wait_for_event(ALLEGRO_EVENT_QUEUE *queue, ALLEGRO_EVENT *ret_event)
```

Wait until the event queue specified is non-empty. If `ret_event` is not NULL, the first event in the queue will be copied into `ret_event` and removed from the queue. If `ret_event` is NULL the first event is left at the head of the queue.

See also: ALLEGRO_EVENT (5.1), al_wait_for_event_timed (5.24), al_wait_for_event_until (5.25)

5.24 al_wait_for_event_timed

```
bool al_wait_for_event_timed(ALLEGRO_EVENT_QUEUE *queue,
   ALLEGRO_EVENT *ret_event, float secs)
```

Wait until the event queue specified is non-empty. If `ret_event` is not NULL, the first event in the queue will be copied into `ret_event` and removed from the queue. If `ret_event` is NULL the first event is left at the head of the queue.

`timeout_msecs` determines approximately how many seconds to wait. If the call times out, false is returned. Otherwise true is returned.

See also: ALLEGRO_EVENT (5.1), al_wait_for_event (5.23), al_wait_for_event_until (5.25)

5.25 al_wait_for_event_until

```
bool al_wait_for_event_until(ALLEGRO_EVENT_QUEUE *queue,
   ALLEGRO_EVENT *ret_event, ALLEGRO_TIMEOUT *timeout)
```

Wait until the event queue specified is non-empty. If `ret_event` is not NULL, the first event in the queue will be copied into `ret_event` and removed from the queue. If `ret_event` is NULL the first event is left at the head of the queue.

`timeout` determines how long to wait. If the call times out, false is returned. Otherwise true is returned.

See also: ALLEGRO_EVENT (5.1), ALLEGRO_TIMEOUT (21.1), al_init_timeout (21.3), al_wait_for_event (5.23), al_wait_for_event_timed (5.24)

6 File I/O

These functions are declared in the main Allegro header file:

```
#include <allegro5/allegro.h>
```

6.1 ALLEGRO_FILE

```
typedef struct ALLEGRO_FILE ALLEGRO_FILE;
```

An opaque object representing an open file. This could be a real file on disk or a virtual file.

6.2 ALLEGRO_FILE_INTERFACE

```
typedef struct ALLEGRO_FILE_INTERFACE
```

A structure containing function pointers to handle a type of "file", real or virtual. See the full discussion in al_set_new_file_interface (6.29.1).

The fields are:

```
ALLEGRO_FILE* (*fi_fopen)(const char *path, const char *mode);
void          (*fi_fclose)(ALLEGRO_FILE *handle);
size_t        (*fi_fread)(ALLEGRO_FILE *f, void *ptr, size_t size);
size_t        (*fi_fwrite)(ALLEGRO_FILE *f, const void *ptr, size_t size);
bool          (*fi_fflush)(ALLEGRO_FILE *f);
int64_t       (*fi_ftell)(ALLEGRO_FILE *f);
bool          (*fi_fseek)(ALLEGRO_FILE *f, int64_t offset, int whence);
bool          (*fi_feof)(ALLEGRO_FILE *f);
bool          (*fi_ferror)(ALLEGRO_FILE *f);
int           (*fi_fungetc)(ALLEGRO_FILE *f, int c);
off_t         (*fi_fsize)(ALLEGRO_FILE *f);
```

6.3 ALLEGRO_SEEK

```
typedef enum ALLEGRO_SEEK
```

- ALLEGRO_SEEK_SET - Seek to pos from beginning of file
- ALLEGRO_SEEK_CUR - Seek to pos from curent position
- ALLEGRO_SEEK_END - Seek to pos from end of file

6.4 al_fopen

```
ALLEGRO_FILE *al_fopen(const char *path, const char *mode)
```

Creates and opens a file (real or virtual) given the path and mode. The current file interface is used to open the file.

'path' - the path to open

'mode' - mode to open the entry in ("r", "w", etc.)

Depending on the stream type and the mode string, files may be opened in "text" mode. The handling of newlines is particularly important. For example, using the default stdio-based streams on DOS and Windows platforms, where the native end-of-line terminators are CR+LF sequences, a call to al_fgetc (6.15) may return just one character ('\n') where there were two bytes (CR+LF) in the file. When writing out '\n', two bytes would be written instead. (As an aside, '\n' is not defined to be equal to LF either.)

Newline translations can be useful for text files but is disastrous for binary files. To avoid this behaviour you need to open file streams in binary mode by using a mode argument containing a "b", e.g. "rb", "wb".

See also: al_set_new_file_interface (6.29.1).

6.5 al_fclose

```
void al_fclose(ALLEGRO_FILE *f)
```

Close the given file.

6.6 al_fread

```
size_t al_fread(ALLEGRO_FILE *f, void *ptr, size_t size)
```

Read 'size' bytes into 'ptr' from entry 'fp'

Return number of bytes actually read.

6.7 al_fwrite

```
size_t al_fwrite(ALLEGRO_FILE *f, const void *ptr, size_t size)
```

Write 'size' bytes from 'ptr' into file 'fp'

Return number of bytes actually written or 0 on error.

Does not distinguish between EOF and other errors. Use al_feof (6.11) and al_ferror (6.12) to tell them apart.

6.8 al_fflush

```
bool al_fflush(ALLEGRO_FILE *f)
```

Flush any pending writes to 'fp' to disk.

Returns true on success, false otherwise, and errno is set to indicate the error.

See also: al_get_errno (18.5)

6.9 al_ftell

```
int64_t al_ftell(ALLEGRO_FILE *f)
```

Returns the current position in file, or −1 on error. errno is set to indicate the error.

On some platforms this function may not support large files.

See also: al_get_errno (18.5)

6.10 al_fseek

```
bool al_fseek(ALLEGRO_FILE *f, int64_t offset, int whence)
```

Seek to 'offset' in file based on 'whence'.

'whence' can be:

- ALLEGRO_SEEK_SET - Seek from beggining of file
- ALLEGRO_SEEK_CUR - Seek from current position
- ALLEGRO_SEEK_END - Seek from end of file

Returns true on success, false on failure and errno is set to indicate the error.

On some platforms this function may not support large files.

See also: al_get_errno (18.5)

6.11 al_feof

```
bool al_feof(ALLEGRO_FILE *f)
```

Returns true if the end-of-file indicator has been set on the file, i.e. we have attempted to read *past* the end of the file.

This does *not* return true if we simply are at the end of the file. The following code correctly reads two bytes, even when the file contains exactly two bytes:

```
int b1 = al_fgetc(f);
int b2 = al_fgetc(f);
if (al_feof(f)) {
   /* At least one byte was unsuccessfully read. */
   report_error();
}
```

See also: al_ferror (6.12)

6.12 al_ferror

```
bool al_ferror(ALLEGRO_FILE *f)
```

Returns true if there was some sort of previous error.

See also: al_feof (6.11)

6.13 al_fungetc

```
int al_fungetc(ALLEGRO_FILE *f, int c)
```

Ungets a single byte from a file. Does not write to file, it only places the char back into the entry's buffer.

See also: al_fgetc (6.15), al_get_errno (18.5)

6.14 al_fsize

```
int64_t al_fsize(ALLEGRO_FILE *f)
```

Return the size of the file, if it can be determined, or −1 otherwise.

6.15 al_fgetc

```
int al_fgetc(ALLEGRO_FILE *f)
```

Read and return next byte in entry 'f'. Returns EOF on end of file or if an error occurred.

See also: al_fungetc (6.13)

6.16 al_fputc

```
int al_fputc(ALLEGRO_FILE *f, int c)
```

Write a single byte to entry.

Parameters:

- c - byte value to write
- f - entry to write to

Returns: EOF on error

6.17 al_fread16le

```
int16_t al_fread16le(ALLEGRO_FILE *f)
```

Reads a 16-bit word in little-endian format (LSB first).

On success, returns the 16-bit word. On failure, returns EOF (–1). Since –1 is also a valid return value, use al_feof (6.11) to check if the end of the file was reached prematurely, or al_ferror (6.12) to check if an error occurred.

See also: al_fread16be (6.18)

6.18 al_fread16be

```
int16_t al_fread16be(ALLEGRO_FILE *f)
```

Reads a 16-bit word in big-endian format (MSB first).

On success, returns the 16-bit word. On failure, returns EOF (–1). Since –1 is also a valid return value, use al_feof (6.11) to check if the end of the file was reached prematurely, or al_ferror (6.12) to check if an error occurred.

See also: al_fread16le (6.17)

6.19 al_fwrite16le

```
size_t al_fwrite16le(ALLEGRO_FILE *f, int16_t w)
```

Writes a 16-bit word in little-endian format (LSB first).

Returns the number of bytes written: 2 on success, less than 2 on an error.

See also: al_fwrite16be (6.20)

6.20 al_fwrite16be

```
size_t al_fwrite16be(ALLEGRO_FILE *f, int16_t w)
```

Writes a 16-bit word in big-endian format (MSB first).

Returns the number of bytes written: 2 on success, less than 2 on an error.

See also: al_fwrite16le (6.19)

6.21 al_fread32le

```
int32_t al_fread32le(ALLEGRO_FILE *f)
```

Reads a 32-bit word in little-endian format (LSB first).

On success, returns the 32-bit word. On failure, returns EOF (–1). Since –1 is also a valid return value, use al_feof (6.11) to check if the end of the file was reached prematurely, or al_ferror (6.12) to check if an error occurred.

See also: al_fread32be (6.22)

6.22 al_fread32be

```
int32_t al_fread32be(ALLEGRO_FILE *f)
```

Read a 32-bit word in big-endian format (MSB first).

On success, returns the 32-bit word. On failure, returns EOF (–1). Since –1 is also a valid return value, use al_feof (6.11) to check if the end of the file was reached prematurely, or al_ferror (6.12) to check if an error occurred.

See also: al_fread32le (6.21)

6.23 al_fwrite32le

```
size_t al_fwrite32le(ALLEGRO_FILE *f, int32_t l)
```

Writes a 32-bit word in little-endian format (LSB first).

Returns the number of bytes written: 4 on success, less than 4 on an error.

See also: al_fwrite32be (6.24)

6.24 al_fwrite32be

```
size_t al_fwrite32be(ALLEGRO_FILE *f, int32_t l)
```

Writes a 32-bit word in big-endian format (MSB first).

Returns the number of bytes written: 4 on success, less than 4 on an error.

See also: al_fwrite32le (6.23)

6.25 al_fgets

```
char *al_fgets(ALLEGRO_FILE *f, char * const buf, size_t max)
```

Read a string of bytes terminated with a newline or end-of-file into the buffer given. The line terminator(s), if any, are included in the returned string. A maximum of max–1 bytes are read, with one byte being reserved for a NUL terminator.

Parameters:

- f - file to read from
- buf - buffer to fill
- max - maximum size of buffer

Returns the pointer to buf on success. Returns NULL if an error occurred or if the end of file was reached without reading any bytes.

See al_fopen (6.4) about translations of end-of-line characters.

6.26 al_fget_ustr

```
ALLEGRO_USTR *al_fget_ustr(ALLEGRO_FILE *f)
```

Read a string of bytes terminated with a newline or end-of-file. The line terminator(s), if any, are included in the returned string.

On success returns a pointer to a new ALLEGRO_USTR structure. This must be freed eventually with al_ustr_free (24.3.4). Returns NULL if an error occurred or if the end of file was reached without reading any bytes.

See al_fopen (6.4) about translations of end-of-line characters.

6.27 al_fputs

```
int al_fputs(ALLEGRO_FILE *f, char const *p)
```

Writes a string to file. Apart from the return value, this is equivalent to:

```
al_fwrite(f, p, strlen(p));
```

Parameters:

- f - file handle to write to
- p - string to write

Returns a non-negative integer on success, EOF on error.

Note: depending on the stream type and the mode passed to al_fopen (6.4), newline characters in the string may or may not be automatically translated to native end-of-line sequences, e.g. CR/LF instead of LF.

6.28 Standard I/O specific routines

6.28.1 al_fopen_fd

```
ALLEGRO_FILE *al_fopen_fd(int fd, const char *mode)
```

Create an ALLEGRO_FILE (6.1) object that operates on an open file descriptor using stdio routines. See the documentation of fdopen() for a description of the 'mode' argument.

Returns an ALLEGRO_FILE object on success or NULL on an error. On an error, the Allegro errno will be set and the file descriptor will not be closed.

The file descriptor will be closed by al_fclose (6.5) so you should not call close() on it.

6.28.2 al_make_temp_file

```
ALLEGRO_FILE *al_make_temp_file(const char *template, ALLEGRO_PATH **ret_path)
```

Make a temporary randomly named file given a filename 'template'.

'template' is a string giving the format of the generated filename and should include one or more capital Xs. The Xs are replaced with random alphanumeric characters. There should be no path separators.

If 'ret_path' is not NULL, the address it points to will be set to point to a new path structure with the name of the temporary file.

Returns the opened ALLEGRO_FILE (6.1) on success, NULL on failure.

6.29 Alternative file streams

By default, the Allegro file I/O routines use the C library I/O routines, hence work with files on the local filesystem, but can be overridden so that you can read and write to other streams. For example, you can work with block of memory or sub-files inside .zip files.

There are two ways to get an ALLEGRO_FILE (6.1) that doesn't use stdio. An addon library may provide a function that returns a new ALLEGRO_FILE directly, after which, all al_f* calls on that object will use overridden functions for that type of stream. Alternatively, al_set_new_file_interface (6.29.1) changes which function will handle the following al_fopen (6.4) calls for the current thread.

6.29.1 al_set_new_file_interface

```
void al_set_new_file_interface(const ALLEGRO_FILE_INTERFACE *file_interface)
```

Set the ALLEGRO_FILE_INTERFACE (6.2) table for the calling thread. This will change the handler for later calls to al_fopen (6.4).

See also: al_set_standard_file_interface (6.29.2), al_store_state (18.4), al_restore_state (18.3).

6.29.2 al_set_standard_file_interface

```
void al_set_standard_file_interface(void)
```

Set the ALLEGRO_FILE_INTERFACE (6.2) table to the default, for the calling thread. This will change the handler for later calls to al_fopen (6.4).

See also: al_set_new_file_interface (6.29.1)

6.29.3 al_get_new_file_interface

```
const ALLEGRO_FILE_INTERFACE *al_get_new_file_interface(void)
```

Return a pointer to the ALLEGRO_FILE_INTERFACE (6.2) table in effect for the calling thread.

See also: al_store_state (18.4), al_restore_state (18.3).

7 File system

These functions are declared in the main Allegro header file:

```
#include <allegro5/allegro.h>
```

These functions allow access to the filesystem. This can either be the real filesystem like your harddrive, or a virtual filesystem like a .zip archive (or whatever else you or an addon makes it do).

7.1 ALLEGRO_FS_ENTRY

```
typedef struct ALLEGRO_FS_ENTRY ALLEGRO_FS_ENTRY;
```

Opaque filesystem entry object. Represents a file or a directory (check with al_fs_entry_is_directory (7.14) or al_fs_entry_is_file (7.13)). There are no user accessible member variables.

7.2 ALLEGRO_FILE_MODE

```
typedef enum ALLEGRO_FILE_MODE
```

Filesystem modes/types

- ALLEGRO_FILEMODE_READ - Readable
- ALLEGRO_FILEMODE_WRITE - Writable
- ALLEGRO_FILEMODE_EXECUTE - Executable
- ALLEGRO_FILEMODE_HIDDEN - Hidden
- ALLEGRO_FILEMODE_ISFILE - Regular file
- ALLEGRO_FILEMODE_ISDIR - Directory

7.3 al_create_fs_entry

```
ALLEGRO_FS_ENTRY *al_create_fs_entry(const char *path)
```

Creates an ALLEGRO_FS_ENTRY (7.1) object pointing to path on the filesystem. 'path' can be a file or a directory and must not be NULL.

7.4 al_destroy_fs_entry

```
void al_destroy_fs_entry(ALLEGRO_FS_ENTRY *fh)
```

Destroys a fs entry handle. The file or directory represented by it is not destroyed.

Does nothing if passed NULL.

7.5 al_get_fs_entry_name

```
const ALLEGRO_PATH *al_get_fs_entry_name(ALLEGRO_FS_ENTRY *e)
```

Returns the entry's filename path. Note that the path will not be an absolute path if the entry wasn't created from an absolute path. Also not that the filesystem encoding may not be known and the conversion to UTF–8 could in very rare cases cause this to return an invalid path. Therefore it's always safest to access the file over its ALLEGRO_FS_ENTRY (7.1) and not the path.

On success returns a read only path structure, which you must not modify or destroy. Returns NULL on failure; errno is set to indicate the error.

7.6 al_update_fs_entry

```
bool al_update_fs_entry(ALLEGRO_FS_ENTRY *e)
```

Updates file status information for a filesystem entry. File status information is automatically updated when the entry is created, however you may update it again with this function, e.g. in case it changed.

Returns true on success, false on failure. Fills in errno to indicate the error.

See also: al_get_errno (18.5), al_get_fs_entry_atime (7.8), al_get_fs_entry_ctime (7.9), al_fs_entry_is_directory (7.14), al_fs_entry_is_file (7.13), al_get_fs_entry_mode (7.7)

7.7 al_get_fs_entry_mode

```
uint32_t al_get_fs_entry_mode(ALLEGRO_FS_ENTRY *e)
```

Returns the entry's mode flags, i.e. permissions and whether the entry refers to a file or directory.

See also: al_get_errno (18.5), ALLEGRO_FILE_MODE (7.2)

7.8 al_get_fs_entry_atime

```
time_t al_get_fs_entry_atime(ALLEGRO_FS_ENTRY *e)
```

Returns the time in seonds since the epoch since the entry was last accessed.

Warning: some filesystem either don't support this flag, or people turn it off to increase performance. It may not be valid in all circumstances.

See also: al_get_fs_entry_ctime (7.9), al_get_fs_entry_mtime (7.10), al_update_fs_entry (7.6)

7.9 al_get_fs_entry_ctime

```
time_t al_get_fs_entry_ctime(ALLEGRO_FS_ENTRY *e)
```

Returns the time in seconds since the epoch this entry was created on the filsystem.

See also: al_get_fs_entry_atime (7.8), al_get_fs_entry_mtime (7.10), al_update_fs_entry (7.6)

7.10 al_get_fs_entry_mtime

```
time_t al_get_fs_entry_mtime(ALLEGRO_FS_ENTRY *e)
```

Returns the time in seconds since the epoch since the entry was last modified.

See also: al_get_fs_entry_atime (7.8), al_get_fs_entry_ctime (7.9), al_update_fs_entry (7.6)

7.11 al_get_fs_entry_size

```
off_t al_get_fs_entry_size(ALLEGRO_FS_ENTRY *e)
```

Returns the size, in bytes, of the given entry. May not return anything sensible for a directory entry.

See also: al_update_fs_entry (7.6)

7.12 al_fs_entry_exists

```
bool al_fs_entry_exists(ALLEGRO_FS_ENTRY *e)
```

Check if the given entry exists on in the filesystem. Returns true if it does exist or false if it doesn't exist, or an error occured. Error is indicated in Allegro' errno.

7.13 al_fs_entry_is_file

```
bool al_fs_entry_is_file(ALLEGRO_FS_ENTRY *e)
```

Return true iff this entry is a regular file.

See also: al_get_fs_entry_mode (7.7)

7.14 al_fs_entry_is_directory

```
bool al_fs_entry_is_directory(ALLEGRO_FS_ENTRY *e)
```

Return true iff this entry is a directory.

See also: al_get_fs_entry_mode (7.7)

7.15 al_remove_fs_entry

```
bool al_remove_fs_entry(ALLEGRO_FS_ENTRY *e)
```

Delete this filesystem entry from the filesystem. Only files and empty directories may be deleted.

Returns true on success, and false on failure, error is indicated in Allegro' errno.

See also: al_filename_exists (7.16)

7.16 al_filename_exists

```
bool al_filename_exists(const char *path)
```

Check if the path exists on the filesystem, without creating an ALLEGRO_FS_ENTRY (7.1) object explicitly.

See also: al_fs_entry_exists (7.12)

7.17 al_remove_filename

```
bool al_remove_filename(const char *path)
```

Delete the given path from the filesystem, which may be a file or an empty directory. This is the same as al_remove_fs_entry (7.15), except it expects the path as a string.

Returns true on success, and false on failure. Allegro's errno is filled in to indicate the error.

See also: al_remove_fs_entry (7.15)

7.18 Directory functions

7.18.1 al_open_directory

```
bool al_open_directory(ALLEGRO_FS_ENTRY *e)
```

Opens a directory entry object. You must call this before using al_read_directory (7.18.2) on an entry and you must call al_close_directory (7.18.3) when you no longer need it.

Returns true on success.

See also: al_read_directory (7.18.2), al_close_directory (7.18.3)

7.18.2 al_read_directory

```
ALLEGRO_FS_ENTRY *al_read_directory(ALLEGRO_FS_ENTRY *e)
```

Reads the next directory item and returns a filesystem entry for it.

Returns NULL if there are no more entries or if an error occurs. Call al_close_directory (7.18.3) on the directory handle when you are done.

See also: al_open_directory (7.18.1), al_close_directory (7.18.3)

7.18.3 al_close_directory

```
bool al_close_directory(ALLEGRO_FS_ENTRY *e)
```

Closes a previously opened directory entry object.

Returns true on success, false on failure and fills in Allegro's errno to indicate the error.

See also: al_open_directory (7.18.1), al_read_directory (7.18.2)

7.18.4 al_get_current_directory

```
ALLEGRO_PATH *al_get_current_directory(void)
```

Returns the path to the current working directory, or NULL on failure. Allegro's errno is filled in to indicate the error.

See also: al_get_errno (18.5)

7.18.5 al_change_directory

```
bool al_change_directory(const char *path)
```

Changes the current working directory to 'path'.

Returns true on success, false on error.

7.18.6 al_make_directory

```
bool al_make_directory(const char *path)
```

Creates a new directory on the filesystem.

Returns true on success, false on error. Fills in Allegro's errno to indicate the error.

See also: al_get_errno (18.5)

7.19 Alternative filesystem functions

By default, Allegro uses platform specific filesystem functions for things like directory access. However if for example the files of your game are not in the local filesystem but inside some file archive, you can provide your own set of functions (or use an addon which does this for you, for example our physfs addon allows access to the most common archive formats).

7.19.1 ALLEGRO_FS_INTERFACE

```
typedef struct ALLEGRO_FS_INTERFACE ALLEGRO_FS_INTERFACE;
```

The available functions you can provide for a filesystem. They are:

~ ALLEGRO_FS_ENTRY * fs_create_entry (const char *path); void fs_destroy_entry (ALLEGRO_FS_ENTRY* e); const ALLEGRO_PATH *fs_entry_name (ALLEGRO_FS_ENTRY* e); bool fs_update_entry (ALLEGRO_FS_ENTRY *e); uint32_t fs_entry_mode (ALLEGRO_FS_ENTRY* e); time_t fs_entry_atime (ALLEGRO_FS_ENTRY *e); time_t fs_entry_mtime (ALLEGRO_FS_ENTRY* e); time_t fs_entry_ctime (ALLEGRO_FS_ENTRY *e); off_t fs_entry_size (ALLEGRO_FS_ENTRY* e); bool fs_entry_exists (ALLEGRO_FS_ENTRY *e); bool fs_remove_entry (ALLEGRO_FS_ENTRY* e);

bool fs_open_directory (ALLEGRO_FS_ENTRY *e); ALLEGRO_FS_ENTRY* fs_read_directory (ALLEGRO_FS_ENTRY *e); bool fs_close_directory(ALLEGRO_FS_ENTRY* e);

bool fs_filename_exists(const char *path); bool fs_remove_filename(const char* path); ALLEGRO_PATH * fs_get_current_directory
bool fs_change_directory(const char *path); bool fs_make_directory(const char* path); ~

7.19.2 al_set_fs_interface

```
void al_set_fs_interface(const ALLEGRO_FS_INTERFACE *fs_interface)
```

Set the ALLEGRO_FS_INTERFACE (7.19.1) table for the calling thread.

See also: al_set_standard_fs_interface (7.19.3), al_store_state (18.4), al_restore_state (18.3).

7.19.3 al_set_standard_fs_interface

```
void al_set_standard_fs_interface(void)
```

Return the ALLEGRO_FS_INTERFACE (7.19.1) table to the default, for the calling thread.

See also: al_set_fs_interface (7.19.2).

7.19.4 al_get_fs_interface

```
const ALLEGRO_FS_INTERFACE *al_get_fs_interface(void)
```

Return a pointer to the ALLEGRO_FS_INTERFACE (7.19.1) table in effect for the calling thread.

See also: al_store_state (18.4), al_restore_state (18.3).

8 Fixed point math routines

These functions are declared in the main Allegro header file:

```
#include <allegro5/allegro.h>
```

8.1 al_fixed

```
typedef int32_t al_fixed;
```

A fixed point number.

Allegro provides some routines for working with fixed point numbers, and defines the type `al_fixed` to be a signed 32-bit integer. The high word is used for the integer part and the low word for the fraction, giving a range of –32768 to 32767 and an accuracy of about four or five decimal places. Fixed point numbers can be assigned, compared, added, subtracted, negated and shifted (for multiplying or dividing by powers of two) using the normal integer operators, but you should take care to use the appropriate conversion routines when mixing fixed point with integer or floating point values. Writing `fixed_point_1 + fixed_point_2` is OK, but `fixed_point + integer` is not.

The only advantage of fixed point math routines is that you don't require a floating point coprocessor to use them. This was great in the time period of i386 and i486 machines, but stopped being so useful with the coming of the Pentium class of processors. From Pentium onwards, CPUs have increased their strength in floating point operations, equaling or even surpassing integer math performance.

Depending on the type of operations your program may need, using floating point types may be faster than fixed types if you are targeting a specific machine class. Many embedded processors have no FPUs so fixed point maths can be useful there.

8.2 al_itofix

```
al_fixed al_itofix(int x);
```

Converts an integer to fixed point. This is the same thing as x<<16. Remember that overflows (trying to convert an integer greater than 32767) and underflows (trying to convert an integer lesser than –32768) are not detected even in debug builds! The values simply "wrap around".

Example:

~~ al_fixed number;

```
/* This conversion is OK. */
number = al_itofix(100);
assert(al_fixtoi(number) == 100);

number = al_itofix(64000);

/* This check will fail in debug builds. */
assert(al_fixtoi(number) == 64000);
```

~~

Return value: Returns the value of the integer converted to fixed point ignoring overflows.

See also: al_fixtoi (8.3), al_ftofix (8.6), al_fixtof (8.7).

8.3 al_fixtoi

```
int al_fixtoi(al_fixed x);
```

Converts fixed point to integer, rounding as required to the nearest integer.

Example:

~~ int result;

```
/* This will put 33 into 'result'. */
result = al_fixtoi(al_itofix(100) / 3);

/* But this will round up to 17. */
result = al_fixtoi(al_itofix(100) / 6);
```

~~

See also: al_itofix (8.2), al_ftofix (8.6), al_fixtof (8.7), al_fixfloor (8.4), al_fixceil (8.5).

8.4 al_fixfloor

```
int al_fixfloor(al_fixed x);
```

Returns the greatest integer not greater than x. That is, it rounds towards negative infinity.

Example:

∼∼ int result;

```
/* This will put 33 into ‘result’. */
result = al_fixfloor(al_itofix(100) / 3);

/* And this will round down to 16. */
result = al_fixfloor(al_itofix(100) / 6);
```

∼∼

See also: al_fixtoi (8.3), al_fixceil (8.5).

8.5 al_fixceil

```
int al_fixceil(al_fixed x);
```

Returns the smallest integer not less than x. That is, it rounds towards positive infinity.

Example:

∼∼ int result;

```
/* This will put 34 into ‘result’. */
result = al_fixceil(al_itofix(100) / 3);

/* This will round up to 17. */
result = al_fixceil(al_itofix(100) / 6);
```

∼∼

See also: al_fixtoi (8.3), al_fixfloor (8.4).

8.6 al_ftofix

```
al_fixed al_ftofix(double x);
```

Converts a floating point value to fixed point. Unlike al_itofix (8.2), this function clamps values which could overflow the type conversion, setting Allegro’s errno to ERANGE in the process if this happens.

Example:

∼∼ al_fixed number;

```
number = al_itofix(-40000);
assert(al_fixfloor(number) == -32768);

number = al_itofix(64000);
assert(al_fixfloor(number) == 32767);
assert(!al_get_errno()); /* This will fail. */
```

∼∼

Return value: Returns the value of the floating point value converted to fixed point clamping overflows (and setting Allegro's errno).

See also: al_fixtof (8.7), al_itofix (8.2), al_fixtoi (8.3), al_get_errno (18.5)

8.7 al_fixtof

```
double al_fixtof(al_fixed x);
```

Converts fixed point to floating point.

Example:

∼∼ float result;

```
/* This will put 33.33333 into 'result'. */
result = al_fixtof(al_itofix(100) / 3);

/* This will put 16.66666 into 'result'. */
result = al_fixtof(al_itofix(100) / 6);
```

∼∼

See also: al_ftofix (8.6), al_itofix (8.2), al_fixtoi (8.3).

8.8 al_fixmul

```
al_fixed al_fixmul(al_fixed x, al_fixed y);
```

A fixed point value can be multiplied or divided by an integer with the normal `*` and `/` operators. To multiply two fixed point values, though, you must use this function.

If an overflow occurs, Allegro's errno will be set and the maximum possible value will be returned, but errno is not cleared if the operation is successful. This means that if you are going to test for overflow you should call `al_set_errno(0)` before calling al_fixmul (8.8).

Example:

∼∼ al_fixed result;

```
/* This will put 30000 into 'result'. */
result = al_fixmul(al_itofix(10), al_itofix(3000));

/* But this overflows, and sets errno. */
result = al_fixmul(al_itofix(100), al_itofix(3000));
assert(!al_get_errno());
```

∼∼

Return value: Returns the clamped result of multiplying `x` by `y`, setting Allegro's errno to ERANGE if there was an overflow.

See also: al_fixadd (8.10), al_fixsub (8.11), al_fixdiv (8.9), al_get_errno (18.5).

8.9 al_fixdiv

```
al_fixed al_fixdiv(al_fixed x, al_fixed y);
```

A fixed point value can be divided by an integer with the normal `/` operator. To divide two fixed point values, though, you must use this function. If a division by zero occurs, Allegro's errno will be set and the maximum possible value will be returned, but errno is not cleared if the operation is successful. This means that if you are going to test for division by zero you should call `al_set_errno(0)` before calling al_fixdiv (8.9).

Example:

~~ al_fixed result;

```
/* This will put 0.06060 'result'. */
result = al_fixdiv(al_itofix(2), al_itofix(33));

/* This will put 0 into 'result'. */
result = al_fixdiv(0, al_itofix(-30));

/* Sets errno and puts -32768 into 'result'. */
result = al_fixdiv(al_itofix(-100), al_itofix(0));
assert(!al_get_errno()); /* This will fail. */
```

~~

Return value: Returns the result of dividing `x` by `y`. If `y` is zero, returns the maximum possible fixed point value and sets Allegro's errno to ERANGE.

See also: al_fixadd (8.10), al_fixsub (8.11), al_fixmul (8.8), al_get_errno (18.5).

8.10 al_fixadd

```
al_fixed al_fixadd(al_fixed x, al_fixed y);
```

Although fixed point numbers can be added with the normal `+` integer operator, that doesn't provide any protection against overflow. If overflow is a problem, you should use this function instead. It is slower than using integer operators, but if an overflow occurs it will set Allegro's errno and clamp the result, rather than just letting it wrap.

Example:

~~ al_fixed result;

```
/* This will put 5035 into 'result'. */
result = al_fixadd(al_itofix(5000), al_itofix(35));

/* Sets errno and puts -32768 into 'result'. */
result = al_fixadd(al_itofix(-31000), al_itofix(-3000));
assert(!al_get_errno()); /* This will fail. */
```

~~

Return value: Returns the clamped result of adding `x` to `y`, setting Allegro's errno to ERANGE if there was an overflow.

See also: al_fixsub (8.11), al_fixmul (8.8), al_fixdiv (8.9).

8.11 al_fixsub

```
al_fixed al_fixsub(al_fixed x, al_fixed y);
```

Although fixed point numbers can be subtracted with the normal `-` integer operator, that doesn't provide any protection against overflow. If overflow is a problem, you should use this function instead. It is slower than using integer operators, but if an overflow occurs it will set Allegro's errno and clamp the result, rather than just letting it wrap.

Example:

~~ al_fixed result;

```
/* This will put 4965 into ‘result’. */
result = al_fixsub(al_itofix(5000), al_itofix(35));

/* Sets errno and puts -32768 into ‘result’. */
result = al_fixsub(al_itofix(-31000), al_itofix(3000));
assert(!al_get_errno()); /* This will fail. */
```

~~

Return value: Returns the clamped result of subtracting `y` from `x`, setting Allegro's errno to ERANGE if there was an overflow.

See also: al_fixadd (8.10), al_fixmul (8.8), al_fixdiv (8.9), al_get_errno (18.5).

8.12 Fixed point trig

The fixed point square root, sin, cos, tan, inverse sin, and inverse cos functions are implemented using lookup tables, which are very fast but not particularly accurate. At the moment the inverse tan uses an iterative search on the tan table, so it is a lot slower than the others. On machines with good floating point processors using these functions could be slower Always profile your code.

Angles are represented in a binary format with 256 equal to a full circle, 64 being a right angle and so on. This has the advantage that a simple bitwise ‘and’ can be used to keep the angle within the range zero to a full circle.

8.12.1 al_fixtorad_r

