The Allegro 5 Library

Reference Manual

Contents

0.1 Getting started guide

0.1.1 Introduction

Welcome to Allegro 5.0!

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 Be aware that, being on the wiki, it may be a little out of date, but the changes should be minor.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.

0.1.2 Structure of the library and its addons

Allegro 5.0 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
allegro_ttf -> allegro_font -> allegro

allegro_audio -> allegro
allegro_acodec -> 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.

0.1.3 The main function

For the purposes of cross-platform compatibility Allegro puts some requirements on your main function. First, you must include the core header (`allegro5/allegro.h`) in the same file as your main function. Second, if your main function is inside a C++ file, then it must have this signature: `int main(int argc, char **argv)`. Third, if you're using C/C++ then you need to link with the allegro_main addon when building your program.

0.1.4 Initialisation

Before using Allegro you must call al_init. Some addons have their own initialisation, e.g. al_init_image_addon, al_init_font_addon, al_init_ttf_addon.

To receive input, you need to initialise some subsystems like al_install_keyboard, al_install_mouse, al_install_joystick.

0.1.5 Opening a window

al_create_display will open a window and return an ALLEGRO_DISPLAY.

To clear the display, call al_clear_to_color. Use al_map_rgba or al_map_rgba_f to obtain an ALLEGRO_COLOR parameter.

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

0.1.6 Display an image

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

Use al_draw_bitmap, al_draw_scaled_bitmap or al_draw_scaled_rotated_bitmap to draw the image to the backbuffer. Remember to call al_flip_display.

0.1.7 Changing the drawing target

Notice that al_clear_to_color and al_draw_bitmap 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.

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

Other bitmaps can be created with al_create_bitmap, with options which can be adjusted with al_set_new_bitmap_flags and al_set_new_bitmap_format.

0.1.8 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, then tell input sources to place new events into that queue using al_register_event_source. The usual input event sources can be retrieved with al_get_keyboard_event_source, al_get_mouse_event_source and al_get_joystick_event_source.

Events can be retrieved with al_wait_for_event or al_get_next_event. Check the event type and other fields of ALLEGRO_EVENT 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 before creating the display, then register the display with an event queue. When you get a resize event, call al_acknowledge_resize.

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_create_timer.

al_get_time and al_rest are more direct ways to deal with time.

0.1.9 Displaying some text

To display some text, initialise the image and font addons with al_init_image_addon and al_init_font_addon, then load a bitmap font with al_load_font. Use al_draw_text or al_draw_textf.

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

0.1.10 Drawing primitives

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

0.1.11 Blending

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

As with al_set_target_bitmap, 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 and al_restore_state provide a convenient way to do that.

0.1.12 Sound

Use al_install_audio to initialize sound. To load any sample formats, you will need to initialise the acodec addon with al_init_acodec_addon.

After that, you can simply use al_reserve_samples and pass the number of sound effects typically playing at the same time. Then load your sound effects with al_load_sample and play them with al_play_sample. To stream large pieces of music from disk, you can use al_load_audio_stream 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 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 (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 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 or play complete sounds using an ALLEGRO_SAMPLE_INSTANCE. The latter simply points to an ALLEGRO_SAMPLE and will stream it for you.

0.1.13 Not the end

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

Enjoy!

0.2 Configuration files

These functions are declared in the main Allegro header file:

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

Allegro supports reading and writing of configuration files with a simple, INI file-like format.

A configuration file consists of key-value pairs separated by newlines. Keys are separated from values by an equals sign (=). All whitespace before the key, after the value and immediately adjacent to the equals sign is ignored. Keys and values may have whitespace characters within them. Keys do not need to be unique, but all but the last one are ignored.

The hash (#) character is used a comment when it is the first non-whitespace character on the line. All characters following that character are ignored to the end of the line. The hash character anywhere else on the line has no special significance.

Key-value pairs can be optionally grouped into sections, which are declared by surrounding a section name with square brackets ([and]) on a single line. Whitespace before the opening bracket is ignored. All characters after the trailing bracket are also ignored.

All key-value pairs that follow a section declaration belong to the last declared section. Key-value pairs that don't follow any section declarations belong to the global section. Sections do not nest.

Here is an example configuration file:

```
# Monster description
monster name = Allegro Developer

[weapon 0]
damage = 443

[weapon 1]
damage = 503
```

It can then be accessed like this (make sure to check for errors in an actual program):

```
ALLEGRO_CONFIG* cfg = al_load_config_file("test.cfg");
printf("%s\n", al_get_config_value(cfg, "", "monster name")); /* Prints: Allegro Developer */
printf("%s\n", al_get_config_value(cfg, "weapon 0", "damage")); /* Prints: 443 */
printf("%s\n", al_get_config_value(cfg, "weapon 1", "damage")); /* Prints: 503 */
al_destroy_config(cfg);
```

0.2.1 ALLEGRO_CONFIG

```
typedef struct ALLEGRO_CONFIG ALLEGRO_CONFIG;
```

An abstract configuration structure.

0.2.2 al_create_config

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

Create an empty configuration structure.

See also: al_load_config_file, al_destroy_config

0.2.3 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, al_load_config_file

0.2.4 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.

See also: al_load_config_file_f, al_save_config_file

0.2.5 al_load_config_file_f

```
ALLEGRO_CONFIG *al_load_config_file_f(ALLEGRO_FILE *file)
```

Read a configuration file from an already open file.

Returns NULL on error. The configuration structure should be destroyed with al_destroy_config. The file remains open afterwards.

See also: al_load_config_file

0.2.6 al_save_config_file

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

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

See also: al_save_config_file_f, al_load_config_file

0.2.7 al_save_config_file_f

```
bool al_save_config_file_f(ALLEGRO_FILE *file, const ALLEGRO_CONFIG *config)
```

Write out a configuration file to an already open file.

Returns true on success, false on error. The file remains open afterwards.

See also: al_save_config_file

0.2.8 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.

0.2.9 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

0.2.10 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

0.2.11 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

0.2.12 al_get_first_config_section

```
char const *al_get_first_config_section(ALLEGRO_CONFIG const *config,
   ALLEGRO_CONFIG_SECTION **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 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

0.2.13 al_get_next_config_section

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

Returns the name of the next section in the given config file or NULL if there are no more sections. The `iterator` must have been obtained with al_get_first_config_section first.

See also: al_get_first_config_section

0.2.14 al_get_first_config_entry

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

Returns the name of the first key in the given section in the given config or NULL if the section is empty. The `iterator` works like the one for al_get_first_config_section.

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_entry

0.2.15 al_get_next_config_entry

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

Returns the next key for the iterator obtained by al_get_first_config_entry. The `iterator` works like the one for al_get_next_config_section.

0.2.16 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

0.2.17 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

0.3 Display routines

These functions are declared in the main Allegro header file:

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

0.3.1 Display creation

ALLEGRO_DISPLAY

```
typedef struct ALLEGRO_DISPLAY ALLEGRO_DISPLAY;
```

An opaque type representing an open display or window.

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.

Each display has a distinct OpenGL rendering context associated with it. See al_set_target_bitmap for the discussion about rendering contexts.

See also: al_set_new_display_flags, al_set_new_display_option, al_set_new_display_refresh_rate, al_set_new_display_adapter, al_set_window_position

al_destroy_display

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

Destroy a display.

If the target bitmap of the calling thread is tied to the display, then it implies a call to "al_set_target_bitmap(NULL);" before the display is destroyed.

That special case notwithstanding, you should make sure no threads are currently targeting a bitmap which is tied to the display before you destroy it.

See also: al_set_target_bitmap

al_get_new_display_flags

```
int al_get_new_display_flags(void)
```

Get the display flags to be used when creating new displays on the calling thread.

See also: al_set_new_display_flags, al_set_display_flag

al_get_new_display_refresh_rate

```
int al_get_new_display_refresh_rate(void)
```

Get the requested refresh rate to be used when creating new displays on the calling thread.

See also: al_set_new_display_refresh_rate

al_get_new_window_position

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

Get the position where new non-fullscreen displays created by the calling thread will be placed.

See also: al_set_new_window_position

al_set_new_display_option

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

Set an extra display option, to be used when creating new displays on the calling thread. Display options differ from display flags, and specify some details of the context to be created within the window itself. These mainly have no effect on Allegro 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_COLOR_SIZE
This can be used to ask for a specific bit depth. For example to force a 16-bit framebuffer set this to 16.

ALLEGRO_RED_SIZE, ALLEGRO_GREEN_SIZE, ALLEGRO_BLUE_SIZE, ALLEGRO_ALPHA_SIZE
Individual color component size in bits.

ALLEGRO_RED_SHIFT, ALLEGRO_GREEN_SHIFT, ALLEGRO_BLUE_SHIFT, ALLEGRO_ALPHA_SHIFT
Together with the previous settings these can be used to specify the exact pixel layout the display should use. Normally there is no reason to use these.

ALLEGRO_ACC_RED_SIZE, ALLEGRO_ACC_GREEN_SIZE, ALLEGRO_ACC_BLUE_SIZE, ALLEGRO_ACC_ALPHA_SIZE
This 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 display is capable of updating just a region, and 0 if calling al_update_display_region is equivalent to al_flip_display.

ALLEGRO_VSYNC
: Set to 1 to tell the driver to wait for vsync in al_flip_display, or to 2 to force vsync off. The default of 0 means that Allegro does not try to modify the vsync behavior so it may be on or off. Note that even in the case of 1 or 2 it is possible to override the vsync behavior in the graphics driver so you should not rely on it.

ALLEGRO_MAX_BITMAP_SIZE
: When queried this returns the maximum size (width as well as height) a bitmap can have for this display. Calls to al_create_bitmap or al_load_bitmap for bitmaps larger than this size will fail. It does not apply to memory bitmaps which always can have arbitrary size (but are slow for drawing).

ALLEGRO_SUPPORT_NPOT_BITMAP
: Set to 1 if textures used for bitmaps on this display can have a size which is not a power of two. This is mostly useful if you use Allegro to load textures as otherwise only power-of-two textures will be used internally as bitmap storage.

ALLEGRO_CAN_DRAW_INTO_BITMAP
: Set to 1 if you can use al_set_target_bitmap on bitmaps of this display to draw into them. If this is not the case software emulation will be used when drawing into display bitmaps (which can be very slow).

ALLEGRO_SUPPORT_SEPARATE_ALPHA
: This is set to 1 if the al_set_separate_blender function is supported. Otherwise the alpha parameters will be ignored.

See also: al_set_new_display_flags

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.

al_reset_new_display_options

```
void al_reset_new_display_options(void)
```

This undoes any previous call to al_set_new_display_option on the calling thread.

al_set_new_display_flags

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

Sets various flags to be used when creating new displays on the calling thread. flags is a bitfield containing any reasonable combination of the following:

ALLEGRO_WINDOWED
: Prefer a windowed mode.

  Under multi-head X (not XRandR/TwinView), the use of more than one adapter is impossible due to bugs in X and GLX. al_create_display will fail if more than one adapter is attempted to be used.

ALLEGRO_FULLSCREEN
: Prefer a fullscreen mode.

  Under X the use of more than one FULLSCREEN display when using multi-head X, or true Xinerama is not possible due to bugs in X and GLX, display creation will fail if more than one adapter is attempted to be used.

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.

  Additionally under X, the use of more than one adapter in multi-head mode or with true Xinerama enabled is impossible due to bugs in X/GLX, creation will fail if more than one adapter is attempted to be used.

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
: If this flag is set, the OpenGL context created with ALLEGRO_OPENGL_3_0 will be forward compatible *only*, meaning that all of the OpenGL API declared deprecated in OpenGL 3.0 will not be supported. Currently, a display created with this flag will *not* be compatible with Allegro drawing routines; the display 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_FRAMELESS
: Try to create a window without a frame (i.e. no border or titlebar). This usually does nothing for fullscreen modes, and even in windowed modes it depends on the underlying platform whether it is supported or not. Since: 5.1.2

ALLEGRO_NOFRAME
: Original name for ALLEGRO_FRAMELESS. This works with older versions of Allegro.

ALLEGRO_GENERATE_EXPOSE_EVENTS
Let the display generate expose events.

0 can be used for default values.

See also: al_set_new_display_option, al_get_display_option

al_set_new_display_refresh_rate

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

Sets the refresh rate to use when creating new displays on the calling thread. If the refresh rate is not available, al_create_display will fail. A list of modes with refresh rates can be found with al_get_num_display_modes and al_get_display_mode.

The default setting is zero (don't care).

See also: al_get_new_display_refresh_rate

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, for displays created by the calling thread. Negative values allowed on some multihead systems.

To reset to the default behaviour, pass (INT_MAX, INT_MAX).

See also: al_get_new_window_position

0.3.2 Display operations

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, ALLEGRO_EVENT

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 buffer bitmap remain valid and retain their semantics as the back buffer, although the 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 will wait for vsync depending on the settings set in the system's graphics preferences.

See also: al_set_new_display_flags, al_set_new_display_option

al_get_backbuffer

```
ALLEGRO_BITMAP *al_get_backbuffer(ALLEGRO_DISPLAY *display)
```

Return a special bitmap representing the back-buffer of the 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). Only untransformed operations are hardware accelerated. This consists of al_draw_bitmap and al_draw_bitmap_region 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 and al_draw_bitmap_region), and then use that temporary bitmap as the source bitmap.

al_get_display_flags

```
int al_get_display_flags(ALLEGRO_DISPLAY *display)
```

Gets the flags of the display.

In addition to the flags set for the display at creation time with al_set_new_display_flags it can also have the ALLEGRO_MINIMIZED flag set, indicating that the window is currently minimized. This flag is very platform-dependent as even a minimized application may still render a preview version so normally you should not care whether it is minimized or not.

See also: al_set_new_display_flags

al_get_display_format

```
int al_get_display_format(ALLEGRO_DISPLAY *display)
```

Gets the pixel format of the display.

See also: ALLEGRO_PIXEL_FORMAT

al_get_display_height

```
int al_get_display_height(ALLEGRO_DISPLAY *display)
```

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

See also: al_get_display_width

al_get_display_refresh_rate

```
int al_get_display_refresh_rate(ALLEGRO_DISPLAY *display)
```

Gets the refresh rate of the display.

See also: al_set_new_display_refresh_rate

al_get_display_width

```
int al_get_display_width(ALLEGRO_DISPLAY *display)
```

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

See also: al_get_display_height

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

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.

al_resize_display

```
bool al_resize_display(ALLEGRO_DISPLAY *display, int width, int height)
```

Resize the 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

al_set_display_icon

```
void al_set_display_icon(ALLEGRO_DISPLAY *display, ALLEGRO_BITMAP *icon)
```

Changes the icon associated with the display (window).

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

See also: al_set_window_title

al_get_display_option

```
int al_get_display_option(ALLEGRO_DISPLAY *display, int option)
```

Return an extra display setting of the display.

See also: al_set_new_display_option

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

al_set_window_title

```
void al_set_window_title(ALLEGRO_DISPLAY *display, const char *title)
```

Set the title on a display.

See also: al_set_display_icon

al_set_display_flag

```
bool al_set_display_flag(ALLEGRO_DISPLAY *display, int flag, bool onoff)
```

Enable or disable one of the display flags. The flags are the same as for al_set_new_display_flags. The only flags that can be changed after creation are:

- ALLEGRO_FULLSCREEN_WINDOW
- ALLEGRO_FRAMELESS

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

Since: 5.1.2

See also: al_set_new_display_flags, al_get_display_flags

al_toggle_display_flag

```
bool al_toggle_display_flag(ALLEGRO_DISPLAY *display, int flag, bool onoff)
```

Deprecated synonym for al_set_display_flag.

al_update_display_region

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

Does the same as al_flip_display, 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) will specify the behavior of this function in the display.

See also: al_flip_display, al_get_display_option

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 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

al_get_display_event_source

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

Retrieve the associated event source.

0.3.3 Fullscreen display modes

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;
```

The `refresh_rate` may be zero if unknown.

See also: al_get_display_mode

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 and al_get_display_mode. 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, al_get_num_display_modes

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, and al_set_new_display_flags 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

0.3.4 Monitors

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

al_get_new_display_adapter

```
int al_get_new_display_adapter(void)
```

Gets the video adapter index where new displays will be created by the calling thread, if previously set with al_set_new_display_adapter. Otherwise returns `ALLEGRO_DEFAULT_DISPLAY_ADAPTER`.

See also: al_set_new_display_adapter

al_set_new_display_adapter

```
void al_set_new_display_adapter(int adapter)
```

Sets the adapter to use for new displays created by the calling thread. The adapter has a monitor attached to it. Information about the monitor can be gotten using al_get_num_video_adapters and al_get_monitor_info.

To return to the default behaviour, pass `ALLEGRO_DEFAULT_DISPLAY_ADAPTER`.