```
const al_fixed al_fixtorad_r = (al_fixed)1608;
```

This constant gives a ratio which can be used to convert a fixed point number in binary angle format to a fixed point number in radians.

Example:

~~ al_fixed rad_angle, binary_angle;

```
/* Set the binary angle to 90 degrees. */
binary_angle = 64;

/* Now convert to radians (about 1.57). */
rad_angle = al_fixmul(binary_angle, al_fixtorad_r);
```

∼∼

See also: al_fixmul (8.8), al_radtofix_r (8.12.2).

8.12.2 al_radtofix_r

```
const al_fixed al_radtofix_r = (al_fixed)2670177;
```

This constant gives a ratio which can be used to convert a fixed point number in radians to a fixed point number in binary angle format.

Example:

~~∼∼ al_fixed rad_angle, binary_angle; . . . binary_angle = al_fixmul(rad_angle, radtofix_r);~~∼∼

See also: al_fixmul (8.8), al_fixtorad_r (8.12.1).

8.12.3 al_fixsin

```
al_fixed al_fixsin(al_fixed x);
```

This function finds the sine of a value using a lookup table. The input value must be a fixed point binary angle.

Example:

∼∼ al_fixed angle; int result;

```
/* Set the binary angle to 90 degrees. */
angle = al_itofix(64);

/* The sine of 90 degrees is one. */
result = al_fixtoi(al_fixsin(angle));
assert(result == 1);
```

∼∼

Return value: Returns the sine of a fixed point binary format angle. The return value will be in radians.

8.12.4 al_fixcos

```
al_fixed al_fixcos(al_fixed x);
```

This function finds the cosine of a value using a lookup table. The input value must be a fixed point binary angle.

Example:

∼∼ al_fixed angle; float result;

```
/* Set the binary angle to 45 degrees. */
angle = al_itofix(32);

/* The cosine of 45 degrees is about 0.7071. */
result = al_fixtof(al_fixcos(angle));
assert(result > 0.7 && result < 0.71);
```

∼∼

Return value: Returns the cosine of a fixed point binary format angle. The return value will be in radians.

8.12.5 al_fixtan

```
al_fixed al_fixtan(al_fixed x);
```

This function finds the tangent of a value using a lookup table. The input value must be a fixed point binary angle.

Example:

∼∼ al_fixed angle, res_a, res_b; float dif;

```
angle = al_itofix(37);
/* Prove that tan(angle) == sin(angle) / cos(angle). */
res_a = al_fixdiv(al_fixsin(angle), al_fixcos(angle));
res_b = al_fixtan(angle);
dif = al_fixtof(al_fixsub(res_a, res_b));
printf("Precision error: %f\n", dif);
```

∼∼

Return value: Returns the tangent of a fixed point binary format angle. The return value will be in radians.

8.12.6 al_fixasin

```
al_fixed al_fixasin(al_fixed x);
```

This function finds the inverse sine of a value using a lookup table. The input value must be a fixed point value. The inverse sine is defined only in the domain from –1 to 1. Outside of this input range, the function will set Allegro's errno to EDOM and return zero.

Example:

∼∼ float angle; al_fixed val;

```
/* Sets 'val' to a right binary angle (64). */
val = al_fixasin(al_itofix(1));

/* Sets 'angle' to 0.2405. */
angle = al_fixtof(al_fixmul(al_fixasin(al_ftofix(0.238)), al_fixtorad_r));

/* This will trigger the assert. */
val = al_fixasin(al_ftofix(-1.09));
assert(!al_get_errno());
```

~~

Return value: Returns the inverse sine of a fixed point value, measured as fixed point binary format angle, or zero if the input was out of the range. All return values of this function will be in the range –64 to 64.

8.12.7 al_fixacos

```
al_fixed al_fixacos(al_fixed x);
```

This function finds the inverse cosine of a value using a lookup table. The input value must be a fixed point radian. The inverse cosine is defined only in the domain from −1 to 1. Outside of this input range, the function will set Allegro's errno to EDOM and return zero.

Example:

~~ al_fixed result;

```
/* Sets result to binary angle 128. */
result = al_fixacos(al_itofix(-1));
```

~~

Return value: Returns the inverse sine of a fixed point value, measured as fixed point binary format angle, or zero if the input was out of range. All return values of this function will be in the range 0 to 128.

8.12.8 al_fixatan

```
al_fixed al_fixatan(al_fixed x)
```

This function finds the inverse tangent of a value using a lookup table. The input value must be a fixed point radian. The inverse tangent is the value whose tangent is `x`.

Example:

~~ al_fixed result;

```
/* Sets result to binary angle 13. */
result = al_fixatan(al_ftofix(0.326));
```

~~

Return value: Returns the inverse tangent of a fixed point value, measured as a fixed point binary format angle.

8.12.9 al_fixatan2

```
al_fixed al_fixatan2(al_fixed y, al_fixed x)
```

This is a fixed point version of the libc atan2() routine. It computes the arc tangent of `y / x`, but the signs of both arguments are used to determine the quadrant of the result, and `x` is permitted to be zero. This function is useful to convert Cartesian coordinates to polar coordinates.

Example:

~~ al_fixed result;

```
/* Sets ‘result’ to binary angle 64. */
result = al_fixatan2(al_itofix(1), 0);

/* Sets ‘result’ to binary angle -109. */
result = al_fixatan2(al_itofix(-1), al_itofix(-2));

/* Fails the assert. */
result = al_fixatan2(0, 0);
assert(!al_get_errno());
```

~~

Return value: Returns the arc tangent of `y / x` in fixed point binary format angle, from –128 to 128. If both `x` and `y` are zero, returns zero and sets Allegro's errno to EDOM.

8.12.10 al_fixsqrt

```
al_fixed al_fixsqrt(al_fixed x)
```

This finds out the non negative square root of `x`. If `x` is negative, Allegro's errno is set to EDOM and the function returns zero.

8.12.11 al_fixhypot

```
al_fixed al_fixhypot(al_fixed x, al_fixed y)
```

Fixed point hypotenuse (returns the square root of `x*x + y*y`). This should be better than calculating the formula yourself manually, since the error is much smaller.

9 Graphics

These functions are declared in the main Allegro header file:

```
#include <allegro5/allegro.h>
```

9.1 Colors

9.1.1 ALLEGRO_COLOR

```
typedef struct ALLEGRO_COLOR ALLEGRO_COLOR;
```

An ALLEGRO_COLOR structure describes a color in a device independant way. Use al_map_rgb (9.1.2) et al. and al_unmap_rgb (9.1.6) et al. to translate from and to various color representations.

9.1.2 al_map_rgb

```
ALLEGRO_COLOR al_map_rgb(
   unsigned char r, unsigned char g, unsigned char b)
```

Convert r, g, b (ranging from 0–255) into an ALLEGRO_COLOR, using 255 for alpha.

See also: al_map_rgba (9.1.4), al_map_rgba_f (9.1.5), al_map_rgb_f (9.1.3)

9.1.3 al_map_rgb_f

```
ALLEGRO_COLOR al_map_rgb_f(float r, float g, float b)
```

Convert r, g, b, (ranging from 0.0f–1.0f) into an ALLEGRO_COLOR, using 1.0f for alpha.

See also: al_map_rgba (9.1.4), al_map_rgb (9.1.2), al_map_rgba_f (9.1.5)

9.1.4 al_map_rgba

```
ALLEGRO_COLOR al_map_rgba(
   unsigned char r, unsigned char g, unsigned char b, unsigned char a)
```

Convert r, g, b, a (ranging from 0–255) into an ALLEGRO_COLOR.

See also: al_map_rgb (9.1.2), al_map_rgba_f (9.1.5), al_map_rgb_f (9.1.3)

9.1.5 al_map_rgba_f

```
ALLEGRO_COLOR al_map_rgba_f(float r, float g, float b, float a)
```

Convert r, g, b, a (ranging from 0.0f–1.0f) into an ALLEGRO_COLOR.

See also: al_map_rgba (9.1.4), al_map_rgb (9.1.2), al_map_rgb_f (9.1.3)

9.1.6 al_unmap_rgb

```
void al_unmap_rgb(ALLEGRO_COLOR color,
   unsigned char *r, unsigned char *g, unsigned char *b)
```

Retrieves components of an ALLEGRO_COLOR, ignoring alpha Components will range from 0–255.

See also: al_unmap_rgba (9.1.8), al_unmap_rgba_f (9.1.9), al_unmap_rgb_f (9.1.7)

9.1.7 al_unmap_rgb_f

```
void al_unmap_rgb_f(ALLEGRO_COLOR color, float *r, float *g, float *b)
```

Retrieves components of an ALLEGRO_COLOR, ignoring alpha. Components will range from 0.0f–1.0f.

See also: al_unmap_rgba (9.1.8), al_unmap_rgb (9.1.6), al_unmap_rgba_f (9.1.9)

9.1.8 al_unmap_rgba

```
void al_unmap_rgba(ALLEGRO_COLOR color,
   unsigned char *r, unsigned char *g, unsigned char *b, unsigned char *a)
```

Retrieves components of an ALLEGRO_COLOR. Components will range from 0–255.

See also: al_unmap_rgb (9.1.6), al_unmap_rgba_f (9.1.9), al_unmap_rgb_f (9.1.7)

9.1.9 al_unmap_rgba_f

```
void al_unmap_rgba_f(ALLEGRO_COLOR color,
   float *r, float *g, float *b, float *a)
```

Retrieves components of an ALLEGRO_COLOR. Components will range from 0.0f–1.0f.

See also: al_unmap_rgba (9.1.8), al_unmap_rgb (9.1.6), al_unmap_rgb_f (9.1.7)

9.2 Locking and pixel formats

9.2.1 ALLEGRO_LOCKED_REGION

```
typedef struct ALLEGRO_LOCKED_REGION ALLEGRO_LOCKED_REGION;
```

Users who wish to manually edit or read from a bitmap are required to lock it first. The ALLEGRO_LOCKED_REGION structure represents the locked region of the bitmap. This call will work with any bitmap, including memory bitmaps.

```
typedef struct ALLEGRO_LOCKED_REGION {
        void *data; // the bitmap data
        int format; // the pixel format of the data
        int pitch;  // the size in bytes of a single line
                    // pitch may be greater than pixel_size*bitmap->w
                    // i.e. padded with extra bytes
}
```

See also: al_lock_bitmap (9.2.5), al_lock_bitmap_region (9.2.6), al_unlock_bitmap (9.2.7), ALLEGRO_PIXEL_FORMAT (9.2.2)

9.2.2 ALLEGRO_PIXEL_FORMAT

```
typedef enum ALLEGRO_PIXEL_FORMAT
```

Pixel formats. Each pixel format specifies the exact size and bit layout of a pixel in memory. Components are specified from high bits to low bits, so for example a fully opaque red pixel in ARGB_8888 format is 0xFFFF0000.

Note:

The pixel format is independent of endianness. That is, in the above example you can always get the red component with (pixel & 0x00ff0000) >> 16

But you can *not* rely on this code:

*(pixel + 2)

It will return the red component on little endian systems, but the green component on big endian systems.

Also note that Allegro's naming is different from OpenGL naming here, where a format of GL_RGBA8 merely defines the component order and the exact layout including endianness treatment is specified separately. Usually GL_RGBA8 will correspond to ALLEGRO_PIXEL_ABGR_8888 though on little endian systems, so care must be taken (note the reversal of RGBA <-> ABGR).

The only exception to this ALLEGRO_PIXEL_FORMAT_ABGR_8888_LE which will always have the components as 4 bytes corresponding to red, green, blue and alpha, in this order, independent of the endianness.

Format	Notes
ALLEGRO_PIXEL_FORMAT_ANY	Let the driver choose a format. This is the default format at program start.
ALLEGRO_PIXEL_FORMAT_ANY_NO_ALPHA	Let the driver choose a format without alpha.
ALLEGRO_PIXEL_FORMAT_ANY_WITH_ALPHA	Let the driver choose a format with alpha.
ALLEGRO_PIXEL_FORMAT_ANY_15_NO_ALPHA	Let the driver choose a 15 bit format without alpha.
ALLEGRO_PIXEL_FORMAT_ANY_15_WITH_ALPHA	Let the driver choose a 15 bit format with alpha.
ALLEGRO_PIXEL_FORMAT_ANY_16_NO_ALPHA	Let the driver choose a 16 bit format without alpha.
ALLEGRO_PIXEL_FORMAT_ANY_16_WITH_ALPHA	Let the driver choose a 16 bit format with alpha.
ALLEGRO_PIXEL_FORMAT_ANY_24_NO_ALPHA	Let the driver choose a 24 bit format without alpha.
ALLEGRO_PIXEL_FORMAT_ANY_24_WITH_ALPHA	Let the driver choose a 24 bit format with alpha.
ALLEGRO_PIXEL_FORMAT_ANY_32_NO_ALPHA	Let the driver choose a 32 bit format without alpha.
ALLEGRO_PIXEL_FORMAT_ANY_32_WITH_ALPHA	Let the driver choose a 32 bit format with alpha.
ALLEGRO_PIXEL_FORMAT_ARGB_8888	32 bit
ALLEGRO_PIXEL_FORMAT_RGBA_8888	32 bit
ALLEGRO_PIXEL_FORMAT_ARGB_4444	16 bit
ALLEGRO_PIXEL_FORMAT_RGB_888	24 bit
ALLEGRO_PIXEL_FORMAT_RGB_565	16 bit
ALLEGRO_PIXEL_FORMAT_RGB_555	15 bit
ALLEGRO_PIXEL_FORMAT_RGBA_5551	16 bit
ALLEGRO_PIXEL_FORMAT_ARGB_1555	16 bit
ALLEGRO_PIXEL_FORMAT_ABGR_8888	32 bit
ALLEGRO_PIXEL_FORMAT_XBGR_8888	32 bit
ALLEGRO_PIXEL_FORMAT_BGR_888	24 bit
ALLEGRO_PIXEL_FORMAT_BGR_565	16 bit
ALLEGRO_PIXEL_FORMAT_BGR_555	15 bit
ALLEGRO_PIXEL_FORMAT_RGBX_8888	32 bit
ALLEGRO_PIXEL_FORMAT_XRGB_8888	32 bit
ALLEGRO_PIXEL_FORMAT_ABGR_F32	128 bit
ALLEGRO_PIXEL_FORMAT_ABGR_8888_LE	Like the version without _LE, but the component order is guaranteed to be red, green, blue, alpha. This only makes a difference on big endian systems, on little endian it is just an alias.

See also: al_set_new_bitmap_format (9.3.9), al_get_bitmap_format (9.4.2)

9.2.3 al_get_pixel_size

```
int al_get_pixel_size(int format)
```

Return the number of bytes that a pixel of the given format occupies.

See also: ALLEGRO_PIXEL_FORMAT (9.2.2), al_get_pixel_format_bits (9.2.4)

9.2.4 al_get_pixel_format_bits

```
int al_get_pixel_format_bits(int format)
```

Return the number of bits that a pixel of the given format occupies.

See also: ALLEGRO_PIXEL_FORMAT (9.2.2), al_get_pixel_size (9.2.3)

9.2.5 al_lock_bitmap

```
ALLEGRO_LOCKED_REGION *al_lock_bitmap(ALLEGRO_BITMAP *bitmap,
   int format, int flags)
```

Lock an entire bitmap for reading or writing. If the bitmap is a display bitmap it will be updated from system memory after the bitmap is unlocked (unless locked read only). Returns NULL if the bitmap cannot be locked, e.g. the bitmap was locked previously and not unlocked.

Flags are:

- ALLEGRO_LOCK_READONLY - The locked region will not be written to. This can be faster if the bitmap is a video texture, as it can be discarded after the lock instead of uploaded back to the card.
- ALLEGRO_LOCK_WRITEONLY - The locked region will not be read from. This can be faster if the bitmap is a video texture, as no data need to be read from the video card. You are required to fill in all pixels before unlocking the bitmap again, so be careful when using this flag.
- ALLEGRO_LOCK_READWRITE - The locked region can be written to and read from.

'format' indicates the pixel format that the returned buffer will be in. To lock in the same format as the bitmap stores it's data internally, call with `al_get_bitmap_format(bitmap)` as the format or use ALLEGRO_PIXEL_FORMAT_ANY. Locking in the native format will usually be faster.

Note:

While a bitmap is locked, you can not use any drawing operations on it (with the sole exception of al_put_pixel (9.5.9) and al_put_blended_pixel (9.5.10)).

See also: ALLEGRO_LOCKED_REGION (9.2.1), ALLEGRO_PIXEL_FORMAT (9.2.2), al_unlock_bitmap (9.2.7)

9.2.6 al_lock_bitmap_region

```
ALLEGRO_LOCKED_REGION *al_lock_bitmap_region(ALLEGRO_BITMAP *bitmap,
   int x, int y, int width, int height, int format, int flags)
```

Like al_lock_bitmap (9.2.5), but only locks a specific area of the bitmap. If the bitmap is a display bitmap, only that area of the texture will be updated when it is unlocked. Locking only the region you indend to modify will be faster than locking the whole bitmap.

See also: ALLEGRO_LOCKED_REGION (9.2.1), ALLEGRO_PIXEL_FORMAT (9.2.2), al_unlock_bitmap (9.2.7)

9.2.7 al_unlock_bitmap

```
void al_unlock_bitmap(ALLEGRO_BITMAP *bitmap)
```

Unlock a previously locked bitmap or bitmap region. If the bitmap is a display bitmap, the texture will be updated to match the system memory copy (unless it was locked read only).

See also: al_lock_bitmap (9.2.5), al_lock_bitmap_region (9.2.6)

9.3 Bitmap creation

9.3.1 ALLEGRO_BITMAP

```
typedef struct ALLEGRO_BITMAP ALLEGRO_BITMAP;
```

Abstract type representing a bitmap (2D image).

9.3.2 al_clone_bitmap

```
ALLEGRO_BITMAP *al_clone_bitmap(ALLEGRO_BITMAP *bitmap)
```

Clone a bitmap "exactly", formats can be different.

XXX document this better

See also: al_create_bitmap (9.3.3)

9.3.3 al_create_bitmap

```
ALLEGRO_BITMAP *al_create_bitmap(int w, int h)
```

Creates a new bitmap using the bitmap format and flags for the current thread. Blitting between bitmaps of differing formats, or blitting between memory bitmaps and display bitmaps may be slow.

Unless you set the ALLEGRO_MEMORY_BITMAP flag, the bitmap is created for the current display. Blitting to another display may be slow.

If a display bitmap is created, there may be limitations on the allowed dimensions. For example a DirectX or OpenGL backend usually has a maximum allowed texture size - so if bitmap creation fails for very large dimensions, you may want to re-try with a smaller bitmap.

See also: al_set_new_bitmap_format (9.3.9), al_set_new_bitmap_flags (9.3.8), al_clone_bitmap (9.3.2), al_create_sub_bitmap (9.3.

9.3.4 al_create_sub_bitmap

```
ALLEGRO_BITMAP *al_create_sub_bitmap(ALLEGRO_BITMAP *parent,
   int x, int y, int w, int h)
```

Creates a sub-bitmap of the parent, at the specified coordinates and of the specified size. A sub-bitmap is a bitmap that shares drawing memory with a pre-existing (parent) bitmap, but possibly with a different size and clipping settings.

If the sub-bitmap does not lie completely inside the parent bitmap, then it is automatically clipped so that it does.

See the discussion in al_get_backbuffer (4.2.3) about using sub-bitmaps of the backbuffer.

The parent bitmap's clipping rectangles are ignored.

If a sub-bitmap was not or cannot be created then NULL is returned.

Note that destroying parents of sub-bitmaps will not destroy the sub-bitmaps; instead the sub-bitmaps become invalid and should no longer be used.

See also: al_create_bitmap (9.3.3)

9.3.5 al_destroy_bitmap

```
void al_destroy_bitmap(ALLEGRO_BITMAP *bitmap)
```

Destroys the given bitmap, freeing all resources used by it. Does nothing if given the null pointer.

9.3.6 al_get_new_bitmap_flags

```
int al_get_new_bitmap_flags(void)
```

Returns the flags used for newly created bitmaps.

See also: al_set_new_bitmap_flags (9.3.8)

9.3.7 al_get_new_bitmap_format

```
int al_get_new_bitmap_format(void)
```

Returns the format used for newly created bitmaps.

See also: ALLEGRO_PIXEL_FORMAT (9.2.2), al_set_new_bitmap_format (9.3.9)

9.3.8 al_set_new_bitmap_flags

```
void al_set_new_bitmap_flags(int flags)
```

Sets the flags to use for newly created bitmaps. Valid flags are:

- ALLEGRO_VIDEO_BITMAP - This is the default flag. Creates a bitmap that resides in the video card memory. These types of bitmaps receive the greatest benefit from hardware acceleration.
- ALLEGRO_MEMORY_BITMAP - Create a bitmap residing in system memory. Operations on, and with, memory bitmaps will not be hardware accelerated. However, direct pixel access can be relatively quick compared to video bitmaps, which depend on the display driver in use. *Note: Allegro's software rendering routines are currently very unoptimised.*
- ALLEGRO_KEEP_BITMAP_FORMAT - Only used when loading bitmaps from disk files, forces the resulting ALLEGRO_BITMAP to use the same format as the file.
- ALLEGRO_FORCE_LOCKING - When drawing to a bitmap with this flag set, always use pixel locking and draw to it using Allegro's software drawing primitives. This should never be used if you plan to draw to the bitmap using Allegro's graphics primitives as it would cause severe performance penalties. However if you know that the bitmap will only ever be accessed by locking it, no unneeded FBOs will be created for it in the OpenGL drivers.
- ALLEGRO_NO_PRESERVE_TEXTURE - Normally, every effort is taken to preserve the contents of bitmaps since some drivers may forget them. This can take extra time. If you know it doesn't matter if a bitmap keeps its image, for example a temporary buffer, use this flag to tell Allegro not to attempt to preserve the contents of bitmaps created after this flag is set. This can lead to speed improvements in your program.
- ALLEGRO_ALPHA_TEST - This is a driver hint only. It tells the graphics driver to do alpha testing instead of alpha blending on bitmaps created with this flag. Alpha testing is usually faster and preferred if your bitmaps have only one level of alpha (0). This flag is currently not widely implemented (i.e., only for memory bitmaps).

See also: al_get_new_bitmap_flags (9.3.6), al_get_bitmap_flags (9.4.1)

9.3.9 al_set_new_bitmap_format

```
void al_set_new_bitmap_format(int format)
```

Sets the pixel format for newly created bitmaps. The default format is 0 and means the display driver will choose the best format.

See also: ALLEGRO_PIXEL_FORMAT (9.2.2), al_get_new_bitmap_format (9.3.7), al_get_bitmap_format (9.4.2)

9.4 Bitmap properties

9.4.1 al_get_bitmap_flags

```
int al_get_bitmap_flags(ALLEGRO_BITMAP *bitmap)
```

Return the flags user to create the bitmap.

See also: al_set_new_bitmap_flags (9.3.8)

9.4.2 al_get_bitmap_format

```
int al_get_bitmap_format(ALLEGRO_BITMAP *bitmap)
```

Returns the pixel format of a bitmap.

See also: ALLEGRO_PIXEL_FORMAT (9.2.2), al_set_new_bitmap_flags (9.3.8)

9.4.3 al_get_bitmap_height

```
int al_get_bitmap_height(ALLEGRO_BITMAP *bitmap)
```

Returns the height of a bitmap in pixels.

9.4.4 al_get_bitmap_width

```
int al_get_bitmap_width(ALLEGRO_BITMAP *bitmap)
```

Returns the width of a bitmap in pixels.

9.4.5 al_get_pixel

```
ALLEGRO_COLOR al_get_pixel(ALLEGRO_BITMAP *bitmap, int x, int y)
```

Get a pixel's color value from the specified bitmap. This operation is slow on non-memory bitmaps. Consider locking the bitmap if you are going to use this function multiple times on the same bitmap.

See also: ALLEGRO_COLOR (9.1.1), al_put_pixel (9.5.9)

9.4.6 al_is_bitmap_locked

```
bool al_is_bitmap_locked(ALLEGRO_BITMAP *bitmap)
```

Returns whether or not a bitmap is already locked.

See also: al_lock_bitmap (9.2.5), al_lock_bitmap_region (9.2.6), al_unlock_bitmap (9.2.7)

9.4.7 al_is_compatible_bitmap

```
bool al_is_compatible_bitmap(ALLEGRO_BITMAP *bitmap)
```

D3D and OpenGL allow sharing a texture in a way so it can be used for multiple windows. Each ALLEGRO_BITMAP created with al_create_bitmap (9.3.3) however is usually tied to a single ALLEGRO_DISPLAY. This function can be used to know if the bitmap is compatible with the current display, even if it is another display than the one it was created with. It returns true if the bitmap is compatible (things like a cached texture version can be used) and false otherwise (blitting in the current display will be slow).

The only time this function is useful is if you are using multiple windows and need accelerated blitting of the same bitmaps to both.

Returns true if the bitmap is compatible with the current display, false otherwise. If there is no current display, false is returned.

9.4.8 al_is_sub_bitmap

```
bool al_is_sub_bitmap(ALLEGRO_BITMAP *bitmap)
```

Returns true if the specified bitmap is a sub-bitmap, false otherwise.

See also: al_create_sub_bitmap (9.3.4)

9.5 Drawing operations

All drawing operations draw to the current "target bitmap" of the current thread. Initially, the target bitmap will be the backbuffer of the last display created in a thread.

9.5.1 al_clear_to_color

```
void al_clear_to_color(ALLEGRO_COLOR color)
```

Clear the complete target bitmap, but confined by the clipping rectangle.

See also: ALLEGRO_COLOR (9.1.1), al_set_clipping_rectangle (9.7.2)

9.5.2 al_draw_bitmap

```
void al_draw_bitmap(ALLEGRO_BITMAP *bitmap, float dx, float dy, int flags)
```

Draws an unscaled, unrotated bitmap at the given position to the current target bitmap (see al_set_target_bitmap (9.5.11)). flags can be a combination of:

- ALLEGRO_FLIP_HORIZONTAL - flip the bitmap about the y-axis
- ALLEGRO_FLIP_VERTICAL - flip the bitmap about the x-axis

See also: al_draw_bitmap_region (9.5.3), al_draw_scaled_bitmap (9.5.7), al_draw_rotated_bitmap (9.5.5), al_draw_rotated_scaled

9.5.3 al_draw_bitmap_region

```
void al_draw_bitmap_region(ALLEGRO_BITMAP *bitmap,
   float sx, float sy, float sw, float sh, float dx, float dy, int flags)
```

Draws a region of the given bitmap to the target bitmap.

- sx - source x
- sy - source y
- sw - source width (width of region to blit)
- sh - source height (height of region to blit)
- dx - destination x

- dy - destination y
- flags - same as for al_draw_bitmap (9.5.2)

See also: al_draw_bitmap (9.5.2), al_draw_scaled_bitmap (9.5.7), al_draw_rotated_bitmap (9.5.5), al_draw_rotated_scaled_bitma

9.5.4 al_draw_pixel

```
void al_draw_pixel(float x, float y, ALLEGRO_COLOR color)
```

Draws a single pixel at x, y. This function, unlike al_put_pixel (9.5.9), does blending and, unlike al_put_blended_pixel (9.5.10), respects the transformations. This function can be slow if called often; if you need to draw a lot of pixels consider using the primitives addon.

- x - destination x
- y - destination y
- color - color of the pixel

See also: ALLEGRO_COLOR (9.1.1), al_put_pixel (9.5.9)

9.5.5 al_draw_rotated_bitmap

```
void al_draw_rotated_bitmap(ALLEGRO_BITMAP *bitmap,
   float cx, float cy, float dx, float dy, float angle, int flags)
```

Draws a rotated version of the given bitmap to the target bitmap. The bitmap is rotated by 'angle' radians clockwise.

The point at cx/cy inside the bitmap will be drawn at dx/dy and the bitmap is rotated around this point.

- cx - center x
- cy - center y
- dx - destination x
- dy - destination y
- angle - angle by which to rotate
- flags - same as for al_draw_bitmap (9.5.2)

See also: al_draw_bitmap (9.5.2), al_draw_bitmap_region (9.5.3), al_draw_scaled_bitmap (9.5.7), al_draw_rotated_scaled_bitmap

9.5.6 al_draw_rotated_scaled_bitmap

```
void al_draw_rotated_scaled_bitmap(ALLEGRO_BITMAP *bitmap,
   float cx, float cy, float dx, float dy, float xscale, float yscale,
   float angle, int flags)
```

Like al_draw_rotated_bitmap (9.5.5), but can also scale the bitmap.

The point at cx/cy in the bitmap will be drawn at dx/dy and the bitmap is rotated and scaled around this point.

- cx - center x
- cy - center y
- dx - destination x
- dy - destination y
- xscale - how much to scale on the x-axis (e.g. 2 for twice the size)
- yscale - how much to scale on the y-axis
- angle - angle by which to rotate
- flags - same as for al_draw_bitmap (9.5.2)

See also: al_draw_bitmap (9.5.2), al_draw_bitmap_region (9.5.3), al_draw_scaled_bitmap (9.5.7), al_draw_rotated_bitmap (9.5.5)

9.5.7 al_draw_scaled_bitmap

```
void al_draw_scaled_bitmap(ALLEGRO_BITMAP *bitmap,
   float sx, float sy, float sw, float sh,
   float dx, float dy, float dw, float dh, int flags)
```

Draws a scaled version of the given bitmap to the target bitmap.

- sx - source x
- sy - source y
- sw - source width
- sh - source height
- dx - destination x
- dy - destination y
- dw - destination width
- dh - destination height
- flags - same as for al_draw_bitmap (9.5.2)

See also: al_draw_bitmap (9.5.2), al_draw_bitmap_region (9.5.3), al_draw_rotated_bitmap (9.5.5), al_draw_rotated_scaled_bitma

9.5.8 al_get_target_bitmap

```
ALLEGRO_BITMAP *al_get_target_bitmap(void)
```

Return the target bitmap of the current display.

See also: al_set_target_bitmap (9.5.11)

9.5.9 al_put_pixel

```
void al_put_pixel(int x, int y, ALLEGRO_COLOR color)
```

Draw a single pixel on the target bitmap. This operation is slow on non-memory bitmaps. Consider locking the bitmap if you are going to use this function multiple times on the same bitmap. This function is not affected by neither the transformations nor the color blenders.

See also: ALLEGRO_COLOR (9.1.1), al_get_pixel (9.4.5), al_put_blended_pixel (9.5.10)

9.5.10 al_put_blended_pixel

```
void al_put_blended_pixel(int x, int y, ALLEGRO_COLOR color)
```

Like al_put_pixel (9.5.9), but the pixel color is blended using the current blenders before being drawn.

See also: ALLEGRO_COLOR (9.1.1), al_put_pixel (9.5.9)

9.5.11 al_set_target_bitmap

```
void al_set_target_bitmap(ALLEGRO_BITMAP *bitmap)
```

Select the bitmap to which all subsequent drawing operations in the calling thread will draw. Select the backbuffer (see al_get_backbuffer (4.2.3)) to return to drawing to the screen normally.

OpenGL note:

Framebuffer objects (FBOs) allow OpenGL to directly draw to a bitmap, which is very fast. However, each created FBO needs additional resources, therefore an FBO is not automatically assigned to each non-memory bitmap when it is created (as is done with textures).

When using an OpenGL display, only if all of the following conditions are met an FBO will be created for the bitmap:

- The GL_EXT_framebuffer_object OpenGL extension is available.
- The bitmap is not a memory bitmap.
- The bitmap is not currently locked.

Once created, the FBO is kept around until the bitmap is destroyed or you explicitely call al_remove_opengl_fbo (15.7) on the bitmap.

In the following example, no FBO will be created:

```
lock = al_lock_bitmap(bitmap);
al_set_target_bitmap(bitmap);
al_put_pixel(x, y, color);
al_unlock_bitmap(bitmap);
```

The above allows using al_put_pixel (9.5.9) on a locked bitmap without creating an FBO.

In this example an FBO is created however:

```
al_set_target_bitmap(bitmap);
al_draw_line(x1, y1, x2, y2, color, 0);
```

And an OpenGL command will be used to directly draw the line into the bitmap's associated texture.

See also: al_get_target_bitmap (9.5.8)

9.6 Blending modes

9.6.1 al_get_blender

```
void al_get_blender(int *op, int *src, int *dst, ALLEGRO_COLOR *color)
```

Returns the active blender for the current thread. You can pass NULL for values you are not interested in.

See also: al_set_blender (9.6.3), al_get_separate_blender (9.6.2)

9.6.2 al_get_separate_blender

```
void al_get_separate_blender(int *op, int *src, int *dst,
   int *alpha_op, int *alpha_src, int *alpha_dst, ALLEGRO_COLOR *color)
```

Returns the active blender for the current thread. You can pass NULL for values you are not interested in.

See also: al_set_separate_blender (9.6.4), al_get_blender (9.6.1)

9.6.3 al_set_blender

```
void al_set_blender(int op, int src, int dst, ALLEGRO_COLOR color)
```

Sets the function to use for blending for the current thread.

Blending means, the source and destination colors are combined in drawing operations.

Assume the source color (e.g. color of a rectangle to draw, or pixel of a bitmap to draw) is given as its red/green/blue/alpha components (if the bitmap has no alpha it always is assumed to be fully opaque, so 255 for 8-bit or 1.0 for floating point): *sr, sg, sb, sa*. And this color is drawn to a destination, which already has a color: *dr, dg, db, da*.

The conceptional formula used by Allegro to draw any pixel then depends on the `op` parameter:

- ALLEGRO_ADD