See also: al_get_num_video_adapters, al_get_monitor_info

al_get_monitor_info

```
bool 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.

Returns true on success, false on failure.

See also: ALLEGRO_MONITOR_INFO, al_get_num_video_adapters

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

0.4 Event system and events

These functions are declared in the main Allegro header file:

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

0.4.1 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:

type (ALLEGRO_EVENT_TYPE)
Indicates the type of event.

any.source (ALLEGRO_EVENT_SOURCE *)
The event source which generated the event.

any.timestamp (double)
When the event was generated.

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_get_time.

Each event is of one of the following types, with the usable fields given.

ALLEGRO_EVENT_JOYSTICK_AXIS

A joystick axis value changed.

joystick.id (ALLEGRO_JOYSTICK *)
: The joystick which generated the event. This is not the same as the event source `joystick.source`.

joystick.stick (int)
: The stick number, counting from zero. Axes on a joystick are grouped into “sticks”.

joystick.axis (int)
: The axis number on the stick, counting from zero.

joystick.pos (float)
: The axis position, from -1.0 to +1.0.

ALLEGRO_EVENT_JOYSTICK_BUTTON_DOWN

A joystick button was pressed.

joystick.id (ALLEGRO_JOYSTICK *)
: The joystick which generated the event.

joystick.button (int)
: The button which was pressed, counting from zero.

ALLEGRO_EVENT_JOYSTICK_BUTTON_UP

A joystick button was released.

joystick.id (ALLEGRO_JOYSTICK *)
: The joystick which generated the event.

joystick.button (int)
: The button which was released, counting from zero.

ALLEGRO_EVENT_JOYSTICK_CONFIGURATION

A joystick was plugged in or unplugged. See al_reconfigure_joysticks for details.

ALLEGRO_EVENT_KEY_DOWN

A keyboard key was pressed.

keyboard.keycode (int)
: The code corresponding to the physical key which was pressed. See the “Key codes” section for the list of ALLEGRO_KEY_* constants.

keyboard.display (ALLEGRO_DISPLAY *)
: The display which had keyboard focus when the event occurred.

> *Note:* this event is about the physical keys being press on the keyboard. Look for ALLEGRO_EVENT_KEY_CHAR events for character input.

ALLEGRO_EVENT_KEY_UP

A keyboard key was released.

keyboard.keycode (int)
: The code corresponding to the physical key which was released. See the “Key codes” section for the list of ALLEGRO_KEY_* constants.

keyboard.display (ALLEGRO_DISPLAY *)
: The display which had keyboard focus when the event occurred.

ALLEGRO_EVENT_KEY_CHAR

A character was typed on the keyboard, or a character was auto-repeated.

keyboard.keycode (int)
: The code corresponding to the physical key which was last pressed. See the “Key codes” section for the list of ALLEGRO_KEY_* constants.

keyboard.unichar (int)
: A Unicode code point (character). This *may* be zero or negative if the event was generated for a non-visible “character”, such as an arrow or Function key. In that case you can act upon the `keycode` field.

  Some special keys will set the `unichar` field to their standard ASCII values: Tab=9, Return=13, Escape=27. In addition if you press the Control key together with A to Z the `unichar` field will have the values 1 to 26. For example Ctrl-A will set `unichar` to 1 and Ctrl-H will set it to 8.

  As of Allegro 5.0.2 there are some inconsistencies in the treatment of Backspace (8 or 127) and Delete (127 or 0) keys on different platforms. These can be worked around by checking the `keycode` field.

keyboard.modifiers (unsigned)
: This is a bitfield of the modifier keys which were pressed when the event occurred. See “Keyboard modifier flags” for the constants.

keyboard.repeat (bool)
: Indicates if this is a repeated character.

keyboard.display (ALLEGRO_DISPLAY *)
: The display which had keyboard focus when the event occurred.

Note: in many input methods, characters are *not* entered one-for-one with physical key presses. Multiple key presses can combine to generate a single character, e.g. apostrophe + e may produce ‘é’. Fewer key presses can also generate more characters, e.g. macro sequences expanding to common phrases.

ALLEGRO_EVENT_MOUSE_AXES

One or more mouse axis values changed.

mouse.x (int)
: x-coordinate

mouse.y (int)
: y-coordinate

mouse.z (int)
z-coordinate. This usually means the vertical axis of a mouse wheel, where up is positive and down is negative.

mouse.w (int)
w-coordinate. This usually means the horizontal axis of a mouse wheel.

mouse.dx (int)
Change in the x-coordinate value since the previous ALLEGRO_EVENT_MOUSE_AXES event.

mouse.dy (int)
Change in the y-coordinate value since the previous ALLEGRO_EVENT_MOUSE_AXES event.

mouse.dz (int)
Change in the z-coordinate value since the previous ALLEGRO_EVENT_MOUSE_AXES event.

mouse.dw (int)
Change in the w-coordinate value since the previous ALLEGRO_EVENT_MOUSE_AXES event.

mouse.display (ALLEGRO_DISPLAY *)
The display which had mouse focus.

> *Note:* Calling al_set_mouse_xy 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.

ALLEGRO_EVENT_MOUSE_BUTTON_DOWN

A mouse button was pressed.

mouse.x (int)
x-coordinate

mouse.y (int)
y-coordinate

mouse.z (int)
z-coordinate

mouse.w (int)
w-coordinate

mouse.button (unsigned)
The mouse button which was pressed, numbering from 1.

mouse.display (ALLEGRO_DISPLAY *)
The display which had mouse focus.

ALLEGRO_EVENT_MOUSE_BUTTON_UP

A mouse button was released.

mouse.x (int)
x-coordinate

mouse.y (int)
y-coordinate

mouse.z (int)
: z-coordinate

mouse.w (int)
: w-coordinate

mouse.button (unsigned)
: The mouse button which was released, numbering from 1.

mouse.display (ALLEGRO_DISPLAY *)
: The display which had mouse focus.

ALLEGRO_EVENT_MOUSE_WARPED

al_set_mouse_xy 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.

mouse.x (int)
: x-coordinate

mouse.y (int)
: y-coordinate

mouse.z (int)
: z-coordinate

mouse.w (int)
: w-coordinate

mouse.display (ALLEGRO_DISPLAY *)
: The display which had mouse focus.

ALLEGRO_EVENT_MOUSE_LEAVE_DISPLAY

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

mouse.x (int)
: x-coordinate

mouse.y (int)
: y-coordinate

mouse.z (int)
: z-coordinate

mouse.w (int)
: w-coordinate

mouse.display (ALLEGRO_DISPLAY *)
: The display which had mouse focus.

ALLEGRO_EVENT_TIMER

A timer counter incremented.

timer.source (ALLEGRO_TIMER *)
The timer which generated the event.

timer.count (int64_t)
The timer count value.

ALLEGRO_EVENT_DISPLAY_EXPOSE

The display (or a portion thereof) has become visible.

display.source (ALLEGRO_DISPLAY *)
The display which was exposed.

display.x (int)

display.y (int)

The top-left corner of the display which was exposed.

display.width (int)

display.height (int)
The width and height of the rectangle which was exposed.

Note: The display needs to be created with ALLEGRO_GENERATE_EXPOSE_EVENTS flag for these events to be generated.

ALLEGRO_EVENT_DISPLAY_RESIZE

The window has been resized.

display.source (ALLEGRO_DISPLAY *)
The display which was resized.

display.x (int)

display.y (int)
The position of the top-level corner of the display.

display.width (int)
The new width of the display.

display.height (int)
The new height of the display.

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.

display.source (ALLEGRO_DISPLAY *)
The display which was closed.

ALLEGRO_EVENT_DISPLAY_LOST

When using Direct3D, displays can enter a "lost" state. In that state, drawing calls are ignored, and upon entering the state, bitmap's pixel data can become undefined. Allegro does its best to preserve the correct contents of bitmaps (see ALLEGRO_NO_PRESERVE_TEXTURE) and restore them when the device is "found" (see ALLEGRO_EVENT_DISPLAY_FOUND). However, this is not 100% fool proof.

To ensure that all bitmap contents are restored accurately, one must take additional steps. The best procedure to follow if bitmap constancy is important to you is as follows: first, always have the ALLEGRO_NO_PRESERVE_TEXTURE flag set to true when creating bitmaps, as it incurs pointless overhead when using this method. Second, create a mechanism in your game for easily reloading all of your bitmaps – for example, wrap them in a class or data structure and have a "bitmap manager" that can reload them back to the desired state. Then, when you receive an ALLEGRO_EVENT_DISPLAY_FOUND event, tell the bitmap manager (or whatever your mechanism is) to restore your bitmaps.

display.source (ALLEGRO_DISPLAY *)
The display which was lost.

ALLEGRO_EVENT_DISPLAY_FOUND

Generated when a lost device is restored to operating state. See ALLEGRO_EVENT_DISPLAY_LOST.

display.source (ALLEGRO_DISPLAY *)
The display which was found.

ALLEGRO_EVENT_DISPLAY_SWITCH_OUT

The window is no longer active, that is the user might have clicked into another window or "tabbed" away.

display.source (ALLEGRO_DISPLAY *)
The display which was switched out of.

ALLEGRO_EVENT_DISPLAY_SWITCH_IN

The window is the active one again.

display.source (ALLEGRO_DISPLAY *)
The display which was switched into.

ALLEGRO_EVENT_DISPLAY_ORIENTATION

Generated when the rotation or orientation of a display changes.

display.source (ALLEGRO_DISPLAY *)
: The display which generated the event.

event.display.orientation
: Contains one of the following values:

  - ALLEGRO_DISPLAY_ORIENTATION_0_DEGREES
  - ALLEGRO_DISPLAY_ORIENTATION_90_DEGREES
  - ALLEGRO_DISPLAY_ORIENTATION_180_DEGREES
  - ALLEGRO_DISPLAY_ORIENTATION_270_DEGREES
  - ALLEGRO_DISPLAY_ORIENTATION_FACE_UP
  - ALLEGRO_DISPLAY_ORIENTATION_FACE_DOWN

See also: ALLEGRO_EVENT_SOURCE, ALLEGRO_EVENT_TYPE, ALLEGRO_USER_EVENT, ALLEGRO_GET_EVENT_TYPE

0.4.2 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;

Like all other event types this structure is a part of the ALLEGRO_EVENT union. To access the fields in an ALLEGRO_EVENT variable ev, you would use:

- ev.user.source
- ev.user.data1
- ev.user.data2
- ev.user.data3
- ev.user.data4

To create a new user event you would do this:

```
ALLEGRO_EVENT_SOURCE my_event_source;
ALLEGRO_EVENT my_event;
float some_var;

al_init_user_event_source(&my_event_source);

my_event.user.type = ALLEGRO_GET_EVENT_TYPE('M','I','N','E');
my_event.user.data1 = 1;
my_event.user.data2 = &some_var;

al_emit_user_event(&my_event_source, &my_event, NULL);
```

Event type identifiers for user events are assigned by the user. Please see the documentation for ALLEGRO_GET_EVENT_TYPE for the rules you should follow when assigning identifiers.

See also: al_emit_user_event, ALLEGRO_GET_EVENT_TYPE

0.4.3 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, al_destroy_event_queue

0.4.4 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.

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

See also: ALLEGRO_EVENT, al_init_user_event_source, al_emit_user_event

0.4.5 ALLEGRO_EVENT_TYPE

```
typedef unsigned int ALLEGRO_EVENT_TYPE;
```

An integer used to distinguish between different types of events.

See also: ALLEGRO_EVENT, ALLEGRO_GET_EVENT_TYPE, ALLEGRO_EVENT_TYPE_IS_USER

0.4.6 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, but not necessarily, this will be made from four 8-bit character codes, for example:

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

IDs less than 1024 are reserved for Allegro or its addons. Don't use anything lower than `ALLEGRO_GET_EVENT_TYPE(0, 0, 4, 0)`.

You should try to make your IDs unique so they don't clash with any 3rd party code you may be using. Be creative. Numbering from 1024 is not creative.

If you need multiple identifiers, you could define them like this:

```
#define BASE_EVENT   ALLEGRO_GET_EVENT_TYPE('M','I','N','E')
#define BARK_EVENT   (BASE_EVENT + 0)
#define MEOW_EVENT   (BASE_EVENT + 1)
#define SQUAWK_EVENT (BASE_EVENT + 2)

/* Alternatively */
enum {
   BARK_EVENT = ALLEGRO_GET_EVENT_TYPE('M','I','N','E'),
   MEOW_EVENT,
   SQUAWK_EVENT
};
```

See also: ALLEGRO_EVENT, ALLEGRO_USER_EVENT, ALLEGRO_EVENT_TYPE_IS_USER

0.4.7 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.

0.4.8 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: al_register_event_source, al_destroy_event_queue, ALLEGRO_EVENT_QUEUE

0.4.9 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: al_create_event_queue, ALLEGRO_EVENT_QUEUE

0.4.10 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: al_unregister_event_source, ALLEGRO_EVENT_SOURCE

0.4.11 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.

See also: al_register_event_source

0.4.12 al_is_event_queue_empty

```
bool al_is_event_queue_empty(ALLEGRO_EVENT_QUEUE *queue)
```

Return true if the event queue specified is currently empty.

See also: al_get_next_event, al_peek_next_event

0.4.13 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, al_peek_next_event, al_wait_for_event

0.4.14 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, al_get_next_event, al_drop_next_event

0.4.15 al_drop_next_event

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

Drop (remove) the next event from the queue. If the queue is empty, nothing happens. Returns true if an event was dropped.

See also: al_flush_event_queue, al_is_event_queue_empty

0.4.16 al_flush_event_queue

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

Drops all events, if any, from the queue.

See also: al_drop_next_event, al_is_event_queue_empty

0.4.17 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, al_wait_for_event_timed, al_wait_for_event_until, al_get_next_event

0.4.18 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, al_wait_for_event, al_wait_for_event_until

0.4.19 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, ALLEGRO_TIMEOUT, al_init_timeout, al_wait_for_event, al_wait_for_event_timed

0.4.20 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.

The user event source will never be destroyed automatically. You must destroy it manually with al_destroy_user_event_source.

See also: ALLEGRO_EVENT_SOURCE, al_destroy_user_event_source, al_emit_user_event, ALLEGRO_USER_EVENT

0.4.21 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.

This does not free the memory, as that was user allocated to begin with.

See also: ALLEGRO_EVENT_SOURCE

0.4.22 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. Returns `false` if the event source isn't registered with any queues, hence the event wouldn't have been delivered into any queues.

Events are *copied* in and out of event queues, so after this function returns the memory pointed to by `event` may be freed or reused. Some fields of the event being passed in may be modified by the function.

Reference counting will be performed if `dtor` is not NULL. Whenever a copy of the event is made, the reference count increases. You need to call al_unref_user_event to decrease the reference count once you are done with a user event that you have received from al_get_next_event, al_peek_next_event, al_wait_for_event, etc.

Once 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, but *not* the event itself (since it is just a copy).

If dtor is NULL then reference counting will not be performed. It is safe, but unnecessary, to call al_unref_user_event on non-reference counted user events.

See also: ALLEGRO_USER_EVENT, al_unref_user_event

0.4.23 al_unref_user_event

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

Decrease the reference count of a user-defined event. This must be called on any user event that you get from al_get_next_event, al_peek_next_event, al_wait_for_event, etc. which is reference counted. This function does nothing if the event is not reference counted.

See also: al_emit_user_event, ALLEGRO_USER_EVENT

0.4.24 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

0.4.25 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

0.5 File I/O

These functions are declared in the main Allegro header file:

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

0.5.1 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.

0.5.2 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.

The fields are:

```
void*        (*fi_fopen)(const char *path, const char *mode);
void         (*fi_fclose)(ALLEGRO_FILE *f);
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);
void         (*fi_fclearerr)(ALLEGRO_FILE *f);
int          (*fi_fungetc)(ALLEGRO_FILE *f, int c);
off_t        (*fi_fsize)(ALLEGRO_FILE *f);
```

The fi_open function must allocate memory for whatever userdata structure it needs. The pointer to that memory must be returned; it will then be associated with the file. The other functions can access that data by calling al_get_file_userdata on the file handle. If fi_open returns NULL then al_fopen will also return NULL.

The fi_fclose function must clean up and free the userdata, but Allegro will free the ALLEGRO_FILE handle.

If fi_fungetc is NULL, then Allegro's default implementation of a 16 char long buffer will be used.

0.5.3 ALLEGRO_SEEK

```
typedef enum ALLEGRO_SEEK
```

- ALLEGRO_SEEK_SET - seek relative to beginning of file
- ALLEGRO_SEEK_CUR - seek relative to current file position
- ALLEGRO_SEEK_END - seek relative to end of file

See also: al_fseek

0.5.4 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.

Parameters:

- path - path to the file to open
- mode - access mode to open the file 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 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".

Returns a file handle on success, or NULL on error.

See also: al_set_new_file_interface, al_fclose.

0.5.5 al_fopen_interface

```
ALLEGRO_FILE *al_fopen_interface(const ALLEGRO_FILE_INTERFACE *drv,
   const char *path, const char *mode)
```

Opens a file using the specified interface, instead of the interface set with al_set_new_file_interface.

See also: al_fopen

0.5.6 al_fopen_slice

```
ALLEGRO_FILE *al_fopen_slice(ALLEGRO_FILE *fp, size_t initial_size, const char *mode)
```

Opens a slice (subset) of an already open random access file as if it were a stand alone file. While the slice is open, the parent file handle must not be used in any way.

The slice is opened at the current location of the parent file, up through `initial_size` bytes. The `initial_size` may be any non-negative integer that will not exceed the bounds of the parent file.

Seeking with `ALLEGRO_SEEK_SET` will be relative to this starting location. `ALLEGRO_SEEK_END` will be relative to the starting location plus the size of the slice.

The mode can be any combination of:

- r: read access
- w: write access
- e: expandable

For example, a mode of "rw" indicates the file can be read and written. (Note that this is slightly different from the stdio modes.) Keep in mind that the parent file must support random access and be open in normal write mode (not append) for the slice to work in a well defined way.

If the slice is marked as expandable, then reads and writes can happen after the initial end point, and the slice will grow accordingly. Otherwise, all activity is restricted to the initial size of the slice.

A slice must be closed with al_fclose. The parent file will then be positioned immediately after the end of the slice.

Since: 5.0.6, 5.1.0

See also: al_fopen

0.5.7 al_fclose

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

Close the given file, writing any buffered output data (if any).

0.5.8 al_fread

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

Read 'size' bytes into the buffer pointed to by 'ptr', from the given file.

Returns the number of bytes actually read. If an error occurs, or the end-of-file is reached, the return value is a short byte count (or zero).

al_fread() does not distinguish between EOF and other errors. Use al_feof and al_ferror to determine which occurred.

See also: al_fgetc, al_fread16be, al_fread16le, al_fread32be, al_fread32le

0.5.9 al_fwrite

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

Write 'size' bytes from the buffer pointed to by 'ptr' into the given file.

Returns the number of bytes actually written. If an error occurs, the return value is a short byte count (or zero).

See also: al_fputc, al_fputs, al_fwrite16be, al_fwrite16le, al_fwrite32be, al_fwrite32le

0.5.10 al_fflush

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

Flush any pending writes to the given file.

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

See also: al_get_errno

0.5.11 al_ftell

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

Returns the current position in the given 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_fseek, al_get_errno

0.5.12 al_fseek

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

Set the current position of the given file to a position relative to that specified by 'whence', plus 'offset' number of bytes.

'whence' can be:

- ALLEGRO_SEEK_SET - seek relative to beginning of file
- ALLEGRO_SEEK_CUR - seek relative to current file position
- ALLEGRO_SEEK_END - seek relative to end of file

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

After a successful seek, the end-of-file indicator is cleared and all pushback bytes are forgotten.

On some platforms this function may not support large files.

See also: al_ftell, al_get_errno

0.5.13 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, al_fclearerr

0.5.14 al_ferror

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

Returns true if the error indicator is set on the given file, i.e. there was some sort of previous error.

See also: al_feof, al_fclearerr

0.5.15 al_fclearerr

```
void al_fclearerr(ALLEGRO_FILE *f)
```

Clear the error indicator for the given file.

The standard I/O backend also clears the end-of-file indicator, and other backends *should* try to do this. However, they may not if it would require too much effort (e.g. PhysicsFS backend), so your code should not rely on it if you need your code to be portable to other backends.

See also: al_ferror, al_feof

0.5.16 al_fungetc

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

Ungets a single byte from a file. Pushed-back bytes are not written to the file, only made available for subsequent reads, in reverse order.

The number of pushbacks depends on the backend. The standard I/O backend only guarantees a single pushback; this depends on the libc implementation.

For backends that follow the standard behavior, the pushback buffer will be cleared after any seeking or writing; also calls to al_fseek and al_ftell are relative to the number of pushbacks. If a pushback causes the position to become negative, the behavior of al_fseek and al_ftell are undefined.

See also: al_fgetc, al_get_errno

0.5.17 al_fsize

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

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

0.5.18 al_fgetc

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

Read and return next byte in the given file. Returns EOF on end of file or if an error occurred.

See also: al_fungetc

0.5.19 al_fputc

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

Write a single byte to the given file.

Parameters:

- c - byte value to write
- f - file to write to

Returns c on success, or EOF on error.

0.5.20 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 to check if the end of the file was reached prematurely, or al_ferror to check if an error occurred.

See also: al_fread16be

0.5.21 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 to check if the end of the file was reached prematurely, or al_ferror to check if an error occurred.

See also: al_fread16le

0.5.22 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

0.5.23 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

0.5.24 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 to check if the end of the file was reached prematurely, or al_ferror to check if an error occurred.

See also: al_fread32be

0.5.25 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 to check if the end of the file was reached prematurely, or al_ferror to check if an error occurred.

See also: al_fread32le

0.5.26 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

0.5.27 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

0.5.28 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 about translations of end-of-line characters.

See also: al_fget_ustr

0.5.29 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. Returns NULL if an error occurred or if the end of file was reached without reading any bytes.

See al_fopen about translations of end-of-line characters.

See also: al_fgetc, al_fgets

0.5.30 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, 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.

See also: al_fwrite

0.5.31 Standard I/O specific routines

al_fopen_fd

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

Create an ALLEGRO_FILE 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 so you should not call close() on it.

See also: al_fopen

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 on success, NULL on failure.

0.5.32 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 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 changes which function will handle the following al_fopen calls for the current thread.

al_set_new_file_interface

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

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

See also: al_set_standard_file_interface, al_store_state, al_restore_state.

al_set_standard_file_interface

```
void al_set_standard_file_interface(void)
```

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

See also: al_set_new_file_interface

al_get_new_file_interface

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

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

See also: al_store_state, al_restore_state.

al_create_file_handle

```
ALLEGRO_FILE *al_create_file_handle(const ALLEGRO_FILE_INTERFACE *drv,
   void *userdata)
```

Creates an empty, opened file handle with some abstract user data. This allows custom interfaces to extend the ALLEGRO_FILE struct with their own data. You should close the handle with the standard al_fclose function when you are finished with it.

See also: al_fopen, al_fclose, al_set_new_file_interface

al_get_file_userdata

```
void *al_get_file_userdata(ALLEGRO_FILE *f)
```

Returns a pointer to the custom userdata that is attached to the file handle. This is intended to be used by functions that extend ALLEGRO_FILE_INTERFACE.

0.6 File system routines

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).

0.6.1 ALLEGRO_FS_ENTRY

```
typedef struct ALLEGRO_FS_ENTRY ALLEGRO_FS_ENTRY;
```

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

0.6.2 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

0.6.3 al_create_fs_entry

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

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

0.6.4 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. If the entry was opened, it is closed before being destroyed.

Does nothing if passed NULL.

0.6.5 al_get_fs_entry_name

```
const char *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 note 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 and not the path.