```
r = dr * dst + sr * src
g = dg * dst + sg * src
b = db * dst + sb * src
a = da * dst + sa * src
```

- ALLEGRO_DEST_MINUS_SRC

  ```
  r = dr * dst - sr * src
  g = dg * dst - sg * src
  b = db * dst - sb * src
  a = da * dst - sa * src
  ```

- ALLEGRO_SRC_MINUS_DEST

  ```
  r = sr * src - dr * dst
  g = sg * src - dg * dst
  b = sb * src - db * dst
  a = sa * src - da * dst
  ```

Valid values for `src` and `dst` passed to this function are

- ALLEGRO_ZERO

  ```
  src = 0
  dst = 0
  ```

- ALLEGRO_ONE

  ```
  src = 1
  dst = 1
  ```

- ALLEGRO_ALPHA

  ```
  src = sa
  dst = sa
  ```

- ALLEGRO_INVERSE_ALPHA

  ```
  src = 1 - sa
  dst = 1 - sa
  ```

The color parameter specifies the blend color, it is multipled with the source color before the above blending operation.

Blending examples:

So for example, to restore the default of using alpha blending, you would use (pseudo code)

```
al_set_blender(ALLEGRO_ADD, ALLEGRO_ALPHA, ALLEGRO_INVERSE_ALPHA, {1, 1, 1, 1})
```

If in addition you want to draw half transparently

```
al_set_blender(ALLEGRO_ADD, ALLEGRO_ALPHA, ALLEGRO_INVERSE_ALPHA, {1, 1, 1, 0.5})
```

Additive blending would be achieved with

```
al_set_blender(ALLEGRO_ADD, ALLEGRO_ONE, ALLEGRO_ONE, {1, 1, 1, 1})
```

Copying the source to the destination (including alpha) unmodified

```
al_set_blender(ALLEGRO_ADD, ALLEGRO_ONE, ALLEGRO_ZERO, {1, 1, 1, 1})
```

See also: al_set_separate_blender (9.6.4), al_get_blender (9.6.1)

9.6.4 al_set_separate_blender

```
void al_set_separate_blender(int op, int src, int dst,
   int alpha_op, int alpha_src, int alpha_dst, ALLEGRO_COLOR color)
```

Like al_set_blender (9.6.3), but allows specifying a separate blending operation for the alpha channel.

See also: al_set_blender (9.6.3), al_get_blender (9.6.1), al_get_separate_blender (9.6.2)

9.7 Clipping

9.7.1 al_get_clipping_rectangle

```
void al_get_clipping_rectangle(int *x, int *y, int *w, int *h)
```

Gets the clipping rectangle of the target bitmap.

See also: al_set_clipping_rectangle (9.7.2)

9.7.2 al_set_clipping_rectangle

```
void al_set_clipping_rectangle(int x, int y, int width, int height)
```

Set the region of the target bitmap or display that pixels get clipped to. The default is to clip pixels to the entire bitmap.

See also: al_get_clipping_rectangle (9.7.1)

9.8 Graphics utility functions

9.8.1 al_convert_mask_to_alpha

```
void al_convert_mask_to_alpha(ALLEGRO_BITMAP *bitmap, ALLEGRO_COLOR mask_color)
```

Convert the given mask color to an alpha channel in the bitmap. Can be used to convert older 4.2-style bitmaps with magic pink to alpha-ready bitmaps.

See also: ALLEGRO_COLOR (9.1.1)

9.9 Deferred drawing

9.9.1 al_hold_bitmap_drawing

```
void al_hold_bitmap_drawing(bool hold)
```

Enables or disables deferred bitmap drawing. This allows for efficient drawing of many bitmaps that share a parent bitmap, such as sub-bitmaps from a tilesheet or simply identical bitmaps. Drawing bitmaps that do not share a parent is less efficient, so it is advisable to stagger bitmap drawing calls such that the parent bitmap is the same for large number of those calls. While deferred bitmap drawing is enabled, the only functions that can be used are the bitmap drawing functions and font drawing functions. Changing the state such as the blending modes will result in undefined behaviour. However, changing the blending color, but keeping the blending modes the same will work as expected.

No drawing is guaranteed to take place until you disable the hold. Thus, the idiom of this function's usage is to enable the deferred bitmap drawing, draw as many bitmaps as possible, taking care to stagger bitmaps that share parent bitmaps, and then disable deferred drawing. As mentioned above, this function also works with bitmap and truetype fonts, so if multiple lines of text need to be drawn, this function can speed things up.

See also: al_is_bitmap_drawing_held (9.9.2)

9.9.2 al_is_bitmap_drawing_held

```
bool al_is_bitmap_drawing_held(void)
```

Returns whether the deferred bitmap drawing mode is turned on or off.

See also: al_hold_bitmap_drawing (9.9.1)

10 Joystick

These functions are declared in the main Allegro header file:

```
#include <allegro5/allegro.h>
```

10.1 ALLEGRO_JOYSTICK

```
typedef struct ALLEGRO_JOYSTICK ALLEGRO_JOYSTICK;
```

This is an abstract data type representing a physical joystick.

See also: al_get_joystick (10.7)

10.2 ALLEGRO_JOYSTICK_STATE

```
typedef struct ALLEGRO_JOYSTICK_STATE ALLEGRO_JOYSTICK_STATE;
```

This is a structure that is used to hold a "snapshot" of a joystick's axes and buttons at a particular instant. All fields public and read-only.

```
struct {
      float axis[num_axes];             // -1.0 to 1.0
} stick[num_sticks];
int button[num_buttons];                // 0 to 32767
```

See also: al_get_joystick_state (10.18)

10.3 ALLEGRO_JOYFLAGS

```
enum
```

- ALLEGRO_JOYFLAG_DIGITAL - the stick provides digital input
- ALLEGRO_JOYFLAG_ANALOGUE - the stick provides analogue input

(this enum is a holdover from the old API and may be removed)

See also: al_get_joystick_stick_flags (10.14)

10.4 al_install_joystick

```
bool al_install_joystick(void)
```

Install a joystick driver, returning true if successful. If a joystick driver was already installed, returns true immediately.

See also: al_uninstall_joystick (10.5)

10.5 al_uninstall_joystick

```
void al_uninstall_joystick(void)
```

Uninstalls the active joystick driver. All outstanding ALLEGRO_JOYSTICK (10.1) structures are automatically released. If no joystick driver was active, this function does nothing.

This function is automatically called when Allegro is shut down.

See also: al_install_joystick (10.4)

10.6 al_get_num_joysticks

```
int al_get_num_joysticks(void)
```

Return the number of joysticks on the system (depending on the OS this may not be accurate). Returns 0 if there is no joystick driver installed.

See also: al_get_joystick (10.7)

10.7 al_get_joystick

```
ALLEGRO_JOYSTICK *al_get_joystick(int num)
```

Get a handle for joystick number *num* on the system. If successful a pointer to a joystick object is returned. Otherwise NULL is returned.

If the joystick was previously 'gotten' (and not yet released) then the returned pointer will be the same as in previous calls.

See also: al_get_num_joysticks (10.6)

10.8 al_release_joystick

```
void al_release_joystick(ALLEGRO_JOYSTICK *joy)
```

Release a previously 'gotten' joystick object. You do not normally need to do this explicitly as al_uninstall_joystick (10.5) will automatically release joysticks.

See also: al_get_joystick (10.7)

10.9 al_get_joystick_name

```
const char *al_get_joystick_name(ALLEGRO_JOYSTICK *joy)
```

Return the name of the given joystick.

See also: al_get_joystick_stick_name (10.10), al_get_joystick_axis_name (10.11), al_get_joystick_button_name (10.12)

10.10 al_get_joystick_stick_name

```
const char *al_get_joystick_stick_name(const ALLEGRO_JOYSTICK *joy, int stick)
```

Return the name of the given "stick". If the stick doesn't exist, NULL is returned.

See also: al_get_joystick_axis_name (10.11), al_get_joystick_num_sticks (10.15)

10.11 al_get_joystick_axis_name

```
const char *al_get_joystick_axis_name(const ALLEGRO_JOYSTICK *joy, int stick, int axis)
```

Return the name of the given axis. If the axis doesn't exist, NULL is returned.

See also: al_get_joystick_stick_name (10.10), al_get_joystick_num_axes (10.16)

10.12 al_get_joystick_button_name

```
const char *al_get_joystick_button_name(const ALLEGRO_JOYSTICK *joy, int button)
```

Return the name of the given button. If the button doesn't exist, NULL is returned.

See also: al_get_joystick_stick_name (10.10), al_get_joystick_axis_name (10.11), al_get_joystick_num_buttons (10.17)

10.13 al_get_joystick_number

```
int al_get_joystick_number(ALLEGRO_JOYSTICK *joy)
```

Return the joystick number, i.e. the parameter passed to al_get_joystick (10.7).

10.14 al_get_joystick_stick_flags

```
int al_get_joystick_stick_flags(const ALLEGRO_JOYSTICK *joy, int stick)
```

Return the flags of the given "stick". If the stick doesn't exist, NULL is returned.

See also: ALLEGRO_JOYFLAGS (10.3)

10.15 al_get_joystick_num_sticks

```
int al_get_joystick_num_sticks(const ALLEGRO_JOYSTICK *joy)
```

Return the number of "sticks" on the given joystick. A stick has one or more axes.

See also: al_get_joystick_num_axes (10.16), al_get_joystick_num_buttons (10.17)

10.16 al_get_joystick_num_axes

```
int al_get_joystick_num_axes(const ALLEGRO_JOYSTICK *joy, int stick)
```

Return the number of axes on the given "stick". If the stick doesn't exist, 0 is returned.

See also: al_get_joystick_num_sticks (10.15)

10.17 al_get_joystick_num_buttons

```
int al_get_joystick_num_buttons(const ALLEGRO_JOYSTICK *joy)
```

Return the number of buttons on the joystick.

See also: al_get_joystick_num_sticks (10.15)

10.18 al_get_joystick_state

```
void al_get_joystick_state(ALLEGRO_JOYSTICK *joy, ALLEGRO_JOYSTICK_STATE *ret_state)
```

Get the current joystick state.

See also: ALLEGRO_JOYSTICK_STATE (10.2), al_get_joystick_num_buttons (10.17), al_get_joystick_num_axes (10.16)

10.19 al_get_joystick_event_source

```
ALLEGRO_EVENT_SOURCE *al_get_joystick_event_source(ALLEGRO_JOYSTICK *joystick)
```

Retrieve the associated event source.

11 Keyboard

These functions are declared in the main Allegro header file:

```
#include <allegro5/allegro.h>
```

11.1 ALLEGRO_KEYBOARD_STATE

```
typedef struct ALLEGRO_KEYBOARD_STATE ALLEGRO_KEYBOARD_STATE;
```

This is a structure that is used to hold a "snapshot" of a keyboard's state at a particular instant. It contains the following publically readable fields:

- display - points to the display that had keyboard focus at the time the state was saved. If no display was focused, this points to NULL.

You cannot read the state of keys directly. Use the function al_key_down (11.8).

11.2 Key codes

These are the list of key codes used by Allegro, which you can pass to al_key_down (11.8).

```
ALLEGRO_KEY_A ... ALLEGRO_KEY_Z,
ALLEGRO_KEY_0 ... ALLEGRO_KEY_9,
ALLEGRO_KEY_PAD_0 ... ALLEGRO_KEY_PAD_9,
ALLEGRO_KEY_F1 ... ALLEGRO_KEY_F12,
ALLEGRO_KEY_ESCAPE,
ALLEGRO_KEY_TILDE,
ALLEGRO_KEY_MINUS,
ALLEGRO_KEY_EQUALS,
ALLEGRO_KEY_BACKSPACE,
ALLEGRO_KEY_TAB,
ALLEGRO_KEY_OPENBRACE, ALLEGRO_KEY_CLOSEBRACE,
ALLEGRO_KEY_ENTER,
ALLEGRO_KEY_SEMICOLON,
ALLEGRO_KEY_QUOTE,
ALLEGRO_KEY_BACKSLASH, ALLEGRO_KEY_BACKSLASH2,
ALLEGRO_KEY_COMMA,
ALLEGRO_KEY_FULLSTOP,
ALLEGRO_KEY_SLASH,
ALLEGRO_KEY_SPACE,
ALLEGRO_KEY_INSERT, ALLEGRO_KEY_DELETE,
ALLEGRO_KEY_HOME, ALLEGRO_KEY_END,
ALLEGRO_KEY_PGUP, ALLEGRO_KEY_PGDN,
ALLEGRO_KEY_LEFT, ALLEGRO_KEY_RIGHT,
ALLEGRO_KEY_UP, ALLEGRO_KEY_DOWN,
ALLEGRO_KEY_PAD_SLASH, ALLEGRO_KEY_PAD_ASTERISK,
ALLEGRO_KEY_PAD_MINUS, ALLEGRO_KEY_PAD_PLUS,
ALLEGRO_KEY_PAD_DELETE, ALLEGRO_KEY_PAD_ENTER,
ALLEGRO_KEY_PRINTSCREEN, ALLEGRO_KEY_PAUSE,
ALLEGRO_KEY_ABNT_C1, ALLEGRO_KEY_YEN, ALLEGRO_KEY_KANA,
ALLEGRO_KEY_CONVERT, ALLEGRO_KEY_NOCONVERT,
ALLEGRO_KEY_AT, ALLEGRO_KEY_CIRCUMFLEX,
ALLEGRO_KEY_COLON2, ALLEGRO_KEY_KANJI,
ALLEGRO_KEY_LSHIFT, ALLEGRO_KEY_RSHIFT,
ALLEGRO_KEY_LCTRL, ALLEGRO_KEY_RCTRL,
ALLEGRO_KEY_ALT, ALLEGRO_KEY_ALTGR,
ALLEGRO_KEY_LWIN, ALLEGRO_KEY_RWIN,
ALLEGRO_KEY_MENU,
ALLEGRO_KEY_SCROLLLOCK,
ALLEGRO_KEY_NUMLOCK,
ALLEGRO_KEY_CAPSLOCK
ALLEGRO_KEY_EQUALS_PAD,
ALLEGRO_KEY_BACKQUOTE,
ALLEGRO_KEY_SEMICOLON2,
ALLEGRO_KEY_COMMAND
```

11.3 Keyboard modifier flags

```
ALLEGRO_KEYMOD_SHIFT
ALLEGRO_KEYMOD_CTRL
ALLEGRO_KEYMOD_ALT
ALLEGRO_KEYMOD_LWIN
ALLEGRO_KEYMOD_RWIN
ALLEGRO_KEYMOD_MENU
ALLEGRO_KEYMOD_ALTGR
ALLEGRO_KEYMOD_COMMAND
ALLEGRO_KEYMOD_SCROLLLOCK
ALLEGRO_KEYMOD_NUMLOCK
ALLEGRO_KEYMOD_CAPSLOCK
ALLEGRO_KEYMOD_INALTSEQ
ALLEGRO_KEYMOD_ACCENT1
ALLEGRO_KEYMOD_ACCENT2
ALLEGRO_KEYMOD_ACCENT3
ALLEGRO_KEYMOD_ACCENT4
```

11.4 al_install_keyboard

```
bool al_install_keyboard(void)
```

Install a keyboard driver. Returns true if successful. If a driver was already installed, nothing happens and true is returned.

See also: al_uninstall_keyboard (11.6), al_is_keyboard_installed (11.5)

11.5 al_is_keyboard_installed

```
bool al_is_keyboard_installed(void)
```

Returns true if al_install_keyboard (11.4) was called successfully.

11.6 al_uninstall_keyboard

```
void al_uninstall_keyboard(void)
```

Uninstalls the active keyboard driver, if any. This will automatically unregister the keyboard event source with any event queues.

This function is automatically called when Allegro is shut down.

See also: al_install_keyboard (11.4)

11.7 al_get_keyboard_state

```
void al_get_keyboard_state(ALLEGRO_KEYBOARD_STATE *ret_state)
```

Save the state of the keyboard specified at the time the function is called into the structure pointed to by *ret_state*.

See also: al_key_down (11.8), ALLEGRO_KEYBOARD_STATE (11.1)

11.8 al_key_down

```
bool al_key_down(const ALLEGRO_KEYBOARD_STATE *state, int keycode)
```

Return true if the key specified was held down in the state specified.

See also: ALLEGRO_KEYBOARD_STATE (11.1)

11.9 al_keycode_to_name

```
const char *al_keycode_to_name(int keycode)
```

Converts the given keycode to a description of the key.

11.10 al_set_keyboard_leds

```
bool al_set_keyboard_leds(int leds)
```

Overrides the state of the keyboard LED indicators. Set to –1 to return to default behavior. False is returned if the current keyboard driver cannot set LED indicators.

11.11 al_get_keyboard_event_source

```
ALLEGRO_EVENT_SOURCE *al_get_keyboard_event_source(void)
```

Retrieve the keyboard event source.

Returns NULL if the keyboard subsystem was not installed.

12 Memory

These functions are declared in the main Allegro header file:

```
#include <allegro5/allegro.h>
```

12.1 al_set_memory_management_functions

```
void al_set_memory_management_functions(
   void *(*malloc)(void *opaque, size_t size),
   void *(*malloc_atomic)(void *opaque, size_t size),
   void (*free)(void *opaque, void *ptr),
   void *(*realloc)(void *opaque, void *ptr, size_t size),
   void *(*debug_malloc)(int line, const char *file, const char *func,
      void *opaque, size_t size),
   void *(*debug_malloc_atomic)(int line, const char *file, const char *func,
      void *opaque, size_t size),
   void (*debug_free)(int line, const char *file, const char *func,
      void *opaque, void *ptr),
   void *(*debug_realloc)(int line, const char *file, const char *func,
      void *opaque, void *ptr, size_t size),
   void *user_opaque)
```

Customise the memory management functions used by the library. A default function will be used in place of any function pointer which is NULL. The default debug variants simply call the non-debug variants.

- malloc_atomic - malloc() replacement for objects not containing pointers
- user_opaque - will be passed through to all the functions above

13 Miscellaneous

These functions are declared in the main Allegro header file:

```
#include <allegro5/allegro.h>
```

13.1 ALLEGRO_PI

```
#define ALLEGRO_PI        3.14159265358979323846
```

C99 compilers have no predefined value like M_PI for the constant , but you can use this one instead.

13.2 al_run_main

```
int al_run_main(int argc, char **argv, int (*user_main)(int, char **))
```

This function is useful in cases where you don't have a main() function but want to run Allegro (mostly useful in a wrapper library). Under Windows and Linux this is no problem because you simply can call al_install_system (19.1). But some other system (like OSX) don't allow calling al_install_system (19.1) in the main thread. al_run_main will know what to do in that case.

The passed argc and argv will simply be passed on to user_main and the return value of user_main will be returned.

14 Mouse

These functions are declared in the main Allegro header file:

```
#include <allegro5/allegro.h>
```

14.1 ALLEGRO_MOUSE_STATE

```
typedef struct ALLEGRO_MOUSE_STATE ALLEGRO_MOUSE_STATE;
```

Public fields (read only):

- x - mouse x
- y - mouse y
- w, z - Mouse wheel position (2D 'ball')

buttons - mouse buttons

See also: al_get_mouse_state (14.7), al_get_mouse_state_axis (14.8), al_mouse_button_down (14.9)

14.2 al_install_mouse

```
bool al_install_mouse(void)
```

Install a mouse driver.

Returns true if successful. If a driver was already installed, nothing happens and true is returned.

14.3 al_is_mouse_installed

```
bool al_is_mouse_installed(void)
```

Returns true if al_install_mouse (14.2) was called successfully.

14.4 al_uninstall_mouse

```
void al_uninstall_mouse(void)
```

Uninstalls the active mouse driver, if any. This will automatically unregister the mouse event source with any event queues.

This function is automatically called when Allegro is shut down.

14.5 al_get_mouse_num_axes

```
unsigned int al_get_mouse_num_axes(void)
```

Return the number of buttons on the mouse.

14.6 al_get_mouse_num_buttons

```
unsigned int al_get_mouse_num_buttons(void)
```

Return the number of buttons on the mouse.

14.7 al_get_mouse_state

```
void al_get_mouse_state(ALLEGRO_MOUSE_STATE *ret_state)
```

Save the state of the mouse specified at the time the function is called into the given structure.

See also: ALLEGRO_MOUSE_STATE (14.1), al_get_mouse_state_axis (14.8)

14.8 al_get_mouse_state_axis

```
int al_get_mouse_state_axis(ALLEGRO_MOUSE_STATE *ret_state, int axis)
```

Extract the mouse axis value from the saved state.

See also: ALLEGRO_MOUSE_STATE (14.1), al_get_mouse_state (14.7), al_mouse_button_down (14.9)

14.9 al_mouse_button_down

```
bool al_mouse_button_down(ALLEGRO_MOUSE_STATE *state, int button)
```

Return true if the mouse button specified was held down in the state specified.

See also: ALLEGRO_MOUSE_STATE (14.1), al_get_mouse_state (14.7), al_get_mouse_state_axis (14.8)

14.10 al_set_mouse_axis

```
bool al_set_mouse_axis(int which, int value)
```

Set the given mouse axis to the given value.

For now: the axis number must not be 0 or 1, which are the X and Y axes.

Returns true on success, false on failure.

14.11 al_set_mouse_xy

```
bool al_set_mouse_xy(int x, int y)
```

Try to position the mouse at the given coordinates on the current display. The mouse movement resulting from a successful move will generate an ALLEGRO_EVENT_MOUSE_WARPED event.

Returns true on success, false on failure.

See also: al_set_mouse_z (14.12), al_set_mouse_w (14.13)

14.12 al_set_mouse_z

```
bool al_set_mouse_z(int z)
```

Set the mouse wheel position to the given value.

Returns true on success, false on failure.

See also: al_set_mouse_w (14.13)

14.13 al_set_mouse_w

```
bool al_set_mouse_w(int w)
```

Set the second mouse wheel position to the given value.

Returns true on success, false on failure.

See also: al_set_mouse_z (14.12)

14.14 al_get_mouse_event_source

```
ALLEGRO_EVENT_SOURCE *al_get_mouse_event_source(void)
```

Retrieve the mouse event source.

Returns NULL if the mouse subsystem was not installed.

14.15 Mouse cursors

14.15.1 al_create_mouse_cursor

```
ALLEGRO_MOUSE_CURSOR *al_create_mouse_cursor(ALLEGRO_BITMAP *bmp,
   int x_focus, int y_focus)
```

Create a mouse cursor from the bitmap provided. There must be a current display in effect.

Returns a pointer to the cursor on success, or NULL on failure.

See also: al_set_mouse_cursor (14.15.3), al_destroy_mouse_cursor (14.15.2)

14.15.2 al_destroy_mouse_cursor

```
void al_destroy_mouse_cursor(ALLEGRO_MOUSE_CURSOR *cursor)
```

Free the memory used by the given cursor.

The display that was in effect when the cursor was created must still be in effect. XXX that's terrible and should be changed

Has no effect if `cursor` is NULL.

See also: al_create_mouse_cursor (14.15.1)

14.15.3 al_set_mouse_cursor

```
bool al_set_mouse_cursor(ALLEGRO_MOUSE_CURSOR *cursor)
```

Set the given mouse cursor to be the current mouse cursor for the current display.

The display that was in effect when the cursor was created must still be in effect. XXX that's terrible and should be changed

If the cursor is currently 'shown' (as opposed to 'hidden') the change is immediately visible.

Returns true on success, false on failure.

See also: al_set_system_mouse_cursor (14.15.4), al_show_mouse_cursor (14.15.7), al_hide_mouse_cursor (14.15.6)

14.15.4 al_set_system_mouse_cursor

```
bool al_set_system_mouse_cursor(ALLEGRO_SYSTEM_MOUSE_CURSOR cursor_id)
```

Set the given system mouse cursor to be the current mouse cursor for the current display. If the cursor is currently 'shown' (as opposed to 'hidden') the change is immediately visible.

If the cursor doesn't exist on the current platform another cursor will be silently be substituted.

The cursors are:

- ALLEGRO_SYSTEM_MOUSE_CURSOR_DEFAULT
- ALLEGRO_SYSTEM_MOUSE_CURSOR_ARROW
- ALLEGRO_SYSTEM_MOUSE_CURSOR_BUSY
- ALLEGRO_SYSTEM_MOUSE_CURSOR_QUESTION
- ALLEGRO_SYSTEM_MOUSE_CURSOR_EDIT
- ALLEGRO_SYSTEM_MOUSE_CURSOR_MOVE
- ALLEGRO_SYSTEM_MOUSE_CURSOR_RESIZE_N
- ALLEGRO_SYSTEM_MOUSE_CURSOR_RESIZE_W
- ALLEGRO_SYSTEM_MOUSE_CURSOR_RESIZE_S

- ALLEGRO_SYSTEM_MOUSE_CURSOR_RESIZE_E
- ALLEGRO_SYSTEM_MOUSE_CURSOR_RESIZE_NW
- ALLEGRO_SYSTEM_MOUSE_CURSOR_RESIZE_SW
- ALLEGRO_SYSTEM_MOUSE_CURSOR_RESIZE_SE
- ALLEGRO_SYSTEM_MOUSE_CURSOR_RESIZE_NE
- ALLEGRO_SYSTEM_MOUSE_CURSOR_PROGRESS
- ALLEGRO_SYSTEM_MOUSE_CURSOR_PRECISION
- ALLEGRO_SYSTEM_MOUSE_CURSOR_LINK
- ALLEGRO_SYSTEM_MOUSE_CURSOR_ALT_SELECT
- ALLEGRO_SYSTEM_MOUSE_CURSOR_UNAVAILABLE

Returns true on success, false on failure.

See also: al_set_mouse_cursor (14.15.3), al_show_mouse_cursor (14.15.7), al_hide_mouse_cursor (14.15.6)

14.15.5 al_get_mouse_cursor_position

```
bool al_get_mouse_cursor_position(int *ret_x, int *ret_y)
```

On platforms where this information is available, this function returns the global location of the mouse cursor, relative to the desktop. You should not normally use this function, as the information is not useful except for special scenarios as moving a window.

Returns true on success, false on failure.

14.15.6 al_hide_mouse_cursor

```
bool al_hide_mouse_cursor(void)
```

Hide the mouse cursor in the current display of the calling thread. This has no effect on what the current mouse cursor looks like; it just makes it disappear.

Returns true on success (or if the cursor already was hidden), false otherwise.

See also: al_show_mouse_cursor (14.15.7)

14.15.7 al_show_mouse_cursor

```
bool al_show_mouse_cursor(void)
```

Make a mouse cursor visible in the current display of the calling thread.

Returns true if a mouse cursor is shown as a result of the call (or one already was visible), false otherwise.

See also: al_hide_mouse_cursor (14.15.6)

15 OpenGL

These functions are declared in the following header file:

```
#include <allegro5/allegro_opengl.h>
```

15.1 al_get_opengl_extension_list

```
ALLEGRO_OGL_EXT_LIST *al_get_opengl_extension_list(void)
```

Returns the list of OpenGL extensions supported by Allegro, for the current display.

Allegro will keep information about all extensions it knows about in a structure returned by `al_get_opengl_extension_list`.

For example:

```
if (al_get_opengl_extension_list()->ALLEGRO_GL_ARB_multitexture) {
    use it
}
```

The extension will be set to true if available for the current display and false otherwise. This means to use the definitions and functions from an OpenGL extension, all you need to do is to check for it as above at run time, after acquiring the OpenGL display from Allegro.

Under Windows, this will also work with WGL extensions, and under Unix with GLX extensions.

In case you want to manually check for extensions and load function pointers yourself (say, in case the Allegro developers did not include it yet), you can use the al_is_opengl_extension_supported (15.8) and al_get_opengl_proc_address (15.2) functions instead.

15.2 al_get_opengl_proc_address

```
void *al_get_opengl_proc_address(const char *name)
```

Helper to get the address of an OpenGL symbol

Example:

How to get the function *glMultiTexCoord3fARB* that comes with ARB's Multitexture extension:

```
// define the type of the function
   ALLEGRO_DEFINE_PROC_TYPE(void, MULTI_TEX_FUNC,
      (GLenum, GLfloat, GLfloat, GLfloat));
// declare the function pointer
   MULTI_TEX_FUNC glMultiTexCoord3fARB;
// get the address of the function
   glMultiTexCoord3fARB = (MULTI_TEX_FUNC) al_get_opengl_proc_address(
      "glMultiTexCoord3fARB");
```

If *glMultiTexCoord3fARB* is not NULL then it can be used as if it has been defined in the OpenGL core library. Note that the use of the ALLEGRO_DEFINE_PROC_TYPE macro is mandatory if you want your program to be portable.

Parameters:

name - The name of the symbol you want to link to.

Return value:

A pointer to the symbol if available or NULL otherwise.

15.3 al_get_opengl_texture

```
GLuint al_get_opengl_texture(ALLEGRO_BITMAP *bitmap)
```

Returns the OpenGL texture id internally used by the given bitmap if it uses one, else 0.

Example:

```
bitmap = al_load_bitmap("my_texture.png");
texture = al_get_opengl_texture(bitmap);
if (texture != 0)
    glBindTexture(GL_TEXTURE_2D, texture);
```

15.4 al_get_opengl_texture_size

```
void al_get_opengl_texture_size(ALLEGRO_BITMAP *bitmap, int *w, int *h)
```

Retrieves the size of the texture used for the bitmap. This can be different from the bitmap size if OpenGL only supports power-of-two sizes or if it is a sub-bitmap.

15.5 al_get_opengl_texture_position

```
void al_get_opengl_texture_position(ALLEGRO_BITMAP *bitmap, int *u, int *v)
```

Returns the u/v coordinates for the top/left corner of the bitmap within the used texture, in pixels.

15.6 al_get_opengl_fbo

```
GLuint al_get_opengl_fbo(ALLEGRO_BITMAP *bitmap)
```

Returns the OpenGL FBO id internally used by the given bitmap if it uses one, else 0. An FBO is created for a bitmap when you call al_set_target_bitmap (9.5.11) for it.

15.7 al_remove_opengl_fbo

```
void al_remove_opengl_fbo(ALLEGRO_BITMAP *bitmap)
```

If the bitmap has an OpenGL FBO created for it (see al_set_target_bitmap (9.5.11)), it is freed. It also is freed automatically when the bitmap is destroyed.

15.8 al_is_opengl_extension_supported

```
int al_is_opengl_extension_supported(const char *extension)
```

This function is a helper to determine whether an OpenGL extension is available on the current display or not.

Example:

```
int packedpixels = al_is_opengl_extension_supported("GL_EXT_packed_pixels");
```

If *packedpixels* is TRUE then you can safely use the constants related to the packed pixels extension.

Parameters:

extension - The name of the extension that is needed

Return value:

TRUE if the extension is available FALSE otherwise.

15.9 al_get_opengl_version

```
float al_get_opengl_version(void)
```

Returns the OpenGL version number of the client (the computer the program is running on), for the current DISPLAY. “1.0” is returned as 1.0, “1.2.1” is returned as 1.21, and “1.2.2” as 1.22, etc.

A valid OpenGL context must exist for this function to work, which means you may *not* call it before al_create_display (4.1.2).

15.10 OpenGL configuration

You can disable the detection of any OpenGL extension by Allegro with a section like this in allegro5.cfg:

```
[opengl_disabled_extensions]
GL_ARB_texture_non_power_of_two=0
GL_EXT_framebuffer_object=0
```

Any extension which appears in the section is treated as not available (it does not matter if you set it to 0 or any other value).

16 Path

These functions are declared in the main Allegro header file:

```
#include <allegro5/allegro.h>
```

We define a path as an optional *drive*, followed by zero or more *directory components*, followed by an optional *filename*. The filename may be broken up into a *basename* and an *extension*, where the basename includes the start of the filename up to, but not including, the last dot (.) character. If no dot character exists the basename is the whole filename. The extension is everything from the last dot character to the end of the filename.

16.1 al_create_path

```
ALLEGRO_PATH *al_create_path(const char *str)
```

Create a path structure from a string. The string may be NULL for an empty path.

See also: al_destroy_path (16.3)

16.2 al_create_path_for_directory

```
ALLEGRO_PATH *al_create_path_for_directory(const char *str)
```

This is the same as al_create_path (16.1), but interprets the passed string as a directory path. The filename component of the returned path will always be empty.

See also: al_destroy_path (16.3)

16.3 al_destroy_path

```
void al_destroy_path(ALLEGRO_PATH *path)
```

Free a path structure. Does nothing if passed NULL.

See also: al_create_path (16.1), al_create_path_for_directory (16.2)

16.4 al_clone_path

```
ALLEGRO_PATH *al_clone_path(const ALLEGRO_PATH *path)
```

Clones an ALLEGRO_PATH structure. Returns NULL on failure.

See also: al_destroy_path (16.3)

16.5 al_join_paths

```
bool al_join_paths(ALLEGRO_PATH *path, const ALLEGRO_PATH *tail)
```

Concatenate two path structures. The first path structure is modified. If 'tail' is an absolute path, this function does nothing.

If 'tail' is a relative path, all of its directory components will be appended to 'path'. tail's filename will also overwrite path's filename, even if it is just the empty string.

Tail's drive is ignored.

Returns true if 'tail' was a relative path and so concatenated to 'path', otherwise returns false.

16.6 al_get_path_drive

```
const char *al_get_path_drive(const ALLEGRO_PATH *path)
```

Return the drive letter on a path, or the empty string if there is none.

The "drive letter" is only used on Windows, and is usually a string like "c:", but may be something like "\\Computer Name" in the case of UNC (Uniform Naming Convention) syntax.

16.7 al_get_path_num_components

```
int al_get_path_num_components(const ALLEGRO_PATH *path)
```

Return the number of directory components in a path.

The directory components do not include the final part of a path (the filename).

See also: al_get_path_component (16.8)

16.8 al_get_path_component

```
const char *al_get_path_component(const ALLEGRO_PATH *path, int i)
```

Return the i'th directory component of a path, counting from zero. If the index is negative then count from the right, i.e. −1 refers to the last path component. It is an error to pass an index which is out of bounds.

See also: al_get_path_num_components (16.7), al_get_path_tail (16.9)

16.9 al_get_path_tail

```
const char *al_get_path_tail(const ALLEGRO_PATH *path)
```

Returns the last directory component, or NULL if there are no directory components.

16.10 al_get_path_filename

```
const char *al_get_path_filename(const ALLEGRO_PATH *path)
```

Return the filename part of the path, or the empty string if there is none.

The returned pointer is valid only until the filename part of the path is modified in any way, or until the path is destroyed.

See also: al_get_path_basename (16.11), al_get_path_extension (16.12), al_get_path_component (16.8)

16.11 al_get_path_basename

```
const char *al_get_path_basename(const ALLEGRO_PATH *path)
```

Return the basename, i.e. filename with the extension removed. If the filename doesn't have an extension, the whole filename is the basename. If there is no filename part then the empty string is returned.

The returned pointer is valid only until the filename part of the path is modified in any way, or until the path is destroyed.

See also: al_get_path_filename (16.10), al_get_path_extension (16.12)

16.12 al_get_path_extension

```
const char *al_get_path_extension(const ALLEGRO_PATH *path)
```

Return a pointer to the start of the extension of the filename, i.e. everything from the final dot ('.') character onwards. If no dot exists, returns an empty string.

The returned pointer is valid only until the filename part of the path is modified in any way, or until the path is destroyed.

See also: al_get_path_filename (16.10), al_get_path_basename (16.11)

16.13 al_set_path_drive

```
void al_set_path_drive(ALLEGRO_PATH *path, const char *drive)
```

Set the drive string on a path. The drive may be NULL, which is equivalent to setting the drive string to the empty string.

See also: al_get_path_drive (16.6)

16.14 al_append_path_component

```
void al_append_path_component(ALLEGRO_PATH *path, const char *s)
```

Append a directory component.

See also: al_insert_path_component (16.15)

16.15 al_insert_path_component

```
void al_insert_path_component(ALLEGRO_PATH *path, int i, const char *s)
```

Insert a directory component at index i. If the index is negative then count from the right, i.e. -1 refers to the last path component.

It is an error to pass an index i which is not within these bounds: 0 <= i <= al_get_path_num_components(path).

See also: al_append_path_component (16.14), al_replace_path_component (16.16), al_remove_path_component (16.17)

16.16 al_replace_path_component

```
void al_replace_path_component(ALLEGRO_PATH *path, int i, const char *s)
```

Replace the i'th directory component by another string. If the index is negative then count from the right, i.e. −1 refers to the last path component. It is an error to pass an index which is out of bounds.

See also: al_insert_path_component (16.15), al_remove_path_component (16.17)

16.17 al_remove_path_component

```
void al_remove_path_component(ALLEGRO_PATH *path, int i)
```

Delete the i'th directory component. If the index is negative then count from the right, i.e. −1 refers to the last path component. It is an error to pass an index which is out of bounds.

See also: al_insert_path_component (16.15), al_replace_path_component (16.16), al_drop_path_tail (16.18)

16.18 al_drop_path_tail

```
void al_drop_path_tail(ALLEGRO_PATH *path)
```

Remove the last directory component, if any.

See also: al_remove_path_component (16.17)

16.19 al_set_path_filename

```
void al_set_path_filename(ALLEGRO_PATH *path, const char *filename)
```

Set the optional filename part of the path. The filename may be NULL, which is equivalent to setting the filename to the empty string.

See also: al_set_path_extension (16.20), al_get_path_filename (16.10)

16.20 al_set_path_extension

```
bool al_set_path_extension(ALLEGRO_PATH *path, char const *extension)
```

Replaces the extension of the path with the given one, i.e. replaces everything from the final dot ('.') character onwards, including the dot. If the filename of the path has no extension, the given one is appended. Usually the new extension you supply should include a leading dot.

Returns false if the path contains no filename part, i.e. the filename part is the empty string.

See also: al_set_path_filename (16.19), al_get_path_extension (16.12)

16.21 al_path_cstr

```
const char *al_path_cstr(const ALLEGRO_PATH *path, char delim)
```

Convert a path to its string representation, i.e. optional drive, followed by directory components separated by 'delim', followed by an optional filename.

To use the current native path separator, use ALLEGRO_NATIVE_PATH_SEP for 'delim'.

The returned pointer is valid only until the path is modified in any way, or until the path is destroyed.

16.22 al_make_path_absolute

```
bool al_make_path_absolute(ALLEGRO_PATH *path)
```

Prepends the current working directory to 'path' if it is a relative path. The drive is also set to the driver of the current working directory. Does nothing if 'path' is an absolute path.

See also: al_make_path_canonical (16.23)

16.23 al_make_path_canonical

```
bool al_make_path_canonical(ALLEGRO_PATH *path)
```

Removes any leading '..' directory components in absolute paths. Removes all '.' directory components.

Note that this does *not* collapse "x/../y" sections into "y". This is by design. If "/foo" on your system is a symlink to "/bar/baz", then "/foo/../quux" is actually "/bar/quux", not "/quux" as a naive removal of ".." components would give you.

See also: al_make_path_absolute (16.22)

16.24 al_is_path_present

```
bool al_is_path_present(const ALLEGRO_PATH *path)
```

Return true if path represents an existing file on the system, or false if it doesn't exist.

17 Platform-specific functions

17.1 Windows

These functions are declared in the following header file:

```
#include <allegro5/allegro_windows.h>
```

17.1.1 al_get_win_window_handle

```
HWND al_get_win_window_handle(ALLEGRO_DISPLAY *display)
```

Returns the handle to the window that the passed display is using.

18 State

These functions are declared in the main Allegro header file:

```
#include <allegro5/allegro.h>
```

18.1 ALLEGRO_STATE

```
typedef struct ALLEGRO_STATE ALLEGRO_STATE;
```

Opaque type which is passed to al_store_state (18.4)/al_restore_state (18.3).

The various state kept internally by Allegro can be displayed like this:

```
  global
      active system driver
          current config
  per thread
      new bitmap params
      new display params
      active file interface
      errno
      current display
          current transformation
          current blending mode
          deferred drawing
      current target bitmap
          current clipping rectangle
          bitmap locking
```

In general, the only real global state is the active system driver. All other global state is per-thread, so if your application has multiple separate threads they never will interfere with each other. (Except if there are objects accessed by multiple threads of course. Usually you want to minimize that though and for the remaining cases use synchronization primitives described in the threads section or events described in the events section to control inter-thread communication.)

18.2 ALLEGRO_STATE_FLAGS

```
typedef enum ALLEGRO_STATE_FLAGS
```

Flags which can be passed to al_store_state (18.4)/al_restore_state (18.3) as bit combinations. The following flags store or restore settings corresponding to the following al_set_/al_get_ calls:

- ALLEGRO_STATE_NEW_DISPLAY_PARAMETERS - new_display_format, new_display_refresh_rate, new_display_flags
- ALLEGRO_STATE_NEW_BITMAP_PARAMETERS - new_bitmap_format, new_bitmap_flags
- ALLEGRO_STATE_DISPLAY - current_display
- ALLEGRO_STATE_TARGET_BITMAP - target_bitmap
- ALLEGRO_STATE_BLENDER - blender
- ALLEGRO_STATE_TRANSFORM - current_transformation
- ALLEGRO_STATE_NEW_FILE_INTERFACE - new_file_interface
- ALLEGRO_STATE_BITMAP - same as ALLEGRO_STATE_NEW_BITMAP_PARAMETERS and ALLEGRO_STATE_TARGET_BITMAP
- ALLEGRO_STATE_ALL - all of the above

18.3 al_restore_state

```
void al_restore_state(ALLEGRO_STATE const *state)
```

Restores part of the state of the current thread from the given ALLEGRO_STATE (18.1) object.

See also: al_store_state (18.4), ALLEGRO_STATE_FLAGS (18.2)

18.4 al_store_state

```
void al_store_state(ALLEGRO_STATE *state, int flags)
```

Stores part of the state of the current thread in the given ALLEGRO_STATE (18.1) objects. The flags parameter can take any bit-combination of the flags described under ALLEGRO_STATE_FLAGS (18.2).

See also: al_restore_state (18.3)

18.5 al_get_errno

```
int al_get_errno(void)
```

Some Allegro functions will set an error number as well as returning an error code. Call this function to retrieve the last error number set for the calling thread.

18.6 al_set_errno

```
void al_set_errno(int errnum)
```

Set the error number for for the calling thread.

19 System

These functions are declared in the main Allegro header file:

```
#include <allegro5/allegro.h>
```

19.1 al_install_system

```
bool al_install_system(int version, int (*atexit_ptr)(void (*)(void)))
```

Initialize the Allegro system. No other Allegro functions can be called before this (with one or two exceptions).

The version field should always be set to ALLEGRO_VERSION_INT.

If atexit_ptr is non-NULL, and if hasn't been done already, al_uninstall_system (19.3) will be registered as an atexit function.

Returns true if Allegro was successfully initialized by this function call (or already was initialized previously), false if Allegro cannot be used.

See also: al_init (19.2)

19.2 al_init

```
#define al_init()    (al_install_system(ALLEGRO_VERSION_INT, atexit))
```

Like al_install_system (19.1), but automatically passes in the version and uses the atexit function visible in the current compilation unit.

See also: al_install_system (19.1)

19.3 al_uninstall_system

```
void al_uninstall_system(void)
```

Closes down the Allegro system.

Note: al_uninstall_system() can be called without a corresponding al_install_system (19.1) call, e.g. from atexit().

19.4 al_get_allegro_version

```
uint32_t al_get_allegro_version(void)
```

Returns the (compiled) version of the Allegro library, packed into a single integer as groups of 8 bits in the form `(major << 24) | (minor << 16) | (revision << 8) | release`.

You can use code like this to extract them:

```
uint32_t version = al_get_allegro_version();
int major = version >> 24;
int minor = (version >> 16) & 255;
int revision = (version >> 8) & 255;
int release = version & 255;
```

The `release` number is 0 for an unofficial version and 1 or greater for an official release. For example "5.0.2[1]" would be the (first) official 5.0.2 release while "5.0.2[0]" would be a compile of a version from the "5.0.2" branch before the official release.

19.5 al_get_standard_path

```
ALLEGRO_PATH *al_get_standard_path(int id)
```

Gets a system path, depending on the `id` parameter:

id	description
ALLEGRO_PROGRAM_PATH	Directory with the executed program.
ALLEGRO_TEMP_PATH	Path to the directory for temporary files.
ALLEGRO_SYSTEM_DATA_PATH	Data path for system-wide installation.
ALLEGRO_USER_DATA_PATH	Data path for per-user installation.
ALLEGRO_USER_HOME_PATH	Path to the user's home directory.
ALLEGRO_USER_SETTINGS_PATH	Path to per-user settings directory.
ALLE-GRO_SYSTEM_SETTINGS_PATH	Path to system-wide settings directory.
ALLEGRO_EXENAME_PATH	The full path to the executable.

Returns NULL on failure. The returned path should be freed with al_destroy_path (16.3).

See also: al_set_appname (19.6), al_set_orgname (19.7), al_destroy_path (16.3)

19.6 al_set_appname

```
void al_set_appname(const char *appname)
```

Sets the global application name.

The application name is used by al_get_standard_path (19.5) to build the full path to an application's files.

This function may be called before al_init (19.2) or al_install_system (19.1).

See also: al_get_appname (19.8), al_set_orgname (19.7)

19.7 al_set_orgname

```
void al_set_orgname(const char *orgname)
```

Sets the global organization name.

The organization name is used by al_get_standard_path (19.5) to build the full path to an application's files.

This function may be called before al_init (19.2) or al_install_system (19.1).

See also: al_get_orgname (19.9), al_set_appname (19.6)

19.8 al_get_appname

```
const char *al_get_appname(void)
```

Returns the global application name string.

See also: al_set_appname (19.6)

19.9 al_get_orgname

```
const char *al_get_orgname(void)
```

Returns the global organization name string.

See also: al_set_orgname (19.7)

19.10 al_get_system_driver

```
ALLEGRO_SYSTEM *al_get_system_driver(void)
```

Returns the currently active system driver, or NULL.

19.11 al_get_system_config

```
ALLEGRO_CONFIG *al_get_system_config(void)
```

Returns the configuration for the installed system, if any, or NULL otherwise. This is mainly used for configuring Allegro and its addons.

20 Threads

Allegro 4.9 includes a simple cross-platform threading interface. It is a thin layer on top of two threading APIs: Windows threads and POSIX Threads (pthreads). Enforcing a consistent semantics on all platforms would be difficult at best, hence the behaviour of the following functions will differ subtly on different platforms (more so than usual). Your best bet is to be aware of this and code to the intersection of the semantics and avoid edge cases.

These functions are declared in the main Allegro header file:

```
#include <allegro5/allegro.h>
```

20.1 ALLEGRO_THREAD

```
typedef struct ALLEGRO_THREAD ALLEGRO_THREAD;
```

An opaque structure representing a thread.

20.2 ALLEGRO_MUTEX

```
typedef struct ALLEGRO_MUTEX ALLEGRO_MUTEX;
```

An opaque structure representing a mutex.

20.3 ALLEGRO_COND

```
typedef struct ALLEGRO_COND ALLEGRO_COND;
```

An opaque structure representing a condition variable.

20.4 al_create_thread

```
ALLEGRO_THREAD *al_create_thread(
   void *(*proc)(ALLEGRO_THREAD *thread, void *arg), void *arg)
```

Spawn a new thread which begins executing `proc`. The new thread is passed its own thread handle and the value `arg`.

Returns true if the thread was created, false if there was an error.

See also: al_start_thread (20.5), al_join_thread (20.6).

20.5 al_start_thread

```
void al_start_thread(ALLEGRO_THREAD *thread)
```

When a thread is created, it is initially in a suspended state. Calling al_start_thread (20.5) will start its actual execution.

Starting a thread which has already been started does nothing.

See also: al_create_thread (20.4).

20.6 al_join_thread

```
void al_join_thread(ALLEGRO_THREAD *thread, void **ret_value)
```

Wait for the thread to finish executing. This implicitly calls al_set_thread_should_stop (20.7) first.

If `ret_value` is non-`NULL`, the value returned by the thread function will be stored at the location pointed to by `ret_value`.

See also: al_set_thread_should_stop (20.7), al_get_thread_should_stop (20.8), al_destroy_thread (20.9).

20.7 al_set_thread_should_stop

```
void al_set_thread_should_stop(ALLEGRO_THREAD *thread)
```

Set the flag to indicate `thread` should stop. Returns immediately.

See also: al_join_thread (20.6), al_get_thread_should_stop (20.8).

20.8 al_get_thread_should_stop

```
bool al_get_thread_should_stop(ALLEGRO_THREAD *thread)
```

Check if another thread is waiting for `thread` to stop. Threads which run in a loop should check this periodically and act on it when convenient.

Returns true if another thread has called al_join_thread (20.6) or al_set_thread_should_stop (20.7) on this thread.

See also: al_join_thread (20.6), al_set_thread_should_stop (20.7).

Note: We don't support forceful killing of threads.

20.9 al_destroy_thread

```
void al_destroy_thread(ALLEGRO_THREAD *thread)
```

Free the resources used by a thread. Implicitly performs al_join_thread (20.6) on the thread if it hasn't been done already.

Does nothing if `thread` is NULL.

See also: al_join_thread (20.6).

20.10 al_run_detached_thread

```
void al_run_detached_thread(void *(*proc)(void *arg), void *arg)
```

Runs the passed function in its own thread, with `arg` passed to it as only parameter. This is similar to calling al_create_thread (20.4), al_start_thread (20.5) and (after the thread has finished) al_destroy_thread (20.9) - but you don't have the possibility of ever calling al_join_thread (20.6) on the thread any longer.

20.11 al_create_mutex

```
ALLEGRO_MUTEX *al_create_mutex(void)
```

Create the mutex object (a mutual exclusion device). The mutex may or may not support "recursive" locking.

Returns the mutex on success or `NULL` on error.

See also: al_create_mutex_recursive (20.12).

20.12 al_create_mutex_recursive

```
ALLEGRO_MUTEX *al_create_mutex_recursive(void)
```

Create the mutex object (a mutual exclusion device), with support for "recursive" locking. That is, the mutex will count the number of times it has been locked by the same thread. If the caller tries to acquire a lock on the mutex when it already holds the lock then the count is incremented. The mutex is only unlocked when the thread releases the lock on the mutex an equal number of times, i.e. the count drops down to zero.

See also: al_create_mutex (20.11).

20.13 al_lock_mutex

```
void al_lock_mutex(ALLEGRO_MUTEX *mutex)
```

Acquire the lock on `mutex`. If the mutex is already locked by another thread, the call will block until the mutex becomes available and locked.

If the mutex is already locked by the calling thread, then the behaviour depends on whether the mutex was created with al_create_mutex (20.11) or al_create_mutex_recursive (20.12). In the former case, the behaviour is undefined; the most likely behaviour is deadlock. In the latter case, the count in the mutex will be incremented and the call will return immediately.

See also: al_unlock_mutex (20.14).

We don't yet have al_mutex_trylock.

20.14 al_unlock_mutex

```
void al_unlock_mutex(ALLEGRO_MUTEX *mutex)
```

Release the lock on `mutex` if the calling thread holds the lock on it.

If the calling thread doesn't hold the lock, or if the mutex is not locked, undefined behaviour results.

See also: al_lock_mutex (20.13).

20.15 al_destroy_mutex

```
void al_destroy_mutex(ALLEGRO_MUTEX *mutex)
```

Free the resources used by the mutex. The mutex should be unlocked. Destroying a locked mutex results in undefined behaviour.

Does nothing if `mutex` is `NULL`.

20.16 al_create_cond

```
ALLEGRO_COND *al_create_cond(void)
```

Create a condition variable.

Returns the condition value on success or `NULL` on error.

20.17 al_destroy_cond

```
void al_destroy_cond(ALLEGRO_COND *cond)
```

Destroy a condition variable.

Destroying a condition variable which has threads block on it results in undefined behaviour.

Does nothing if `cond` is `NULL`.

20.18 al_wait_cond

```
void al_wait_cond(ALLEGRO_COND *cond, ALLEGRO_MUTEX *mutex)
```

On entering this function, `mutex` must be locked by the calling thread. The function will atomically release `mutex` and block on `cond`. The function will return when `cond` is "signalled", acquiring the lock on the mutex in the process.

Example of proper use:

```
al_lock_mutex(mutex);
while (something_not_true) {
    al_wait_cond(cond, mutex);
}
do_something();
al_unlock_mutex(mutex);
```

The mutex should be locked before checking the condition, and should be rechecked al_wait_cond (20.18) returns. al_wait_cond (20.18) can return for other reasons than the condition becoming true (e.g. the process was signalled). If multiple threads are blocked on the condition variable, the condition may no longer be true by the time the second and later threads are unblocked. Remember not to unlock the mutex prematurely.

See also: al_wait_cond_timed (20.19), al_broadcast_cond (20.20), al_signal_cond (20.21).

20.19 al_wait_cond_timed

```
int al_wait_cond_timed(ALLEGRO_COND *cond, ALLEGRO_MUTEX *mutex,
   const ALLEGRO_TIMEOUT *timeout)
```

Like al_wait_cond (20.18) but the call can return if the absolute time passes `timeout` before the condition is signalled.

Returns zero on success, non-zero if the call timed out.

Fix up the return value. pthread_cond_timedwait returns ETIMEDOUT but can return other errors. Do we need to return other errors?

20.20 al_broadcast_cond

```
void al_broadcast_cond(ALLEGRO_COND *cond)
```

Unblock all threads currently waiting on a condition variable. That is, broadcast that some condition which those threads were waiting for has become true.

See also: al_signal_cond (20.21).

Note: The pthreads spec says to lock the mutex associated with `cond` before signalling for predictable scheduling behaviour.

20.21 al_signal_cond

```
void al_signal_cond(ALLEGRO_COND *cond)
```

Unblock at least one thread waiting on a condition variable.

Generally you should use al_broadcast_cond (20.20) but al_signal_cond (20.21) may be more efficient when it's applicable.

See also: al_broadcast_cond (20.20).

21 Time routines

These functions are declared in the main Allegro header file:

```
#include <allegro5/allegro.h>
```

21.1 ALLEGRO_TIMEOUT

```
typedef struct ALLEGRO_TIMEOUT ALLEGRO_TIMEOUT;
```

Represent a timeout value. The size of the structure is known so can be statically allocated. The contents are private.

See also: al_init_timeout (21.3)

21.2 al_current_time

```
double al_current_time(void)
```

Return the number of seconds since the Allegro library was initialised. The return value is undefined if Allegro is uninitialised. The resolution depends on the used driver, but typically can be in the order of microseconds.

21.3 al_init_timeout

```
void al_init_timeout(ALLEGRO_TIMEOUT *timeout, double seconds)
```

Set timeout value of some number of seconds after the function call.

See also: ALLEGRO_TIMEOUT (21.1)

21.4 al_rest

```
void al_rest(double seconds)
```

Waits for the specified number seconds. This tells the system to pause the current thread for the given amount of time. With some operating systems, the accuracy can be in the order of 10ms. That is, even

```
al_rest(0.000001)
```

might pause for something like 10ms. Also see the section on easier ways to time your program without using up all CPU.

22 Timer

These functions are declared in the main Allegro header file:

```
#include <allegro5/allegro.h>
```

22.1 ALLEGRO_TIMER

```
typedef struct ALLEGRO_TIMER ALLEGRO_TIMER;
```

This is an abstract data type representing a timer object.

22.2 ALLEGRO_USECS_TO_SECS

```
#define ALLEGRO_USECS_TO_SECS(x)      ((x) / 1000000.0)
```

Convert microseconds to seconds.

22.3 ALLEGRO_MSECS_TO_SECS

```
#define ALLEGRO_MSECS_TO_SECS(x)      ((x) / 1000.0)
```

Convert milliseconds to seconds.

22.4 ALLEGRO_BPS_TO_SECS

```
#define ALLEGRO_BPS_TO_SECS(x)        (1.0 / (x))
```

Convert beats per second to seconds.

22.5 ALLEGRO_BPM_TO_SECS

```
#define ALLEGRO_BPM_TO_SECS(x)        (60.0 / (x))
```

Convert beats per minute to seconds.

22.6 al_install_timer

```
ALLEGRO_TIMER* al_install_timer(double speed_secs)
```

Install a new timer. If successful, a pointer to a new timer object is returned, otherwise NULL is returned. *speed_secs* is in seconds per "tick", and must be positive. The new timer is initially stopped.

The system driver must be installed before this function can be called.

Usage note: typical granularity is on the order of microseconds, but with some drivers might only be milliseconds.

See also: al_start_timer (22.7), al_uninstall_timer (22.10)

22.7 al_start_timer

```
void al_start_timer(ALLEGRO_TIMER *timer)
```

Start the timer specified. From then, the timer's counter will increment at a constant rate, and it will begin generating events. Starting a timer that is already started does nothing.

See also: al_stop_timer (22.8), al_timer_is_started (22.9)

22.8 al_stop_timer

```
void al_stop_timer(ALLEGRO_TIMER *timer)
```

Stop the timer specified. The timer's counter will stop incrementing and it will stop generating events. Stopping a timer that is already stopped does nothing.

See also: al_start_timer (22.7), al_timer_is_started (22.9)

22.9 al_timer_is_started

```
bool al_timer_is_started(const ALLEGRO_TIMER *timer)
```

Return true if the timer specified is currently started.

22.10 al_uninstall_timer

```
void al_uninstall_timer(ALLEGRO_TIMER *timer)
```

Uninstall the timer specified. If the timer is started, it will automatically be stopped before uninstallation. It will also automatically unregister the timer with any event queues.

Does nothing if passed the NULL pointer.

See also: al_install_timer (22.6)

22.11 al_get_timer_count

```
int64_t al_get_timer_count(const ALLEGRO_TIMER *timer)
```

Return the timer's counter value. The timer can be started or stopped.

See also: al_set_timer_count (22.12)

22.12 al_set_timer_count

```
void al_set_timer_count(ALLEGRO_TIMER *timer, int64_t new_count)
```

Change a timer's counter value. The timer can be started or stopped. The count value may be positive or negative, but will always be incremented by +1.

See also: al_get_timer_count (22.11)

22.13 al_get_timer_speed

```
double al_get_timer_speed(const ALLEGRO_TIMER *timer)
```

Return the timer's speed, in seconds.

See also: al_set_timer_speed (22.14)

22.14 al_set_timer_speed

```
void al_set_timer_speed(ALLEGRO_TIMER *timer, double new_speed_secs)
```

Set the timer's speed, i.e. the rate at which its counter will be incremented when it is started. This can be done when the timer is started or stopped. If the timer is currently running, it is made to look as though the speed change occured precisely at the last tick.

speed_secs has exactly the same meaning as with al_install_timer (22.6).

See also: al_get_timer_speed (22.13)

22.15 al_get_timer_event_source

```
ALLEGRO_EVENT_SOURCE *al_get_timer_event_source(ALLEGRO_TIMER *timer)
```

Retrieve the associated event source.

23 Transformations

These functions are declared in the main Allegro header file:

```
#include <allegro5/allegro.h>
```

23.1 ALLEGRO_TRANSFORM

```
typedef struct ALLEGRO_TRANSFORM ALLEGRO_TRANSFORM;
```

Defines the generic transformation type, a 4x4 matrix. 2D transforms use only a small subsection of this matrix, namely the top left 2x2 matrix, and the right most 2x1 matrix, for a total of 6 values.

Fields:

- m - A 4x4 float matrix

23.2 al_copy_transform

```
void al_copy_transform(const ALLEGRO_TRANSFORM *src, ALLEGRO_TRANSFORM *dest)
```

Makes a copy of a transformation.

Parameters:

- src - Source transformation
- dest - Destination transformation

23.3 al_use_transform

```
void al_use_transform(const ALLEGRO_TRANSFORM *trans)
```

Sets the transformation to be used for the the drawing operations. Every drawing operation after this call will be transformed using this transformation. Call this function with an identity transformation to return to the default behaviour.

Parameters:

- trans - Transformation to use

23.4 al_get_current_transform

```
const ALLEGRO_TRANSFORM *al_get_current_transform()
```

Returns the current transformation, as set by al_use_transform (23.3).

Returns: A pointer to the current transformation.

23.5 al_invert_transform

```
void al_invert_transform(ALLEGRO_TRANSFORM *trans)
```

Inverts the passed transformation. If the transformation is nearly singular (close to not having an inverse) then the returned transformation may be invalid. Use al_check_inverse (23.6) to assertain if the transformation has an inverse before inverting it if you are in doubt.

Parameters:

- trans - Transformation to invert

See Also: al_check_inverse (23.6)

23.6 al_check_inverse

```
int al_check_inverse(const ALLEGRO_TRANSFORM *trans, float tol)
```

Checks if the transformation has an inverse using the supplied tolerance. Tolerance should be a small value between 0 and 1, with 0.001 being sufficient for most applications. Note that this check is superfluous most of the time if you never touched the transformation matrix values yourself. The only thing that would cause the transformation to not have an inverse is if you applied a 0 (or very small) scale to the transformation. As long as the scale is comfortably above 0, the transformation will be invertible.

Parameters:

- trans - Transformation to check
- tol - Tolerance

Returns: 1 if the transformation is invertible, 0 otherwise

See Also: al_invert_transform (23.5)

23.7 al_identity_transform

```
void al_identity_transform(ALLEGRO_TRANSFORM *trans)
```

Sets the transformation to be the identity transformation.

Parameters:

- trans - Transformation to alter

23.8 al_build_transform

```
void al_build_transform(ALLEGRO_TRANSFORM *trans, float x, float y,
   float sx, float sy, float theta)
```

Builds a transformation given some parameters. This call is equivalent to calling the transformations in this order: make identity, scale, rotate, translate. This method is faster, however, than actually calling those functions.

Parameters:

- trans - Transformation to alter
- x, y - Translation
- sx, sy - Scale
- theta - Rotation angle

23.9 al_translate_transform

```
void al_translate_transform(ALLEGRO_TRANSFORM *trans, float x, float y)
```

Apply a translation to a transformation.

Parameters:

- trans - Transformation to alter
- x, y - Translation

23.10 al_rotate_transform

```
void al_rotate_transform(ALLEGRO_TRANSFORM *trans, float theta)
```

Apply a rotation to a transformation.

Parameters:

- trans - Transformation to alter
- theta - Rotation angle

23.11 al_scale_transform

```
void al_scale_transform(ALLEGRO_TRANSFORM *trans, float sx, float sy)
```

Apply a scale to a transformation.

Parameters:

- trans - Transformation to alter
- sx, sy - Scale

23.12 al_transform_coordinates

```
void al_transform_coordinates(const ALLEGRO_TRANSFORM *trans, float *x, float *y)
```

Transform a pair of coordinates.

Parameters:

- trans - Transformation to use
- x, y - Pointers to the coordinates

23.13 al_transform_transform

```
void al_transform_transform(const ALLEGRO_TRANSFORM *trans, ALLEGRO_TRANSFORM *trans2)
```

Transform a transformation.

Parameters:

- trans - Transformation to use
- trans2 - Transformation to transform

24 UTF–8 string routines

These functions are declared in the main Allegro header file:

```
#include <allegro5/allegro.h>
```

24.1 About Unicode

Here we should give a short overview of Unicode/UCS and in particular UTF–8 encoding.

Explain about code points and relationship to "characters".

Explain about half-open intervals.

Always be careful whether a function takes byte offsets or code-point indices. In general, all position parameters are in byte offsets, not code point indices. This may be surprising, but it is because the functions are designed to be highly performant and to also work with arbitrary byte buffers. Therefore UTF8 decoding is not done by default.

For actual text processing, where you want to specify positions with code point indices, you should use al_ustr_offset (24.6.3) to find the byte position of a code point.

24.2 UTF–8 string types

24.2.1 ALLEGRO_USTR

```
typedef struct _al_tagbstring ALLEGRO_USTR;
```

24.2.2 ALLEGRO_USTR_INFO

```
typedef struct ALLEGRO_USTR_INFO ALLEGRO_USTR_INFO;
```

24.3 Creating and destroying strings

24.3.1 al_ustr_new

```
ALLEGRO_USTR *al_ustr_new(const char *s)
```

Create a new string containing a copy of the C-style string `s`. The string must eventually be freed with al_ustr_free (24.3.4).

24.3.2 al_ustr_new_from_buffer

```
ALLEGRO_USTR *al_ustr_new_from_buffer(const char *s, size_t size)
```

Create a new string containing a copy of the buffer pointed to by `s` of the given size. The string must eventually be freed with al_ustr_free (24.3.4).

24.3.3 al_ustr_newf

```
ALLEGRO_USTR *al_ustr_newf(const char *fmt, ...)
```

Create a new string using a printf-style format string.

Notes:

That "%s" specifier takes C string arguments, not ALLEGRO_USTRs. Therefore to pass an ALLEGRO_USTR as a parameter you must use al_cstr (24.3.5), and it must be NUL terminated. If the string contains an embedded NUL byte everything from that byte onwards will be ignored.

The "%c" specifier outputs a single byte, not the UTF–8 encoding of a code point. Therefore it's only usable for ASCII characters (value <= 127) or if you really mean to output byte values from 128—255. To insert the UTF–8 encoding of a code point, encode it into a memory buffer using al_utf8_encode (24.16.2) then use the "%s" specifier. Remember to NUL terminate the buffer.

24.3.4 al_ustr_free

```
void al_ustr_free(ALLEGRO_USTR *us)
```

Free a previously allocated string.

24.3.5 al_cstr

```
const char *al_cstr(const ALLEGRO_USTR *us)
```

Get a `char *` pointer to the data in a string. This pointer will only be valid while the underlying string is not modified and not destroyed. The pointer may be passed to functions expecting C-style strings, with the following caveats:

- ALLEGRO_USTRs are allowed to contain embedded NUL ('\0') bytes. That means `al_ustr_size(u)` and `strlen(al_cstr(u))` may not agree.
- An ALLEGRO_USTR may be created in such a way that it is not NUL terminated. A string which is dynamically allocated will always be NUL terminated, but a string which references the middle of another string or region of memory will *not* be NUL terminated.
- If the ALLEGRO_USTR references another string, the returned c-string will point into the referenced string, the length of the string will be ignored.

24.3.6 al_ustr_to_buffer

```
void al_ustr_to_buffer(const ALLEGRO_USTR *us, char *buffer, int size)
```

Write the contents of the string into a pre-allocated buffer of the given size in bytes. The result will always be 0-terminated.

24.3.7 al_cstr_dup

```
char *al_cstr_dup(const ALLEGRO_USTR *us)
```

Create a NUL ('\0') terminated copy of the string. Any embedded NUL bytes will still be presented in the returned string. The new string must eventually be freed with free(). If an error occurs NULL is returned.

[after we introduce al_free it should be freed with al_free]

24.3.8 al_ustr_dup

```
ALLEGRO_USTR *al_ustr_dup(const ALLEGRO_USTR *us)
```

Return a duplicate copy of a string. The new string will need to be freed with al_ustr_free (24.3.4).

24.3.9 al_ustr_dup_substr