On success returns a read only string which you must not modify or destroy. Returns NULL on failure.

0.6.6 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, al_get_fs_entry_atime, al_get_fs_entry_ctime, al_get_fs_entry_mode

0.6.7 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, ALLEGRO_FILE_MODE

0.6.8 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, al_get_fs_entry_mtime, al_update_fs_entry

0.6.9 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, al_get_fs_entry_mtime, al_update_fs_entry

0.6.10 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, al_get_fs_entry_ctime, al_update_fs_entry

0.6.11 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

0.6.12 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.

0.6.13 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

0.6.14 al_filename_exists

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

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

See also: al_fs_entry_exists

0.6.15 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, 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

0.6.16 Directory functions

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 on an entry and you must call al_close_directory when you no longer need it.

Returns true on success.

See also: al_read_directory, al_close_directory

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_destroy_fs_entry on the returned entry when you are done with it.

See also: al_open_directory, al_close_directory

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, al_read_directory

al_get_current_directory

```
char *al_get_current_directory(void)
```

Returns the path to the current working directory, or NULL on failure. The returned path is dynamically allocated and must be destroyed with al_free.

Allegro's errno is filled in to indicate the error if there is a failure. This function may not be implemented on some (virtual) filesystems.

See also: al_get_errno, al_free

al_change_directory

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

Changes the current working directory to 'path'.

Returns true on success, false on error.

al_make_directory

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

Creates a new directory on the filesystem. This function also creates any parent directories as needed.

Returns true on success (including if the directory already exists), otherwise returns false on error. Fills in Allegro's errno to indicate the error.

See also: al_get_errno

al_open_fs_entry

```
ALLEGRO_FILE *al_open_fs_entry(ALLEGRO_FS_ENTRY *e, const char *mode)
```

Open an ALLEGRO_FILE handle to a filesystem entry, for the given access mode. This is like calling al_fopen with the name of the filesystem entry, but uses the appropriate file interface, not whatever was set with the latest call to al_set_new_file_interface.

Returns the handle on success, NULL on error.

See also: al_fopen

0.6.17 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).

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 char *        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);
   char *              fs_get_current_directory(void);
   bool                fs_change_directory(const char *path);
   bool                fs_make_directory(const char *path);

   ALLEGRO_FILE *      fs_open_file(ALLEGRO_FS_ENTRY *e);
```

al_set_fs_interface

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

Set the ALLEGRO_FS_INTERFACE table for the calling thread.

See also: al_set_standard_fs_interface, al_store_state, al_restore_state.

al_set_standard_fs_interface

```
void al_set_standard_fs_interface(void)
```

Return the ALLEGRO_FS_INTERFACE table to the default, for the calling thread.

See also: al_set_fs_interface.

al_get_fs_interface

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

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

See also: al_store_state, al_restore_state.

0.7 Fixed point math routines

These functions are declared in the main Allegro header file:

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

0.7.1 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.

0.7.2 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, al_ftofix, al_fixtof.

0.7.3 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, al_ftofix, al_fixtof, al_fixfloor, al_fixceil.

0.7.4 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, al_fixceil.

0.7.5 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, al_fixfloor.

0.7.6 al_ftofix

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

Converts a floating point value to fixed point. Unlike al_itofix, 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, al_itofix, al_fixtoi, al_get_errno

0.7.7 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, al_itofix, al_fixtoi.

0.7.8 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.

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, al_fixsub, al_fixdiv, al_get_errno.

0.7.9 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.

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, al_fixsub, al_fixmul, al_get_errno.

0.7.10 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, al_fixmul, al_fixdiv.

0.7.11 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, al_fixmul, al_fixdiv, al_get_errno.

0.7.12 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.

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, al_radtofix_r.

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, al_fixtorad_r.

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.

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.

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.

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.

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.

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.

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.

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.

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.

0.8 Graphics routines

These functions are declared in the main Allegro header file:

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

0.8.1 Colors

ALLEGRO_COLOR

```
typedef struct ALLEGRO_COLOR ALLEGRO_COLOR;
```

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

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, al_map_rgba_f, al_map_rgb_f

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, al_map_rgb, al_map_rgba_f

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, al_map_rgba_f, al_map_rgb_f

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, al_map_rgb, al_map_rgb_f

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, al_unmap_rgba_f, al_unmap_rgb_f

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, al_unmap_rgb, al_unmap_rgba_f

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, al_unmap_rgba_f, al_unmap_rgb_f

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, al_unmap_rgb, al_unmap_rgb_f

0.8.2 Locking and pixel formats

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;
    int format;
    int pitch;
    int pixel_size;
} ALLEGRO_LOCKED_REGION;
```

- *data* points to the leftmost pixel of the first row (row 0) of the locked region.
- *format* indicates the pixel format of the data.
- *pitch* gives the size in bytes of a single row (also known as the stride). The pitch may be greater than `width * pixel_size` due to padding; this is not uncommon. It is also *not* uncommon for the pitch to be negative (the bitmap may be upside down).
- *pixel_size* is the number of bytes used to represent a single pixel.

See also: al_lock_bitmap, al_lock_bitmap_region, al_unlock_bitmap, ALLEGRO_PIXEL_FORMAT

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.

- 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_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_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.
- ALLEGRO_PIXEL_FORMAT_RGBA_4444 - 16bit

See also: al_set_new_bitmap_format, al_get_bitmap_format

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, al_get_pixel_format_bits

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, al_get_pixel_size

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. Use this flag if a partial number of pixels need to be written to, even if reading is not needed.

'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 and al_put_blended_pixel).

See also: ALLEGRO_LOCKED_REGION, ALLEGRO_PIXEL_FORMAT, al_unlock_bitmap

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, 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, ALLEGRO_PIXEL_FORMAT, al_unlock_bitmap

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, al_lock_bitmap_region

0.8.3 Bitmap creation

ALLEGRO_BITMAP

```
typedef struct ALLEGRO_BITMAP ALLEGRO_BITMAP;
```

Abstract type representing a bitmap (2D image).

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. Some platforms also dictate a minimum texture size, which is relevant if you plan to use this bitmap with the primitives addon. If you try to create a bitmap smaller than this, this call will not fail but the returned bitmap will be a section of a larger bitmap with the minimum size. This minimum size is 16 by 16.

Some platforms do not directly support display bitmaps whose dimensions are not powers of two. Allegro handles this by creating a larger bitmap that has dimensions that are powers of two and then returning a section of that bitmap with the dimensions you requested. This can be relevant if you plan to use this bitmap with the primitives addon but shouldn't be an issue otherwise.

See also: al_set_new_bitmap_format, al_set_new_bitmap_flags, al_clone_bitmap, al_create_sub_bitmap

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.

The sub-bitmap may originate off or extend past the parent bitmap.

See the discussion in al_get_backbuffer 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

al_clone_bitmap

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

Create a new bitmap with al_create_bitmap, and copy the pixel data from the old bitmap across.

See also: al_create_bitmap, al_set_new_bitmap_format, al_set_new_bitmap_flags

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.

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

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, al_set_new_bitmap_format

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
: Creates a bitmap that resides in the video card memory. These types of bitmaps receive the greatest benefit from hardware acceleration. al_set_new_bitmap_flags will implicitly set this flag unless ALLEGRO_MEMORY_BITMAP is present.

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.

  This is not yet honoured.

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 Direct3D may forget them. This can take extra processing time. If you know it doesn't matter if a bitmap keeps its pixel data, for example its a temporary buffer, use this flag to tell Allegro not to attempt to preserve its contents. This can increase performance of your game or application, but there is a catch. See ALLEGRO_EVENT_DISPLAY_LOST for further information.

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).

ALLEGRO_MIN_LINEAR
: When drawing a scaled down version of the bitmap, use linear filtering. This usually looks better. You can also combine it with the MIPMAP flag for even better quality.

ALLEGRO_MAG_LINEAR
: When drawing a magnified version of a bitmap, use linear filtering. This will cause the picture to get blurry instead of creating a big rectangle for each pixel. It depends on how you want things to look like whether you want to use this or not.

ALLEGRO_MIPMAP
: This can only be used for bitmaps whose width and height is a power of two. In that case, it will generate mipmaps and use them when drawing scaled down versions. For example if the bitmap is 64x64, then extra bitmaps of sizes 32x32, 16x16, 8x8, 4x4, 2x2 and 1x1 will be created always containing a scaled down version of the original.

ALLEGRO_NO_PREMULTIPLIED_ALPHA
: By default, Allegro pre-multiplies the alpha channel of an image with the images color data when it loads it. Typically that would look something like this:

```
r = get_float_byte();
g = get_float_byte();
b = get_float_byte();
a = get_float_byte();

r = r * a;
g = g * a;
b = b * a;

set_image_pixel(x, y, r, g, b, a);
```

The reason for this can be seen in the Allegro example ex_premulalpha, ie, using pre-multiplied alpha gives more accurate color results in some cases. To use alpha blending with images loaded with pre-multiplied alpha, you would use the default blending mode, which is set with al_set_blender(ALLEGRO_ADD, ALLEGRO_ONE, ALLEGRO_INVERSE_ALPHA).

The ALLEGRO_NO_PREMULTIPLIED_ALPHA flag being set will ensure that images are not loaded with alpha pre-multiplied, but are loaded with color values direct from the image. That looks like this:

```
r = get_float_byte();
g = get_float_byte();
b = get_float_byte();
a = get_float_byte();

set_image_pixel(x, y, r, g, b, a);
```

To draw such an image using regular alpha blending, you would use al_set_blender(ALLEGRO_ADD, ALLEGRO_ALPHA, ALLEGRO_INVERSE_ALPHA) to set the correct blender. This has some caveats. First, as mentioned above, drawing such an image can result in less accurate color blending (when drawing an image with linear filtering on, the edges will be darker than they should be). Second, the behaviour is somewhat confusing, which is explained in the example below.

```
// Load and create bitmaps with an alpha channel
al_set_new_bitmap_format(ALLEGRO_PIXEL_FORMAT_ANY_32_WITH_ALPHA);
// Load some bitmap with alpha in it
bmp = al_load_bitmap("some_alpha_bitmap.png");
// We will draw to this buffer and then draw this buffer to the screen
tmp_buffer = al_create_bitmap(SCREEN_W, SCREEN_H);
// Set the buffer as the target and clear it
al_set_target_bitmap(tmp_buffer);
al_clear_to_color(al_map_rgba_f(0, 0, 0, 1));
// Draw the bitmap to the temporary buffer
al_draw_bitmap(bmp, 0, 0, 0);
// Finally, draw the buffer to the screen
// The output will look incorrect (may take close inspection
// depending on the bitmap -- it may also be very obvious)
al_set_target_bitmap(al_get_backbuffer(display));
al_draw_bitmap(tmp_buffer, 0, 0, 0);
```

To explain further, if you have a pixel with 0.5 alpha, and you're using (ALLEGRO_ADD, ALLEGRO_ALPHA, ALLEGRO_INVERSE_ALPHA) for blending, the formula is:

```
a = da * dst + sa * src
```

Expands to:

```
    result_a = dst_a * (1-0.5) + 0.5 * 0.5;
```

So if you draw the image to the temporary buffer, it is blended once resulting in 0.75 alpha, then drawn again to the screen, blended in the same way, resulting in a pixel has 0.1875 as an alpha value.

See also: al_get_new_bitmap_flags, al_get_bitmap_flags

al_add_new_bitmap_flag

```
void al_add_new_bitmap_flag(int flag)
```

A convenience function which does the same as

```
al_set_new_bitmap_flags(al_get_new_bitmap_flags() | flag);
```

See also: al_set_new_bitmap_flags, al_get_new_bitmap_flags, al_get_bitmap_flags

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, al_get_new_bitmap_format, al_get_bitmap_format

0.8.4 Bitmap properties

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

al_get_bitmap_format

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

Returns the pixel format of a bitmap.

See also: ALLEGRO_PIXEL_FORMAT, al_set_new_bitmap_flags

al_get_bitmap_height

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

Returns the height of a bitmap in pixels.

al_get_bitmap_width

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

Returns the width of a bitmap in pixels.

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, al_put_pixel, al_lock_bitmap

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, al_lock_bitmap_region, al_unlock_bitmap

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 however is usually tied to a single ALLEGRO_DISPLAY. This function can be used to know if the bitmap is compatible with the given display, even if it is a different display to 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.

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, al_get_parent_bitmap

al_get_parent_bitmap

```
ALLEGRO_BITMAP *al_get_parent_bitmap(ALLEGRO_BITMAP *bitmap)
```

Returns the bitmap this bitmap is a sub-bitmap of. Returns NULL if this bitmap is not a sub-bitmap.

Since: 5.0.6, 5.1.2

See also: al_create_sub_bitmap, al_is_sub_bitmap

0.8.5 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.

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, al_set_clipping_rectangle

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).

`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

> *Note:* The current target bitmap must be a different bitmap. Drawing a bitmap to itself (or to a sub-bitmap of itself) or drawing a sub-bitmap to its parent (or another sub-bitmap of its parent) are not currently supported. To copy part of a bitmap into the same bitmap simply use a temporary bitmap instead.

> *Note:* The backbuffer (or a sub-bitmap thereof) can not be transformed, blended or tinted. If you need to draw the backbuffer draw it to a temporary bitmap first with no active transformation (except translation). Blending and tinting settings/parameters will be ignored. This does not apply when drawing into a memory bitmap.

See also: al_draw_bitmap_region, al_draw_scaled_bitmap, al_draw_rotated_bitmap, al_draw_scaled_rotated_bitmap

al_draw_tinted_bitmap

```
void al_draw_tinted_bitmap(ALLEGRO_BITMAP *bitmap, ALLEGRO_COLOR tint,
   float dx, float dy, int flags)
```

Like al_draw_bitmap but multiplies all colors in the bitmap with the given color. For example:

```
al_draw_tinted_bitmap(bitmap, al_map_rgba_f(1, 1, 1, 0.5), x, y, 0);
```

The above will draw the bitmap 50% transparently.

```
al_draw_tinted_bitmap(bitmap, al_map_rgba_f(1, 0, 0, 1), x, y, 0);
```

The above will only draw the red component of the bitmap.

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

See also: al_draw_bitmap, al_draw_scaled_bitmap, al_draw_rotated_bitmap, al_draw_scaled_rotated_bitmap

al_draw_tinted_bitmap_region

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

Like al_draw_bitmap_region but multiplies all colors in the bitmap with the given color.

See also: al_draw_tinted_bitmap

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, does blending and, unlike al_put_blended_pixel, respects the transformations. This function can be slow if called often; if you need to draw a lot of pixels consider using al_draw_prim with ALLEGRO_PRIM_POINT_LIST from the primitives addon.

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

> *Note:* This function may not draw exactly where you expect it to. See the pixel-precise output section on the primitives addon documentation for details on how to control exactly where the pixel is drawn.

See also: ALLEGRO_COLOR, al_put_pixel

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 relative to the upper left corner of the bitmap will be drawn at dx/dy and the bitmap is rotated around this point. If cx,cy is 0,0 the bitmap will rotate around its upper left corner.

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

Example

```
float w = al_get_bitmap_width(bitmap);
float h = al_get_bitmap_height(bitmap);
al_draw_rotated_bitmap(bitmap, w / 2, h / 2, x, y, ALLEGRO_PI / 2, 0);
```

The above code draws the bitmap centered on x/y and rotates it 90° clockwise.

See also: al_draw_bitmap, al_draw_bitmap_region, al_draw_scaled_bitmap, al_draw_scaled_rotated_bitmap

al_draw_tinted_rotated_bitmap

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

Like al_draw_rotated_bitmap but multiplies all colors in the bitmap with the given color.

See also: al_draw_tinted_bitmap

al_draw_scaled_rotated_bitmap

```
void al_draw_scaled_rotated_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, 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

See also: al_draw_bitmap, al_draw_bitmap_region, al_draw_scaled_bitmap, al_draw_rotated_bitmap

al_draw_tinted_scaled_rotated_bitmap

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

Like al_draw_scaled_rotated_bitmap but multiplies all colors in the bitmap with the given color.

See also: al_draw_tinted_bitmap

al_draw_tinted_scaled_rotated_bitmap_region

```
void al_draw_tinted_scaled_rotated_bitmap_region(ALLEGRO_BITMAP *bitmap,
   float sx, float sy, float sw, float sh,
   ALLEGRO_COLOR tint,
   float cx, float cy, float dx, float dy, float xscale, float yscale,
   float angle, int flags)
```

Like al_draw_tinted_scaled_rotated_bitmap but you specify an area within the bitmap to be drawn.

You can get the same effect with a sub bitmap:

```
al_draw_tinted_scaled_rotated_bitmap(bitmap, sx, sy, sw, sh, tint,
    cx, cy, dx, dy, xscale, yscale, angle, flags);

/* This draws the same: */
sub_bitmap = al_create_sub_bitmap(bitmap, sx, sy, sw, sh);
al_draw_tinted_scaled_rotated_bitmap(sub_bitmap, tint, cx, cy,
    dx, dy, xscale, yscale, angle, flags);
```

Since: 5.0.6, 5.1.0

See also: al_draw_tinted_bitmap

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

See also: al_draw_bitmap, al_draw_bitmap_region, al_draw_rotated_bitmap, al_draw_scaled_rotated_bitmap,

al_draw_tinted_scaled_bitmap

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

Like al_draw_scaled_bitmap but multiplies all colors in the bitmap with the given color.

See also: al_draw_tinted_bitmap

al_get_target_bitmap

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

Return the target bitmap of the calling thread.

See also: al_set_target_bitmap

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 the transformations or the color blenders.

See also: ALLEGRO_COLOR, al_get_pixel, al_put_blended_pixel, al_lock_bitmap

al_put_blended_pixel

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

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

See also: ALLEGRO_COLOR, al_put_pixel

al_set_target_bitmap

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

This function selects the bitmap to which all subsequent drawing operations in the calling thread will draw to. To return to drawing to a display, set the backbuffer of the display as the target bitmap, using al_get_backbuffer. As a convenience, you may also use al_set_target_backbuffer.

Each video bitmap is tied to a display. When a video bitmap is set to as the target bitmap, the display that the bitmap belongs to is automatically made "current" for the calling thread (if it is not current already). Then drawing other bitmaps which are tied to the same display can be hardware accelerated.

A single display cannot be current for multiple threads simultaneously. If you need to release a display, so it is not current for the calling thread, call `al_set_target_bitmap(NULL);`

Setting a memory bitmap as the target bitmap will not change which display is current for the calling thread.

OpenGL note:

Framebuffer objects (FBOs) allow OpenGL to directly draw to a bitmap, which is very fast. When using an OpenGL display, if all of the following conditions are met an FBO will be created for use with the bitmap:

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

In Allegro 5.0.0, you had to be careful as an FBO would be kept around until the bitmap is destroyed or you explicitly called al_remove_opengl_fbo on the bitmap, wasting resources. In newer versions, FBOs will be freed automatically when the bitmap is no longer the target bitmap, *unless* you have called al_get_opengl_fbo to retrieve the FBO id.

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 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);
```

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

See also: al_get_target_bitmap, al_set_target_backbuffer

al_set_target_backbuffer

```
void al_set_target_backbuffer(ALLEGRO_DISPLAY *display)
```

Same as `al_set_target_bitmap(al_get_backbuffer(display));`

See also: al_set_target_bitmap, al_get_backbuffer

al_get_current_display

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

Return the display that is "current" for the calling thread, or NULL if there is none.