```
ALLEGRO_USTR *al_ustr_dup_substr(const ALLEGRO_USTR *us, int start_pos,
   int end_pos)
```

Return a new copy of a string, containing its contents in the byte interval [start_pos, end_pos). The new string will be NUL terminated and will need to be freed with al_ustr_free (24.3.4).

If you need a range of code-points instead of bytes, use al_ustr_offset (24.6.3) to find the byte offsets.

24.4 Predefined strings

24.4.1 al_ustr_empty_string

```
ALLEGRO_USTR *al_ustr_empty_string(void)
```

Return a pointer to a static empty string. The string is read only.

24.5 Creating strings by referencing other data

24.5.1 al_ref_cstr

```
ALLEGRO_USTR *al_ref_cstr(ALLEGRO_USTR_INFO *info, const char *s)
```

Create a string that references the storage of a C-style string. The information about the string (e.g. its size) is stored in the structure pointed to by the `info` parameter. The string will not have any other storage allocated of its own, so if you allocate the `info` structure on the stack then no explicit "free" operation is required.

The string is valid until the underlying C string disappears.

Example:

```
ALLEGRO_USTR_INFO info;
ALLEGRO_USTR us = al_ref_cstr(&info, "my string");
```

24.5.2 al_ref_buffer

```
ALLEGRO_USTR *al_ref_buffer(ALLEGRO_USTR_INFO *info, const char *s, size_t size)
```

Like al_ref_cstr (24.5.1) but the size of the string data is passed in as a parameter. Hence you can use it to reference only part of a string or an arbitrary region of memory.

The string is valid while the underlying C string is valid.

24.5.3 al_ref_ustr

```
ALLEGRO_USTR *al_ref_ustr(ALLEGRO_USTR_INFO *info, const ALLEGRO_USTR *us,
   int start_pos, int end_pos)
```

Create a read-only string that references the storage of another string. The information about the string (e.g. its size) is stored in the structure pointed to by the `info` parameter. The string will not have any other storage allocated of its own, so if you allocate the `info` structure on the stack then no explicit "free" operation is required.

The referenced interval is [start_pos, end_pos).

The string is valid until the underlying string is modified or destroyed.

If you need a range of code-points instead of bytes, use al_ustr_offset (24.6.3) to find the byte offsets.

24.6 Sizes and offsets

24.6.1 al_ustr_size

```
size_t al_ustr_size(const ALLEGRO_USTR *us)
```

Return the size of the string in bytes. This is equal to the number of code points in the string if the string is empty or contains only 7-bit ASCII characters.

24.6.2 al_ustr_length

```
size_t al_ustr_length(const ALLEGRO_USTR *us)
```

Return the number of code points in the string.

24.6.3 al_ustr_offset

```
int al_ustr_offset(const ALLEGRO_USTR *us, int index)
```

Return the offset (in bytes from the start of the string) of the code point at the specified index in the string. A zero index parameter will return the first character of the string. If index is negative, it counts backward from the end of the string, so an index of −1 will return an offset to the last code point.

If the index is past the end of the string, returns the offset of the end of the string.

24.6.4 al_ustr_next

```
bool al_ustr_next(const ALLEGRO_USTR *us, int *pos)
```

Find the byte offset of the next code point in string, beginning at `*pos`. `*pos` does not have to be at the beginning of a code point. Returns true on success, then value pointed to by `pos` will be updated to the found offset. Otherwise returns false if `*pos` was already at the end of the string, then `*pos` is unmodified.

This function just looks for an appropriate byte; it doesn't check if found offset is the beginning of a valid code point. If you are working with possibly invalid UTF–8 strings then it could skip over some invalid bytes.

24.6.5 al_ustr_prev

```
bool al_ustr_prev(const ALLEGRO_USTR *us, int *pos)
```

Find the byte offset of the previous code point in string, before `*pos`. `*pos` does not have to be at the beginning of a code point. Returns true on success, then value pointed to by `pos` will be updated to the found offset. Otherwise returns false if `*pos` was already at the end of the string, then `*pos` is unmodified.

This function just looks for an appropriate byte; it doesn't check if found offset is the beginning of a valid code point. If you are working with possibly invalid UTF–8 strings then it could skip over some invalid bytes.

24.7 Getting code points

24.7.1 al_ustr_get

```
int32_t al_ustr_get(const ALLEGRO_USTR *ub, int pos)
```

Return the code point in `us` beginning at `pos`.

On success returns the code point value. If `pos` was out of bounds (e.g. past the end of the string), return –1. On an error, such as an invalid byte sequence, return –2.

24.7.2 al_ustr_get_next

```
int32_t al_ustr_get_next(const ALLEGRO_USTR *us, int *pos)
```

Find the code point in `us` beginning at `*pos`, then advance to the next code point.

On success return the code point value. If `pos` was out of bounds (e.g. past the end of the string), return –1. On an error, such as an invalid byte sequence, return –2. As with al_ustr_next (24.6.4), invalid byte sequences may be skipped while advancing.

24.7.3 al_ustr_prev_get

```
int32_t al_ustr_prev_get(const ALLEGRO_USTR *us, int *pos)
```

Find the beginning of a code point before `*pos`, then return it. Note this performs a *pre-increment*.

On success returns the code point value. If `pos` was out of bounds (e.g. past the end of the string), return –1. On an error, such as an invalid byte sequence, return –2. As with al_ustr_prev (24.6.5), invalid byte sequences may be skipped while advancing.

24.8 Inserting into strings

24.8.1 al_ustr_insert

```
bool al_ustr_insert(ALLEGRO_USTR *us1, int pos, const ALLEGRO_USTR *us2)
```

Insert `us2` into `us1` beginning at `pos`. `pos` cannot be less than 0. If `pos` is past the end of `us1` then the space between the end of the string and `pos` will be padded with NUL ('\0') bytes. `pos` is specified in bytes.

Use al_ustr_offset (24.6.3) to find the byte offset for a code-points offset

Returns true on success, false on error.

24.8.2 al_ustr_insert_cstr

```
bool al_ustr_insert_cstr(ALLEGRO_USTR *us, int pos, const char *s)
```

Like al_ustr_insert (24.8.1) but inserts a C-style string.

24.8.3 al_ustr_insert_chr

```
size_t al_ustr_insert_chr(ALLEGRO_USTR *us, int pos, int32_t c)
```

Insert a code point into `us` beginning at byte offset `pos`. `pos` cannot be less than 0. If `pos` is past the end of `us` then the space between the end of the string and `pos` will be padded with NUL ('\0') bytes.

Returns the number of bytes inserted, or 0 on error.

24.9 Appending to strings

24.9.1 al_ustr_append

```
bool al_ustr_append(ALLEGRO_USTR *us1, const ALLEGRO_USTR *us2)
```

Append `us2` to the end of `us1`.

Returns true on success, false on error.

24.9.2 al_ustr_append_cstr

```
bool al_ustr_append_cstr(ALLEGRO_USTR *us, const char *s)
```

Append C-style string `s` to the end of `us`.

Returns true on success, false on error.

24.9.3 al_ustr_append_chr

```
size_t al_ustr_append_chr(ALLEGRO_USTR *us, int32_t c)
```

Append a code point to the end of `us`.

Returns the number of bytes added, or 0 on error.

24.9.4 al_ustr_appendf

```
bool al_ustr_appendf(ALLEGRO_USTR *us, const char *fmt, ...)
```

This function appends formatted output to the string `us`. `fmt` is a printf-style format string. See al_ustr_newf (24.3.3) about the "%s" and "%c" specifiers.

Returns true on success, false on error.

24.9.5 al_ustr_vappendf

```
bool al_ustr_vappendf(ALLEGRO_USTR *us, const char *fmt, va_list ap)
```

Like al_ustr_appendf (24.9.4) but you pass the variable argument list directly, instead of the arguments themselves. See al_ustr_newf (24.3.3) about the "%s" and "%c" specifiers.

Returns true on success, false on error.

24.10 Removing parts of strings

24.10.1 al_ustr_remove_chr

```
bool al_ustr_remove_chr(ALLEGRO_USTR *us, int pos)
```

Remove the code point beginning at byte offset `pos`. Returns true on success. If `pos` is out of range or `pos` is not the beginning of a valid code point, returns false leaving the string unmodified.

Use al_ustr_offset (24.6.3) to find the byte offset for a code-points offset.

24.10.2 al_ustr_remove_range

```
bool al_ustr_remove_range(ALLEGRO_USTR *us, int start_pos, int end_pos)
```

Remove the interval [start_pos, end_pos) (in bytes) from a string. `start_pos` and `end_pos` may both be past the end of the string but cannot be less than 0 (the start of the string).

Returns true on success, false on error.

24.10.3 al_ustr_truncate

```
bool al_ustr_truncate(ALLEGRO_USTR *us, int start_pos)
```

Truncate a portion of a string at byte offset `start_pos` onwards. `start_pos` can be past the end of the string (has no effect) but cannot be less than 0.

Returns true on success, false on error.

24.10.4 al_ustr_ltrim_ws

```
bool al_ustr_ltrim_ws(ALLEGRO_USTR *us)
```

Remove leading whitespace characters from a string, as defined by the C function `isspace()`.

Returns true on success, or false if the function was passed an empty string.

24.10.5 al_ustr_rtrim_ws

```
bool al_ustr_rtrim_ws(ALLEGRO_USTR *us)
```

Remove trailing ("right") whitespace characters from a string, as defined by the C function `isspace()`.

Returns true on success, or false if the function was passed an empty string.

24.10.6 al_ustr_trim_ws

```
bool al_ustr_trim_ws(ALLEGRO_USTR *us)
```

Remove both leading and trailing whitespace characters from a string.

Returns true on success, or false if the function was passed an empty string.

24.11 Assigning one string to another

24.11.1 al_ustr_assign

```
bool al_ustr_assign(ALLEGRO_USTR *us1, const ALLEGRO_USTR *us2)
```

Overwrite the string `us1` with another string `us2`. Returns true on success, false on error.

24.11.2 al_ustr_assign_substr

```
bool al_ustr_assign_substr(ALLEGRO_USTR *us1, const ALLEGRO_USTR *us2,
   int start_pos, int end_pos)
```

Overwrite the string `us1` with the contents of `us2` in the byte interval [start_pos, end_pos). The end points will be clamed to the bounds of `us2`.

Usually you will first have to use al_ustr_offset (24.6.3) to find the byte offsets.

Returns true on success, false on error.

24.11.3 al_ustr_assign_cstr

```
bool al_ustr_assign_cstr(ALLEGRO_USTR *us1, const char *s)
```

Overwrite the string `us` with the contents of the C-style string `s`. Returns true on success, false on error.

24.12 Replacing parts of string

24.12.1 al_ustr_set_chr

```
size_t al_ustr_set_chr(ALLEGRO_USTR *us, int start_pos, int32_t c)
```

Replace the code point beginning at byte offset `pos` with `c`. `pos` cannot be less than 0. If `pos` is past the end of `us1` then the space between the end of the string and `pos` will be padded with NUL ('\0') bytes. If `pos` is not the start of a valid code point, that is an error and the string will be unmodified.

On success, returns the number of bytes written, i.e. the offset to the following code point. On error, returns 0.

24.12.2 al_ustr_replace_range

```
bool al_ustr_replace_range(ALLEGRO_USTR *us1, int start_pos1, int end_pos1,
   const ALLEGRO_USTR *us2)
```

Replace the part of `us1` in the byte interval [start_pos, end_pos) with the contents of `us2`. `start_pos` cannot be less than 0. If `start_pos` is past the end of `us1` then the space between the end of the string and `start_pos` will be padded with NUL ('\0') bytes.

Use al_ustr_offset (24.6.3) to find the byte offsets.

Returns true on success, false on error.

24.13 Searching

24.13.1 al_ustr_find_chr

```
int al_ustr_find_chr(const ALLEGRO_USTR *us, int start_pos, int32_t c)
```

Search for the encoding of code point `c` in `us` from byte offset `start_pos` (inclusive).

Returns the position where it is found or –1 if it is not found.

24.13.2 al_ustr_rfind_chr

```
int al_ustr_rfind_chr(const ALLEGRO_USTR *us, int end_pos, int32_t c)
```

Search for the encoding of code point `c` in `us` backwards from byte offset `end_pos` (exclusive). Returns the position where it is found or –1 if it is not found.

24.13.3 al_ustr_find_set

```
int al_ustr_find_set(const ALLEGRO_USTR *us, int start_pos,
   const ALLEGRO_USTR *accept)
```

This function finds the first code point in `us`, beginning from byte offset `start_pos`, that matches any code point in `accept`. Returns the position if a code point was found. Otherwise returns –1.

24.13.4 al_ustr_find_set_cstr

```
int al_ustr_find_set_cstr(const ALLEGRO_USTR *us, int start_pos,
   const char *accept)
```

Like al_ustr_find_set (24.13.3) but takes a C-style string for `accept`.

24.13.5 al_ustr_find_cset

```
int al_ustr_find_cset(const ALLEGRO_USTR *us, int start_pos,
   const ALLEGRO_USTR *reject)
```

This function finds the first code point in `us`, beginning from byte offset `start_pos`, that does *not* match any code point in `reject`. In other words it finds a code point in the complementary set of `reject`. Returns the byte position of that code point, if any. Otherwise returns –1.

24.13.6 al_ustr_find_cset_cstr

```
int al_ustr_find_cset_cstr(const ALLEGRO_USTR *us, int start_pos,
   const char *reject)
```

Like al_ustr_find_cset (24.13.5) but takes a C-style string for `reject`.

24.13.7 al_ustr_find_str

```
int al_ustr_find_str(const ALLEGRO_USTR *haystack, int start_pos,
   const ALLEGRO_USTR *needle)
```

Find the first occurrence of string `needle` in `haystack`, beginning from byte offset `pos` (inclusive). Return the byte offset of the occurrence if it is found, otherwise return –1.

24.13.8 al_ustr_find_cstr

```
int al_ustr_find_cstr(const ALLEGRO_USTR *haystack, int start_pos,
   const char *needle)
```

Like al_ustr_find_str (24.13.7) but takes a C-style string for `needle`.

24.13.9 al_ustr_rfind_str

```
int al_ustr_rfind_str(const ALLEGRO_USTR *haystack, int end_pos,
   const ALLEGRO_USTR *needle)
```

Find the last occurrence of string `needle` in `haystack` before byte offset `end_pos` (exclusive). Return the byte offset of the occurrence if it is found, otherwise return –1.

24.13.10 al_ustr_rfind_cstr

```
int al_ustr_rfind_cstr(const ALLEGRO_USTR *haystack, int end_pos,
   const char *needle)
```

Like al_ustr_rfind_str (24.13.9) but takes a C-style string for `needle`.

24.13.11 al_ustr_find_replace

```
bool al_ustr_find_replace(ALLEGRO_USTR *us, int start_pos,
   const ALLEGRO_USTR *find, const ALLEGRO_USTR *replace)
```

Replace all occurrences of `find` in `us` with `replace`, beginning at byte offset `start_pos`. The `find` string must be non-empty. Returns true on success, false on error.

24.13.12 al_ustr_find_replace_cstr

```
bool al_ustr_find_replace_cstr(ALLEGRO_USTR *us, int start_pos,
   const char *find, const char *replace)
```

Like al_ustr_find_replace (24.13.11) but takes C-style strings for `find` and `replace`.

24.14 Comparing

24.14.1 al_ustr_equal

```
bool al_ustr_equal(const ALLEGRO_USTR *us1, const ALLEGRO_USTR *us2)
```

Return true iff the two strings are equal. This function is more efficient than al_ustr_compare (24.14.2) so is preferable if ordering is not important.

24.14.2 al_ustr_compare

```
int al_ustr_compare(const ALLEGRO_USTR *us1, const ALLEGRO_USTR *us2)
```

This function compares `us1` and `us2` by code point values. Returns zero if the strings are equal, a positive number if `us1` comes after `us2`, else a negative number.

This does *not* take into account locale-specific sorting rules. For that you will need to use another library.

24.14.3 al_ustr_ncompare

```
int al_ustr_ncompare(const ALLEGRO_USTR *us1, const ALLEGRO_USTR *us2, int n)
```

Like al_ustr_compare (24.14.2) but only compares up to the first `n` code points of both strings.

Returns zero if the strings are equal, a positive number if `us1` comes after `us2`, else a negative number.

24.14.4 al_ustr_has_prefix

```
bool al_ustr_has_prefix(const ALLEGRO_USTR *us1, const ALLEGRO_USTR *us2)
```

Returns true iff `us1` begins with `us2`.

24.14.5 al_ustr_has_prefix_cstr

```
bool al_ustr_has_prefix_cstr(const ALLEGRO_USTR *us1, const char *s2)
```

Returns true iff `us1` begins with `s2`.

24.14.6 al_ustr_has_suffix

```
bool al_ustr_has_suffix(const ALLEGRO_USTR *us1, const ALLEGRO_USTR *us2)
```

Returns true iff `us1` ends with `us2`.

24.14.7 al_ustr_has_suffix_cstr

```
bool al_ustr_has_suffix_cstr(const ALLEGRO_USTR *us1, const char *s2)
```

Returns true iff `us1` ends with `s2`.

24.15 UTF–16 conversion

24.15.1 al_ustr_new_from_utf16

```
ALLEGRO_USTR *al_ustr_new_from_utf16(uint16_t const *s)
```

Create a new string containing a copy of the 0-terminated string `s` which must be encoded as UTF–16. The string must eventually be freed with al_ustr_free (24.3.4).

24.15.2 al_ustr_size_utf16

```
size_t al_ustr_size_utf16(const ALLEGRO_USTR *us)
```

Returns the number of bytes required to encode the string in UTF–16 (including the terminating 0). Usually called before al_ustr_encode_utf16 (24.15.3) to determine the size of the buffer to allocate.

24.15.3 al_ustr_encode_utf16

```
size_t al_ustr_encode_utf16(const ALLEGRO_USTR *us, uint16_t *s,
   size_t n)
```

Encode the string into the given buffer, in UTF–16. Returns the number of bytes written. There are never more than `n` bytes written. The minimum size to encode the complete string can be queried with al_ustr_size_utf16 (24.15.2). If the `n` parameter is smaller than that, the string will be truncated but still always 0 terminated.

24.16 Low-level UTF–8 routines

24.16.1 al_utf8_width

```
size_t al_utf8_width(int c)
```

Returns the number of bytes that would be occupied by the specified code point when encoded in UTF–8. This is between 1 and 4 bytes for legal code point values. Otherwise returns 0.

24.16.2 al_utf8_encode

```
size_t al_utf8_encode(char s[], int32_t c)
```

Encode the specified code point to UTF–8 into the buffer `s`. The buffer must have enough space to hold the encoding, which takes between 1 and 4 bytes. This routine will refuse to encode code points above 0x10FFFF.

Returns the number of bytes written, which is the same as that returned by al_utf8_width (24.16.1).

24.17 Low-level UTF–16 routines

24.17.1 al_utf16_width

```
size_t al_utf16_width(int c)
```

Returns the number of bytes that would be occupied by the specified code point when encoded in UTF–16. This is either 2 or 4 bytes for legal code point values. Otherwise returns 0.

24.17.2 al_utf16_encode

```
size_t al_utf16_encode(uint16_t s[], int32_t c)
```

Encode the specified code point to UTF–8 into the buffer `s`. The buffer must have enough space to hold the encoding, which takes either 2 or 4 bytes. This routine will refuse to encode code points above 0x10FFFF.

Returns the number of bytes written, which is the same as that returned by al_utf16_width (24.17.1).

25 Audio addon

These functions are declared in the following header file. Link with allegro_audio.

```
#include <allegro5/allegro_audio.h>
```

25.1 Audio types

25.1.1 ALLEGRO_AUDIO_DEPTH

```
enum ALLEGRO_AUDIO_DEPTH
```

Sample depth and type, and signedness. Mixers only use 32-bit signed float (–1..+1), or 16-bit signed integers. The unsigned value is a bit-flag applied to the depth value.

- ALLEGRO_AUDIO_DEPTH_INT8
- ALLEGRO_AUDIO_DEPTH_INT16
- ALLEGRO_AUDIO_DEPTH_INT24
- ALLEGRO_AUDIO_DEPTH_FLOAT32
- ALLEGRO_AUDIO_DEPTH_UNSIGNED

For convenience:

- ALLEGRO_AUDIO_DEPTH_UINT8
- ALLEGRO_AUDIO_DEPTH_UINT16
- ALLEGRO_AUDIO_DEPTH_UINT24

25.1.2 ALLEGRO_AUDIO_DRIVER_ENUM

```
enum ALLEGRO_AUDIO_DRIVER_ENUM
```

The sound driver to use. It is *highly* recommended to use ALLEGRO_AUDIO_DRIVER_AUTODETECT whenever possible.

- ALLEGRO_AUDIO_DRIVER_AUTODETECT
- ALLEGRO_AUDIO_DRIVER_OPENAL
- ALLEGRO_AUDIO_DRIVER_ALSA
- ALLEGRO_AUDIO_DRIVER_DSOUND
- ALLEGRO_AUDIO_DRIVER_OSS
- ALLEGRO_AUDIO_DRIVER_PULSEAUDIO
- ALLEGRO_AUDIO_DRIVER_AQUEUE

25.1.3 ALLEGRO_AUDIO_PAN_NONE

```
#define ALLEGRO_AUDIO_PAN_NONE      (-1000.0f)
```

Special value for the ALLEGRO_AUDIOPROP_PAN property. Use this value to play samples at their original volume with panning disabled.

25.1.4 ALLEGRO_CHANNEL_CONF

```
enum ALLEGRO_CHANNEL_CONF
```

Speaker configuration (mono, stereo, 2.1, 3, etc).

- ALLEGRO_CHANNEL_CONF_1
- ALLEGRO_CHANNEL_CONF_2
- ALLEGRO_CHANNEL_CONF_3
- ALLEGRO_CHANNEL_CONF_4
- ALLEGRO_CHANNEL_CONF_5_1
- ALLEGRO_CHANNEL_CONF_6_1
- ALLEGRO_CHANNEL_CONF_7_1

25.1.5 ALLEGRO_MIXER

```
typedef struct ALLEGRO_MIXER ALLEGRO_MIXER;
```

A mixer is a type of stream which mixes together attached streams into a single buffer.

25.1.6 ALLEGRO_MIXER_QUALITY

```
enum ALLEGRO_MIXER_QUALITY
```

- ALLEGRO_MIXER_QUALITY_POINT
- ALLEGRO_MIXER_QUALITY_LINEAR

25.1.7 ALLEGRO_PLAYMODE

```
enum ALLEGRO_PLAYMODE
```

Sample and stream looping mode.

- ALLEGRO_PLAYMODE_ONCE
- ALLEGRO_PLAYMODE_LOOP
- ALLEGRO_PLAYMODE_BIDIR

25.1.8 ALLEGRO_SAMPLE_ID

```
typedef struct ALLEGRO_SAMPLE_ID ALLEGRO_SAMPLE_ID;
```

An ALLEGRO_SAMPLE_ID represents a sample being played via al_play_sample (25.4.3). It can be used to later stop the sample with al_stop_sample (25.4.4).

25.1.9 ALLEGRO_SAMPLE

```
typedef struct ALLEGRO_SAMPLE ALLEGRO_SAMPLE;
```

An ALLEGRO_SAMPLE object stores the data necessary for playing pre-defined digital audio. It holds information pertaining to data length, frequency, channel configuration, etc. You can have an ALLEGRO_SAMPLE object playing multiple times simultaneously. The object holds a user-specified PCM data buffer, of the format the object is created with.

25.1.10 ALLEGRO_SAMPLE_INSTANCE

```
typedef struct ALLEGRO_SAMPLE_INSTANCE ALLEGRO_SAMPLE_INSTANCE;
```

An ALLEGRO_SAMPLE_INSTANCE object represents a playable instance of a predefined sound effect. It holds information pertaining to the looping mode, loop start/end points, playing position, etc. An instance uses the data from an ALLEGRO_SAMPLE (25.1.9) object. Multiple instances may be created from the same ALLEGRO_SAMPLE. An ALLEGRO_SAMPLE must not be destroyed while there are instances which reference it.

To be played, an ALLEGRO_SAMPLE_INSTANCE object must be attached to an ALLEGRO_VOICE (25.1.12) object, or to an ALLEGRO_MIXER (25.1.5) object which is itself attached to an ALLEGRO_VOICE object (or to another ALLEGRO_MIXER object which is attached to an ALLEGRO_VOICE object, etc).

25.1.11 ALLEGRO_AUDIO_STREAM

```
typedef struct ALLEGRO_AUDIO_STREAM ALLEGRO_AUDIO_STREAM;
```

An ALLEGRO_AUDIO_STREAM object is used to stream generated audio to the sound device, in real-time. This is done by reading from a buffer, which is split into a number of fragments. Whenever a fragment has finished playing, the user can refill it with new data.

As with ALLEGRO_SAMPLE_INSTANCE (25.1.10) objects, streams store information necessary for playback, so you may not play the same stream multiple times simultaneously. Streams also need to be attached to an ALLEGRO_VOICE (25.1.12) object, or to an ALLEGRO_MIXER (25.1.5) object which, eventually, reaches an ALLEGRO_VOICE object.

While playing, you must periodically fill fragments with new audio data. To know when a new fragment is ready to be filled, you can either directly check with al_get_available_audio_stream_fragments (25.7.25), or listen to events from the stream.

You can register an audio stream event source to an event queue; see al_get_audio_stream_event_source (25.7.3). An ALLEGRO_EVENT_AUDIO_STREAM_FRAGMENT event is generated whenever a new fragment is ready. When you receive an event, use al_get_audio_stream_fragment (25.7.22) to obtain a pointer to the fragment to be filled. The size and format are determined by the parameters passed to al_create_audio_stream (25.7.1).

If you're late with supplying new data, the stream will be silent until new data is provided. You must call al_drain_audio_stream (25.7.4) when you're finished with supplying data to the stream.

If the stream is created by al_load_audio_stream (25.8.9) then it can also generate an ALLEGRO_EVENT_audio_stream_FINISH event if it reaches the end of the file and is not set to loop.

25.1.12 ALLEGRO_VOICE

```
typedef struct ALLEGRO_VOICE ALLEGRO_VOICE;
```

A voice structure that you'd attach a mixer or sample to. Ideally there would be one ALLEGRO_VOICE per system/hardware voice.

25.2 Setting up

25.2.1 al_install_audio

```
bool al_install_audio(ALLEGRO_AUDIO_DRIVER_ENUM mode)
```

Install the audio subsystem.

Parameters:

- mode - see ALLEGRO_AUDIO_DRIVER_ENUM (25.1.2). It is recommended to pass ALLEGRO_AUDIO_DRIVER_AU

Returns true on success, false on failure.

See also: al_reserve_samples (25.2.4), al_uninstall_audio (25.2.2), al_is_audio_installed (25.2.3)

25.2.2 al_uninstall_audio

```
void al_uninstall_audio(void)
```

Uninstalls the audio subsystem.

See also: al_install_audio (25.2.1)

25.2.3 al_is_audio_installed

```
bool al_is_audio_installed(void)
```

Returns true if al_install_audio (25.2.1) was called previously and returned successfully.

25.2.4 al_reserve_samples

```
bool al_reserve_samples(int reserve_samples)
```

Reserves 'reserve_samples' number of samples attached to the default mixer. al_install_audio (25.2.1) must have been called first. If no default mixer is set, then this function will create a voice with an attached mixer.

Returns true on success, false on error.

See also: al_set_default_mixer (25.6.4)

25.2.5 al_get_allegro_audio_version

```
uint32_t al_get_allegro_audio_version(void)
```

Returns the (compiled) version of the addon, in the same format as al_get_allegro_version (19.4).

25.2.6 al_get_depth_size

Return the size of a sample, in bytes, for the given format. The format is one of the values listed under ALLEGRO_AUDIO_DEPTH (25.1.1).

25.2.7 al_get_channel_count

Return the number of channels for the given channel configuration, which is one of the values listed under ALLEGRO_CHANNEL_CONF (25.1.4).

25.3 Voice functions

25.3.1 al_create_voice

```
ALLEGRO_VOICE *al_create_voice(unsigned int freq,
   ALLEGRO_AUDIO_DEPTH depth, ALLEGRO_CHANNEL_CONF chan_conf)
```

Creates a voice struct and allocates a voice from the digital sound driver. The passed frequency, sample format and channel configuration are used as a hint to what kind of data will be sent to the voice. However, the underlying sound driver is free to use non-matching values. For example it may be the native format of the sound hardware. If you attach a mixer to the voice, the mixer will convert from the mixer's format to the voice format and you do not have to care about this. If you access the voice directly, make sure to not rely on the parameters passed to this function, but query the returned voice for the actual settings.

See also: al_destroy_voice (25.3.2)

25.3.2 al_destroy_voice

```
void al_destroy_voice(ALLEGRO_VOICE *voice)
```

Destroys the voice and deallocates it from the digital driver. Does nothing if the voice is NULL.

See also: al_create_voice (25.3.1)

25.3.3 al_detach_voice

```
void al_detach_voice(ALLEGRO_VOICE *voice)
```

Detaches the sample or mixer stream from the voice.

25.3.4 al_attach_audio_stream_to_voice

```
bool al_attach_audio_stream_to_voice(ALLEGRO_AUDIO_STREAM *stream,
   ALLEGRO_VOICE *voice)
```

Attaches an audio stream to a voice. The same rules as al_attach_sample_instance_to_voice (25.3.6) apply. This may fail if the driver can't create a voice with the buffer count and buffer size the stream uses.

An audio stream attached directly to a voice has a number of limitations. The audio stream plays immediately and cannot be stopped. The stream position, speed, gain, panning, cannot be changed. At this time, we don't recommend attaching audio streams directly to voices. Use a mixer in between.

Returns true on success, false on failure.

25.3.5 al_attach_mixer_to_voice

```
bool al_attach_mixer_to_voice(ALLEGRO_MIXER *mixer, ALLEGRO_VOICE *voice)
```

Attaches a mixer to a voice. The same rules as al_attach_sample_instance_to_voice (25.3.6) apply, with the exception of the depth requirement.

Returns true on success, false on failure.

25.3.6 al_attach_sample_instance_to_voice

```
bool al_attach_sample_instance_to_voice(ALLEGRO_SAMPLE_INSTANCE *spl,
   ALLEGRO_VOICE *voice)
```

Attaches a sample to a voice, and allows it to play. The sample's volume and loop mode will be ignored, and it must have the same frequency and depth (including signed-ness) as the voice. This function may fail if the selected driver doesn't support preloading sample data.

At this time, we don't recommend attaching samples directly to voices. Use a mixer in between.

Returns true on success, false on failure.

25.3.7 al_get_voice_frequency

```
unsigned int al_get_voice_frequency(const ALLEGRO_VOICE *voice)
```

Return the frequency of the voice, e.g. 44100.

25.3.8 al_get_voice_channels

```
ALLEGRO_CHANNEL_CONF al_get_voice_channels(const ALLEGRO_VOICE *voice)
```

Return the channel configuration of the voice.

See also: ALLEGRO_CHANNEL_CONF (25.1.4).

25.3.9 al_get_voice_depth

```
ALLEGRO_AUDIO_DEPTH al_get_voice_depth(const ALLEGRO_VOICE *voice)
```

Return the audio depth of the voice.

See also: ALLEGRO_AUDIO_DEPTH (25.1.1).

25.3.10 al_get_voice_playing

```
bool al_get_voice_playing(const ALLEGRO_VOICE *voice)
```

Return true if the voice is currently playing.

25.3.11 al_set_voice_playing

```
bool al_set_voice_playing(ALLEGRO_VOICE *voice, bool val)
```

Change whether a voice is playing or not. The voice must have a sample or mixer attached to it.

Returns true on success, false on failure.

25.3.12 al_get_voice_position

```
unsigned int al_get_voice_position(const ALLEGRO_VOICE *voice)
```

When the voice has a non-streaming object attached to it, e.g. a sample, returns the voice's current sample position. Otherwise, returns zero.

See also: al_set_voice_position (25.3.13).

25.3.13 al_set_voice_position

```
bool al_set_voice_position(ALLEGRO_VOICE *voice, unsigned int val)
```

Set the voice position. This can only work if the voice has a non-streaming object attached to it, e.g. a sample.

Returns true on success, false on failure.

See also: al_get_voice_position (25.3.12).

25.4 Sample functions

25.4.1 al_create_sample

```
ALLEGRO_SAMPLE *al_create_sample(void *buf, unsigned int samples,
   unsigned int freq, ALLEGRO_AUDIO_DEPTH depth,
   ALLEGRO_CHANNEL_CONF chan_conf, bool free_buf)
```

Create a sample data structure from the supplied buffer. If `free_buf` is true then the buffer will be freed as well when the sample data structure is destroyed.

To allocate a buffer of the correct size, you can use something like this:

```
sample_size = al_get_channel_count(chan_conf) * al_get_depth_size(depth);
bytes = samples * sample_size;
buffer = al_malloc(bytes)
```

See also: al_destroy_sample (25.4.2).

25.4.2 al_destroy_sample

```
void al_destroy_sample(ALLEGRO_SAMPLE *spl)
```

Free the sample data structure. If it was created with the `free_buf` parameter set to true, then the buffer will be freed as well.

You must stop or destroy any ALLEGRO_SAMPLE_INSTANCE (25.1.10) structures which reference this ALLEGRO_SAMPLE (25.1.9) beforehand.

If you have used al_play_sample (25.4.3) at all, it is a very good idea to call al_stop_samples (25.4.5) before destroying samples at the end of the program, in case any are still playing.

See also: al_destroy_sample_instance (25.5.2), al_stop_sample (25.4.4), al_stop_samples (25.4.5)

25.4.3 al_play_sample