See also: al_set_target_bitmap

0.8.6 Blending modes

al_get_blender

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

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

See also: al_set_blender, al_get_separate_blender

al_get_separate_blender

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

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, al_get_blender

al_set_blender

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

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
  ```

Blending examples:

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

```
al_set_blender(ALLEGRO_ADD, ALLEGRO_ONE, ALLEGRO_INVERSE_ALPHA)
```

If you are using non-pre-multiplied alpha, you could use

```
al_set_blender(ALLEGRO_ADD, ALLEGRO_ALPHA, ALLEGRO_INVERSE_ALPHA)
```

Additive blending would be achieved with

```
al_set_blender(ALLEGRO_ADD, ALLEGRO_ONE, ALLEGRO_ONE)
```

Copying the source to the destination (including alpha) unmodified

```
al_set_blender(ALLEGRO_ADD, ALLEGRO_ONE, ALLEGRO_ZERO)
```

See also: al_set_separate_blender, al_get_blender

al_set_separate_blender

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

Like al_set_blender, but allows specifying a separate blending operation for the alpha channel. This is useful if your target bitmap also has an alpha channel and the two alpha channels need to be combined in a different way than the color components.

See also: al_set_blender, al_get_blender, al_get_separate_blender

0.8.7 Clipping

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

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, al_reset_clipping_rectangle

al_reset_clipping_rectangle

```
void al_reset_clipping_rectangle(void)
```

Equivalent to calling 'al_set_clipping_rectangle(0, 0, w, h)' where *w* and *h* are the width and height of the target bitmap respectively.

Does nothing if there is no target bitmap.

See also: al_set_clipping_rectangle

Since: 5.0.6, 5.1.0

0.8.8 Graphics utility functions

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

0.8.9 Deferred drawing

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. One exception to this rule are the transformations. It is possible to set a new transformation while the drawing is held.

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

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

0.8.10 Image I/O

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. 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.

See also: al_register_bitmap_saver, al_register_bitmap_loader_f

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. 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.

See also: al_register_bitmap_loader, al_register_bitmap_saver_f

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. 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.

See also: al_register_bitmap_loader

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. 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.

See also: al_register_bitmap_saver

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.

> *Note:* the core Allegro library does not support any image file formats by default. You must use the allegro_image addon, or register your own format handler.

See also: al_load_bitmap_f, al_register_bitmap_loader, al_set_new_bitmap_format, al_set_new_bitmap_flags, al_init_image_addon

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. The file remains open afterwards.

> *Note:* the core Allegro library does not support any image file formats by default. You must use the allegro_image addon, or register your own format handler.

See also: al_load_bitmap, al_register_bitmap_loader_f, al_init_image_addon

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.

> *Note:* the core Allegro library does not support any image file formats by default. You must use the allegro_image addon, or register your own format handler.

See also: al_save_bitmap_f, al_register_bitmap_saver, al_init_image_addon

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. The file remains open afterwards.

> *Note:* the core Allegro library does not support any image file formats by default. You must use the allegro_image addon, or register your own format handler.

See also: al_save_bitmap, al_register_bitmap_saver_f, al_init_image_addon

0.9 Joystick routines

These functions are declared in the main Allegro header file:

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

0.9.1 ALLEGRO_JOYSTICK

```
typedef struct ALLEGRO_JOYSTICK ALLEGRO_JOYSTICK;
```

This is an abstract data type representing a physical joystick.

See also: al_get_joystick

0.9.2 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

0.9.3 ALLEGRO_JOYFLAGS

```
enum ALLEGRO_JOYFLAGS
```

- 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

0.9.4 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

0.9.5 al_uninstall_joystick

```
void al_uninstall_joystick(void)
```

Uninstalls the active joystick driver. All outstanding ALLEGRO_JOYSTICK structures are invalidated. 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

0.9.6 al_is_joystick_installed

```
bool al_is_joystick_installed(void)
```

Returns true if al_install_joystick was called successfully.

0.9.7 al_reconfigure_joysticks

```
bool al_reconfigure_joysticks(void)
```

Allegro is able to cope with users connecting and disconnected joystick devices on-the-fly. On existing platforms, the joystick event source will generate an event of type `ALLEGRO_EVENT_JOYSTICK_CONFIGURATION` when a device is plugged in or unplugged. In response, you should call al_reconfigure_joysticks.

Afterwards, the number returned by al_get_num_joysticks may be different, and the handles returned by al_get_joystick may be different or be ordered differently.

All ALLEGRO_JOYSTICK handles remain valid, but handles for disconnected devices become inactive: their states will no longer update, and al_get_joystick will not return the handle. Handles for devices which remain connected will continue to represent the same devices. Previously inactive handles may become active again, being reused to represent newly connected devices.

Returns true if the joystick configuration changed, otherwise returns false.

It is possible that on some systems, Allegro won't be able to generate `ALLEGRO_EVENT_JOYSTICK_CONFIGURATION` events. If your game has an input configuration screen or similar, you may wish to call al_reconfigure_joysticks when entering that screen.

See also: al_get_joystick_event_source, ALLEGRO_EVENT

0.9.8 al_get_num_joysticks

```
int al_get_num_joysticks(void)
```

Return the number of joysticks currently on the system (or potentially on the system). This number can change after al_reconfigure_joysticks is called, in order to support hotplugging.

Returns 0 if there is no joystick driver installed.

See also: al_get_joystick, al_get_joystick_active

0.9.9 al_get_joystick

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

Get a handle for a joystick on the system. The number may be from 0 to al_get_num_joysticks-1. If successful a pointer to a joystick object is returned, which represents a physical device. Otherwise NULL is returned.

The handle and the index are only incidentally linked. After al_reconfigure_joysticks is called, al_get_joystick may return handles in a different order, and handles which represent disconnected devices will not be returned.

See also: al_get_num_joysticks, al_reconfigure_joysticks, al_get_joystick_active

0.9.10 al_release_joystick

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

This function currently does nothing.

See also: al_get_joystick

0.9.11 al_get_joystick_active

```
bool al_get_joystick_active(ALLEGRO_JOYSTICK *joy)
```

Return if the joystick handle is "active", i.e. in the current configuration, the handle represents some physical device plugged into the system. al_get_joystick returns active handles. After reconfiguration, active handles may become inactive, and vice versa.

See also: al_reconfigure_joysticks

0.9.12 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, al_get_joystick_axis_name, al_get_joystick_button_name

0.9.13 al_get_joystick_stick_name

```
const char *al_get_joystick_stick_name(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, al_get_joystick_num_sticks

0.9.14 al_get_joystick_axis_name

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

Return the name of the given axis. If the axis doesn't exist, NULL is returned. Indices begin from 0.

See also: al_get_joystick_stick_name, al_get_joystick_num_axes

0.9.15 al_get_joystick_button_name

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

Return the name of the given button. If the button doesn't exist, NULL is returned. Indices begin from 0.

See also: al_get_joystick_stick_name, al_get_joystick_axis_name, al_get_joystick_num_buttons

0.9.16 al_get_joystick_stick_flags

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

Return the flags of the given "stick". If the stick doesn't exist, NULL is returned. Indices begin from 0.

See also: ALLEGRO_JOYFLAGS

0.9.17 al_get_joystick_num_sticks

```
int al_get_joystick_num_sticks(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, al_get_joystick_num_buttons

0.9.18 al_get_joystick_num_axes

```
int al_get_joystick_num_axes(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

0.9.19 al_get_joystick_num_buttons

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

Return the number of buttons on the joystick.

See also: al_get_joystick_num_sticks

0.9.20 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, al_get_joystick_num_buttons, al_get_joystick_num_axes

0.9.21 al_get_joystick_event_source

```
ALLEGRO_EVENT_SOURCE *al_get_joystick_event_source(void)
```

Returns the global joystick event source. All joystick events are generated by this event source.

0.10 Keyboard routines

These functions are declared in the main Allegro header file:

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

0.10.1 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.

0.10.2 Key codes

The constant ALLEGRO_KEY_MAX is always one higher than the highest key code. So if you want to use the key code as array index you can do something like this:

```
bool pressed_keys[ALLEGRO_KEY_MAX];
...
pressed_keys[key_code] = true;
```

These are the list of key codes used by Allegro, which are returned in the event.keyboard.keycode field of the ALLEGRO_KEY_DOWN and ALLEGRO_KEY_UP events and which you can pass to al_key_down:

```
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_PAD_EQUALS,
ALLEGRO_KEY_BACKQUOTE,
ALLEGRO_KEY_SEMICOLON2,
ALLEGRO_KEY_COMMAND
```

0.10.3 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
```

The event field 'keyboard.modifiers' is a bitfield composed of these constants. These indicate the modifier keys which were pressed at the time a character was typed.

0.10.4 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, al_is_keyboard_installed

0.10.5 al_is_keyboard_installed

```
bool al_is_keyboard_installed(void)
```

Returns true if al_install_keyboard was called successfully.

0.10.6 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

0.10.7 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, ALLEGRO_KEYBOARD_STATE

0.10.8 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

0.10.9 al_keycode_to_name

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

Converts the given keycode to a description of the key.

0.10.10 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.

0.10.11 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.

0.11 Memory management routines

These functions are declared in the main Allegro header file:

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

0.11.1 al_malloc

```
#define al_malloc(n) \
   (al_malloc_with_context((n), __LINE__, __FILE__, __func__))
```

Like malloc() in the C standard library, but the implementation may be overridden.

This is a macro.

See also: al_free, al_realloc, al_calloc, al_malloc_with_context, al_set_memory_interface

0.11.2 al_free

```
#define al_free(p) \
   (al_free_with_context((p), __LINE__, __FILE__, __func__))
```

Like free() in the C standard library, but the implementation may be overridden.

Additionally, on Windows, a memory block allocated by one DLL must be freed from the same DLL. In the few places where an Allegro function returns a pointer that must be freed, you must use al_free for portability to Windows.

This is a macro.

See also: al_malloc, al_free_with_context

0.11.3 al_realloc

```
#define al_realloc(p, n) \
   (al_realloc_with_context((p), (n), __LINE__, __FILE__, __func__))
```

Like realloc() in the C standard library, but the implementation may be overridden.

This is a macro.

See also: al_malloc, al_realloc_with_context

0.11.4 al_calloc

```
#define al_calloc(c, n) \
   (al_calloc_with_context((c), (n), __LINE__, __FILE__, __func__))
```

Like calloc() in the C standard library, but the implementation may be overridden.

This is a macro.

See also: al_malloc, al_calloc_with_context

0.11.5 al_malloc_with_context

```
void *al_malloc_with_context(size_t n,
   int line, const char *file, const char *func)
```

This calls malloc() from the Allegro library (this matters on Windows), unless overriden with al_set_memory_interface,

Generally you should use the al_malloc macro.

0.11.6 al_free_with_context

```
void al_free_with_context(void *ptr,
   int line, const char *file, const char *func)
```

This calls free() from the Allegro library (this matters on Windows), unless overriden with al_set_memory_interface.

Generally you should use the al_free macro.

0.11.7 al_realloc_with_context

```
void *al_realloc_with_context(void *ptr, size_t n,
   int line, const char *file, const char *func)
```

This calls realloc() from the Allegro library (this matters on Windows), unless overriden with al_set_memory_interface,

Generally you should use the al_realloc macro.

0.11.8 al_calloc_with_context

```
void *al_calloc_with_context(size_t count, size_t n,
   int line, const char *file, const char *func)
```

This calls calloc() from the Allegro library (this matters on Windows), unless overriden with al_set_memory_interface,

Generally you should use the al_calloc macro.

0.11.9 ALLEGRO_MEMORY_INTERFACE

```
typedef struct ALLEGRO_MEMORY_INTERFACE ALLEGRO_MEMORY_INTERFACE;
```

This structure has the following fields.

```
void *(*mi_malloc)(size_t n, int line, const char *file, const char *func);
void (*mi_free)(void *ptr, int line, const char *file, const char *func);
void *(*mi_realloc)(void *ptr, size_t n, int line, const char *file,
                    const char *func);
void *(*mi_calloc)(size_t count, size_t n, int line, const char *file,
                   const char *func);
```

See also: al_set_memory_interface

0.11.10 al_set_memory_interface

```
void al_set_memory_interface(ALLEGRO_MEMORY_INTERFACE *memory_interface)
```

Override the memory management functions with implementations of al_malloc_with_context, al_free_with_context, al_realloc_with_context and al_calloc_with_context. The context arguments may be used for debugging.

If the pointer is NULL, the default behaviour will be restored.

See also: ALLEGRO_MEMORY_INTERFACE

0.12 Miscellaneous routines

These functions are declared in the main Allegro header file:

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

0.12.1 ALLEGRO_PI

```
#define ALLEGRO_PI        3.14159265358979323846
```

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

0.12.2 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. But some other system (like OSX) don't allow calling al_install_system 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.

0.13 Mouse routines

These functions are declared in the main Allegro header file:

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

0.13.1 ALLEGRO_MOUSE_STATE

```
typedef struct ALLEGRO_MOUSE_STATE ALLEGRO_MOUSE_STATE;
```

Public fields (read only):

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

The zeroth bit is set if the primary mouse button is held down, the first bit is set if the secondary mouse button is held down, and so on.

See also: al_get_mouse_state, al_get_mouse_state_axis, al_mouse_button_down

0.13.2 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.

0.13.3 al_is_mouse_installed

```
bool al_is_mouse_installed(void)
```

Returns true if al_install_mouse was called successfully.

0.13.4 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.

0.13.5 al_get_mouse_num_axes

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

Return the number of buttons on the mouse. The first axis is 0.

See also: al_get_mouse_num_buttons

0.13.6 al_get_mouse_num_buttons

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

Return the number of buttons on the mouse. The first button is 1.

See also: al_get_mouse_num_axes

0.13.7 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.

Example:

```
ALLEGRO_MOUSE_STATE state;

al_get_mouse_state(&state);
if (state.buttons & 1) {
    /* Primary (e.g. left) mouse button is held. */
    printf("Mouse position: (%d, %d)\n", state.x, state.y);
}
if (state.buttons & 2) {
    /* Secondary (e.g. right) mouse button is held. */
}
if (state.buttons & 4) {
    /* Tertiary (e.g. middle) mouse button is held. */
}
```

See also: ALLEGRO_MOUSE_STATE, al_get_mouse_state_axis, al_mouse_button_down

0.13.8 al_get_mouse_state_axis

```
int al_get_mouse_state_axis(const ALLEGRO_MOUSE_STATE *state, int axis)
```

Extract the mouse axis value from the saved state. The axes are numbered from 0, in this order: x-axis, y-axis, z-axis, w-axis.

See also: ALLEGRO_MOUSE_STATE, al_get_mouse_state, al_mouse_button_down

0.13.9 al_mouse_button_down

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

Return true if the mouse button specified was held down in the state specified. Unlike most things, the first mouse button is numbered 1.

See also: ALLEGRO_MOUSE_STATE, al_get_mouse_state, al_get_mouse_state_axis

0.13.10 al_set_mouse_xy

```
bool al_set_mouse_xy(ALLEGRO_DISPLAY *display, int x, int y)
```

Try to position the mouse at the given coordinates on the given 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, al_set_mouse_w

0.13.11 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

0.13.12 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

0.13.13 al_set_mouse_axis

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

Set the given mouse axis to the given value.

The axis number must not be 0 or 1, which are the X and Y axes. Use al_set_mouse_xy for that.

Returns true on success, false on failure.

See also: al_set_mouse_xy, al_set_mouse_z, al_set_mouse_w

0.13.14 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.

0.13.15 Mouse cursors

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.

Returns a pointer to the cursor on success, or NULL on failure.

See also: al_set_mouse_cursor, al_destroy_mouse_cursor

al_destroy_mouse_cursor

```
void al_destroy_mouse_cursor(ALLEGRO_MOUSE_CURSOR *cursor)
```

Free the memory used by the given cursor.

Has no effect if `cursor` is NULL.

See also: al_create_mouse_cursor

al_set_mouse_cursor

```
bool al_set_mouse_cursor(ALLEGRO_DISPLAY *display, ALLEGRO_MOUSE_CURSOR *cursor)
```

Set the given mouse cursor to be the current mouse cursor for the given display.

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, al_show_mouse_cursor, al_hide_mouse_cursor

al_set_system_mouse_cursor

```
bool al_set_system_mouse_cursor(ALLEGRO_DISPLAY *display,
   ALLEGRO_SYSTEM_MOUSE_CURSOR cursor_id)
```

Set the given system mouse cursor to be the current mouse cursor for the given 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, al_show_mouse_cursor, al_hide_mouse_cursor

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.

al_hide_mouse_cursor

```
bool al_hide_mouse_cursor(ALLEGRO_DISPLAY *display)
```

Hide the mouse cursor in the given display. 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

al_show_mouse_cursor

```
bool al_show_mouse_cursor(ALLEGRO_DISPLAY *display)
```

Make a mouse cursor visible in the given display.

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

al_grab_mouse

```
bool al_grab_mouse(ALLEGRO_DISPLAY *display)
```

Confine the mouse cursor to the given display. The mouse cursor can only be confined to one display at a time.

Returns true if successful, otherwise returns false. Do not assume that the cursor will remain confined until you call al_ungrab_mouse. It may lose the confined status at any time for other reasons.

> *Note:* not yet implemented on Mac OS X.

See also: al_ungrab_mouse

al_ungrab_mouse

```
bool al_ungrab_mouse(void)
```

Stop confining the mouse cursor to any display belonging to the program.

> *Note:* not yet implemented on Mac OS X.

See also: al_grab_mouse

0.14 Path structures

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.

0.14.1 al_create_path

```
ALLEGRO_PATH *al_create_path(const char *str)
```

Create a path structure from a string. The last component, if it is followed by a directory separator and is neither "." nor "..", is treated as the last directory name in the path. Otherwise the last component is treated as the filename. The string may be NULL for an empty path.

See also: al_create_path, al_destroy_path

0.14.2 al_create_path_for_directory

```
ALLEGRO_PATH *al_create_path_for_directory(const char *str)
```

This is the same as al_create_path, but interprets the passed string as a directory path. The filename component of the returned path will always be empty.

See also: al_create_path, al_destroy_path

0.14.3 al_destroy_path

```
void al_destroy_path(ALLEGRO_PATH *path)
```

Free a path structure. Does nothing if passed NULL.

See also: al_create_path, al_create_path_for_directory

0.14.4 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

0.14.5 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.

See also: al_rebase_path

0.14.6 al_rebase_path

```
bool al_rebase_path(const ALLEGRO_PATH *head, ALLEGRO_PATH *tail)
```

Concatenate two path structures, modifying the second path structure. If *tail* is an absolute path, this function does nothing. Otherwise, the drive and path components in *head* are inserted at the start of *tail*.

For example, if *head* is "/anchor/" and *tail* is "data/file.ext", then after the call *tail* becomes "/anchor/data/file.ext".

See also: al_join_paths

0.14.7 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.

0.14.8 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

0.14.9 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, al_get_path_tail

0.14.10 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.

0.14.11 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, al_get_path_extension, al_get_path_component

0.14.12 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, al_get_path_extension

0.14.13 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, al_get_path_basename

0.14.14 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

0.14.15 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

0.14.16 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, al_replace_path_component, al_remove_path_component

0.14.17 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, al_remove_path_component

0.14.18 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, al_replace_path_component, al_drop_path_tail

0.14.19 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

0.14.20 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, al_get_path_filename

0.14.21 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, al_get_path_extension

0.14.22 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.

0.14.23 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.

0.15 State

These functions are declared in the main Allegro header file:

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

0.15.1 ALLEGRO_STATE

```
typedef struct ALLEGRO_STATE ALLEGRO_STATE;
```

Opaque type which is passed to al_store_state/al_restore_state.

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 blending mode
    current display
        deferred drawing
    current target bitmap
        current transformation
        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.)

0.15.2 ALLEGRO_STATE_FLAGS

```
typedef enum ALLEGRO_STATE_FLAGS
```

Flags which can be passed to al_store_state/al_restore_state as bit combinations. See al_store_state for the list of flags.

0.15.3 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 object.

See also: al_store_state, ALLEGRO_STATE_FLAGS

0.15.4 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 objects. The flags parameter can take any bit-combination of these flags:

- 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

See also: al_restore_state, ALLEGRO_STATE

0.15.5 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.

0.15.6 al_set_errno

```
void al_set_errno(int errnum)
```

Set the error number for for the calling thread.

0.16 System routines

These functions are declared in the main Allegro header file:

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

0.16.1 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 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

0.16.2 al_init

```
#define al_init()    (al_install_system(ALLEGRO_VERSION_INT, atexit))
```

Like al_install_system, but automatically passes in the version and uses the atexit function visible in the current compilation unit.

See also: al_install_system

0.16.3 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 call, e.g. from atexit().

0.16.4 al_is_system_installed

```
bool al_is_system_installed(void)
```

Returns true if Allegro is initialized, otherwise returns false.

0.16.5 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.

0.16.6 al_get_standard_path

```
ALLEGRO_PATH *al_get_standard_path(int id)
```

Gets a system path, depending on the `id` parameter. Some of these paths may be affected by the organization and application name, so be sure to set those before calling this function.

The paths are not guaranteed to be unique (e.g., SETTINGS and DATA may be the same on some platforms), so you should be sure your filenames are unique if you need to avoid naming collisions. Also, a returned path may not actually exist on the file system.

ALLEGRO_RESOURCES_PATH
: If you bundle data in a location relative to your executable, then you should use this path to locate that data. On most platforms, this is the directory that contains the executable file.

  If ran from an OS X app bundle, then this will point to the internal resource directory (/Contents/Resources). To maintain consistency, if you put your resources into a directory called "data" beneath the executable on some other platform (like Windows), then you should also create a directory called "data" under the OS X app bundle's resource folder.

  You should not try to write to this path, as it is very likely read-only.

  If you install your resources in some other system directory (e.g., in /usr/share or C:\ProgramData), then you are responsible for keeping track of that yourself.

ALLEGRO_TEMP_PATH
: Path to the directory for temporary files.

ALLEGRO_USER_HOME_PATH
: This is the user's home directory. You should not normally write files into this directory directly, or create any sub folders in it, without explicit permission from the user. One practical application of this path would be to use it as the starting place of a file selector in a GUI.

ALLEGRO_USER_DOCUMENTS_PATH
: This location is easily accessible by the user, and is the place to store documents and files that the user might want to later open with an external program or transfer to another place.

  You should not save files here unless the user expects it, usually by explicit permission.

ALLEGRO_USER_DATA_PATH
: If your program saves any data that the user doesn't need to access externally, then you should place it here. This is generally the least intrusive place to store data.

ALLEGRO_USER_SETTINGS_PATH
: If you are saving configuration files (especially if the user may want to edit them outside of your program), then you should place them here.

ALLEGRO_EXENAME_PATH
: The full path to the executable.

Returns NULL on failure. The returned path should be freed with al_destroy_path.

See also: al_set_app_name, al_set_org_name, al_destroy_path, al_set_exe_name

0.16.7 al_set_exe_name

```
void al_set_exe_name(char const *path)
```

This override the executable name used by al_get_standard_path for ALLEGRO_EXENAME_PATH and ALLEGRO_RESOURCES_PATH.

One possibility where changing this can be useful is if you use the Python wrapper. Allegro would then by default think that the system's Python executable is the current executable - but you can set it to the .py file being executed instead.

Since: 5.0.6, 5.1.0

See also: al_get_standard_path

0.16.8 al_set_app_name

```
void al_set_app_name(const char *app_name)
```

Sets the global application name.

The application name is used by al_get_standard_path to build the full path to an application's files.

This function may be called before al_init or al_install_system.

See also: al_get_app_name, al_set_org_name

0.16.9 al_set_org_name

```
void al_set_org_name(const char *org_name)
```

Sets the global organization name.

The organization name is used by al_get_standard_path to build the full path to an application's files.

This function may be called before al_init or al_install_system.

See also: al_get_org_name, al_set_app_name

0.16.10 al_get_app_name

```
const char *al_get_app_name(void)
```

Returns the global application name string.

See also: al_set_app_name

0.16.11 al_get_org_name

```
const char *al_get_org_name(void)
```

Returns the global organization name string.

See also: al_set_org_name

0.16.12 al_get_system_config

```
ALLEGRO_CONFIG *al_get_system_config(void)
```

Returns the current system configuration structure, or NULL if there is no active system driver. This is mainly used for configuring Allegro and its addons.

0.16.13 al_register_assert_handler

```
void al_register_assert_handler(void (*handler)(char const *expr,
   char const *file, int line, char const *func))
```

Register a function to be called when an internal Allegro assertion fails. Pass NULL to reset to the default behaviour, which is to do whatever the standard `assert()` macro does.

Since: 5.0.6, 5.1.0

0.17 Threads

Allegro 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>
```

0.17.1 ALLEGRO_THREAD

```
typedef struct ALLEGRO_THREAD ALLEGRO_THREAD;
```

An opaque structure representing a thread.

0.17.2 ALLEGRO_MUTEX

```
typedef struct ALLEGRO_MUTEX ALLEGRO_MUTEX;
```

An opaque structure representing a mutex.

0.17.3 ALLEGRO_COND

```
typedef struct ALLEGRO_COND ALLEGRO_COND;
```

An opaque structure representing a condition variable.

0.17.4 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, al_join_thread.

0.17.5 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 will start its actual execution.

Starting a thread which has already been started does nothing.

See also: al_create_thread.

0.17.6 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 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, al_get_thread_should_stop, al_destroy_thread.

0.17.7 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, al_get_thread_should_stop.

0.17.8 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 or al_set_thread_should_stop on this thread.

See also: al_join_thread, al_set_thread_should_stop.

> *Note:* We don't support forceful killing of threads.

0.17.9 al_destroy_thread

```
void al_destroy_thread(ALLEGRO_THREAD *thread)
```

Free the resources used by a thread. Implicitly performs al_join_thread on the thread if it hasn't been done already.

Does nothing if `thread` is NULL.

See also: al_join_thread.

0.17.10 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, al_start_thread and (after the thread has finished) al_destroy_thread - but you don't have the possibility of ever calling al_join_thread on the thread any longer.

0.17.11 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.

0.17.12 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.

0.17.13 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 or al_create_mutex_recursive. 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.

We don’t yet have al_mutex_trylock.

0.17.14 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.

0.17.15 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`.

0.17.16 al_create_cond

```
ALLEGRO_COND *al_create_cond(void)
```

Create a condition variable.

Returns the condition value on success or NULL on error.

0.17.17 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.

0.17.18 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 returns. al_wait_cond 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_until, al_broadcast_cond, al_signal_cond.

0.17.19 al_wait_cond_until

```
int al_wait_cond_until(ALLEGRO_COND *cond, ALLEGRO_MUTEX *mutex,
   const ALLEGRO_TIMEOUT *timeout)
```

Like al_wait_cond 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.

See also: al_wait_cond

0.17.20 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.

> *Note:* The pthreads spec says to lock the mutex associated with `cond` before signalling for predictable scheduling behaviour.

0.17.21 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 but al_signal_cond may be more efficient when it's applicable.

See also: al_broadcast_cond.

0.18 Time routines

These functions are declared in the main Allegro header file:

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

0.18.1 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

0.18.2 al_get_time

```
double al_get_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.

0.18.3 al_current_time

Alternate spelling of al_get_time.

0.18.4 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, al_wait_for_event_until

0.18.5 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.

0.19 Timer routines

These functions are declared in the main Allegro header file:

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

0.19.1 ALLEGRO_TIMER

```
typedef struct ALLEGRO_TIMER ALLEGRO_TIMER;
```

This is an abstract data type representing a timer object.

0.19.2 ALLEGRO_USECS_TO_SECS

```
#define ALLEGRO_USECS_TO_SECS(x)      ((x) / 1000000.0)
```

Convert microseconds to seconds.

0.19.3 ALLEGRO_MSECS_TO_SECS

```
#define ALLEGRO_MSECS_TO_SECS(x)      ((x) / 1000.0)
```

Convert milliseconds to seconds.

0.19.4 ALLEGRO_BPS_TO_SECS

```
#define ALLEGRO_BPS_TO_SECS(x)        (1.0 / (x))
```

Convert beats per second to seconds.

0.19.5 ALLEGRO_BPM_TO_SECS

```
#define ALLEGRO_BPM_TO_SECS(x)        (60.0 / (x))
```

Convert beats per minute to seconds.

0.19.6 al_create_timer

```
ALLEGRO_TIMER *al_create_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, al_destroy_timer

0.19.7 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, al_get_timer_started

0.19.8 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, al_get_timer_started

0.19.9 al_get_timer_started

```
bool al_get_timer_started(const ALLEGRO_TIMER *timer)
```

Return true if the timer specified is currently started.

0.19.10 al_destroy_timer

```
void al_destroy_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_create_timer

0.19.11 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

0.19.12 al_set_timer_count

```
void al_set_timer_count(ALLEGRO_TIMER *timer, int64_t new_count)
```

Set the 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 at each tick.

See also: al_get_timer_count, al_add_timer_count

0.19.13 al_add_timer_count

```
void al_add_timer_count(ALLEGRO_TIMER *timer, int64_t diff)
```

Add *diff* to the timer's counter value. This is similar to writing:

```
al_set_timer_count(timer, al_get_timer_count(timer) + diff);
```

except that the addition is performed atomically, so no ticks will be lost.

See also: al_set_timer_count

0.19.14 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

0.19.15 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_create_timer.

See also: al_get_timer_speed

0.19.16 al_get_timer_event_source

```
ALLEGRO_EVENT_SOURCE *al_get_timer_event_source(ALLEGRO_TIMER *timer)
```

Retrieve the associated event source.

0.20 Transformations

These functions are declared in the main Allegro header file:

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

The transformations are combined in the order of the function invocations. Thus to create a transformation that first rotates a point and then translates it, you would (starting with an identity transformation) call al_rotate_transform and then al_translate_transform. This approach is opposite of what OpenGL uses but similar to what Direct3D uses.

For those who known the matrix algebra going behind the scenes, what the transformation functions in Allegro do is "pre-multiply" the successive transformations. So, for example, if you have code that does:

```
al_identity_transform(&T);

al_compose_transform(&T, &T1);
al_compose_transform(&T, &T2);
al_compose_transform(&T, &T3);
al_compose_transform(&T, &T4);
```

The resultant matrix multiplication expression will look like this:

```
T4 * T3 * T2 * T1
```

Since the point coordinate vector term will go on the right of that sequence of factors, the transformation that is called first, will also be applied first.

This means if you have code like this:

```
al_identity_transform(&T1);
al_scale_transform(&T1, 2, 2);
al_identity_transform(&T2);
al_translate_transform(&T2, 100, 0);

al_identity_transform(&T);

al_compose_transform(&T, &T1);
al_compose_transform(&T, &T2);

al_use_transform(T);
```

it does exactly the same as:

```
al_identity_transform(&T);
al_scale_transform(&T, 2, 2);
al_translate_transform(&T, 100, 0);
al_use_transform(T);
```

0.20.1 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

0.20.2 al_copy_transform

```
void al_copy_transform(ALLEGRO_TRANSFORM *dest, const ALLEGRO_TRANSFORM *src)
```

Makes a copy of a transformation.

Parameters:

- dest - Source transformation
- src - Destination transformation

0.20.3 al_use_transform

```
void al_use_transform(const ALLEGRO_TRANSFORM *trans)
```

Sets the transformation to be used for the the drawing operations on the target bitmap (each bitmap maintains its own transformation). 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.

This function does nothing if there is no target bitmap.

The parameter is passed by reference as an optimization to avoid the overhead of stack copying. The reference will not be stored in the Allegro library so it is safe to pass references to local variables.

```
void setup_my_transformation(void)
{
   ALLEGRO_TRANSFORM transform;
   al_translate_transform(&transform, 5, 10);
   al_use_transform(&transform);
}
```

Parameters:

- trans - Transformation to use

See also: al_get_current_transform, al_transform_coordinates

0.20.4 al_get_current_transform

```
const ALLEGRO_TRANSFORM *al_get_current_transform(void)
```

Returns the transformation of the current target bitmap, as set by al_use_transform. If there is no target bitmap, this function returns NULL.

Returns: A pointer to the current transformation.

0.20.5 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 to ascertain if the transformation has an inverse before inverting it if you are in doubt.

Parameters:

- trans - Transformation to invert

See also: al_check_inverse

0.20.6 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 1e-7 being sufficient for most applications.

In this function tolerance specifies how close the determinant can be to 0 (if the determinant is 0, the transformation has no inverse). Thus the smaller the tolerance you specify, the "worse" transformations will pass this test. Using a tolerance of 1e-7 will catch errors greater than 1/1000's of a pixel, but let smaller errors pass. That means that if you transformed a point by a transformation and then transformed it again by the inverse transformation that passed this check, the resultant point should less than 1/1000's of a pixel away from the original point.

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 or you have a really large translation. 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

0.20.7 al_identity_transform

```
void al_identity_transform(ALLEGRO_TRANSFORM *trans)
```

Sets the transformation to be the identity transformation. This is the default transformation. Use al_use_transform on an identity transformation to return to the default.

```
ALLEGRO_TRANSFORM t;
al_identity_transform(&t);
al_use_transform(&t);
```

Parameters:

- trans - Transformation to alter

See also: al_translate_transform, al_rotate_transform, al_scale_transform

0.20.8 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

See also: al_translate_transform, al_rotate_transform, al_scale_transform, al_compose_transform

0.20.9 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

See also: al_rotate_transform, al_scale_transform, al_build_transform

0.20.10 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

See also: al_translate_transform, al_scale_transform, al_build_transform

0.20.11 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

See also: al_translate_transform, al_rotate_transform, al_build_transform

0.20.12 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

See also: al_use_transform

0.20.13 al_compose_transform

```
void al_compose_transform(ALLEGRO_TRANSFORM *trans, const ALLEGRO_TRANSFORM *other)
```

Compose (combine) two transformations by a matrix multiplication.

```
trans := trans other
```

Note that the order of matrix multiplications is important. The effect of applying the combined transform will be as if first applying `trans` and then applying `other` and not the other way around.

Parameters:

- trans - Transformation to alter
- other - Transformation used to transform `trans`

0.21 UTF-8 string routines

These functions are declared in the main Allegro header file:

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

0.21.1 About UTF-8 string routines

Some parts of the Allegro API, such as the font rountines, expect Unicode strings encoded in UTF-8. These basic routines are provided to help you work with UTF-8 strings, however it does *not* mean you need to use them. You should consider another library (e.g. ICU) if you require more functionality.

You should also see elsewhere for a proper introduction to Unicode. Briefly, Unicode is a standard consisting of a large character set of over 100,000 characters, and rules, such as how to sort strings. A *code point* is the integer value of a character, but not all code points are characters, as some code points have other uses. Clearly it is impossible represent each code point with a 8-bit byte or even a 16-bit integer, so there exist different Unicode Transformation Formats, most importantly UTF-8 and UTF-16.

UTF-8 has many nice properties, but the main advantages are that it is backwards compatible with C strings, and ASCII characters (code points <= 127) are encoded in UTF-8 exactly as they would be in ASCII.

Here is a diagram of the representation of the word "ål", with a NUL terminator, in both UTF-8 and UTF-16.

String	å	l	NUL
Code points	U+00E5 (229)	U+006C (108)	U+0000 (0)
UTF-8 encoding	0xC3, 0xA5	0x6C	0x00
UTF-16LE encoding	0xE5, 0x00	0x6C, 0x00	0x00, 0x00

You can see the aforementioned properties of UTF-8. The first character U+00E5 (å) has a value greater than 127, so its encoding takes more than a single byte; it requires two bytes. U+006C (l) and U+0000 (NUL) both exist in the ASCII range, i.e. less than 127, so take exactly one byte each, just as in a pure ASCII string. A zero byte never appears except to represent the NUL character, so many functions which expect C-style strings will work with UTF-8 strings without modification.

On the other hand, UTF-16 represents each code code by either two or four bytes, so functions expecting C-style strings will not work at all.

In the following functions, 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 if you think about, is required for good performance. It also means many functions will work even if they do not contain UTF-8, so you may actually store arbitrary data in the strings.

For actual text processing, where you want to specify positions with code point indices, you should use al_ustr_offset to find the byte position of a code point.

0.21.2 UTF-8 string types

ALLEGRO_USTR

```
typedef struct _al_tagbstring ALLEGRO_USTR;
```

An opaque type representing a string. ALLEGRO_USTRs normally contain UTF-8 encoded strings, but they may be used to hold any byte sequences, including NULs.

ALLEGRO_USTR_INFO

```
typedef struct _al_tagbstring ALLEGRO_USTR_INFO;
```

A type that holds additional information for an ALLEGRO_USTR that references an external memory buffer. See al_ref_cstr, al_ref_buffer and al_ref_ustr.

0.21.3 Creating and destroying strings

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.

See also: al_ustr_new_from_buffer, al_ustr_newf, al_ustr_dup, al_ustr_new_from_utf16

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.

See also: al_ustr_new

al_ustr_newf

```
ALLEGRO_USTR *al_ustr_newf(const char *fmt, ...)
```

Create a new string using a printf-style format string.

Notes:

The "%s" specifier takes C string arguments, not ALLEGRO_USTRs. Therefore to pass an ALLEGRO_USTR as a parameter you must use al_cstr, 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 then use the "%s" specifier. Remember to NUL terminate the buffer.

See also: al_ustr_new, al_ustr_appendf

al_ustr_free

```
void al_ustr_free(ALLEGRO_USTR *us)
```

Free a previously allocated string. Does nothing if the argument is NULL.

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 (‘’) 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. Again, no NUL terminator will be added to the referenced string.

See also: al_ustr_to_buffer, al_cstr_dup

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, so a maximum of `size - 1` bytes will be copied.

See also: al_cstr, al_cstr_dup

al_cstr_dup

```
char *al_cstr_dup(const ALLEGRO_USTR *us)
```

Create a NUL (‘’) 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 al_free. If an error occurs NULL is returned.

See also: al_cstr, al_ustr_to_buffer

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.

See also: al_ustr_dup_substr

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.

If you need a range of code-points instead of bytes, use al_ustr_offset to find the byte offsets.

See also: al_ustr_dup

0.21.4 Predefined strings

al_ustr_empty_string

```
const ALLEGRO_USTR *al_ustr_empty_string(void)
```

Return a pointer to a static empty string. The string is read only and must not be freed.

0.21.5 Creating strings by referencing other data

al_ref_cstr

```
const 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");
```

See also: al_ref_buffer, al_ref_ustr

al_ref_buffer

```
const ALLEGRO_USTR *al_ref_buffer(ALLEGRO_USTR_INFO *info, const char *s, size_t size)
```

Like al_ref_cstr 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.

See also: al_ref_cstr, al_ref_ustr

al_ref_ustr

```
const 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 to find the byte offsets.