```
bool al_play_sample(ALLEGRO_SAMPLE *spl, float gain, float pan, float speed,
   int loop, ALLEGRO_SAMPLE_ID *ret_id)
```

Plays a sample over the default mixer. al_reserve_samples (25.2.4) must have previously been called. Returns true on success, false on failure. Playback may fail because all the reserved samples are currently used.

Parameters:

- gain - relative volume at which the sample is played; 1.0 is normal.
- pan - 0.0 is centred, −1.0 is left, 1.0 is right, or ALLEGRO_AUDIO_PAN_NONE.
- speed - relative speed at which the sample is played; 1.0 is normal.
- loop - the play mode.
- ret_id - if non-NULL the variable which this points to will be assigned an id representing the sample being played.

See also: ALLEGRO_PLAYMODE (25.1.7), ALLEGRO_AUDIO_PAN_NONE (25.1.3), ALLEGRO_SAMPLE_ID (25.1.8), al_stop_sample (25.4.4), al_stop_samples (25.4.5).

25.4.4 al_stop_sample

```
void al_stop_sample(ALLEGRO_SAMPLE_ID *spl_id)
```

Stop the sample started by al_play_sample (25.4.3).

See also: al_stop_samples (25.4.5)

25.4.5 al_stop_samples

```
void al_stop_samples(void)
```

Stop all samples started by al_play_sample (25.4.3).

See also: al_stop_sample (25.4.4)

25.4.6 al_get_sample_channels

```
ALLEGRO_CHANNEL_CONF al_get_sample_channels(const ALLEGRO_SAMPLE *spl)
```

Return the channel configuration.

See also: ALLEGRO_CHANNEL_CONF (25.1.4).

25.4.7 al_get_sample_depth

```
ALLEGRO_AUDIO_DEPTH al_get_sample_depth(const ALLEGRO_SAMPLE *spl)
```

Return the audio depth.

See also: ALLEGRO_AUDIO_DEPTH (25.1.1).

25.4.8 al_get_sample_frequency

```
unsigned int al_get_sample_frequency(const ALLEGRO_SAMPLE *spl)
```

Return the frequency of the sample.

25.4.9 al_get_sample_length

```
unsigned int al_get_sample_length(const ALLEGRO_SAMPLE *spl)
```

Return the length of the sample in sample values.

25.4.10 al_get_sample_data

```
void *al_get_sample_data(const ALLEGRO_SAMPLE *spl)
```

Return a pointer to the raw sample data.

25.5 Sample instance functions

25.5.1 al_create_sample_instance

```
ALLEGRO_SAMPLE_INSTANCE *al_create_sample_instance(ALLEGRO_SAMPLE *sample_data)
```

Creates a sample stream, using the supplied data. This must be attached to a voice or mixer before it can be played. The argument may be NULL. You can then set the data later with al_set_sample (25.5.26).

25.5.2 al_destroy_sample_instance

```
void al_destroy_sample_instance(ALLEGRO_SAMPLE_INSTANCE *spl)
```

Detaches the sample stream from anything it may be attached to and frees it (the sample data is *not* freed!).

25.5.3 al_play_sample_instance

```
bool al_play_sample_instance(ALLEGRO_SAMPLE_INSTANCE *spl)
```

Play an instance of a sample data. Returns true on success, false on failure.

25.5.4 al_stop_sample_instance

```
bool al_stop_sample_instance(ALLEGRO_SAMPLE_INSTANCE *spl)
```

Stop an sample instance playing.

25.5.5 al_get_sample_instance_channels

```
ALLEGRO_CHANNEL_CONF al_get_sample_instance_channels(
   const ALLEGRO_SAMPLE_INSTANCE *spl)
```

Return the channel configuration.

See also: ALLEGRO_CHANNEL_CONF (25.1.4).

25.5.6 al_get_sample_instance_depth

```
ALLEGRO_AUDIO_DEPTH al_get_sample_instance_depth(const ALLEGRO_SAMPLE_INSTANCE *spl)
```

Return the audio depth.

See also: ALLEGRO_AUDIO_DEPTH (25.1.1).

25.5.7 al_get_sample_instance_frequency

```
unsigned int al_get_sample_instance_frequency(const ALLEGRO_SAMPLE_INSTANCE *spl)
```

Return the frequency of the sample instance.

25.5.8 al_get_sample_instance_length

```
unsigned int al_get_sample_instance_length(const ALLEGRO_SAMPLE_INSTANCE *spl)
```

Return the length of the sample instance in sample values.

25.5.9 al_set_sample_instance_length

```
bool al_set_sample_instance_length(ALLEGRO_SAMPLE_INSTANCE *spl,
   unsigned int val)
```

Set the length of the sample instance in sample values.

Return true on success, false on failure. Will fail if the sample instance is currently playing.

25.5.10 al_get_sample_instance_position

```
unsigned int al_get_sample_instance_position(const ALLEGRO_SAMPLE_INSTANCE *spl)
```

Get the playback position of a sample instance.

25.5.11 al_set_sample_instance_position

```
bool al_set_sample_instance_position(ALLEGRO_SAMPLE_INSTANCE *spl,
   unsigned int val)
```

Set the playback position of a sample instance.

Returns true on success, false on failure.

25.5.12 al_get_sample_instance_speed

```
float al_get_sample_instance_speed(const ALLEGRO_SAMPLE_INSTANCE *spl)
```

Return the playback speed.

25.5.13 al_set_sample_instance_speed

```
bool al_set_sample_instance_speed(ALLEGRO_SAMPLE_INSTANCE *spl, float val)
```

Set the playback speed.

Return true on success, false on failure. Will fail if the sample instance is attached directly to a voice.

25.5.14 al_get_sample_instance_gain

```
float al_get_sample_instance_gain(const ALLEGRO_SAMPLE_INSTANCE *spl)
```

Return the playback gain.

25.5.15 al_set_sample_instance_gain

```
bool al_set_sample_instance_gain(ALLEGRO_SAMPLE_INSTANCE *spl, float val)
```

Set the playback gain.

Returns true on success, false on failure. Will fail if the sample instance is attached directly to a voice.

25.5.16 al_get_sample_instance_pan

```
float al_get_sample_instance_pan(const ALLEGRO_SAMPLE_INSTANCE *spl)
```

Get the pan value.

See also: al_set_sample_instance_pan (25.5.17).

25.5.17 al_set_sample_instance_pan

```
bool al_set_sample_instance_pan(ALLEGRO_SAMPLE_INSTANCE *spl, float val)
```

Set the pan value on a sample instance. A value of −1.0 means to play the sample only through the left speaker; +1.0 means only through the right speaker; 0.0 means the sample is centre balanced.

A constant sound power level is maintained as the sample is panned from left to right. As a consequence, a pan value of 0.0 will play the sample 3 dB softer than the original level. To disable panning and play a sample at its original level, set the pan value to ALLEGRO_AUDIO_PAN_NONE (25.1.3).

Returns true on success, false on failure. Will fail if the sample instance is attached directly to a voice.

(A sound guy should explain that better; I only implemented it. Also this might be more properly called a balance control than pan. Also we don't attempt anything with more than two channels yet.)

See also: al_get_sample_instance_pan (25.5.16).

25.5.18 al_get_sample_instance_time

```
float al_get_sample_instance_time(const ALLEGRO_SAMPLE_INSTANCE *spl)
```

Return the length of the sample instance in seconds, assuming a playback speed of 1.0.

25.5.19 al_get_sample_instance_playmode

```
ALLEGRO_PLAYMODE al_get_sample_instance_playmode(const ALLEGRO_SAMPLE_INSTANCE *spl)
```

Return the playback mode.

25.5.20 al_set_sample_instance_playmode

```
bool al_set_sample_instance_playmode(ALLEGRO_SAMPLE_INSTANCE *spl,
   ALLEGRO_PLAYMODE val)
```

Set the playback mode.

Returns true on success, false on failure.

25.5.21 al_get_sample_instance_playing

```
bool al_get_sample_instance_playing(const ALLEGRO_SAMPLE_INSTANCE *spl)
```

Return true if the sample instance is playing.

25.5.22 al_set_sample_instance_playing

```
bool al_set_sample_instance_playing(ALLEGRO_SAMPLE_INSTANCE *spl, bool val)
```

Change whether the sample instance is playing.

Returns true on success, false on failure.

25.5.23 al_get_sample_instance_attached

```
bool al_get_sample_instance_attached(const ALLEGRO_SAMPLE_INSTANCE *spl)
```

Return whether the sample instance is attached to something.

25.5.24 al_detach_sample_instance

```
bool al_detach_sample_instance(ALLEGRO_SAMPLE_INSTANCE *spl)
```

Detach the sample instance from whatever it's attached to, if anything.

Returns true on success.

25.5.25 al_get_sample

```
ALLEGRO_SAMPLE *al_get_sample(ALLEGRO_SAMPLE_INSTANCE *spl)
```

Return the sample data that the sample instance plays.

25.5.26 al_set_sample

```
bool al_set_sample(ALLEGRO_SAMPLE_INSTANCE *spl, ALLEGRO_SAMPLE *data)
```

Change the sample data that a sample instance plays. This can be quite an involved process.

First, the sample is stopped if it is not already.

Next, if data is NULL, the sample is detached from its parent (if any).

If data is not NULL, the sample may be detached and reattached to its parent (if any). This is not necessary if the old sample data and new sample data have the same frequency, depth and channel configuration. Reattaching may not always succeed.

On success, the sample remains stopped. The playback position and loop end points are reset to their default values. The loop mode remains unchanged.

Returns true on success, false on failure. On failure, the sample will be stopped and detached from its parent.

25.6 Mixer functions

25.6.1 al_create_mixer

```
ALLEGRO_MIXER *al_create_mixer(unsigned int freq,
   ALLEGRO_AUDIO_DEPTH depth, ALLEGRO_CHANNEL_CONF chan_conf)
```

Creates a mixer stream, to attach sample streams or other mixers to. It will mix into a buffer at the requested frequency and channel count.

The only supported audio depths are ALLEGRO_AUDIO_DEPTH_FLOAT32 and ALLEGRO_AUDIO_DEPTH_INT16 (not yet complete).

Returns true on success, false on error.

See also: al_destroy_mixer (25.6.2)

25.6.2 al_destroy_mixer

```
void al_destroy_mixer(ALLEGRO_MIXER *mixer)
```

Destroys the mixer stream.

See also: al_create_mixer (25.6.1)

25.6.3 al_get_default_mixer

```
ALLEGRO_MIXER *al_get_default_mixer(void)
```

Return the default mixer, or NULL if one has not been set. Although different configurations of mixers and voices can be used, in most cases a single mixer attached to a voice is what you want. The default mixer is used by al_play_sample (25.4.3).

See also: al_reserve_samples (25.2.4), al_play_sample (25.4.3), al_set_default_mixer (25.6.4), al_restore_default_mixer (25.6.5)

25.6.4 al_set_default_mixer

```
bool al_set_default_mixer(ALLEGRO_MIXER *mixer)
```

Sets the default mixer. All samples started with al_play_sample (25.4.3) will be stopped. If you are using your own mixer, this should be called before al_reserve_samples (25.2.4).

Returns true on success, false on error.

See also: al_reserve_samples (25.2.4), al_play_sample (25.4.3), al_get_default_mixer (25.6.3), al_restore_default_mixer (25.6.5)

25.6.5 al_restore_default_mixer

```
bool al_restore_default_mixer(void)
```

Restores Allegro's default mixer. All samples started with al_play_sample (25.4.3) will be stopped. Returns true on success, false on error.

See also: al_get_default_mixer (25.6.3), al_set_default_mixer (25.6.4), al_reserve_samples (25.2.4).

25.6.6 al_attach_mixer_to_mixer

```
bool al_attach_mixer_to_mixer(ALLEGRO_MIXER *stream, ALLEGRO_MIXER *mixer)
```

Attaches a mixer onto another mixer. The same rules as with al_attach_sample_instance_to_mixer (25.6.7) apply, with the added caveat that both mixers must be the same frequency. Returns true on success, false on error.

Currently both mixers must have the same audio depth, otherwise the function fails.

See also: al_detach_mixer (25.6.18).

25.6.7 al_attach_sample_instance_to_mixer

```
bool al_attach_sample_instance_to_mixer(ALLEGRO_SAMPLE_INSTANCE *spl,
   ALLEGRO_MIXER *mixer)
```

Attach a sample instance to a mixer. The instance must not already be attached to anything.

Returns true on success, false on failure.

See also: al_detach_sample_instance (25.5.24).

25.6.8 al_attach_audio_stream_to_mixer

```
bool al_attach_audio_stream_to_mixer(ALLEGRO_AUDIO_STREAM *stream, ALLEGRO_MIXER *mixer)
```

Attach a stream to a mixer.

Returns true on success, false on failure.

See also: al_detach_audio_stream (25.7.21).

25.6.9 al_get_mixer_frequency

```
unsigned int al_get_mixer_frequency(const ALLEGRO_MIXER *mixer)
```

Return the mixer frequency.

25.6.10 al_set_mixer_frequency

```
bool al_set_mixer_frequency(ALLEGRO_MIXER *mixer, unsigned int val)
```

Set the mixer frequency. This will only work if the mixer is not attached to anything.

Returns true on success, false on failure.

25.6.11 al_get_mixer_channels

```
ALLEGRO_CHANNEL_CONF al_get_mixer_channels(const ALLEGRO_MIXER *mixer)
```

Return the mixer channel configuration.

See also: ALLEGRO_CHANNEL_CONF (25.1.4).

25.6.12 al_get_mixer_depth

```
ALLEGRO_AUDIO_DEPTH al_get_mixer_depth(const ALLEGRO_MIXER *mixer)
```

Return the mixer audio depth.

See also: ALLEGRO_AUDIO_DEPTH (25.1.1).

25.6.13 al_get_mixer_quality

```
ALLEGRO_MIXER_QUALITY al_get_mixer_quality(const ALLEGRO_MIXER *mixer)
```

Return the mixer quality.

See also: ALLEGRO_MIXER_QUALITY (25.1.6).

25.6.14 al_set_mixer_quality

```
bool al_set_mixer_quality(ALLEGRO_MIXER *mixer, ALLEGRO_MIXER_QUALITY new_quality)
```

Set the mixer quality. This can only succeed if the mixer does not have anything attached to it.

Returns true on success, false on failure.

See also: ALLEGRO_MIXER_QUALITY (25.1.6).

25.6.15 al_get_mixer_playing

```
bool al_get_mixer_playing(const ALLEGRO_MIXER *mixer)
```

Return true if the mixer is playing.

See also: al_set_mixer_playing (25.6.16).

25.6.16 al_set_mixer_playing

```
bool al_set_mixer_playing(ALLEGRO_MIXER *mixer, bool val)
```

Change whether the mixer is playing.

Returns true on success, false on failure.

See also: al_get_mixer_playing (25.6.15).

25.6.17 al_get_mixer_attached

```
bool al_get_mixer_attached(const ALLEGRO_MIXER *mixer)
```

Return true if the mixer is attached to something.

See also: al_attach_sample_instance_to_mixer (25.6.7), al_attach_audio_stream_to_mixer (25.6.8), al_attach_mixer_to_mixer (25.

25.6.18 al_detach_mixer

```
bool al_detach_mixer(ALLEGRO_MIXER *mixer)
```

Detach the mixer from whatever it is attached to, if anything.

See also: al_attach_mixer_to_mixer (25.6.6).

25.6.19 al_set_mixer_postprocess_callback

```
bool al_set_mixer_postprocess_callback(ALLEGRO_MIXER *mixer,
   void (*pp_callback)(void *buf, unsigned int samples, void *data),
   void *pp_callback_userdata)
```

Sets a post-processing filter function that's called after the attached streams have been mixed. The buffer's format will be whatever the mixer was created with. The sample count and user-data pointer is also passed.

25.7 Stream functions

25.7.1 al_create_audio_stream

```
ALLEGRO_AUDIO_STREAM *al_create_audio_stream(size_t fragment_count,
   unsigned int samples, unsigned int freq, ALLEGRO_AUDIO_DEPTH depth,
   ALLEGRO_CHANNEL_CONF chan_conf)
```

Creates an ALLEGRO_AUDIO_STREAM (25.1.11). The stream will be set to play by default. It will feed audio data from a buffer, which is split into a number of fragments.

Parameters:

- fragment_count - How many fragments to use for the audio stream. Usually only two fragments are required - splitting the audio buffer in two halves. But it means that the only time when new data can be supplied is whenever one half has finished playing. When using many fragments, you usually will use fewer samples for one, so there always will be (small) fragments available to be filled with new data.
- samples - The size of a fragment in samples. See note below.
- freq - The frequency, in Hertz.
- depth - Must be one of the values listed for ALLEGRO_AUDIO_DEPTH (25.1.1).
- chan_conf - Must be one of the values listed for ALLEGRO_CHANNEL_CONF (25.1.4).

The choice of *fragment_count*, *samples* and *freq* directly influences the audio delay. The delay in seconds can be expressed as:

```
delay = fragment_count * samples / freq
```

This is only the delay due to Allegro's streaming, there may be additional delay caused by sound drivers and/or hardware.

Note: If you know the fragment size in bytes, you can get the size in samples like this:

```
sample_size = al_get_channel_count(chan_conf) * al_get_depth_size(depth);
samples = bytes_per_fragment / sample_size;
```

The size of the complete buffer is:

```
buffer_size = bytes_per_fragment * fragment_count
```

Note: unlike many Allegro objects, audio streams are not implicitly destroyed when Allegro is shut down. You must destroy them manually with al_destroy_audio_stream (25.7.2) before the audio system is shut down.

25.7.2 al_destroy_audio_stream

```
void al_destroy_audio_stream(ALLEGRO_AUDIO_STREAM *stream)
```

Destroy an audio stream which was created with al_create_audio_stream (25.7.1) or al_load_audio_stream (25.8.9).

Note: If the stream is still attached to a mixer or voice, al_detach_audio_stream (25.7.21) is automatically called on it first.

See also: al_drain_audio_stream (25.7.4).

25.7.3 al_get_audio_stream_event_source

```
ALLEGRO_EVENT_SOURCE *al_get_audio_stream_event_source(
   ALLEGRO_AUDIO_STREAM *stream)
```

Retrieve the associated event source.

See al_get_audio_stream_fragment (25.7.22) for a description of the ALLEGRO_EVENT_AUDIO_STREAM_FRAGMENT event that audio streams emit.

25.7.4 al_drain_audio_stream

```
void al_drain_audio_stream(ALLEGRO_AUDIO_STREAM *stream)
```

You should call this to finalise an audio stream that you will no longer be feeding, to wait for all pending buffers to finish playing. The stream's playing state will change to false.

25.7.5 al_rewind_audio_stream

```
bool al_rewind_audio_stream(ALLEGRO_AUDIO_STREAM *stream)
```

Set the streaming file playing position to the beginning. Returns true on success. Currently this can only be called on streams created with al_load_audio_stream (25.8.9), al_load_audio_stream_f (25.8.10) and the format-specific functions underlying those functions.

25.7.6 al_get_audio_stream_frequency

```
unsigned int al_get_audio_stream_frequency(const ALLEGRO_AUDIO_STREAM *stream)
```

Return the stream frequency.

25.7.7 al_get_audio_stream_channels

```
ALLEGRO_CHANNEL_CONF al_get_audio_stream_channels(
   const ALLEGRO_AUDIO_STREAM *stream)
```

Return the stream channel configuration.

See also: ALLEGRO_CHANNEL_CONF (25.1.4).

25.7.8 al_get_audio_stream_depth

```
ALLEGRO_AUDIO_DEPTH al_get_audio_stream_depth(
   const ALLEGRO_AUDIO_STREAM *stream)
```

Return the stream audio depth.

See also: ALLEGRO_AUDIO_DEPTH (25.1.1).

25.7.9 al_get_audio_stream_length

```
unsigned int al_get_audio_stream_length(const ALLEGRO_AUDIO_STREAM *stream)
```

Return the stream length in samples.

25.7.10 al_get_audio_stream_speed

```
float al_get_audio_stream_speed(const ALLEGRO_AUDIO_STREAM *stream)
```

Return the playback speed.

See also: al_set_audio_stream_speed (25.7.11).

25.7.11 al_set_audio_stream_speed

```
bool al_set_audio_stream_speed(ALLEGRO_AUDIO_STREAM *stream, float val)
```

Set the playback speed.

Return true on success, false on failure. Will fail if the sample instance is attached directly to a voice.

See also: al_get_audio_stream_speed (25.7.10).

25.7.12 al_get_audio_stream_gain

```
float al_get_audio_stream_gain(const ALLEGRO_AUDIO_STREAM *stream)
```

Return the playback gain.

See also: al_set_audio_stream_gain (25.7.13).

25.7.13 al_set_audio_stream_gain

```
bool al_set_audio_stream_gain(ALLEGRO_AUDIO_STREAM *stream, float val)
```

Set the playback gain.

Returns true on success, false on failure. Will fail if the sample instance is attached directly to a voice.

See also: al_get_audio_stream_gain (25.7.12).

25.7.14 al_get_audio_stream_pan

```
float al_get_audio_stream_pan(const ALLEGRO_AUDIO_STREAM *stream)
```

Get the pan value.

See also: al_set_audio_stream_pan (25.7.15).

25.7.15 al_set_audio_stream_pan

```
bool al_set_audio_stream_pan(ALLEGRO_AUDIO_STREAM *stream, float val)
```

Set the pan value on a sample instance. A value of −1.0 means to play the sample only through the left speaker; +1.0 means only through the right speaker; 0.0 means the sample is centre balanced.

Returns true on success, false on failure. Will fail if the sample instance is attached directly to a voice.

See also: al_get_audio_stream_playing (25.7.16).

25.7.16 al_get_audio_stream_playing

```
bool al_get_audio_stream_playing(const ALLEGRO_AUDIO_STREAM *stream)
```

Return true if the stream is playing.

See also: al_set_audio_stream_playing (25.7.17).

25.7.17 al_set_audio_stream_playing

```
bool al_set_audio_stream_playing(ALLEGRO_AUDIO_STREAM *stream, bool val)
```

Change whether the stream is playing.

Returns true on success, false on failure.

25.7.18 al_get_audio_stream_playmode

```
ALLEGRO_PLAYMODE al_get_audio_stream_playmode(
   const ALLEGRO_AUDIO_STREAM *stream)
```

Return the playback mode.

See also: ALLEGRO_PLAYMODE (25.1.7), al_set_audio_stream_playmode (25.7.19).

25.7.19 al_set_audio_stream_playmode

```
bool al_set_audio_stream_playmode(ALLEGRO_AUDIO_STREAM *stream,
   ALLEGRO_PLAYMODE val)
```

Set the playback mode.

Returns true on success, false on failure.

See also: ALLEGRO_PLAYMODE (25.1.7), al_get_audio_stream_playmode (25.7.18).

25.7.20 al_get_audio_stream_attached

```
bool al_get_audio_stream_attached(const ALLEGRO_AUDIO_STREAM *stream)
```

Return whether the stream is attached to something.

See also: al_attach_audio_stream_to_mixer (25.6.8), al_attach_audio_stream_to_voice (25.3.4), al_detach_audio_stream (25.7.21).

25.7.21 al_detach_audio_stream

```
bool al_detach_audio_stream(ALLEGRO_AUDIO_STREAM *stream)
```

Detach the stream from whatever it's attached to, if anything.

See also: al_attach_audio_stream_to_mixer (25.6.8), al_attach_audio_stream_to_voice (25.3.4), al_get_audio_stream_attached (25

25.7.22 al_get_audio_stream_fragment

```
void *al_get_audio_stream_fragment(const ALLEGRO_AUDIO_STREAM *stream)
```

When using Allegro's audio streaming, you will use this function to continuously provide new sample data to a stream.

If the stream is ready for new data, the function will return the address of an internal buffer to be filled with audio data. The length and format of the buffer are specified with al_create_audio_stream (25.7.1) or can be queried with the various functions described here. Once the buffer is filled, you must signal this to Allegro by passing the buffer to al_set_audio_stream_fragment (25.7.23).

If the stream is not ready for new data, the function will return NULL.

Note: If you listen to events from the stream, an ALLEGRO_EVENT_AUDIO_STREAM_FRAGMENT event will be generated whenever a new fragment is ready. However, getting an event is *not* a guarantee that al_get_audio_stream_fragment (25.7.22) will not return NULL, so you still must check for it.

See also: al_set_audio_stream_fragment (25.7.23), al_get_audio_stream_event_source (25.7.3), al_get_audio_stream_frequency (25
al_get_audio_stream_channels (25.7.7), al_get_audio_stream_depth (25.7.8), al_get_audio_stream_length (25.7.9)

25.7.23 al_set_audio_stream_fragment

```
bool al_set_audio_stream_fragment(ALLEGRO_AUDIO_STREAM *stream, void *val)
```

This function needs to be called for every successful call of al_get_audio_stream_fragment (25.7.22) to indicate that the buffer is filled with new data.

25.7.24 al_get_audio_stream_fragments

```
unsigned int al_get_audio_stream_fragments(const ALLEGRO_AUDIO_STREAM *stream)
```

Returns the number of fragments this stream uses. This is the same value as passed to al_create_audio_stream (25.7.1) when a new stream is created.

25.7.25 al_get_available_audio_stream_fragments

```
unsigned int al_get_available_audio_stream_fragments(
   const ALLEGRO_AUDIO_STREAM *stream)
```

Returns the number of available fragments in the stream, that is, fragments which are not currently filled with data for playback.

See also: al_get_audio_stream_fragment (25.7.22)

25.7.26 al_seek_audio_stream_secs

```
bool al_seek_audio_stream_secs(ALLEGRO_AUDIO_STREAM *stream, double time)
```

Set the streaming file playing position to time. Returns true on success. Currently this can only be called on streams created with al_load_audio_stream (25.8.9), al_load_audio_stream_f (25.8.10) and the format-specific functions underlying those functions.

25.7.27 al_get_audio_stream_position_secs

```
double al_get_audio_stream_position_secs(ALLEGRO_AUDIO_STREAM *stream)
```

Return the position of the stream in seconds. Currently this can only be called on streams created with al_load_audio_stream (25.8.9).

25.7.28 al_get_audio_stream_length_secs

```
double al_get_audio_stream_length_secs(ALLEGRO_AUDIO_STREAM *stream)
```

Return the length of the stream in seconds, if known. Otherwise returns zero.

Currently this can only be called on streams created with al_load_audio_stream (25.8.9), al_load_audio_stream_f (25.8.10) and the format-specific functions underlying those functions.

25.7.29 al_set_audio_stream_loop_secs

```
bool al_set_audio_stream_loop_secs(ALLEGRO_AUDIO_STREAM *stream,
   double start, double end)
```

Sets the loop points for the stream in seconds. Currently this can only be called on streams created with al_load_audio_stream (25.8.9), al_load_audio_stream_f (25.8.10) and the format-specific functions underlying those functions.

25.8 Audio file I/O

25.8.1 al_register_sample_loader

```
bool al_register_sample_loader(const char *ext,
   ALLEGRO_SAMPLE *(*loader)(const char *filename))
```

Register a handler for al_load_sample (25.8.7). The given function will be used to handle the loading of sample files with the given extension.

The extension should include the leading dot ('.') character. It will be matched case-insensitively.

The `loader` argument may be NULL to unregister an entry.

Returns true on success, false on error. Returns false if unregistering an entry that doesn't exist.

25.8.2 al_register_sample_loader_f

```
bool al_register_sample_loader_f(const char *ext,
   ALLEGRO_SAMPLE *(*loader)(ALLEGRO_FILE* fp))
```

Register a handler for al_load_sample_f (25.8.8). The given function will be used to handle the loading of sample files with the given extension.

The extension should include the leading dot ('.') character. It will be matched case-insensitively.

The `loader` argument may be NULL to unregister an entry.

Returns true on success, false on error. Returns false if unregistering an entry that doesn't exist.

25.8.3 al_register_sample_saver

```
bool al_register_sample_saver(const char *ext,
   bool (*saver)(const char *filename, ALLEGRO_SAMPLE *spl))
```

Register a handler for al_save_sample (25.8.11). The given function will be used to handle the saving of sample files with the given extension.

The extension should include the leading dot ('.') character. It will be matched case-insensitively.

The `saver` argument may be NULL to unregister an entry.

Returns true on success, false on error. Returns false if unregistering an entry that doesn't exist.

25.8.4 al_register_sample_saver_f

```
bool al_register_sample_saver_f(const char *ext,
   bool (*saver)(ALLEGRO_FILE* fp, ALLEGRO_SAMPLE *spl))
```

Register a handler for al_save_sample_f (25.8.12). The given function will be used to handle the saving of sample files with the given extension.

The extension should include the leading dot ('.') character. It will be matched case-insensitively.

The `saver` argument may be NULL to unregister an entry.

Returns true on success, false on error. Returns false if unregistering an entry that doesn't exist.

25.8.5 al_register_audio_stream_loader

```
bool al_register_audio_stream_loader(const char *ext,
   ALLEGRO_AUDIO_STREAM *(*stream_loader)(const char *filename,
      size_t buffer_count, unsigned int samples))
```

Register a handler for al_load_audio_stream (25.8.9). The given function will be used to open streams from files with the given extension.

The extension should include the leading dot ('.') character. It will be matched case-insensitively.

The `stream_loader` argument may be NULL to unregister an entry.

Returns true on success, false on error. Returns false if unregistering an entry that doesn't exist.

25.8.6 al_register_audio_stream_loader_f

```
bool al_register_audio_stream_loader_f(const char *ext,
   ALLEGRO_AUDIO_STREAM *(*stream_loader)(ALLEGRO_FILE* fp,
      size_t buffer_count, unsigned int samples))
```

Register a handler for al_load_audio_stream_f (25.8.10). The given function will be used to open streams from files with the given extension.

The extension should include the leading dot ('.') character. It will be matched case-insensitively.

The `stream_loader` argument may be NULL to unregister an entry.

Returns true on success, false on error. Returns false if unregistering an entry that doesn't exist.

25.8.7 al_load_sample

```
ALLEGRO_SAMPLE *al_load_sample(const char *filename)
```

Loads a few different audio file formats based on their extension. Some formats require external libraries to be installed prior to compiling the library.

Note that this stores the entire file in memory at once, which may be time consuming. To read the file as it is needed, use al_load_audio_stream (25.8.9).

Returns the sample on success, NULL on failure.

See also: al_register_sample_loader (25.8.1), al_load_wav (25.8.13)

25.8.8 al_load_sample_f

```
ALLEGRO_SAMPLE *al_load_sample_f(ALLEGRO_FILE* fp, const char *ident)
```

Loads an audio file from an ALLEGRO_FILE stream into an ALLEGRO_SAMPLE. The file type is determined by the passed 'ident' parameter, which is a file name extension including the leading dot.

Note that this stores the entire file in memory at once, which may be time consuming. To read the file as it is needed, use al_load_audio_stream_f (25.8.10).

Returns the sample on success, NULL on failure.

See also: al_register_sample_loader_f (25.8.2), al_load_wav_f (25.8.14)

25.8.9 al_load_audio_stream

```
ALLEGRO_AUDIO_STREAM *al_load_audio_stream(const char *filename,
   size_t buffer_count, unsigned int samples)
```

Loads an audio file from disk as it is needed.

Unlike regular streams, the one returned by this function need not be fed by the user; the library will automatically read more of the file as it is needed. The stream will contain *buffer_count* buffers with *samples* samples.

A stream must be attached to a voice to be used. See ALLEGRO_AUDIO_STREAM (25.1.11) for more details.

Returns the stream on success, NULL on failure.

See also: al_register_audio_stream_loader (25.8.5), al_load_wav_audio_stream (25.8.17)

25.8.10 al_load_audio_stream_f

```
ALLEGRO_AUDIO_STREAM *al_load_audio_stream_f(ALLEGRO_FILE* fp, const char *ident,
   size_t buffer_count, unsigned int samples)
```

Loads an audio file from ALLEGRO_FILE stream as it is needed.

Unlike regular streams, the one returned by this function need not be fed by the user; the library will automatically read more of the file as it is needed. The stream will contain *buffer_count* buffers with *samples* samples.

The file type is determined by the passed 'ident' parameter, which is a file name extension including the leading dot.

A stream must be attached to a voice to be used. See ALLEGRO_AUDIO_STREAM (25.1.11) for more details.

Returns the stream on success, NULL on failure.

See also: al_register_audio_stream_loader_f (25.8.6), al_load_wav_audio_stream_f (25.8.18)

25.8.11 al_save_sample

```
bool al_save_sample(const char *filename, ALLEGRO_SAMPLE *spl)
```

Writes a sample into a file. Currently, wav is the only supported format, and the extension must be '.wav'.

Returns true on success, false on error.

See also: al_register_sample_saver (25.8.3), al_save_wav (25.8.15)

25.8.12 al_save_sample_f

```
bool al_save_sample_f(ALLEGRO_FILE *fp, const char *ident, ALLEGRO_SAMPLE *spl)
```

Writes a sample into a ALLEGRO_FILE (6.1) filestream. Currently, wav is the only supported format, and the extension must be '.wav'.

Returns true on success, false on error.

See also: al_register_sample_saver_f (25.8.4), al_save_wav_f (25.8.16)

25.8.13 al_load_wav

```
ALLEGRO_SAMPLE *al_load_wav(const char *filename)
```

Load a sample from a PCM .wav file.

Returns the sample on success, NULL on failure.

See also: al_load_sample (25.8.7), al_load_wav_f (25.8.14)

25.8.14 al_load_wav_f

```
ALLEGRO_SAMPLE *al_load_wav_f(ALLEGRO_FILE *fp)
```

Load a sample from a ALLEGRO_FILE (6.1) stream.

Returns the sample on success, NULL on failure.

See also: al_load_sample (25.8.7), al_load_wav (25.8.13)

25.8.15 al_save_wav

```
bool al_save_wav(const char *filename, ALLEGRO_SAMPLE *spl)
```

Save a sample to a PCM .wav file.

Returns true on success, false on error.

See also: al_save_sample (25.8.11), al_save_wav_f (25.8.16)

25.8.16 al_save_wav_f

```
bool al_save_wav_f(ALLEGRO_FILE *pf, ALLEGRO_SAMPLE *spl)
```

Write a PCM .wav file into the ALLEGRO_FILE (6.1) stream given.

Returns true on success, false on error.

See also: al_save_sample (25.8.11), al_save_wav (25.8.15)

25.8.17 al_load_wav_audio_stream

```
ALLEGRO_AUDIO_STREAM *al_load_wav_audio_stream(const char *filename,
   size_t buffer_count, unsigned int samples)
```

Like al_load_audio_stream (25.8.9) but assumes the file is PCM .wav file.

See also: al_load_audio_stream (25.8.9), al_load_wav_audio_stream_f (25.8.18)

25.8.18 al_load_wav_audio_stream_f

```
ALLEGRO_AUDIO_STREAM *al_load_wav_audio_stream_f(ALLEGRO_FILE* f,
   size_t buffer_count, unsigned int samples)
```

Like al_load_audio_stream_f (25.8.10) but assumes the file is PCM .wav file.

See also: al_load_audio_stream_f (25.8.10), al_load_wav_audio_stream (25.8.17)

26 Audio codecs

26.1 FLAC addon

These functions are declared in the following header file. Link with allegro_flac.

```
#include <allegro5/allegro_flac.h>
```

26.1.1 al_init_flac_addon

```
bool al_init_flac_addon(void)
```

This function registers al_load_flac (26.1.3) with al_load_sample (25.8.7) to handle files with the extension ".flac". You will need to include the `allegro_flac.h` header file and link with the `allegro_flac` library.

Return true on success.

26.1.2 al_get_allegro_flac_version

```
uint32_t al_get_allegro_flac_version(void)
```

Returns the (compiled) version of the addon, in the same format as al_get_allegro_version (19.4).

26.1.3 al_load_flac

```
ALLEGRO_SAMPLE *al_load_flac(const char *filename)
```

Loads a sample in FLAC format.

See also: al_load_sample (25.8.7)

26.1.4 al_load_flac_f

```
ALLEGRO_SAMPLE *al_load_flac_f(ALLEGRO_FILE* f)
```

Loads a sample in FLAC format from the ALLEGRO_FILE (6.1) stream given.

See also: al_load_sample_f (25.8.8)

26.1.5 al_load_flac_audio_stream

```
ALLEGRO_AUDIO_STREAM *al_load_flac_audio_stream(const char *filename,
   size_t buffer_count, unsigned int samples)
```

Loads a stream in FLAC format.

See also: al_load_audio_stream (25.8.9)

26.1.6 al_load_flac_audio_stream_f

```
ALLEGRO_AUDIO_STREAM *al_load_flac_audio_stream_f(ALLEGRO_FILE* f,
   size_t buffer_count, unsigned int samples)
```

Loads a stream in FLAC format from the ALLEGRO_FILE (6.1) stream given.

See also: al_load_audio_stream_f (25.8.10)

26.2 MOD Audio addon

These functions are declared in the following header file. Link with allegro_modaudio.

```
#include <allegro5/allegro_modaudio.h>
```

26.2.1 al_init_modaudio_addon

```
bool al_init_modaudio_addon()
```

This function registers al_load_it_audio_stream (26.2.3), al_load_mod_audio_stream (26.2.5), al_load_s3m_audio_stream (26.2.7), and al_load_xm_audio_stream (26.2.9) with al_load_audio_stream (25.8.9) to handle files with the extensions ".it", ".mod", ".s3m", and ".xm". You will need to include the `allegro_modaudio.h` header file and link with the `allegro_modaudio` library.

Return true if every function is registered successfully.

26.2.2 al_get_allegro_modaudio_version

```
uint32_t al_get_allegro_modaudio_version()
```

Returns the (compiled) version of the addon, in the same format as al_get_allegro_version (19.4).

26.2.3 al_load_it_audio_stream

```
ALLEGRO_AUDIO_STREAM *al_load_it_audio_stream(const char *filename,
   size_t buffer_count, unsigned int samples)
```

Loads a stream in the Impulse Tracker format.

See also: al_load_audio_stream (25.8.9)

26.2.4 al_load_it_audio_stream_f

```
ALLEGRO_AUDIO_STREAM *al_load_it_audio_stream_f(ALLEGRO_FILE *f,
   size_t buffer_count, unsigned int samples)
```

Loads a stream in the Impulse Tracker format from the ALLEGRO_FILE (6.1) stream given.

See also: al_load_audio_stream_f (25.8.10)

26.2.5 al_load_mod_audio_stream

```
ALLEGRO_AUDIO_STREAM *al_load_mod_audio_stream(const char *filename,
   size_t buffer_count, unsigned int samples)
```

Loads a stream in the Amiga Module format.

See also: al_load_audio_stream (25.8.9)

26.2.6 al_load_mod_audio_stream_f

```
ALLEGRO_AUDIO_STREAM *al_load_mod_audio_stream_f(ALLEGRO_FILE *f,
   size_t buffer_count, unsigned int samples)
```

Loads a stream in the Amiga Module format from the ALLEGRO_FILE (6.1) stream given.

See also: al_load_audio_stream_f (25.8.10)

26.2.7 al_load_s3m_audio_stream

```
ALLEGRO_AUDIO_STREAM *al_load_s3m_audio_stream(const char *filename,
   size_t buffer_count, unsigned int samples)
```

Loads a stream in the Scream Tracker 3 format.

See also: al_load_audio_stream (25.8.9)

26.2.8 al_load_s3m_audio_stream_f

```
ALLEGRO_AUDIO_STREAM *al_load_s3m_audio_stream_f(ALLEGRO_FILE *f,
   size_t buffer_count, unsigned int samples)
```

Loads a stream in the Scream Tracker 3 format from the ALLEGRO_FILE (6.1) stream given.

See also: al_load_audio_stream_f (25.8.10)

26.2.9 al_load_xm_audio_stream

```
ALLEGRO_AUDIO_STREAM *al_load_xm_audio_stream(const char *filename,
   size_t buffer_count, unsigned int samples)
```

Loads a stream in the Fast Tracker 2 format.

See also: al_load_audio_stream (25.8.9)

26.2.10 al_load_xm_audio_stream_f

```
ALLEGRO_AUDIO_STREAM *al_load_xm_audio_stream_f(ALLEGRO_FILE *f,
   size_t buffer_count, unsigned int samples)
```

Loads a stream in Fast Tracker 2 format from the ALLEGRO_FILE (6.1) stream given.

See also: al_load_audio_stream_f (25.8.10)

26.3 Ogg Vorbis addon

These functions are declared in the following header file. Link with allegro_vorbis.

```
#include <allegro5/allegro_vorbis.h>
```

26.3.1 al_init_ogg_vorbis_addon

```
bool al_init_ogg_vorbis_addon(void)
```

This function registers al_load_ogg_vorbis (26.3.3) with al_load_sample (25.8.7) and al_load_ogg_vorbis_audio_stream (26.3.5) with al_load_audio_stream (25.8.9) to handle files with the extension ".ogg" (assumed to contain Vorbis data).
You will need to include the `allegro_vorbis.h` header file and link with the `allegro_vorbis` library.

Return true on success.

26.3.2 al_get_allegro_ogg_vorbis_version

```
uint32_t al_get_allegro_ogg_vorbis_version(void)
```

Returns the (compiled) version of the addon, in the same format as al_get_allegro_version (19.4).

26.3.3 al_load_ogg_vorbis

```
ALLEGRO_SAMPLE *al_load_ogg_vorbis(const char *filename)
```

Loads a sample in Ogg Vorbis format.

See also: al_load_sample (25.8.7)

26.3.4 al_load_ogg_vorbis_f

```
ALLEGRO_SAMPLE *al_load_ogg_vorbis_f(ALLEGRO_FILE* file)
```

Loads a sample in Ogg Vorbis format from the ALLEGRO_FILE (6.1) stream given.

See also: al_load_sample_f (25.8.8)

26.3.5 al_load_ogg_vorbis_audio_stream

```
ALLEGRO_AUDIO_STREAM *al_load_ogg_vorbis_audio_stream(const char *filename,
    size_t buffer_count, unsigned int samples)
```

Loads a stream in Ogg Vorbis format.

See also: al_load_audio_stream (25.8.9)

26.3.6 al_load_ogg_vorbis_audio_stream_f

```
ALLEGRO_AUDIO_STREAM *al_load_ogg_vorbis_audio_stream_f(ALLEGRO_FILE* file,
    size_t buffer_count, unsigned int samples)
```

Loads a stream in Ogg Vorbis format from the ALLEGRO_FILE (6.1) stream given.

See also: al_load_audio_stream_f (25.8.10)

27 Color addon

These functions are declared in the following header file. Link with allegro_color.

```
#include <allegro5/allegro_color.h>
```

27.1 al_color_cmyk

```
ALLEGRO_COLOR al_color_cmyk(float c, float m, float y, float k)
```

Return an ALLEGRO_COLOR (9.1.1) structure from CMYK values (cyan, magenta, yellow, black).

See also: al_color_cmyk_to_rgb (27.2), al_color_rgb_to_cmyk (27.12)

27.2 al_color_cmyk_to_rgb

```
void al_color_cmyk_to_rgb(float cyan, float magenta, float yellow,
    float key, float *red, float *green, float *blue)
```

Convert CMYK values to RGB values.

See also: al_color_cmyk (27.1), al_color_rgb_to_cmyk (27.12)

27.3 al_color_hsl

```
ALLEGRO_COLOR al_color_hsl(float h, float s, float l)
```

Return an ALLEGRO_COLOR (9.1.1) structure from HSL (hue, saturation, lightness) values.

See also: al_color_hsl_to_rgb (27.4), al_color_hsv (27.5)

27.4 al_color_hsl_to_rgb

```
void al_color_hsl_to_rgb(float hue, float saturation, float lightness,
   float *red, float *green, float *blue)
```

Convert values in HSL color model to RGB color model.

Parameters:

- hue - Color hue angle in the range 0..360.
- saturation - Color saturation in the range 0..1.
- lightness - Color lightness in the range 0..1.
- red, green, blue - returned RGB values in the range 0..1.

See also: al_color_rgb_to_hsl (27.13), al_color_hsl (27.3), al_color_hsv_to_rgb (27.6)

27.5 al_color_hsv

```
ALLEGRO_COLOR al_color_hsv(float h, float s, float v)
```

Return an ALLEGRO_COLOR (9.1.1) structure from HSV (hue, saturation, value) values.

See also: al_color_hsv_to_rgb (27.6), al_color_hsl (27.3)

27.6 al_color_hsv_to_rgb

```
void al_color_hsv_to_rgb(float hue, float saturation, float value,
   float *red, float *green, float *blue)
```

Convert values in HSV color model to RGB color model.

Parameters:

- hue - Color hue angle in the range 0..360.
- saturation - Color saturation in the range 0..1.
- value - Color value in the range 0..1.
- red, green, blue - returned RGB values in the range 0..1.

See also: al_color_rgb_to_hsv (27.14), al_color_hsv (27.5), al_color_hsl_to_rgb (27.4)

27.7 al_color_html

```
ALLEGRO_COLOR al_color_html(char const *string)
```

Interprets an HTML styled hex number (e.g. #00faff) as a color. Components that are malformed are set to 0.

See also: al_color_html_to_rgb (27.8), al_color_rgb_to_html (27.9)

27.8 al_color_html_to_rgb

```
void al_color_html_to_rgb(char const *string,
   float *red, float *green, float *blue)
```

Interprets an HTML styled hex number (e.g. #00faff) as a color. Components that are malformed are set to 0.

See also: al_color_html (27.7), al_color_rgb_to_html (27.9)

27.9 al_color_rgb_to_html

```
void al_color_rgb_to_html(float red, float green, float blue,
    char *string)
```

Create an HTML-style string representation of an ALLEGRO_COLOR (9.1.1), e.g. #00faff.

Parameters:

- red, green, blue - The color components in the range 0..1.
- string - A string with a size of 8 bytes into which the result will be written.

Example:

```
char html[8];
al_color_rgb_to_html(1, 0, 0, html);
```

Now html will contain "#ff0000".

See also: al_color_html (27.7), al_color_html_to_rgb (27.8)

27.10 al_color_name

```
ALLEGRO_COLOR al_color_name(char const *name)
```

Return an ALLEGRO_COLOR (9.1.1) with the given name. If the color is not found then black is returned.

See al_color_name_to_rgb (27.11) for the list of names.

27.11 al_color_name_to_rgb

```
bool al_color_name_to_rgb(char const *name, float *r, float *g, float *b)
```

Parameters:

- name - The (lowercase) name of the color.
- r, g, b - If one of the recognized color names below is passed, the corresponding RGB values in the range 0..1 are written.

The recognized names are:

> aliceblue, antiquewhite, aqua, aquamarine, azure, beige, bisque, black, blanchedalmond, blue, blueviolet, brown, burlywood, cadetblue, chartreuse, chocolate, coral, cornflowerblue, cornsilk, crimson, cyan, darkblue, darkcyan, darkgoldenrod, darkgray, darkgreen, darkkhaki, darkmagenta, darkolivegreen, darkorange, darkorchid, darkred, darksalmon, darkseagreen, darkslateblue, darkslategray, darkturquoise, darkviolet, deeppink, deepskyblue, dimgray, dodgerblue, firebrick, floralwhite, forestgreen, fuchsia, gainsboro, ghostwhite, goldenrod, gold, gray, green, greenyellow, honeydew, hotpink, indianred, indigo, ivory, khaki, lavenderblush, lavender, lawngreen, lemonchiffon, lightblue, lightcoral, lightcyan, lightgoldenrodyellow, lightgreen, lightgrey, lightpink, lightsalmon, lightseagreen, lightskyblue, lightslategray, lightsteelblue, lightyellow, lime, limegreen, linen, magenta, maroon, mediumaquamarine, mediumblue, mediumorchid, mediumpurple, mediumseagreen, mediumslateblue, mediumspringgreen, mediumturquoise, mediumvioletred, midnightblue, mintcream, mistyrose, moccasin, avajowhite, navy, oldlace, olive, olivedrab, orange, orangered, orchid, palegoldenrod, palegreen, paleturquoise, palevioletred, papayawhip, peachpuff, peru, pink, plum, powderblue, purple, purwablue, red, rosybrown, royalblue, saddlebrown, salmon, sandybrown, seagreen, seashell, sienna, silver, skyblue, slateblue, slategray, snow, springgreen, steelblue, tan, teal, thistle, tomato, turquoise, violet, wheat, white, whitesmoke, yellow, yellowgreen

They are taken from `http://en.wikipedia.org/wiki/X11_color_names`, with CSS names being preferred over X11 ones where there is overlap.

Returns: true if a name from the list above was passed, else false.

See also: al_color_name (27.10)

27.12 al_color_rgb_to_cmyk

```
void al_color_rgb_to_cmyk(float red, float green, float blue,
   float *cyan, float *magenta, float *yellow, float *key)
```

Each RGB color can be represented in CMYK with a K component of 0 with the following formula:

```
C = 1 - R
M = 1 - G
Y = 1 - B
K = 0
```

This function will instead find the representation with the maximal value for K and minimal color components.

See also: al_color_cmyk (27.1), al_color_cmyk_to_rgb (27.2)

27.13 al_color_rgb_to_hsl

```
void al_color_rgb_to_hsl(float red, float green, float blue,
   float *hue, float *saturation, float *lightness)
```

Given an RGB triplet with components in the range 0..1, return the hue in degrees from 0..360 and saturation and lightness in the range 0..1.

See also: al_color_hsl_to_rgb (27.4), al_color_hsl (27.3)

27.14 al_color_rgb_to_hsv

```
void al_color_rgb_to_hsv(float red, float green, float blue,
   float *hue, float *saturation, float *value)
```

Given an RGB triplet with components in the range 0..1, return the hue in degrees from 0..360 and saturation and value in the range 0..1.

See also: al_color_hsv_to_rgb (27.6), al_color_hsv (27.5)

27.15 al_color_rgb_to_name

```
char const *al_color_rgb_to_name(float r, float g, float b)
```

Given an RGB triplet with components in the range 0..1, find a color name describing it approximately.

See also: al_color_name_to_rgb (27.11), al_color_name (27.10)

27.16 al_color_rgb_to_yuv

```
void al_color_rgb_to_yuv(float red, float green, float blue,
   float *y, float *u, float *v)
```

Convert RGB values to YUV color space.

See also: al_color_yuv (27.17), al_color_yuv_to_rgb (27.18)

27.17 al_color_yuv

```
ALLEGRO_COLOR al_color_yuv(float y, float u, float v)
```

Return an ALLEGRO_COLOR (9.1.1) structure from YUV values.

See also: al_color_yuv_to_rgb (27.18), al_color_rgb_to_yuv (27.16)

27.18 al_color_yuv_to_rgb

```
void al_color_yuv_to_rgb(float y, float u, float v,
    float *red, float *green, float *blue)
```

Convert YUV color values to RGB color space.

See also: al_color_yuv (27.17), al_color_rgb_to_yuv (27.16)

27.19 al_get_allegro_color_version

```
uint32_t al_get_allegro_color_version(void)
```

Returns the (compiled) version of the addon, in the same format as al_get_allegro_version (19.4).

28 Font addons

These functions are declared in the following header file. Link with allegro_font.

```
#include <allegro5/allegro_font.h>
```

28.1 General font routines

28.1.1 ALLEGRO_FONT

```
typedef struct ALLEGRO_FONT ALLEGRO_FONT;
```

A handle identifying any kind of font. Usually you will create it with al_load_font (28.1.4) which supports loading all kinds of TrueType fonts supported by the FreeType library. If you instead pass the filename of a bitmap file, it will be loaded with al_load_bitmap (29.7) and a font in Allegro's bitmap font format will be created from it with al_grab_font_from_bitmap (28.2.1).

28.1.2 al_init_font_addon

```
void al_init_font_addon(void)
```

Initialise the font addon.

See also: al_init_ttf_addon (28.3.1), al_shutdown_font_addon (28.1.3)

28.1.3 al_shutdown_font_addon

```
void al_shutdown_font_addon(void)
```

Shut down the font addon. This is done automatically at program exit, but can be called any time the user wishes as well.

See also: al_init_font_addon (28.1.2)

28.1.4 al_load_font

```
ALLEGRO_FONT *al_load_font(char const *filename, int size, int flags)
```

Loads a font from disk. This will use al_load_bitmap_font (28.2.2) if you pass the name of a known bitmap format, or else al_load_ttf_font (28.3.2).

See also: al_destroy_font (28.1.5), al_init_font_addon (28.1.2)

28.1.5 al_destroy_font

```
void al_destroy_font(ALLEGRO_FONT *f)
```

Frees the memory being used by a font structure.

See also: al_load_font (28.1.4)

28.1.6 al_register_font_loader

```
bool al_register_font_loader(char const *extension,
   ALLEGRO_FONT *(*load_font)(char const *filename, int size, int flags))
```

Informs Allegro of a new font file type, telling it how to load files of this format.

The extension should include the leading dot ('.') character. It will be matched case-insensitively.

The `load_font` argument may be NULL to unregister an entry.

Returns true on success, false on error. Returns false if unregistering an entry that doesn't exist.

See also: al_init_font_addon (28.1.2)

28.1.7 al_get_font_line_height

```
int al_get_font_line_height(const ALLEGRO_FONT *f)
```

Returns the usual height of a line of text in the specified font. For bitmap fonts this is simply the height of all glyph bitmaps. For truetype fonts it is whatever the font file specifies. In particular, some special glyphs may be higher than the height returned here.

See also: al_get_text_width (28.1.8), al_get_text_dimensions (28.1.16)

28.1.8 al_get_text_width

```
int al_get_text_width(const ALLEGRO_FONT *f, const char *str)
```

Calculates the length of a string in a particular font, in pixels.

See also: al_get_ustr_width (28.1.9), al_get_font_line_height (28.1.7), al_get_text_dimensions (28.1.16)

28.1.9 al_get_ustr_width

```
int al_get_ustr_width(const ALLEGRO_FONT *f, ALLEGRO_USTR const *ustr)
```

Like al_get_text_width (28.1.8) but expects an ALLEGRO_USTR.

See also: al_get_text_width (28.1.8), al_get_ustr_dimensions (28.1.17)

28.1.10 al_draw_text

```
void al_draw_text(const ALLEGRO_FONT *font, float x, float y, int flags,
   char const *text)
```

Writes the 0-terminated string `text` onto `bmp` at position `x`, `y`, using the specified `font`.

The `flags` parameter can be 0 or one of the following flags:

- ALLEGRO_ALIGN_LEFT - Draw the text left-aligned (same as 0).

- ALLEGRO_ALIGN_CENTRE - Draw the text centered around the given position.
- ALLEGRO_ALIGN_RIGHT - Draw the text right-aligned to the given position.

See also: al_draw_ustr (28.1.11), al_draw_textf (28.1.14), al_draw_justified_text (28.1.12)

28.1.11 al_draw_ustr

```
void al_draw_ustr(const ALLEGRO_FONT *font, float x, float y, int flags,
   const ALLEGRO_USTR *ustr)
```

Like al_draw_text (28.1.10), except the text is passed as an ALLEGRO_USTR instead of a 0-terminated char array.

See also: al_draw_text (28.1.10), al_draw_justified_ustr (28.1.13)

28.1.12 al_draw_justified_text

```
void al_draw_justified_text(const ALLEGRO_FONT *font, float x1, float x2,
   float y, float diff, int flags, const char *text)
```

Like al_draw_text (28.1.10), but justifies the string to the specified area.

See also: al_draw_justified_textf (28.1.15), al_draw_justified_ustr (28.1.13)

28.1.13 al_draw_justified_ustr

```
void al_draw_justified_ustr(const ALLEGRO_FONT *font, float x1, float x2,
   float y, float diff, int flags, const ALLEGRO_USTR *ustr)
```

Like al_draw_ustr (28.1.11), but justifies the string to the specified area.

See also: al_draw_justified_text (28.1.12), al_draw_justified_textf (28.1.15).

28.1.14 al_draw_textf

```
void al_draw_textf(const ALLEGRO_FONT *font, float x, float y, int flags,
   const char *format, ...)
```

Formatted text output, using a printf() style format string, all parameters have the same meaning as with al_draw_text (28.1.10) otherwise.

See also: al_draw_text (28.1.10), al_draw_ustr (28.1.11)

28.1.15 al_draw_justified_textf

```
void al_draw_justified_textf(const ALLEGRO_FONT *f, float x1, float x2, float y,
   float diff, int flags, const char *format, ...)
```

Like al_draw_justified_text (28.1.12) and al_draw_textf (28.1.14).

See also: al_draw_justified_text (28.1.12), al_draw_justified_ustr (28.1.13).

28.1.16 al_get_text_dimensions

```
void al_get_text_dimensions(const ALLEGRO_FONT *f,
   char const *text,
   int *bbx, int *bby, int *bbw, int *bbh, int *ascent, int *descent)
```

Sometimes, the al_get_text_width (28.1.8) and al_get_font_line_height (28.1.7) functions are not enough for exact text placement, so this function returns some additional information.

Returned variables (all in pixel):

- x, y - Offset to upper left corner of bounding box.
- w, h - Dimensions of bounding box.
- ascent - Ascent of the font.
- descent - Descent of the font.

If the X is the position you specify to draw text, the meaning of ascent and descent and the line height is like in the figure below. Note that glyphs may go to the left and upwards of the X, in which case x and y will have negative values.

```
X------------------------
    /\         |        |
   /  \        |        |
  /____\       ascent   |
 /      \      |        |
/        \     |        height
----------------        |
               |        |
               descent  |
               |        |
-------------------------
```

See also: al_get_text_width (28.1.8), al_get_font_line_height (28.1.7), al_get_ustr_dimensions (28.1.17)

28.1.17 al_get_ustr_dimensions

```
void al_get_ustr_dimensions(const ALLEGRO_FONT *f,
   ALLEGRO_USTR const *ustr,
   int *bbx, int *bby, int *bbw, int *bbh, int *ascent, int *descent)
```

Sometimes, the al_get_ustr_width (28.1.9) and al_get_font_line_height (28.1.7) functions are not enough for exact text placement, so this function returns some additional information.

See also: al_get_text_dimensions (28.1.16)

28.1.18 al_get_allegro_font_version

```
uint32_t al_get_allegro_font_version(void)
```

Returns the (compiled) version of the addon, in the same format as al_get_allegro_version (19.4).

28.2 Bitmap fonts

28.2.1 al_grab_font_from_bitmap

```
ALLEGRO_FONT *al_grab_font_from_bitmap(ALLEGRO_BITMAP *bmp,
   int ranges_n, int ranges[])
```

Creates a new font from an Allegro bitmap. You can delete the bitmap after the function returns as the font will contain a copy for itself.

Parameters:

- bmp: The bitmap with the glyphs drawn onto it
- n: Number of unicode ranges in the bitmap.
- ranges: 'n' pairs of first and last unicode point to map glyphs to for each range.

The bitmap format is as in the following example, which contains three glyphs for 1, 2 and 3.

```
.............
. 1 .222.333.
. 1 .  2.  3.
. 1 .222.333.
. 1 .2  .  3.
. 1 .222.333.
.............
```

In the above illustration, the dot is for pixels having the background color. It is determined by the color of the top left pixel in the bitmap. There should be a border of at least 1 pixel with this color to the bitmap edge and between all glyphs.

Each glyph is inside a rectangle of pixels not containing the background color. The height of all glyph rectangles should be the same, but the width can vary.

The placement of the rectangles does not matter, except that glyphs are scanned from left to right and top to bottom to match them to the specified unicode codepoints.

The glyphs will simply be drawn using al_draw_bitmap (9.5.2), so usually you will want the rectangles filled with full transparency and the glyphs drawn in opaque white.

Examples:

```
int ranges[] = {32, 126};
al_font_grab_font_from_bitmap(bitmap, 1, ranges)

int ranges[] = {
    0x0020, 0x007F,  /* ASCII */
    0x00A1, 0x00FF,  /* Latin 1 */
    0x0100, 0x017F,  /* Extended-A */
    0x20AC, 0x20AC}; /* Euro */
al_font_grab_font_from_bitmap(bitmap, 4, ranges)
```

The first example will grab glyphs for the 95 standard printable ASCII characters, beginning with the space character (32) and ending with the tilde character (126). The second example will map the first 96 glyphs found in the bitmap to ASCII range, the next 95 glyphs to Latin 1, the next 128 glyphs to Extended-A, and the last glyph to the Euro character. (This is just the characters found in the Allegro 4 font.)

See also: al_load_bitmap (29.7), al_grab_font_from_bitmap (28.2.1)

28.2.2 al_load_bitmap_font

```
ALLEGRO_FONT *al_load_bitmap_font(const char *fname)
```

Load a bitmap font from. It does this by first calling al_load_bitmap (29.7) and then al_grab_font_from_bitmap (28.2.1). If you want to for example load an old A4 font, you could load the bitmap yourself, then call al_convert_mask_to_alpha (9.8.1) on it and only then pass it to al_grab_font_from_bitmap (28.2.1).

28.3 TTF fonts

These functions are declared in the following header file. Link with allegro_ttf.

```
#include <allegro5/allegro_ttf.h>
```

28.3.1 al_init_ttf_addon

```
bool al_init_ttf_addon(void)
```

Call this after al_init_font_addon (28.1.2) to make al_load_font (28.1.4) recognize .ttf and other formats supported by al_load_ttf_font (28.3.2).

28.3.2 al_load_ttf_font

```
ALLEGRO_FONT *al_load_ttf_font(char const *filename, int size, int flags)
```

Loads a TrueType font from a file using the FreeType library. Quoting from the FreeType FAQ this means support for many different font formats:

TrueType, OpenType, Type1, CID, CFF, Windows FON/FNT, X11 PCF, and others

The *size* parameter determines the size the font will be rendered at, specified in pixel. The standard font size is measured in *units per EM*, if you instead want to specify the size as the total height of glyphs in pixel, pass it as a negative value.

Note: If you want to display text at multiple sizes, load the font multiple times with different size parameters.

The only flag supported right now is:

- ALLEGRO_TTF_NO_KERNING - Do not use any kerning even if the font file supports it.

See also: al_init_ttf_addon (28.3.1), al_load_ttf_font_entry (28.3.3)

28.3.3 al_load_ttf_font_entry

```
ALLEGRO_FONT *al_load_ttf_font_entry(ALLEGRO_FILE *file,
    char const *filename, int size, int flags)
```

Like al_load_ttf_font (28.3.2), but the font is read from the file handle. The filename is only used to find possible additional files next to a font file.

Note: The file handle is owned by the returned ALLEGRO_FONT object and must not be freed by the caller, as FreeType expects to be able to read from it at a later time.

28.3.4 al_get_allegro_ttf_version

```
uint32_t al_get_allegro_ttf_version(void)
```

Returns the (compiled) version of the addon, in the same format as al_get_allegro_version (19.4).

29 Image I/O addon

These functions are declared in the following header file. Link with allegro_image.