See also: al_ref_cstr, al_ref_buffer

0.21.6 Sizes and offsets

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.

See also: al_ustr_length

al_ustr_length

```
size_t al_ustr_length(const ALLEGRO_USTR *us)
```

Return the number of code points in the string.

See also: al_ustr_size, al_ustr_offset

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.

See also: al_ustr_length

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.

See also: al_ustr_prev

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.

See also: al_ustr_next

0.21.7 Getting code points

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.

See also: al_ustr_get_next, al_ustr_prev_get

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, invalid byte sequences may be skipped while advancing.

See also: al_ustr_get, al_ustr_prev_get

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, invalid byte sequences may be skipped while advancing.

See also: al_ustr_get_next

0.21.8 Inserting into strings

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 (‘’) bytes. pos is specified in bytes.

Use al_ustr_offset to find the byte offset for a code-points offset

Returns true on success, false on error.

See also: al_ustr_insert_cstr, al_ustr_insert_chr, al_ustr_append

al_ustr_insert_cstr

```
bool al_ustr_insert_cstr(ALLEGRO_USTR *us, int pos, const char *s)
```

Like al_ustr_insert but inserts a C-style string.

See also: al_ustr_insert, al_ustr_insert_chr

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 (") bytes.

Returns the number of bytes inserted, or 0 on error.

See also: al_ustr_insert, al_ustr_insert_cstr

0.21.9 Appending to strings

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.

This function can be used to append an arbitrary buffer:

```
ALLEGRO_USTR_INFO info;
al_ustr_append(us, al_ref_buffer(&info, buf, size));
```

See also: al_ustr_append_cstr, al_ustr_append_chr, al_ustr_appendf, al_ustr_vappendf

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.

See also: al_ustr_append

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.

See also: al_ustr_append

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 about the "%s" and "%c" specifiers.

Returns true on success, false on error.

See also: al_ustr_vappendf

al_ustr_vappendf

```
bool al_ustr_vappendf(ALLEGRO_USTR *us, const char *fmt, va_list ap)
```

Like al_ustr_appendf but you pass the variable argument list directly, instead of the arguments themselves. See al_ustr_newf about the "%s" and "%c" specifiers.

Returns true on success, false on error.

See also: al_ustr_appendf

0.21.10 Removing parts of strings

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 to find the byte offset for a code-points offset.

See also: al_ustr_remove_range

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.

See also: al_ustr_remove_chr, al_ustr_truncate

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.

See also: al_ustr_remove_range, al_ustr_ltrim_ws, al_ustr_rtrim_ws, al_ustr_trim_ws

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 on error.

See also: al_ustr_rtrim_ws, al_ustr_trim_ws

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 on error.

See also: al_ustr_ltrim_ws, al_ustr_trim_ws

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 on error.

See also: al_ustr_ltrim_ws, al_ustr_rtrim_ws

0.21.11 Assigning one string to another

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.

See also: al_ustr_assign_substr, al_ustr_assign_cstr

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 to find the byte offsets.

Returns true on success, false on error.

See also: al_ustr_assign, al_ustr_assign_cstr

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.

See also: al_ustr_assign_substr, al_ustr_assign_cstr

0.21.12 Replacing parts of string

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 (‘’) 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.

See also: al_ustr_replace_range

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 (‘’) bytes.

Use al_ustr_offset to find the byte offsets.

Returns true on success, false on error.

See also: al_ustr_set_chr

0.21.13 Searching

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.

See also: al_ustr_rfind_chr

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.

See also: al_ustr_find_chr

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.

See also: al_ustr_find_set_cstr, al_ustr_find_cset

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 but takes a C-style string for `accept`.

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.

See also: al_ustr_find_cset_cstr, al_ustr_find_set

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 but takes a C-style string for `reject`.

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.

See also: al_ustr_find_cstr, al_ustr_rfind_str, al_ustr_find_replace

al_ustr_find_cstr

```
int al_ustr_find_cstr(const ALLEGRO_USTR *haystack, int start_pos,
   const char *needle)
```

Like al_ustr_find_str but takes a C-style string for `needle`.

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.

See also: al_ustr_rfind_cstr, al_ustr_find_str

al_ustr_rfind_cstr

```
int al_ustr_rfind_cstr(const ALLEGRO_USTR *haystack, int end_pos,
   const char *needle)
```

Like al_ustr_rfind_str but takes a C-style string for `needle`.

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.

See also: al_ustr_find_replace_cstr

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 but takes C-style strings for `find` and `replace`.

0.21.14 Comparing

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 so is preferable if ordering is not important.

See also: al_ustr_compare

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.

See also: al_ustr_ncompare, al_ustr_equal

al_ustr_ncompare

```
int al_ustr_ncompare(const ALLEGRO_USTR *us1, const ALLEGRO_USTR *us2, int n)
```

Like al_ustr_compare 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.

al_ustr_has_prefix

```
bool al_ustr_has_prefix(const ALLEGRO_USTR *us1, const ALLEGRO_USTR *us2)
```

Returns true iff `us1` begins with `us2`.

See also: al_ustr_has_prefix_cstr, al_ustr_has_suffix

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`.

See also: al_ustr_has_prefix

al_ustr_has_suffix

```
bool al_ustr_has_suffix(const ALLEGRO_USTR *us1, const ALLEGRO_USTR *us2)
```

Returns true iff `us1` ends with `us2`.

See also: al_ustr_has_suffix_cstr, al_ustr_has_prefix

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`.

See also: al_ustr_has_suffix

0.21.15 UTF-16 conversion

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.

See also: al_ustr_new

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 to determine the size of the buffer to allocate.

See also: al_ustr_size

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. If the n parameter is smaller than that, the string will be truncated but still always 0 terminated.

See also: al_ustr_size_utf16, al_utf16_encode

0.21.16 Low-level UTF-8 routines

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.

See also: al_utf8_encode, al_utf16_width

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.

See also: al_utf16_encode

0.21.17 Low-level UTF-16 routines

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.

See also: al_utf16_encode, al_utf8_width

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.

See also: al_utf8_encode, al_ustr_encode_utf16

0.22 Platform-specific functions

0.22.1 Windows

These functions are declared in the following header file:

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

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.

0.22.2 iPhone

These functions are declared in the following header file:

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

al_iphone_program_has_halted

```
void al_iphone_program_has_halted(void)
```

Multitasking on iOS is different than on other platforms. When an application receives an ALLEGRO_DISPLAY_SWITCH_OUT or ALLEGRO_DISPLAY_CLOSE event on a multitasking-capable device, it should cease all activity and do nothing but check for an ALLEGRO_DISPLAY_SWITCH_IN event. To let the iPhone driver know that you've ceased all activity, call this function. You should call this function very soon after receiving the event telling you it's time to switch out (within a couple milliseconds). Certain operations, if done, will crash the program after this call, most notably any function which uses OpenGL. This function is needed because the "switch out" handler on iPhone can't return until these operations have stopped, or a crash as described before can happen.

al_iphone_override_screen_scale

```
void al_iphone_override_screen_scale(float scale)
```

Original iPhones and iPod Touches had a screen resolution of 320x480 (in Portrait mode). When the iPhone 4 and iPod Touch 4th generation devices came out, they were backwards compatible with all old iPhone apps. This means that they assume a 320x480 screen resolution by default, while they actually have a 640x960 pixel screen (exactly 2x on each dimension). An API was added to allow access to the full (or in fact any fraction of the) resolution of the new devices. This function is normally not needed, as in the case when you want a scale of 2.0 for "retina display" resolution (640x960). In that case you would just call al_create_display with the larger width and height parameters. It is not limited to 2.0 scaling factors however. You can use 1.5 or 0.5 or other values in between, however if it's not an exact multiple of the original iPhone resolution, linear filtering will be applied to the final image.

This function should be called BEFORE calling al_create_display.

0.23 Direct3D integration

These functions are declared in the following header file:

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

0.23.1 al_get_d3d_device

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

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

Returns: A pointer to the Direct3D device.

0.23.2 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.

0.23.3 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.

0.23.4 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.

0.23.5 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.

0.23.6 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

0.23.7 al_is_d3d_device_lost

```
bool al_is_d3d_device_lost(ALLEGRO_DISPLAY *display)
```

Returns a boolean indicating whether or not the Direct3D device belonging to the given display is in a lost state.

Parameters:

- display - The display that the device you wish to check is attached to

0.24 OpenGL integration

These functions are declared in the following header file:

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

0.24.1 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 given 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 given 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_have_opengl_extension and al_get_opengl_proc_address functions instead.

0.24.2 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:* Under Windows, OpenGL functions may need a special calling convention, so it's best to always use the ALLEGRO_DEFINE_PROC_TYPE macro when declaring function pointer types for OpenGL functions.

Parameters:

name - The name of the symbol you want to link to.

Return value:

A pointer to the symbol if available or NULL otherwise.

0.24.3 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);
```

0.24.4 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. 0's are returned if the bitmap is not an OpenGL bitmap.

0.24.5 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.

0.24.6 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, otherwise returns zero. No attempt will be made to create an FBO if the bitmap is not owned by the current display.

The FBO returned by this function will only be freed when the bitmap is destroyed, or if you call al_remove_opengl_fbo on the bitmap.

> *Note:* In Allegro 5.0.0 this function only returned an FBO which had previously been created by calling al_set_target_bitmap. It would not attempt to create an FBO itself. This has since been changed.

See also: al_remove_opengl_fbo, al_set_target_bitmap

0.24.7 al_remove_opengl_fbo

```
void al_remove_opengl_fbo(ALLEGRO_BITMAP *bitmap)
```

Explicitly free an OpenGL FBO created for a bitmap, if it has one. Usually you do not need to worry about freeing FBOs, unless you use al_get_opengl_fbo.

See also: al_get_opengl_fbo, al_set_target_bitmap

0.24.8 al_have_opengl_extension

```
bool al_have_opengl_extension(const char *extension)
```

This function is a helper to determine whether an OpenGL extension is available on the given display or not.

Example:

```
bool packedpixels = al_have_opengl_extension("GL_EXT_packed_pixels");
```

If *packedpixels* is true then you can safely use the constants related to the packed pixels extension.

Returns true if the extension is available; false otherwise.

0.24.9 al_get_opengl_version

```
uint32_t al_get_opengl_version(void)
```

Returns the OpenGL or OpenGL ES version number of the client (the computer the program is running on), for the current display. "1.0" is returned as 0x01000000, "1.2.1" is returned as 0x01020100, and "1.2.2" as 0x01020200, etc.

A valid OpenGL context must exist for this function to work, which means you may *not* call it before al_create_display.

See also: al_get_opengl_variant

0.24.10 al_get_opengl_variant

```
int al_get_opengl_variant(void)
```

Returns the variant or type of OpenGL used on the running platform. This function can be called before creating a display or setting properties for new displays. Possible values are:

ALLEGRO_DESKTOP_OPENGL
: Regular OpenGL as seen on desktop/laptop computers.

ALLEGRO_OPENGL_ES
: Trimmed down version of OpenGL used on many small consumer electronic devices such as handheld (and sometimes full size) consoles.

See also: al_get_opengl_version

0.24.11 al_set_current_opengl_context

```
void al_set_current_opengl_context(ALLEGRO_DISPLAY *display)
```

Make the OpenGL context associated with the given display current for the calling thread. If there is a current target bitmap which belongs to a different OpenGL context, the target bitmap will be changed to NULL.

Normally you do not need to use this function, as the context will be made current when you call al_set_target_bitmap or al_set_target_backbuffer. You might need if it you created an OpenGL "forward compatible" context. Then al_get_backbuffer only returns NULL, so it would not work to pass that to al_set_target_bitmap.

0.24.12 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).

0.25 Audio addon

These functions are declared in the following header file. Link with allegro_audio.

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

0.25.1 Audio types

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

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.

ALLEGRO_CHANNEL_CONF

```
enum ALLEGRO_CHANNEL_CONF
```

Speaker configuration (mono, stereo, 2.1, 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

ALLEGRO_MIXER

```
typedef struct ALLEGRO_MIXER ALLEGRO_MIXER;
```

A mixer is a type of stream which mixes together attached streams into a single buffer.

ALLEGRO_MIXER_QUALITY

```
enum ALLEGRO_MIXER_QUALITY
```

- ALLEGRO_MIXER_QUALITY_POINT - point sampling
- ALLEGRO_MIXER_QUALITY_LINEAR - linear interpolation

ALLEGRO_PLAYMODE

```
enum ALLEGRO_PLAYMODE
```

Sample and stream playback mode.

- ALLEGRO_PLAYMODE_ONCE
- ALLEGRO_PLAYMODE_LOOP
- ALLEGRO_PLAYMODE_BIDIR

ALLEGRO_SAMPLE_ID

```
typedef struct ALLEGRO_SAMPLE_ID ALLEGRO_SAMPLE_ID;
```

An ALLEGRO_SAMPLE_ID represents a sample being played via al_play_sample. It can be used to later stop the sample with al_stop_sample.

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.

See also: ALLEGRO_SAMPLE_INSTANCE

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 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 object, or to an ALLEGRO_MIXER 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).

See also: ALLEGRO_SAMPLE

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 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 object, or to an ALLEGRO_MIXER 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, 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. 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 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.

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 when you're finished with supplying data to the stream.

If the stream is created by al_load_audio_stream then it can also generate an ALLEGRO_EVENT_AUDIO_STREAM_FINISHED event if it reaches the end of the file and is not set to loop.

ALLEGRO_VOICE

```
typedef struct ALLEGRO_VOICE ALLEGRO_VOICE;
```

A voice represents an audio device on the system, which may be a real device, or an abstract device provided by the operating system. To play back audio, you would attach a mixer or sample or stream to a voice.

See also: ALLEGRO_MIXER, ALLEGRO_SAMPLE, ALLEGRO_AUDIO_STREAM

0.25.2 Setting up audio

al_install_audio

```
bool al_install_audio(void)
```

Install the audio subsystem.

Returns true on success, false on failure.

Note: most users will call al_reserve_samples and al_init_acodec_addon after this.

See also: al_reserve_samples, al_uninstall_audio, al_is_audio_installed, al_init_acodec_addon

al_uninstall_audio

```
void al_uninstall_audio(void)
```

Uninstalls the audio subsystem.

See also: al_install_audio

al_is_audio_installed

```
bool al_is_audio_installed(void)
```

Returns true if al_install_audio was called previously and returned successfully.

al_reserve_samples

```
bool al_reserve_samples(int reserve_samples)
```

Reserves a number of sample instances, attaching them to the default mixer. If no default mixer is set when this function is called, then it will automatically create a voice with an attached mixer, which becomes the default mixer. This diagram illustrates the structures that are set up:

```
                              sample instance 1
                            / sample instance 2
voice <-- default mixer <---         .
                            \        .
                              sample instance N
```

Returns true on success, false on error. al_install_audio must have been called first.

See also: al_set_default_mixer, al_play_sample

0.25.3 Misc audio functions

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.

al_get_audio_depth_size

```
size_t al_get_audio_depth_size(ALLEGRO_AUDIO_DEPTH depth)
```

Return the size of a sample, in bytes, for the given format. The format is one of the values listed under ALLEGRO_AUDIO_DEPTH.

al_get_channel_count

```
size_t al_get_channel_count(ALLEGRO_CHANNEL_CONF conf)
```

Return the number of channels for the given channel configuration, which is one of the values listed under ALLEGRO_CHANNEL_CONF.

0.25.4 Voice functions

al_create_voice

```
ALLEGRO_VOICE *al_create_voice(unsigned int freq,
   ALLEGRO_AUDIO_DEPTH depth, ALLEGRO_CHANNEL_CONF chan_conf)
```

Creates a voice structure 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 a mixer is attached to the voice, the mixer will convert from the mixer's format to the voice format and care does not have to be taken for this.

However if you access the voice directly, make sure to not rely on the parameters passed to this function, but instead query the returned voice for the actual settings.

See also: al_destroy_voice

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

al_detach_voice

```
void al_detach_voice(ALLEGRO_VOICE *voice)
```

Detaches the mixer or sample or stream from the voice.

See also: al_attach_mixer_to_voice, al_attach_sample_instance_to_voice, al_attach_audio_stream_to_voice

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 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.

See also: al_detach_voice

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 apply, with the exception of the depth requirement.

Returns true on success, false on failure.

See also: al_detach_voice

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.

See also: al_detach_voice

al_get_voice_frequency

```
unsigned int al_get_voice_frequency(const ALLEGRO_VOICE *voice)
```

Return the frequency of the voice, e.g. 44100.

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.

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.

al_get_voice_playing

```
bool al_get_voice_playing(const ALLEGRO_VOICE *voice)
```

Return true if the voice is currently playing.

See also: al_set_voice_playing

al_set_voice_playing

```
bool al_set_voice_playing(ALLEGRO_VOICE *voice, bool val)
```

Change whether a voice is playing or not. This can only work if the voice has a non-streaming object attached to it, e.g. a sample instance. On success the voice's current sample position is reset.

Returns true on success, false on failure.

See also: al_get_voice_playing

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.

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 instance.

Returns true on success, false on failure.

See also: al_get_voice_position.

0.25.5 Sample functions

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 with al_free when the sample data structure is destroyed. For portability (especially Windows), the buffer should have been allocated with al_malloc. Otherwise you should free the sample data yourself.

To allocate a buffer of the correct size, you can use something like this:

```
sample_size = al_get_channel_count(chan_conf) * al_get_audio_depth_size(depth);
bytes = samples * sample_size;
buffer = al_malloc(bytes);
```

See also: al_destroy_sample, ALLEGRO_AUDIO_DEPTH, ALLEGRO_CHANNEL_CONF

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 with al_free.

This function will stop any sample instances which may be playing the buffer referenced by the ALLEGRO_SAMPLE.

See also: al_destroy_sample_instance, al_stop_sample, al_stop_samples

al_play_sample

```
bool al_play_sample(ALLEGRO_SAMPLE *spl, float gain, float pan, float speed,
   ALLEGRO_PLAYMODE loop, ALLEGRO_SAMPLE_ID *ret_id)
```

Plays a sample on one of the sample instances created by al_reserve_samples. Returns true on success, false on failure. Playback may fail because all the reserved sample instances 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 - ALLEGRO_PLAYMODE_ONCE, ALLEGRO_PLAYMODE_LOOP, or ALLEGRO_PLAYMODE_BIDIR
- 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, ALLEGRO_AUDIO_PAN_NONE, ALLEGRO_SAMPLE_ID, al_stop_sample, al_stop_samples.

al_stop_sample

```
void al_stop_sample(ALLEGRO_SAMPLE_ID *spl_id)
```

Stop the sample started by al_play_sample.

See also: al_stop_samples

al_stop_samples

```
void al_stop_samples(void)
```

Stop all samples started by al_play_sample.

See also: al_stop_sample

al_get_sample_channels

```
ALLEGRO_CHANNEL_CONF al_get_sample_channels(const ALLEGRO_SAMPLE *spl)
```

Return the channel configuration.

See also: ALLEGRO_CHANNEL_CONF, al_get_sample_depth, al_get_sample_frequency, al_get_sample_length, al_get_sample_data

al_get_sample_depth

```
ALLEGRO_AUDIO_DEPTH al_get_sample_depth(const ALLEGRO_SAMPLE *spl)
```

Return the audio depth.

See also: ALLEGRO_AUDIO_DEPTH, al_get_sample_channels, al_get_sample_frequency, al_get_sample_length, al_get_sample_data

al_get_sample_frequency

```
unsigned int al_get_sample_frequency(const ALLEGRO_SAMPLE *spl)
```

Return the frequency of the sample.

See also: al_get_sample_channels, al_get_sample_depth, al_get_sample_length, al_get_sample_data

al_get_sample_length

```
unsigned int al_get_sample_length(const ALLEGRO_SAMPLE *spl)
```

Return the length of the sample in sample values.

See also: al_get_sample_channels, al_get_sample_depth, al_get_sample_frequency, al_get_sample_data

al_get_sample_data

```
void *al_get_sample_data(const ALLEGRO_SAMPLE *spl)
```

Return a pointer to the raw sample data.

See also: al_get_sample_channels, al_get_sample_depth, al_get_sample_frequency, al_get_sample_length

0.25.6 Sample instance functions

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.

See also: al_destroy_sample_instance

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!).

See also: al_create_sample_instance

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.

See also: al_stop_sample_instance

al_stop_sample_instance

```
bool al_stop_sample_instance(ALLEGRO_SAMPLE_INSTANCE *spl)
```

Stop an sample instance playing.

See also: al_play_sample_instance

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.

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.

al_get_sample_instance_frequency

```
unsigned int al_get_sample_instance_frequency(const ALLEGRO_SAMPLE_INSTANCE *spl)
```

Return the frequency of the sample instance.

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.

See also: al_set_sample_instance_length, al_get_sample_instance_time

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.

See also: al_get_sample_instance_length

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.

See also: al_set_sample_instance_position

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.

See also: al_get_sample_instance_position

al_get_sample_instance_speed

```
float al_get_sample_instance_speed(const ALLEGRO_SAMPLE_INSTANCE *spl)
```

Return the relative playback speed.

See also: al_set_sample_instance_speed

al_set_sample_instance_speed

```
bool al_set_sample_instance_speed(ALLEGRO_SAMPLE_INSTANCE *spl, float val)
```

Set the relative playback speed. 1.0 is normal speed.

Return true on success, false on failure. Will fail if the sample instance is attached directly to a voice.

See also: al_get_sample_instance_speed

al_get_sample_instance_gain

```
float al_get_sample_instance_gain(const ALLEGRO_SAMPLE_INSTANCE *spl)
```

Return the playback gain.

See also: al_set_sample_instance_gain

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.

See also: al_get_sample_instance_gain

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.

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.

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.

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.

See also: al_get_sample_instance_length

al_get_sample_instance_playmode

```
ALLEGRO_PLAYMODE al_get_sample_instance_playmode(const ALLEGRO_SAMPLE_INSTANCE *spl)
```

Return the playback mode.

See also: ALLEGRO_PLAYMODE, al_set_sample_instance_playmode

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.

See also: ALLEGRO_PLAYMODE, al_get_sample_instance_playmode

al_get_sample_instance_playing

```
bool al_get_sample_instance_playing(const ALLEGRO_SAMPLE_INSTANCE *spl)
```

Return true if the sample instance is playing.

See also: al_set_sample_instance_playing

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.

See also: al_get_sample_instance_playing

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.

See also: al_attach_sample_instance_to_mixer, al_attach_sample_instance_to_voice, al_detach_sample_instance

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.

See also: al_attach_sample_instance_to_mixer, al_attach_sample_instance_to_voice, al_get_sample_instance_attached

al_get_sample

```
ALLEGRO_SAMPLE *al_get_sample(ALLEGRO_SAMPLE_INSTANCE *spl)
```

Return the sample data that the sample instance plays.

Note this returns a pointer to an internal structure, *not* the ALLEGRO_SAMPLE that you may have passed to al_set_sample. You may, however, check which sample buffer is being played by the sample instance with al_get_sample_data, and so on.

See also: al_set_sample

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.

See also: al_get_sample

0.25.7 Mixer functions

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

al_destroy_mixer

```
void al_destroy_mixer(ALLEGRO_MIXER *mixer)
```

Destroys the mixer stream.

See also: al_create_mixer

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.

See also: al_reserve_samples, al_play_sample, al_set_default_mixer, al_restore_default_mixer

al_set_default_mixer

```
bool al_set_default_mixer(ALLEGRO_MIXER *mixer)
```

Sets the default mixer. All samples started with al_play_sample will be stopped. If you are using your own mixer, this should be called before al_reserve_samples.

Returns true on success, false on error.

See also: al_reserve_samples, al_play_sample, al_get_default_mixer, al_restore_default_mixer

al_restore_default_mixer

```
bool al_restore_default_mixer(void)
```

Restores Allegro's default mixer. All samples started with al_play_sample will be stopped. Returns true on success, false on error.

See also: al_get_default_mixer, al_set_default_mixer, al_reserve_samples.

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 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.

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.

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.

al_get_mixer_frequency

```
unsigned int al_get_mixer_frequency(const ALLEGRO_MIXER *mixer)
```

Return the mixer frequency.

See also: al_set_mixer_frequency

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.

See also: al_get_mixer_frequency

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.

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.

al_get_mixer_gain

```
float al_get_mixer_gain(const ALLEGRO_MIXER *mixer)
```

Return the mixer gain (amplification factor). The default is 1.0.

Since: 5.0.6, 5.1.0

See also: al_set_mixer_gain.

al_set_mixer_gain

```
bool al_set_mixer_gain(ALLEGRO_MIXER *mixer, float new_gain)
```

Set the mixer gain (amplification factor).

Returns true on success, false on failure.

Since: 5.0.6, 5.1.0

See also: al_get_mixer_gain

al_get_mixer_quality

```
ALLEGRO_MIXER_QUALITY al_get_mixer_quality(const ALLEGRO_MIXER *mixer)
```

Return the mixer quality.

See also: ALLEGRO_MIXER_QUALITY, al_set_mixer_quality

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, al_get_mixer_quality

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.

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.

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, al_attach_audio_stream_to_mixer, al_attach_mixer_to_mixer, al_detach_mixer

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.

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.

0.25.8 Stream functions

al_create_audio_stream

```
ALLEGRO_AUDIO_STREAM *al_create_audio_stream(size_t fragment_count,
   unsigned int frag_samples, unsigned int freq, ALLEGRO_AUDIO_DEPTH depth,
   ALLEGRO_CHANNEL_CONF chan_conf)
```

Creates an ALLEGRO_AUDIO_STREAM. 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.

- frag_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.
- chan_conf - Must be one of the values listed for ALLEGRO_CHANNEL_CONF.

The choice of *fragment_count*, *frag_samples* and *freq* directly influences the audio delay. The delay in seconds can be expressed as:

```
delay = fragment_count * frag_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_audio_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 before the audio system is shut down.

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 or al_load_audio_stream.

> *Note:* If the stream is still attached to a mixer or voice, al_detach_audio_stream is automatically called on it first.

See also: al_drain_audio_stream.

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 for a description of the ALLEGRO_EVENT_AUDIO_STREAM_FRAGMENT event that audio streams emit.

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.

See also: al_destroy_audio_stream

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, al_load_audio_stream_f and the format-specific functions underlying those functions.

al_get_audio_stream_frequency

```
unsigned int al_get_audio_stream_frequency(const ALLEGRO_AUDIO_STREAM *stream)
```

Return the stream frequency.

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.

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.

al_get_audio_stream_length

```
unsigned int al_get_audio_stream_length(const ALLEGRO_AUDIO_STREAM *stream)
```

Return the stream length in samples.

al_get_audio_stream_speed

```
float al_get_audio_stream_speed(const ALLEGRO_AUDIO_STREAM *stream)
```

Return the relative playback speed.

See also: al_set_audio_stream_speed.

al_set_audio_stream_speed

```
bool al_set_audio_stream_speed(ALLEGRO_AUDIO_STREAM *stream, float val)
```

Set the relative playback speed. 1.0 is normal 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.

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.

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.

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.

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.

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.

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.

See also: al_get_audio_stream_playing

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, al_set_audio_stream_playmode.

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, al_get_audio_stream_playmode.

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, al_attach_audio_stream_to_voice, al_detach_audio_stream.

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, al_attach_audio_stream_to_voice, al_get_audio_stream_attached.

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 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.

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 will not return NULL, so you still must check for it.

See also: al_set_audio_stream_fragment, al_get_audio_stream_event_source, al_get_audio_stream_frequency, al_get_audio_stream_channels, al_get_audio_stream_depth, al_get_audio_stream_length

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 to indicate that the buffer is filled with new data.

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 when a new stream is created.

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

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, al_load_audio_stream_f and the format-specific functions underlying those functions.

See also: al_get_audio_stream_position_secs, al_get_audio_stream_length_secs

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.

See also: al_get_audio_stream_length_secs

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, al_load_audio_stream_f and the format-specific functions underlying those functions.

See also: al_get_audio_stream_position_secs

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, al_load_audio_stream_f and the format-specific functions underlying those functions.

0.25.9 Audio file I/O

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. 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.

See also: al_register_sample_loader_f, al_register_sample_saver

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. 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.

See also: al_register_sample_loader

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. 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.

See also: al_register_sample_saver_f, al_register_sample_loader

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. 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.

See also: al_register_sample_saver

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. 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.

See also: al_register_audio_stream_loader_f

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. 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.

See also: al_register_audio_stream_loader

al_load_sample

```
ALLEGRO_SAMPLE *al_load_sample(const char *filename)
```

Loads a few different audio file formats based on their extension.

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.

Returns the sample on success, NULL on failure.

> *Note:* the allegro_audio library does not support any audio file formats by default. You must use the allegro_acodec addon, or register your own format handler.

See also: al_register_sample_loader, al_init_acodec_addon

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.

Returns the sample on success, NULL on failure. The file remains open afterwards.

> *Note:* the allegro_audio library does not support any audio file formats by default. You must use the allegro_acodec addon, or register your own format handler.

See also: al_register_sample_loader_f, al_init_acodec_addon

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.

The audio stream will start in the playing state. It should be attached to a voice or mixer to generate any output. See ALLEGRO_AUDIO_STREAM for more details.

Returns the stream on success, NULL on failure.

> *Note:* the allegro_audio library does not support any audio file formats by default. You must use the allegro_acodec addon, or register your own format handler.

See also: al_load_audio_stream_f, al_register_audio_stream_loader, al_init_acodec_addon

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.

The audio stream will start in the playing state. It should be attached to a voice or mixer to generate any output. See ALLEGRO_AUDIO_STREAM for more details.

Returns the stream on success, NULL on failure. On success the file should be considered owned by the audio stream, and will be closed when the audio stream is destroyed. On failure the file will be closed.

> *Note:* the allegro_audio library does not support any audio file formats by default. You must use the allegro_acodec addon, or register your own format handler.

See also: al_load_audio_stream, al_register_audio_stream_loader_f, al_init_acodec_addon

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.

> *Note:* the allegro_audio library does not support any audio file formats by default. You must use the allegro_acodec addon, or register your own format handler.

See also: al_save_sample_f, al_register_sample_saver, al_init_acodec_addon

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 filestream. Currently, wav is the only supported format, and the extension must be ".wav".

Returns true on success, false on error. The file remains open afterwards.

> *Note:* the allegro_audio library does not support any audio file formats by default. You must use the allegro_acodec addon, or register your own format handler.

See also: al_save_sample, al_register_sample_saver_f, al_init_acodec_addon

0.26 Audio codecs addon

These functions are declared in the following header file. Link with allegro_acodec.

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

0.26.1 al_init_acodec_addon

```
bool al_init_acodec_addon(void)
```

This function registers all the known audio file type handlers for al_load_sample, al_save_sample, al_load_audio_stream, etc.

Depending on what libraries are available, the full set of recognised extensions is: .wav, .flac, .ogg, .it, .mod, .s3m, .xm.

Return true on success.

0.26.2 al_get_allegro_acodec_version

```
uint32_t al_get_allegro_acodec_version(void)
```

Returns the (compiled) version of the addon, in the same format as al_get_allegro_version.

0.27 Color addon

These functions are declared in the following header file. Link with allegro_color.

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

0.27.1 al_color_cmyk

```
ALLEGRO_COLOR al_color_cmyk(float c, float m, float y, float k)
```

Return an ALLEGRO_COLOR structure from CMYK values (cyan, magenta, yellow, black).

See also: al_color_cmyk_to_rgb, al_color_rgb_to_cmyk

0.27.2 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, al_color_rgb_to_cmyk

0.27.3 al_color_hsl

```
ALLEGRO_COLOR al_color_hsl(float h, float s, float l)
```

Return an ALLEGRO_COLOR structure from HSL (hue, saturation, lightness) values.

See also: al_color_hsl_to_rgb, al_color_hsv

0.27.4 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, al_color_hsl, al_color_hsv_to_rgb

0.27.5 al_color_hsv

```
ALLEGRO_COLOR al_color_hsv(float h, float s, float v)
```

Return an ALLEGRO_COLOR structure from HSV (hue, saturation, value) values.

See also: al_color_hsv_to_rgb, al_color_hsl

0.27.6 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, al_color_hsv, al_color_hsl_to_rgb

0.27.7 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, al_color_rgb_to_html

0.27.8 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, al_color_rgb_to_html

0.27.9 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, 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, al_color_html_to_rgb

0.27.10 al_color_name

```
ALLEGRO_COLOR al_color_name(char const *name)
```

Return an ALLEGRO_COLOR with the given name. If the color is not found then black is returned.

See al_color_name_to_rgb for the list of names.

0.27.11 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://www.w3.org/TR/2010/PR-css3-color-20101028/#svg-color`.

Returns: true if a name from the list above was passed, else false.

See also: al_color_name

0.27.12 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, al_color_cmyk_to_rgb

0.27.13 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, al_color_hsl

0.27.14 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, al_color_hsv

0.27.15 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, al_color_name

0.27.16 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, al_color_yuv_to_rgb

0.27.17 al_color_yuv

```
ALLEGRO_COLOR al_color_yuv(float y, float u, float v)
```

Return an ALLEGRO_COLOR structure from YUV values.

See also: al_color_yuv_to_rgb, al_color_rgb_to_yuv

0.27.18 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, al_color_rgb_to_yuv

0.27.19 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.

0.28 Font addons

These functions are declared in the following header file. Link with allegro_font.

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

0.28.1 General font routines

ALLEGRO_FONT

```
typedef struct ALLEGRO_FONT ALLEGRO_FONT;
```

A handle identifying any kind of font. Usually you will create it with al_load_font 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 and a font in Allegro's bitmap font format will be created from it with al_grab_font_from_bitmap.

al_init_font_addon

```
void al_init_font_addon(void)
```

Initialise the font addon.

Note that if you intend to load bitmap fonts, you will need to initialise allegro_image separately (unless you are using another library to load images).

See also: al_init_image_addon, al_init_ttf_addon, al_shutdown_font_addon

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

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 if you pass the name of a known bitmap format, or else al_load_ttf_font.

Bitmap and TTF fonts are affected by the current bitmap flags at the time the font is loaded.

See also: al_destroy_font, al_init_font_addon, al_register_font_loader

al_destroy_font

```
void al_destroy_font(ALLEGRO_FONT *f)
```

Frees the memory being used by a font structure. Does nothing if passed NULL.

See also: al_load_font

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

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.

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.

```
X------------------------
    /\         |        |
   /  \        |        |
  /____\       ascent   |
 /      \      |        |
/        \     |        height
----------------        |
               |        |
               descent  |
               |        |
-------------------------
```

See also: al_get_text_width, al_get_text_dimensions

al_get_font_ascent

```
int al_get_font_ascent(const ALLEGRO_FONT *f)
```

Returns the ascent of the specified font.

See also: al_get_font_descent, al_get_font_line_height

al_get_font_descent

```
int al_get_font_descent(const ALLEGRO_FONT *f)
```

Returns the descent of the specified font.

See also: al_get_font_ascent, al_get_font_line_height

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, al_get_font_line_height, al_get_text_dimensions

al_get_ustr_width

```
int al_get_ustr_width(const ALLEGRO_FONT *f, ALLEGRO_USTR const *ustr)
```

Like al_get_text_width but expects an ALLEGRO_USTR.

See also: al_get_text_width, al_get_ustr_dimensions

al_draw_text

```
void al_draw_text(const ALLEGRO_FONT *font,
   ALLEGRO_COLOR color, float x, float y, int flags,
   char const *text)
```

Writes the NUL-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, al_draw_textf, al_draw_justified_text

al_draw_ustr

```
void al_draw_ustr(const ALLEGRO_FONT *font,
   ALLEGRO_COLOR color, float x, float y, int flags,
   const ALLEGRO_USTR *ustr)
```

Like al_draw_text, except the text is passed as an ALLEGRO_USTR instead of a NUL-terminated char array.

See also: al_draw_text, al_draw_justified_ustr

al_draw_justified_text

```
void al_draw_justified_text(const ALLEGRO_FONT *font,
   ALLEGRO_COLOR color, float x1, float x2,
   float y, float diff, int flags, const char *text)
```

Like al_draw_text, but justifies the string to the region x1-x2.

The *diff* parameter is the maximum amount of horizontal space to allow between words. If justisfying the text would exceed *diff* pixels, or the string contains less than two words, then the string will be drawn left aligned.

See also: al_draw_justified_textf, al_draw_justified_ustr

al_draw_justified_ustr

```
void al_draw_justified_ustr(const ALLEGRO_FONT *font,
   ALLEGRO_COLOR color, float x1, float x2,
   float y, float diff, int flags, const ALLEGRO_USTR *ustr)
```

Like al_draw_justified_text, except the text is passed as an ALLEGRO_USTR instead of a NUL-terminated char array.

See also: al_draw_justified_text, al_draw_justified_textf.

al_draw_textf

```
void al_draw_textf(const ALLEGRO_FONT *font, ALLEGRO_COLOR color,
   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 otherwise.

See also: al_draw_text, al_draw_ustr

al_draw_justified_textf

```
void al_draw_justified_textf(const ALLEGRO_FONT *f,
   ALLEGRO_COLOR color, float x1, float x2, float y,
   float diff, int flags, const char *format, ...)
```

Formatted text output, using a printf() style format string. All parameters have the same meaning as with al_draw_justified_text otherwise.

See also: al_draw_justified_text, al_draw_justified_ustr.

al_get_text_dimensions

```
void al_get_text_dimensions(const ALLEGRO_FONT *f,
   char const *text,
   int *bbx, int *bby, int *bbw, int *bbh)
```

Sometimes, the al_get_text_width and al_get_font_line_height 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.

Note that glyphs may go to the left and upwards of the X, in which case x and y will have negative values.

See also: al_get_text_width, al_get_font_line_height, al_get_ustr_dimensions

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)
```

Sometimes, the al_get_ustr_width and al_get_font_line_height functions are not enough for exact text placement, so this function returns some additional information.

See also: al_get_text_dimensions

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.

0.28.2 Bitmap fonts

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, so usually you will want the rectangles filled with full transparency and the glyphs drawn in opaque white.