```
#include <allegro5/allegro_image.h>
```

29.1 al_init_image_addon

```
bool al_init_image_addon(void)
```

Initializes the IIO addon.

29.2 al_shutdown_image_addon

```
void al_shutdown_image_addon(void)
```

Shut down the IIO addon. This is done automatically at program exit, but can be called any time the user wishes as well.

29.3 al_register_bitmap_loader

```
bool al_register_bitmap_loader(const char *extension,
   ALLEGRO_BITMAP *(*loader)(const char *filename))
```

Register a handler for al_load_bitmap (29.7). The given function will be used to handle the loading of bitmaps files with the given extension.

The extension should include the leading dot ('.') character. It will be matched case-insensitively.

The `loader` argument may be NULL to unregister an entry.

Returns true on success, false on error. Returns false if unregistering an entry that doesn't exist.

29.4 al_register_bitmap_saver

```
bool al_register_bitmap_saver(const char *extension,
   bool (*saver)(const char *filename, ALLEGRO_BITMAP *bmp))
```

Register a handler for al_save_bitmap (29.9). The given function will be used to handle the loading of bitmaps files with the given extension.

The extension should include the leading dot ('.') character. It will be matched case-insensitively.

The `saver` argument may be NULL to unregister an entry.

Returns true on success, false on error. Returns false if unregistering an entry that doesn't exist.

29.5 al_register_bitmap_loader_f

```
bool al_register_bitmap_loader_f(const char *extension,
   ALLEGRO_BITMAP *(*loader_f)(ALLEGRO_FILE *fp))
```

Register a handler for al_load_bitmap_f (29.8). The given function will be used to handle the loading of bitmaps files with the given extension.

The extension should include the leading dot ('.') character. It will be matched case-insensitively.

The `fs_loader` argument may be NULL to unregister an entry.

Returns true on success, false on error. Returns false if unregistering an entry that doesn't exist.

29.6 al_register_bitmap_saver_f

```
bool al_register_bitmap_saver_f(const char *extension,
   bool (*saver_f)(ALLEGRO_FILE *fp, ALLEGRO_BITMAP *bmp))
```

Register a handler for al_save_bitmap_f (29.10). The given function will be used to handle the loading of bitmaps files with the given extension.

The extension should include the leading dot ('.') character. It will be matched case-insensitively.

The `saver_f` argument may be NULL to unregister an entry.

Returns true on success, false on error. Returns false if unregistering an entry that doesn't exist.

29.7 al_load_bitmap

```
ALLEGRO_BITMAP *al_load_bitmap(const char *filename)
```

Loads an image file into an ALLEGRO_BITMAP. The file type is determined by the extension.

Returns NULL on error.

See also: al_load_bitmap_f (29.8), al_register_bitmap_loader (29.3), al_set_new_bitmap_format (9.3.9), al_set_new_bitmap_flags (

29.8 al_load_bitmap_f

```
ALLEGRO_BITMAP *al_load_bitmap_f(ALLEGRO_FILE *fp, const char *ident)
```

Loads an image from an ALLEGRO_FILE stream into an ALLEGRO_BITMAP. The file type is determined by the passed 'ident' parameter, which is a file name extension including the leading dot.

Returns NULL on error.

See also: al_load_bitmap (29.7), al_register_bitmap_loader_f (29.5)

29.9 al_save_bitmap

```
bool al_save_bitmap(const char *filename, ALLEGRO_BITMAP *bitmap)
```

Saves an ALLEGRO_BITMAP to an image file. The file type is determined by the extension.

Returns true on success, false on error.

See also: al_save_bitmap_f (29.10), al_register_bitmap_saver (29.4)

29.10 al_save_bitmap_f

```
bool al_save_bitmap_f(ALLEGRO_FILE *fp, const char *ident,
   ALLEGRO_BITMAP *bitmap)
```

Saves an ALLEGRO_BITMAP to an ALLEGRO_FILE stream. The file type is determined by the passed 'ident' parameter, which is a file name extension including the leading dot.

Returns true on success, false on error.

See also: al_save_bitmap (29.9), al_register_bitmap_saver_f (29.6)

29.11 al_load_bmp

```
ALLEGRO_BITMAP *al_load_bmp(const char *filename)
```

Create a new ALLEGRO_BITMAP from a BMP file. The bitmap is created with al_create_bitmap (9.3.3).

Returns NULL on error.

See Also: al_load_bitmap (29.7).

29.12 al_load_bmp_f

```
ALLEGRO_BITMAP *al_load_bmp_f(ALLEGRO_FILE *f)
```

See al_load_bmp (29.11) and al_load_bitmap_f (29.8).

29.13 al_load_jpg

```
ALLEGRO_BITMAP *al_load_jpg(char const *filename)
```

Create a new ALLEGRO_BITMAP from a JPEG file. The bitmap is created with al_create_bitmap (9.3.3).
Returns NULL on error.
See Also: al_load_bitmap (29.7).

29.14 al_load_jpg_f

```
ALLEGRO_BITMAP *al_load_jpg_f(ALLEGRO_FILE *fp)
```

See al_load_jpg (29.13) and al_load_bitmap_f (29.8).

29.15 al_load_pcx

```
ALLEGRO_BITMAP *al_load_pcx(const char *filename)
```

Create a new ALLEGRO_BITMAP from a PCX file. The bitmap is created with al_create_bitmap (9.3.3).
Returns NULL on error.
See Also: al_load_bitmap (29.7).

29.16 al_load_pcx_f

```
ALLEGRO_BITMAP *al_load_pcx_f(ALLEGRO_FILE *f)
```

See al_load_pcx (29.15) and al_load_bitmap_f (29.8).

29.17 al_load_png

```
ALLEGRO_BITMAP *al_load_png(const char *filename)
```

Create a new ALLEGRO_BITMAP from a PNG file. The bitmap is created with al_create_bitmap (9.3.3).
Returns NULL on error.
See Also: al_load_bitmap (29.7).

29.18 al_load_png_f

```
ALLEGRO_BITMAP *al_load_png_f(ALLEGRO_FILE *fp)
```

See al_load_png (29.17) and al_load_bitmap_f (29.8).

29.19 al_load_tga

```
ALLEGRO_BITMAP *al_load_tga(const char *filename)
```

Create a new ALLEGRO_BITMAP from a TGA file. The bitmap is created with al_create_bitmap (9.3.3).

Returns NULL on error.

See Also: al_load_bitmap (29.7).

29.20 al_load_tga_f

```
ALLEGRO_BITMAP *al_load_tga_f(ALLEGRO_FILE *f)
```

See al_load_tga (29.19) and al_load_bitmap_f (29.8).

29.21 al_save_bmp

```
bool al_save_bmp(const char *filename, ALLEGRO_BITMAP *bmp)
```

Save an ALLEGRO_BITMAP as a BMP file.

Returns true on success, false on error.

See Also: al_save_bitmap (29.9).

29.22 al_save_bmp_f

```
bool al_save_bmp_f(ALLEGRO_FILE *f, ALLEGRO_BITMAP *bmp)
```

See al_save_bmp (29.21) and al_save_bitmap_f (29.10).

29.23 al_save_jpg

```
bool al_save_jpg(char const *filename, ALLEGRO_BITMAP *bmp)
```

Save an ALLEGRO_BITMAP as a JPEG file.

Returns true on success, false on error.

See Also: al_save_bitmap (29.9).

29.24 al_save_jpg_f

```
bool al_save_jpg_f(ALLEGRO_FILE *fp, ALLEGRO_BITMAP *bmp)
```

See al_save_jpg (29.23) and al_save_bitmap_f (29.10).

29.25 al_save_pcx

```
bool al_save_pcx(const char *filename, ALLEGRO_BITMAP *bmp)
```

Save an ALLEGRO_BITMAP as a PCX file.

Returns true on success, false on error.

See Also: al_save_bitmap (29.9).

29.26 al_save_pcx_f

```
bool al_save_pcx_f(ALLEGRO_FILE *f, ALLEGRO_BITMAP *bmp)
```

See al_save_pcx (29.25) and al_save_bitmap_f (29.10).

29.27 al_save_png

```
bool al_save_png(const char *filename, ALLEGRO_BITMAP *bmp)
```

Save an ALLEGRO_BITMAP as a PNG file.

Returns true on success, false on error.

See Also: al_save_bitmap (29.9).

29.28 al_save_png_f

```
bool al_save_png_f(ALLEGRO_FILE *fp, ALLEGRO_BITMAP *bmp)
```

See al_save_png (29.27) and al_save_bitmap_f (29.10).

29.29 al_save_tga

```
bool al_save_tga(const char *filename, ALLEGRO_BITMAP *bmp)
```

Save an ALLEGRO_BITMAP as a TGA file.

Returns true on success, false on error.

See Also: al_save_bitmap (29.9).

29.30 al_save_tga_f

```
bool al_save_tga_f(ALLEGRO_FILE *f, ALLEGRO_BITMAP *bmp)
```

See al_save_tga (29.29) and al_save_bitmap_f (29.10).

29.31 al_get_allegro_image_version

```
uint32_t al_get_allegro_image_version(void)
```

Returns the (compiled) version of the addon, in the same format as al_get_allegro_version (19.4).

30 Native dialogs support

These functions are declared in the following header file. Link with allegro_dialog.

```
#include <allegro5/allegro_native_dialog.h>
```

30.1 ALLEGRO_NATIVE_DIALOG

```
typedef struct ALLEGRO_NATIVE_DIALOG ALLEGRO_NATIVE_DIALOG;
```

Opaque handle to a native file dialog. You should only have one such dialog opened at a time.

30.2 al_create_native_file_dialog

```
ALLEGRO_NATIVE_DIALOG *al_create_native_file_dialog(
    ALLEGRO_PATH const *initial_path,
    char const *title,
    char const *patterns,
    int mode)
```

Creates a new native file dialog.

Parameters:

- initial_path: The initial search path and filename. Can be NULL.
- title: Title of the dialog.
- patterns: A list of semi-colon separated patterns to match. You should always include the pattern "*.*" as usually the MIME type and not the file pattern is relevant. If no file patterns are supported by the native dialog, this parameter is ignored.
- mode: 0, or a combination of the flags below.

Possible flags for the 'mode' parameter are:

- ALLEGRO_FILECHOOSER_FILE_MUST_EXIST: If supported by the native dialog, it will not allow entering new names, but just allow existing files to be selected. Else it is ignored.
- ALLEGRO_FILECHOOSER_SAVE: If the native dialog system has a different dialog for saving (for example one which allows creating new directories), it is used. Else ignored.

- ALLEGRO_FILECHOOSER_FOLDER: If there is support for a separate dialog to select a folder instead of a file, it will be used.
- ALLEGRO_FILECHOOSER_PICTURES: If a different dialog is available for selecting pictures, it is used. Else ignored.
- ALLEGRO_FILECHOOSER_SHOW_HIDDEN: If the platform supports it, also hidden files will be shown.
- ALLEGRO_FILECHOOSER_MULTIPLE: If supported, allow selecting multiple files.

Returns:

A handle to the dialog which you can pass to al_show_native_file_dialog (30.3) to display it, and from which you then can query the results. When you are done, call [al_destroy_native_file_dialog] on it.

30.3 al_show_native_file_dialog

```
void al_show_native_file_dialog(ALLEGRO_NATIVE_DIALOG *fd)
```

Show the dialog window.

This function blocks the calling thread until it returns, so you may want to spawn a thread with al_create_thread (20.4) and call it from inside that thread.

30.4 al_get_native_file_dialog_count

```
int al_get_native_file_dialog_count(const ALLEGRO_NATIVE_DIALOG *fc)
```

Returns the number of files selected, or 0 if the dialog was cancelled.

30.5 al_get_native_file_dialog_path

```
const ALLEGRO_PATH *al_get_native_file_dialog_path(
   const ALLEGRO_NATIVE_DIALOG *fc, size_t i)
```

Returns one of the selected paths.

30.6 al_destroy_native_dialog

```
void al_destroy_native_dialog(ALLEGRO_NATIVE_DIALOG *fd)
```

Frees up all resources used by the dialog.

30.7 al_show_native_message_box

```
int al_show_native_message_box(
    char const *title, char const *heading, char const *text,
    char const *buttons, int flags)
```

Show a native GUI message box. This can be used for example to display an error message if creation of an initial display fails. You should usually not use this function as long as an ALLEGRO_DISPLAY is active (but you may).

The message box will have a single "OK" button and use the style informative dialog boxes usually have on the native system. If the `buttons` parameter is not NULL, you can instead specify the button text in a string, with buttons separated by a vertical bar (|).

Flags	
ALLEGRO_MESSAGEBOX_WARN	The message is a warning. This may cause a different icon (or other effects).
ALLEGRO_MESSAGEBOX_ERROR	The message is an error.
ALLE-GRO_MESSAGEBOX_QUESTION	The message is a question.
ALLE-GRO_MESSAGEBOX_OK_CANCEL	Instead of the "OK" button also display a cancel button. Ignored if `buttons` is not NULL.
ALLEGRO_MESSAGEBOX_YES_NO	Instead of the "OK" button display Yes/No buttons. Ignored if `buttons` is not NULL.

Returns:

- 0 if the dialog window was closed without activating a button.
- 1 if the OK or Yes button was pressed.
- 2 if the Cancel or No button was pressed.

If `buttons` is not NULL, the number of the pressed button is returned, starting with 1.

Example:

```
  button = al_show_native_message_box("Fullscreen?",
     "Do you want to run this game in fullscreen mode?",
     "Never|Always|Not this time|Only this time",
     ALLEGRO_MESSAGEBOX_QUESTION);
  /* button is 1/2/3/4 if one of the buttons is pressed, 0 if the window
   * is closed.
   */
```

30.8 al_get_allegro_native_dialog_version

```
uint32_t al_get_allegro_native_dialog_version(void)
```

Returns the (compiled) version of the addon, in the same format as al_get_allegro_version (19.4).

31 PhysicsFS integration

PhysicsFS is a library to provide abstract access to various archives. See `http://icculus.org/physfs/` for more information.

This addon makes it possible to read and write files (on disk or inside archives) using PhysicsFS, through Allegro's file I/O API. For example, that means you can use the Image I/O addon to load images from .zip files.

You must set up PhysicsFS through its own API. When you want to open an ALLEGRO_FILE using PhysicsFS, first call al_set_physfs_file_interface (31.1), then al_fopen (6.4) or another function that calls al_fopen (6.4).

These functions are declared in the following header file. Link with allegro_physfs.

```
#include <allegro5/allegro_physfs.h>
```

31.1 al_set_physfs_file_interface

```
void al_set_physfs_file_interface(void)
```

After calling this, subsequent calls to al_fopen (6.4) will be handled by PHYSFS_open(). Operations on the files returned by al_fopen (6.4) will then be performed through PhysicsFS.

At the same time, all filesystem functions like al_read_directory (7.18.2) or al_create_fs_entry (7.3) will use PhysicsFS.

This functions only affects the thread it was called from.

To remember and restore another file I/O backend, you can use al_store_state (18.4)/al_restore_state (18.3).

See also: al_set_new_file_interface (6.29.1).

31.2 al_get_allegro_physfs_version

```
uint32_t al_get_allegro_physfs_version(void)
```

Returns the (compiled) version of the addon, in the same format as al_get_allegro_version (19.4).

32 Primitives addon

These functions are declared in the following header file. Link with allegro_primitives.

```
#include <allegro5/allegro_primitives.h>
```

32.1 General

32.1.1 al_get_allegro_primitives_version

```
uint32_t al_get_allegro_primitives_version(void)
```

Returns the (compiled) version of the addon, in the same format as al_get_allegro_version (19.4).

32.2 High level drawing routines

High level drawing routines encompass the most common usage of this addon: to draw geometric primitives, both smooth (variations on the circle theme) and piecewise linear. Outlined primitives support the concept of thickness with two distinct modes of output: hairline lines and think lines. Hairline lines are specifically designed to be exactly a pixel wide, and are commonly used for drawing outlined figures that need to be a pixel wide. Hairline thickness is designated as thickness less than or equal to 0. Unfortunately, the exact rasterization rules for drawing these hairline lines vary from one video card to another, and sometimes leave gaps where the lines meet. If that matters to you, then you should use thick lines. In many cases, having a thickness of 1 will produce 1 pixel wide lines that look better than hairline lines. Obviously, hairline lines cannot replicate thicknesses greater than 1. Thick lines grow symmetrically around the generating shape as thickness is increased.

32.2.1 al_draw_line

```
void al_draw_line(float x1, float y1, float x2, float y2,
   ALLEGRO_COLOR color, float thickness)
```

Draws a line segment between two points.

Parameters:

- x1, y1, x2, y2 - Start and end points of the line
- color - Color of the line
- thickness - Thickness of the line, pass `<= 0` to draw hairline lines

32.2.2 al_draw_triangle

```
void al_draw_triangle(float x1, float y1, float x2, float y2,
   float x3, float y3, ALLEGRO_COLOR color, float thickness)
```

Draws an outlined triangle.

Parameters:

- x1, y1, x2, y2, x3, y3 - Three points of the triangle
- color - Color of the triangle
- thickness - Thickness of the lines, pass `<= 0` to draw hairline lines

32.2.3 al_draw_filled_triangle

```
void al_draw_filled_triangle(float x1, float y1, float x2, float y2,
   float x3, float y3, ALLEGRO_COLOR color)
```

Draws a filled triangle.

Parameters:

- x1, y1, x2, y2, x3, y3 - Three points of the triangle
- color - Color of the triangle

32.2.4 al_draw_rectangle

```
void al_draw_rectangle(float x1, float y1, float x2, float y2,
   ALLEGRO_COLOR color, float thickness)
```

Draws an outlined rectangle.

Parameters:

- x1, y1, x2, y2 - Upper left and lower right points of the rectangle
- color - Color of the rectangle
- thickness - Thickness of the lines, pass `<= 0` to draw hairline lines

32.2.5 al_draw_filled_rectangle

```
void al_draw_filled_rectangle(float x1, float y1, float x2, float y2,
   ALLEGRO_COLOR color)
```

Draws a filled rectangle.

Parameters:

- x1, y1, x2, y2 - Upper left and lower right points of the rectangle
- color - Color of the rectangle

32.2.6 al_draw_rounded_rectangle

```
void al_draw_rounded_rectangle(float x1, float y1, float x2, float y2,
   float rx, float ry, ALLEGRO_COLOR color, float thickness)
```

Draws an outlined rounded rectangle.

Parameters:

- x1, y1, x2, y2 - Upper left and lower right points of the rectangle
- color - Color of the rectangle
- rx, ry - The radii of the round
- thickness - Thickness of the lines, pass `<= 0` to draw hairline lines

32.2.7 al_draw_filled_rounded_rectangle

```
void al_draw_filled_rounded_rectangle(float x1, float y1, float x2, float y2,
   float rx, float ry, ALLEGRO_COLOR color)
```

Draws an filled rounded rectangle.

Parameters:

- x1, y1, x2, y2 - Upper left and lower right points of the rectangle
- color - Color of the rectangle
- rx, ry - The radii of the round

32.2.8 al_calculate_arc

```
void al_calculate_arc(float* dest, int stride, float cx, float cy,
   float rx, float ry, float start_theta, float delta_theta, float thickness,
   int num_segments)
```

Calculates an elliptical arc, and sets the vertices in the destination buffer to the calculated positions. If `thickness <= 0`, then `num_points` of points are required in the destination, otherwise twice as many are needed. The destination buffer should consist of regularly spaced (by distance of `stride` bytes) doublets of floats, corresponding to x and y coordinates of the vertices.

Parameters:

- dest - The destination buffer
- stride - Distance (in bytes) between starts of successive pairs of coordinates
- cx, cy - Center of the arc
- rx, ry - Radii of the arc
- start_theta - The initial angle from which the arc is calculated
- delta_theta - Angular span of the arc (pass a negative number to switch direction)
- thickness - Thickness of the arc
- num_points - The number of points to calculate

32.2.9 al_draw_ellipse

```
void al_draw_ellipse(float cx, float cy, float rx, float ry,
   ALLEGRO_COLOR color, float thickness)
```

Draws an outlined ellipse.

Parameters:

- cx, cy - Center of the ellipse
- rx, ry - Radii of the ellipse
- color - Color of the ellipse
- thickness - Thickness of the ellipse, pass `<= 0` to draw a hairline ellipse

32.2.10 al_draw_filled_ellipse

```
void al_draw_filled_ellipse(float cx, float cy, float rx, float ry,
   ALLEGRO_COLOR color)
```

Draws a filled ellipse.

Parameters:

- cx, cy - Center of the ellipse
- rx, ry - Radii of the ellipse
- color - Color of the ellipse

32.2.11 al_draw_circle

```
void al_draw_circle(float cx, float cy, float r, ALLEGRO_COLOR color,
   float thickness)
```

Draws an outlined circle.

Parameters:

- cx, cy - Center of the circle
- r - Radius of the circle
- color - Color of the circle
- thickness - Thickness of the circle, pass `<= 0` to draw a hairline circle

32.2.12 al_draw_filled_circle

```
void al_draw_filled_circle(float cx, float cy, float r, ALLEGRO_COLOR color)
```

Draws a filled circle.

Parameters:

- cx, cy - Center of the circle
- r - Radius of the circle
- color - Color of the circle

32.2.13 al_draw_arc

```
void al_draw_arc(float cx, float cy, float r, float start_theta,
   float delta_theta, ALLEGRO_COLOR color, float thickness)
```

Draws an arc.

Parameters:

- cx, cy - Center of the arc
- r - Radius of the arc
- color - Color of the arc
- start_theta - The initial angle from which the arc is calculated
- delta_theta - Angular span of the arc (pass a negative number to switch direction)
- thickness - Thickness of the circle, pass `<= 0` to draw hairline circle

32.2.14 al_calculate_spline

```
void al_calculate_spline(float* dest, int stride, float points[8],
   float thickness, int num_segments)
```

Calculates a Bzier spline given 4 control points. If `thickness <= 0`, then `num_segments` of points are required in the destination, otherwise twice as many are needed. The destination buffer should consist of regularly spaced (by distance of stride bytes) doublets of floats, corresponding to x and y coordinates of the vertices.

Parameters:

- dest - The destination buffer
- stride - Distance (in bytes) between starts of successive pairs of coordinates
- points - An array of 4 pairs of coordinates of the 4 control points
- thickness - Thickness of the spline ribbon
- num_segments - The number of points to calculate

32.2.15 al_draw_spline

```
void al_draw_spline(float points[8], ALLEGRO_COLOR color, float thickness)
```

Draws a Bzier spline given 4 control points.

Parameters:

- points - An array of 4 pairs of coordinates of the 4 control points
- color - Color of the spline
- thickness - Thickness of the spline, pass `<= 0` to draw a hairline spline

32.2.16 al_calculate_ribbon

```
void al_calculate_ribbon(float* dest, int dest_stride, const float *points,
   int points_stride, float thickness, int num_segments)
```

Calculates a ribbon given an array of points. The ribbon will go through all of the passed points. If `thickness <= 0`, then `num_segments` of points are required in the destination buffer, otherwise twice as many are needed. The destination and the points buffer should consist of regularly spaced doublets of floats, corresponding to x and y coordinates of the vertices.

Parameters:

- dest - Pointer to the destination buffer
- dest_stride - Distance (in bytes) between starts of successive pairs of coordinates in the destination buffer
- points - An array of pairs of coordinates for each point
- points_stride - Distance (in bytes) between starts successive pairs of coordinates in the points buffer
- thickness - Thickness of the spline ribbon
- num_segments - The number of points to calculate

32.2.17 al_draw_ribbon

```
void al_draw_ribbon(const float *points, int points_stride, ALLEGRO_COLOR color,
   float thickness, int num_segments)
```

Draws a ribbon given given an array of points. The ribbon will go through all of the passed points.

Parameters:

- points - An array of pairs of coordinates for each point
- color - Color of the spline
- thickness - Thickness of the spline, pass `<= 0` to draw hairline spline

32.3 Low level drawing routines

Low level drawing routines allow for more advanced usage of the addon, allowing you to pass arbitrary sequences of vertices to draw to the screen. These routines also support using textures on the primitives with some restrictions. For maximum portability, you should only use textures that have dimensions that are a power of two, as not every videocard supports them completely. This warning is relaxed, however, if the texture coordinates never exit the boundaries of a single bitmap (i.e. you are not having the texture repeat/tile). As long as that is the case, any texture can be used safely. Sub-bitmaps work as textures, but cannot be tiled.

32.3.1 al_draw_prim

```
int al_draw_prim(const void* vtxs, const ALLEGRO_VERTEX_DECL* decl,
   ALLEGRO_BITMAP* texture, int start, int end, int type)
```

Draws a subset of the passed vertex buffer.

Parameters:

- texture - Texture to use, pass 0 to use only color shaded primitves
- vtxs - Pointer to an array of vertices
- decl - Pointer to a vertex declaration. If set to 0, the vtxs are assumed to be of the ALLEGRO_VERTEX type
- start, end - Start and end of the subset of the vertex buffer to draw
- type - Primitive type to draw

Returns: Number of primitives drawn

See Also: ALLEGRO_VERTEX (32.4.2), ALLEGRO_PRIM_TYPE (32.4.5), ALLEGRO_VERTEX_DECL (32.4.3), al_draw_indexed_prim (32.3.2)

32.3.2 al_draw_indexed_prim

```
int al_draw_indexed_prim(const void* vtxs, const ALLEGRO_VERTEX_DECL* decl,
   ALLEGRO_BITMAP* texture, const int* indices, int num_vtx, int type)
```

Draws a subset of the passed vertex buffer. This function uses an index array to specify which vertices to use.

Parameters:

- texture - Texture to use, pass 0 to use only shaded primitves
- vtxs - Pointer to an array of vertices
- decl - Pointer to a vertex declaration. If set to 0, the vtxs are assumed to be of the ALLEGRO_VERTEX type
- indices - An array of indices into the vertex buffer
- num_vtx - Number of indices from the indices array you want to draw
- type - Primitive type to draw

Returns: Number of primitives drawn

See Also: ALLEGRO_VERTEX (32.4.2), ALLEGRO_PRIM_TYPE (32.4.5), ALLEGRO_VERTEX_DECL (32.4.3), al_draw_prim (32.3.1)

32.3.3 al_get_allegro_color

```
ALLEGRO_COLOR al_get_allegro_color(ALLEGRO_PRIM_COLOR col)
```

Converts an `ALLEGRO_PRIM_COLOR` into a `ALLEGRO_COLOR`.

Parameters:

- col - ALLEGRO_PRIM_COLOR to convert

Returns: Converted ALLEGRO_COLOR

See Also: ALLEGRO_PRIM_COLOR (32.4.1), al_get_prim_color (32.3.4)

32.3.4 al_get_prim_color

```
ALLEGRO_PRIM_COLOR al_get_prim_color(ALLEGRO_COLOR col)
```

Converts an `ALLEGRO_COLOR` into a `ALLEGRO_PRIM_COLOR`.

Parameters:

- col - ALLEGRO_COLOR to convert

Returns: Converted ALLEGRO_PRIM_COLOR

See Also: ALLEGRO_PRIM_COLOR (32.4.1), al_get_allegro_color (32.3.3)

32.3.5 al_create_vertex_decl

```
ALLEGRO_VERTEX_DECL* al_create_vertex_decl(const ALLEGRO_VERTEX_ELEMENT* elements, int stride)
```

Creates a vertex declaration, which describes a custom vertex format.

Parameters:

- elements - An array of ALLEGRO_VERTEX_ELEMENT structures.
- stride - Size of the custom vertex structure

Returns: Newly created vertex declaration.

See Also: ALLEGRO_VERTEX_ELEMENT (32.4.4), ALLEGRO_VERTEX_DECL (32.4.3), al_destroy_vertex_decl (32.3.6)

32.3.6 al_destroy_vertex_decl

```
void al_destroy_vertex_decl(ALLEGRO_VERTEX_DECL* decl)
```

Destroys a vertex declaration.

Parameters:

- decl - Vertex declaration to destroy

See Also: ALLEGRO_VERTEX_ELEMENT (32.4.4), ALLEGRO_VERTEX_DECL (32.4.3), al_create_vertex_decl (32.3.5)

32.3.7 al_draw_soft_triangle

```
void al_draw_soft_triangle(
   ALLEGRO_VERTEX* v1, ALLEGRO_VERTEX* v2, ALLEGRO_VERTEX* v3, uintptr_t state,
   void (*init)(uintptr_t, ALLEGRO_VERTEX*, ALLEGRO_VERTEX*, ALLEGRO_VERTEX*),
   void (*first)(uintptr_t, int, int, int, int),
   void (*step)(uintptr_t, int),
   void (*draw)(uintptr_t, int, int, int))
```

Draws a triangle using the software rasterizer and user supplied pixel functions. For help in understanding what these functions do, see the implementation of the various shading routines in addons/primitives/tri_soft.c. The triangle is drawn in two segments, from top to bottom. The segments are deliniated by the vertically middle vertex of the triangle. One of each segment may be absent if two vertices are horizontally collinear.

Parameters:

- v1, v2, v3 - The three vertices of the triangle
- state - A pointer to a user supplied struct, this struct will be passed to all the pixel functions
- init - Called once per call before any drawing is done. The three points passed to it may be altered by clipping.
- first - Called twice per call, once per triangle segment. It is passed 4 parameters, the first two are the coordinates of the initial pixel drawn in the segment. The second two are the left minor and the left major steps, respectively. They represent the sizes of two steps taken by the rasterizer as it walks on the left side of the triangle. From then on, the each step will either be classified as a minor or a major step, corresponding to the above values.
- step - Called once per scanline. The last parameter is set to 1 if the step is a minor step, and 0 if it is a major step.
- draw - Called once per scanline. The function is expected to draw the scanline starting with a point specified by the first two parameters (corresponding to x and y values) going to the right until it reaches the value of the third parameter (the x value of the end point). All coordinates are inclusive.

32.3.8 al_draw_soft_line

```
void al_draw_soft_line(ALLEGRO_VERTEX* v1, ALLEGRO_VERTEX* v2, uintptr_t state,
   void (*first)(uintptr_t, int, int, ALLEGRO_VERTEX*, ALLEGRO_VERTEX*),
   void (*step)(uintptr_t, int),
   void (*draw)(uintptr_t, int, int))
```

Draws a line using the software rasterizer and user supplied pixel functions. For help in understanding what these functions do, see the implementation of the various shading routines in addons/primitives/line_soft.c. The line is drawn top to bottom.

Parameters:

- v1, v2 - The two vertices of the line
- state - A pointer to a user supplied struct, this struct will be passed to all the pixel functions

- first - Called before drawing the first pixel of the line. It is passed the coordinates of this pixel, as well as the two vertices above. The passed vertices may have been altered by clipping.
- step - Called once per pixel. The second parameter is set to 1 if the step is a minor step, and 0 if this step is a major step. Minor steps are taken only either in x or y directions. Major steps are taken in both directions diagonally. In all cases, the the absolute value of the change in coordinate is at most 1 in either direction.
- draw - Called once per pixel. The function is expected to draw the pixel at the coordinates passed to it.

32.4 Structures and types

32.4.1 ALLEGRO_PRIM_COLOR

```
typedef uint32_t ALLEGRO_PRIM_COLOR;
```

A special structure that defines a color in a way that understandable to both OpenGL and Direct3D backends. You should never access internal fields, instead using the two conversion functions to convert between it and ALLEGRO_COLOR (9.1.1) structure.

See Also: al_get_allegro_color (32.3.3), al_get_prim_color (32.3.4)

32.4.2 ALLEGRO_VERTEX

```
typedef struct ALLEGRO_VERTEX ALLEGRO_VERTEX;
```

Defines the generic vertex type, with a 3D position, color and texture coordinates for a single texture. Note that at this time, the software driver for this addon cannot render 3D primitives. If you want a 2D only primitive, set z to 0.

Fields:

- x, y, z - Position of the vertex
- color - `ALLEGRO_PRIM_COLOR` structure
- u, v - Texture coordinates measured in pixels

See Also: ALLEGRO_PRIM_COLOR (32.4.1), ALLEGRO_PRIM_ATTR (32.4.6)

32.4.3 ALLEGRO_VERTEX_DECL

```
typedef struct ALLEGRO_VERTEX_DECL ALLEGRO_VERTEX_DECL;
```

A vertex declaration. This opaque structure is responsible for describing the format and layout of a user defined custom vertex. It is created and destroyed by specialized functions.

See Also: al_create_vertex_decl (32.3.5), al_destroy_vertex_decl (32.3.6), ALLEGRO_VERTEX_ELEMENT (32.4.4)

32.4.4 ALLEGRO_VERTEX_ELEMENT

```
typedef struct ALLEGRO_VERTEX_ELEMENT ALLEGRO_VERTEX_ELEMENT;
```

A small structure describing a certain element of a vertex. E.g. the position of the vertex, or its color. These structures are used by the al_create_vertex_decl function to create the vertex declaration. For that they generally occur in an array. The last element of such an array should have the attribute field equal to 0, to signify that it is the end of the array. Here is an example code that would create a declaration describing the ALLEGRO_VERTEX structure:

```
ALLEGRO_VERTEX_ELEMENT elems[] = {
   {ALLEGRO_PRIM_POSITION, ALLEGRO_PRIM_FLOAT_3, offsetof(ALLEGRO_VERTEX, x)},
   {ALLEGRO_PRIM_TEX_COORD_PIXEL, ALLEGRO_PRIM_FLOAT_2, offsetof(ALLEGRO_VERTEX, u)},
   {ALLEGRO_PRIM_COLOR_ATTR, 0, offsetof(CUSTOM_VERTEX, color)},
   {0, 0, 0}
};
ALLEGRO_VERTEX_DECL* decl = al_create_vertex_decl(elems, sizeof(ALLEGRO_VERTEX));
```

Fields:

- attribute - A member of the ALLEGRO_PRIM_ATTR enumeration, specifying what this attribute signifies
- storage - A member of the ALLEGRO_PRIM_STORAGE enumeration, specifying how this attribute is stored
- offset - Offset in bytes from the beginning of the custom vertex structure. C function offsetof is very useful here.

See Also: al_create_vertex_decl (32.3.5), ALLEGRO_VERTEX_DECL (32.4.3), ALLEGRO_PRIM_STORAGE (32.4.7)

32.4.5 ALLEGRO_PRIM_TYPE

```
typedef enum ALLEGRO_PRIM_TYPE
```

Enumerates the types of primitives this addon can draw.

- ALLEGRO_PRIM_POINT_LIST - A list of points, each vertex defines a point
- ALLEGRO_PRIM_LINE_LIST - A list of lines, sequential pairs of vertices define disjointed lines
- ALLEGRO_PRIM_LINE_STRIP - A strip of lines, sequential vertices define a strip of lines
- ALLEGRO_PRIM_LINE_LOOP - Like a line strip, except at the end the first and the last vertices are also connected by a line
- ALLEGRO_PRIM_TRIANGLE_LIST - A list of triangles, sequential triplets of vertices define disjointed triangles
- ALLEGRO_PRIM_TRIANGLE_STRIP - A strip of triangles, sequential vertices define a strip of triangles
- ALLEGRO_PRIM_TRIANGLE_FAN - A fan of triangles, all triangles share the first vertex

32.4.6 ALLEGRO_PRIM_ATTR

```
typedef enum ALLEGRO_PRIM_ATTR
```

Enumerates the types of vertex attributes that a custom vertex may have.

- ALLEGRO_PRIM_POSITION - Position information, can be stored in any supported fashion
- ALLEGRO_PRIM_COLOR_ATTR - Color information, stored in an ALLEGRO_PRIM_COLOR. The storage field of ALLEGRO_VERTEX_ELEMENT is ignored
- ALLEGRO_PRIM_TEX_COORD - Texture coordinate information, can be stored only in ALLEGRO_PRIM_FLOAT_2 and ALLEGRO_PRIM_SHORT_2. These coordinates are normalized by the width and height of the texture, meaning that the bottom-right corner has texture coordinates of (1, 1).
- ALLEGRO_PRIM_TEX_COORD_PIXEL - Texture coordinate information, can be stored only in ALLEGRO_PRIM_FLOAT_2 and ALLEGRO_PRIM_SHORT_2. These coordinates are measured in pixels.

A note about pixel coordinates. In OpenGL the texture coordinate (0, 0) refers to the top left corner of the pixel. This confuses some drivers, because due to rounding errors the actual pixel sampled might be the pixel to the top and/or left of the (0, 0) pixel. To make this error less likely it is advisable to offset the texture coordinates you pass to the al_draw_prim by (0.5, 0.5) if you need precise pixel control. E.g. to refer to pixel (5, 10) you'd set the u and v to 5.5 and 10.5 respectively.

See Also: ALLEGRO_VERTEX_DECL (32.4.3), ALLEGRO_PRIM_STORAGE (32.4.7)

32.4.7 ALLEGRO_PRIM_STORAGE

```
typedef enum ALLEGRO_PRIM_STORAGE
```

Enumerates the types of storage an attribute of a custom vertex may be stored in.

- ALLEGRO_PRIM_FLOAT_2 - A doublet of floats
- ALLEGRO_PRIM_FLOAT_3 - A triplet of floats
- ALLEGRO_PRIM_SHORT_2 - A doublet of shorts

See Also: ALLEGRO_PRIM_ATTR (32.4.6)

32.4.8 ALLEGRO_VERTEX_CACHE_SIZE

```
#define ALLEGRO_VERTEX_CACHE_SIZE 256
```

Defines the size of the transformation vertex cache for the software renderer. If you pass less than this many vertices to the primitive rendering functions you will get a speed boost. This also defines the size of the cache vertex buffer, used for the high-level primitives. This corresponds to the maximum number of line segments that will be used to form them.

32.4.9 ALLEGRO_PRIM_QUALITY

```
#define ALLEGRO_PRIM_QUALITY 10
```

Defines the quality of the quadratic primitives. At 10, this roughly corresponds to error of less than half of a pixel.