Examples:

```
int ranges[] = {32, 126};
al_grab_font_from_bitmap(bitmap, 1, ranges)

int ranges[] = {
    0x0020, 0x007F,  /* ASCII */
    0x00A1, 0x00FF,  /* Latin 1 */
    0x0100, 0x017F,  /* Extended-A */
    0x20AC, 0x20AC}; /* Euro */
al_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, al_grab_font_from_bitmap

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 and then al_grab_font_from_bitmap. If you want to for example load an old A4 font, you could load the bitmap yourself, then call al_convert_mask_to_alpha on it and only then pass it to al_grab_font_from_bitmap.

0.28.3 TTF fonts

These functions are declared in the following header file. Link with allegro_ttf.

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

al_init_ttf_addon

```
bool al_init_ttf_addon(void)
```

Call this after al_init_font_addon to make al_load_font recognize ".ttf" and other formats supported by al_load_ttf_font.

al_shutdown_ttf_addon

```
void al_shutdown_ttf_addon(void)
```

Unloads the ttf addon again. You normally don't need to call this.

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 pixels. 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 pixels, 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 following flags are supported:

- ALLEGRO_TTF_NO_KERNING - Do not use any kerning even if the font file supports it.
- ALLEGRO_TTF_MONOCHROME - Load as a monochrome font (which means no anti-aliasing of the font is done).
- ALLEGRO_TTF_NO_AUTOHINT - Disable the Auto Hinter which is enabled by default in newer versions of FreeType. Since: 5.0.6, 5.1.2

See also: al_init_ttf_addon, al_load_ttf_font_f

al_load_ttf_font_f

```
ALLEGRO_FONT *al_load_ttf_font_f(ALLEGRO_FILE *file,
    char const *filename, int size, int flags)
```

Like al_load_ttf_font, 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.

al_load_ttf_font_stretch

```
ALLEGRO_FONT *al_load_ttf_font_stretch(char const *filename, int w, int h,
    int flags)
```

Like al_load_ttf_font, except it takes separate width and height parameters instead of a single size parameter.

If the height is a positive value, and the width zero or positive, then font will be stretched according to those parameters. The width must not be negative if the height is positive.

As with al_load_ttf_font, the height may be a negative value to specify the total height in pixels. Then the width must also be a negative value, or zero.

The behaviour is undefined the height is positive while width is negative, or if the height is negative while the width is positive.

Since: 5.0.6, 5.1.0

See also: al_load_ttf_font, al_load_ttf_font_stretch_f

al_load_ttf_font_stretch_f

```
ALLEGRO_FONT *al_load_ttf_font_stretch_f(ALLEGRO_FILE *file,
    char const *filename, int w, int h, int flags)
```

Like al_load_ttf_font_stretch, 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.

Since: 5.0.6, 5.1.0

See also: al_load_ttf_font_stretch

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.

0.29 Image I/O addon

These functions are declared in the following header file. Link with allegro_image.

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

0.29.1 al_init_image_addon

```
bool al_init_image_addon(void)
```

Initializes the image addon. This registers bitmap format handlers for al_load_bitmap, al_load_bitmap_f, al_save_bitmap, al_save_bitmap_f.

The following types are built into the Allegro image addon and guaranteed to be available: BMP, PCX, TGA. Every platform also supports JPEG and PNG via external dependencies.

Other formats may be available depending on the operating system and installed libraries, but are not guaranteed and should not be assumed to be universally available.

0.29.2 al_shutdown_image_addon

```
void al_shutdown_image_addon(void)
```

Shut down the image addon. This is done automatically at program exit, but can be called any time the user wishes as well.

0.29.3 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.

0.30 Main addon

The `main` addon has no public API, but contains functionality to enable programs using Allegro to build and run without platform-specific changes.

On platforms that require this functionality (e.g. OSX) this addon contains a C `main` function that invokes al_run_main with the user's own `main` function, where the user's `main` function has had its name mangled to something else. The file that defines the user `main` function must include the header file `allegro5/allegro.h`; that header performs the name mangling using some macros.

If the user `main` function is defined in C++, then it must have the following signature for this addon to work:

```
int main(int argc, char **argv)
```

This addon does nothing on platforms that don't require its functionality, but you should keep it in mind in case you need to port to platforms that do require it.

Link with allegro_main.

0.31 Memfile interface

The memfile interface allows you to treat a fixed block of contiguous memory as a file that can be used with Allegro's I/O functions.

These functions are declared in the following header file. Link with allegro_memfile.

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

0.31.1 al_open_memfile

```
ALLEGRO_FILE *al_open_memfile(void *mem, int64_t size, const char *mode)
```

Returns a file handle to the block of memory. All read and write operations act upon the memory directly, so it must not be freed while the file remains open.

The mode can be any combination of "r" (readable) and "w" (writable). Regardless of the mode, the file always opens at position 0. The file size is fixed and cannot be expanded.

It should be closed with al_fclose. After the file is closed, you are responsible for freeing the memory (if needed).

0.31.2 al_get_allegro_memfile_version

```
uint32_t al_get_allegro_memfile_version(void)
```

Returns the (compiled) version of the addon, in the same format as al_get_allegro_version.

0.32 Native dialogs support

These functions are declared in the following header file. Link with allegro_dialog.

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

0.32.1 ALLEGRO_FILECHOOSER

```
typedef struct ALLEGRO_FILECHOOSER ALLEGRO_FILECHOOSER;
```

Opaque handle to a native file dialog.

0.32.2 ALLEGRO_TEXTLOG

```
typedef struct ALLEGRO_TEXTLOG ALLEGRO_TEXTLOG;
```

Opaque handle to a text log window.

0.32.3 al_create_native_file_dialog

```
ALLEGRO_FILECHOOSER *al_create_native_file_dialog(
   char const *initial_path,
   char const *title,
   char const *patterns,
   int mode)
```

Creates a new native file dialog. You should only have one such dialog opened at a time.

Parameters:

- initial_path: The initial search path and filename. Can be NULL. To start with a blank file name the string should end with a directory separator (this should be the common case).
- 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 'flags' 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 to display it, and from which you then can query the results. When you are done, call al_destroy_native_file_dialog on it.

If a dialog window could not be created then this function returns NULL.

0.32.4 al_show_native_file_dialog

```
bool al_show_native_file_dialog(ALLEGRO_DISPLAY *display,
   ALLEGRO_FILECHOOSER *dialog)
```

Show the dialog window. The display may be NULL, otherwise the given display is treated as the parent if possible.

This function blocks the calling thread until it returns, so you may want to spawn a thread with al_create_thread and call it from inside that thread.

Returns true on success, false on failure.

0.32.5 al_get_native_file_dialog_count

```
int al_get_native_file_dialog_count(const ALLEGRO_FILECHOOSER *dialog)
```

Returns the number of files selected, or 0 if the dialog was cancelled.

0.32.6 al_get_native_file_dialog_path

```
const char *al_get_native_file_dialog_path(
   const ALLEGRO_FILECHOOSER *dialog, size_t i)
```

Returns one of the selected paths.

0.32.7 al_destroy_native_file_dialog

```
void al_destroy_native_file_dialog(ALLEGRO_FILECHOOSER *dialog)
```

Frees up all resources used by the file dialog.

0.32.8 al_show_native_message_box

```
int al_show_native_message_box(ALLEGRO_DISPLAY *display,
   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. The display may be NULL, otherwise the given display is treated as the parent if possible.

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 (|).

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.

ALLEGRO_MESSAGEBOX_QUESTION
The message is a question.

ALLEGRO_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.

al_show_native_message_box may be called without Allegro being installed. This is useful to report an error to initialise Allegro itself.

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.

If a message box could not be created then this returns 0, as if the window was dismissed without activating a button.

Example:

```
button = al_show_native_message_box(
   display,
   "Warning",
   "Are you sure?",
   "If you click yes then you are confirming that \"Yes\""
   "is your response to the query which you have"
   "generated by the action you took to open this"
   "message box.",
   NULL,
   ALLEGRO_MESSAGEBOX_YES_NO
);
```

0.32.9 al_open_native_text_log

```
ALLEGRO_TEXTLOG *al_open_native_text_log(char const *title, int flags)
```

Opens a window to which you can append log messages with al_append_native_text_log. This can be useful for debugging if you don't want to depend on a console being available.

Use al_close_native_text_log to close the window again.

The flags available are:

ALLEGRO_TEXTLOG_NO_CLOSE
Prevent the window from having a close button. Otherwise if the close button is pressed an event is generated; see al_get_native_text_log_event_source.

ALLEGRO_TEXTLOG_MONOSPACE
Use a monospace font to display the text.

Returns NULL if there was an error opening the window, or if text log windows are not implemented on the platform.

See also: al_append_native_text_log, al_close_native_text_log

0.32.10 al_close_native_text_log

```
void al_close_native_text_log(ALLEGRO_TEXTLOG *textlog)
```

Closes a message log window opened with al_open_native_text_log earlier.

Does nothing if passed NULL.

See also: al_open_native_text_log

0.32.11 al_append_native_text_log

```
void al_append_native_text_log(ALLEGRO_TEXTLOG *textlog,
   char const *format, ...)
```

Appends a line of text to the message log window and scrolls to the bottom (if the line would not be visible otherwise). This works like printf. A line is continued until you add a newline character.

If the window is NULL then this function will fall back to calling printf. This makes it convenient to support logging to a window or a terminal.

0.32.12 al_get_native_text_log_event_source

```
ALLEGRO_EVENT_SOURCE *al_get_native_text_log_event_source(
   ALLEGRO_TEXTLOG *textlog)
```

Get an event source for a text log window. The possible events are:

ALLEGRO_EVENT_NATIVE_DIALOG_CLOSE
: The window was requested to be closed, either by pressing the close button or pressing Escape on the keyboard. The user.data1 field will hold a pointer to the ALLEGRO_TEXTLOG which generated the event. The user.data2 field will be 1 if the event was generated as a result of a key press; otherwise it will be zero.

0.32.13 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.

0.33 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, then al_fopen or another function that calls al_fopen.

These functions are declared in the following header file. Link with allegro_physfs.

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

0.33.1 al_set_physfs_file_interface

```
void al_set_physfs_file_interface(void)
```

After calling this, subsequent calls to al_fopen will be handled by PHYSFS_open(). Operations on the files returned by al_fopen will then be performed through PhysicsFS.

At the same time, all filesystem functions like al_read_directory or al_create_fs_entry 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/al_restore_state.

See also: al_set_new_file_interface.

0.33.2 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.

0.34 Primitives addon

These functions are declared in the following header file. Link with allegro_primitives.

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

0.34.1 General

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.

al_init_primitives_addon

```
bool al_init_primitives_addon(void)
```

Initializes the primitives addon.

Returns: True on success, false on failure.

See also: al_shutdown_primitives_addon

al_shutdown_primitives_addon

```
void al_shutdown_primitives_addon(void)
```

Shut down the primitives addon. This is done automatically at program exit, but can be called any time the user wishes as well.

See also: al_init_primitives_addon

0.34.2 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 thick 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.

Pixel-precise output

While normally you should not be too concerned with which pixels are displayed when the high level primitives are drawn, it is nevertheless possible to control that precisely by carefully picking the coordinates at which you draw those primitives.

To be able to do that, however, it is critical to understand how GPU cards convert shapes to pixels. Pixels are not the smallest unit that can be addressed by the GPU. Because the GPU deals with floating point coordinates, it can in fact assign different coordinates to different parts of a single pixel. To a GPU, thus, a screen is composed of a grid of squares that have width and length of 1. The top left corner of the top left pixel is located at (0, 0). Therefore, the center of that pixel is at (0.5, 0.5). The basic rule that determines which pixels are associated with which shape is then as follows: a pixel is treated to belong to a shape if the pixel's center is located in that shape. The figure below illustrates the above concepts:

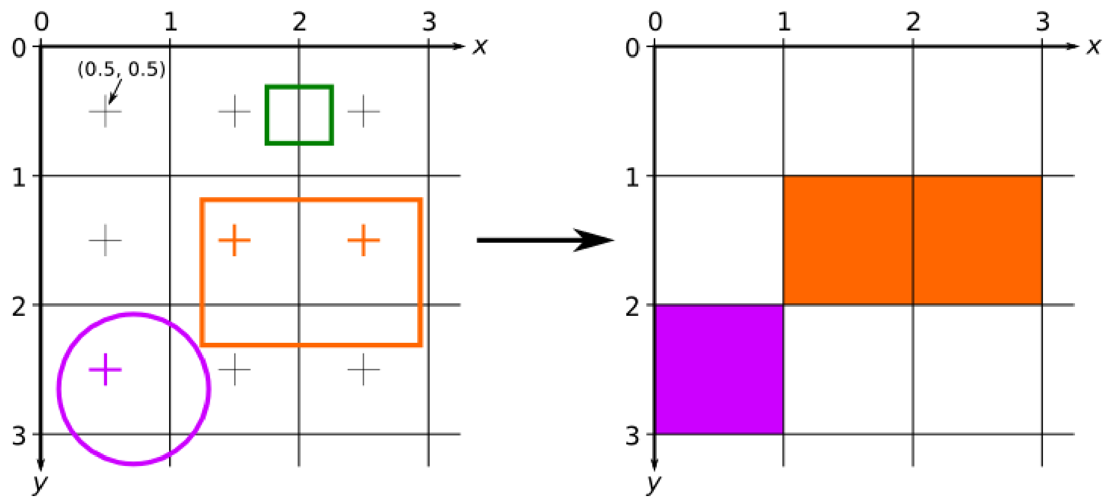

Figure 0.1: *Diagram showing a how pixel output is calculated by the GPU given the mathematical description of several shapes.*

This figure depicts three shapes drawn at the top left of the screen: an orange and green rectangles and a purple circle. On the left are the mathematical descriptions of pixels on the screen and the shapes to be drawn. On the right is the screen output. Only a single pixel has its center inside the circle, and therefore only a single pixel is drawn on the screen. Similarly, two pixels are drawn for the orange rectangle. Since there are no pixels that have their centers inside the green rectangle, the output image has no green pixels.

Here is a more practical example. The image below shows the output of this code:

```
/* blue vertical line */
al_draw_line(0.5, 0, 0.5, 6, color_blue, 1);
/* red horizontal line */
al_draw_line(2, 1, 6, 1, color_red, 2);
/* green filled rectangle */
al_draw_filled_rectangle(3, 4, 5, 5, color_green);
/* purple outlined rectangle */
al_draw_rectangle(2.5, 3.5, 5.5, 5.5, color_purple, 1);
```

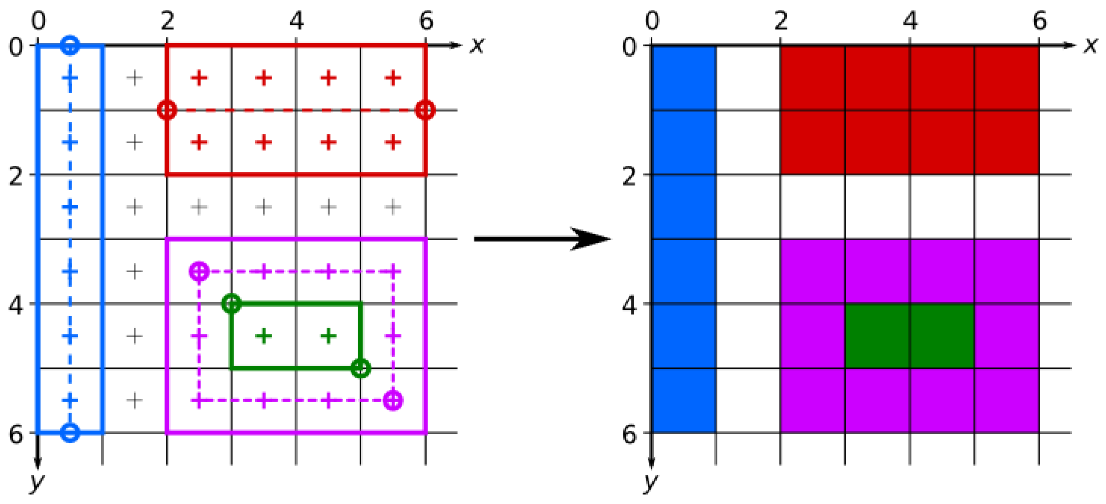

Figure 0.2: *Diagram showing a practical example of pixel output resulting from the invocation of several primitives addon functions.*

It can be seen that lines are generated by making a rectangle based on the dashed line between the two endpoints. The thickness causes the rectangle to grow symmetrically about that generating line, as can be seen by comparing the red and blue lines. Note that to get proper pixel coverage, the coordinates passed to the `al_draw_line` had to be offset by 0.5 in the appropriate dimensions.

Filled rectangles are generated by making a rectangle between the endpoints passed to the `al_draw_filled_rectangle`.

Outlined rectangles are generated by symmetrically expanding an outline of a rectangle. With thickness of 1, as depicted in the diagram, this means that an offset of 0.5 is needed for both sets of endpoint coordinates.

The above rules only apply when multisampling is turned off. When multisampling is turned on, the area of a pixel that is covered by a shape is taken into account when choosing what color to draw there. This also means that shapes no longer have to contain the pixel's center to affect its color. For example, the green rectangle in the first diagram may in fact be drawn as two (or one) semi-transparent pixels. The advantages of multisampling is that slanted shapes will look smoother because they will not have jagged edges. A disadvantage of multisampling is that it may make vertical and horizontal edges blurry. While the exact rules for multisampling are unspecified, and may vary from GPU to GPU it is usually safe to assume that as long as a pixel is either completely covered by a shape or completely not covered, then the shape edges will be sharp. The offsets used in the second diagram were chosen so that this is the case: if you use those offsets, your shapes (if they are oriented the same way as they are on the diagram) should look the same whether multisampling is turned on or off.

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

See also: al_draw_soft_line

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

See also: al_draw_filled_triangle, al_draw_soft_triangle

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

See also: al_draw_triangle

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

See also: al_draw_filled_rectangle, al_draw_rounded_rectangle

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

See also: al_draw_rectangle, al_draw_filled_rounded_rectangle

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

See also: al_draw_filled_rounded_rectangle, al_draw_rectangle

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

See also: al_draw_rounded_rectangle, al_draw_filled_rectangle

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

See also: al_draw_arc, al_calculate_spline, al_calculate_ribbon

al_draw_pieslice

```
void al_draw_pieslice(float cx, float cy, float r, float start_theta,
   float delta_theta, ALLEGRO_COLOR color, float thickness)
```

Draws a pieslice (outlined circular sector).

Parameters:

- cx, cy - Center of the pieslice

- r - Radius of the pieslice
- color - Color of the pieslice
- start_theta - The initial angle from which the pieslice is drawn
- delta_theta - Angular span of the pieslice (pass a negative number to switch direction)
- thickness - Thickness of the circle, pass <= `0` to draw hairline pieslice

Since: 5.0.6, 5.1.0

See also: al_draw_filled_pieslice

al_draw_filled_pieslice

```
void al_draw_filled_pieslice(float cx, float cy, float r, float start_theta,
   float delta_theta, ALLEGRO_COLOR color)
```

Draws a filled pieslice (filled circular sector).

Parameters:

- cx, cy - Center of the pieslice
- r - Radius of the pieslice
- color - Color of the pieslice
- start_theta - The initial angle from which the pieslice is drawn
- delta_theta - Angular span of the pieslice (pass a negative number to switch direction)

Since: 5.0.6, 5.1.0

See also: al_draw_pieslice

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

See also: al_draw_filled_ellipse, al_draw_circle

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

See also: al_draw_ellipse, al_draw_filled_circle

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

See also: al_draw_filled_circle, al_draw_ellipse

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

See also: al_draw_circle, al_draw_filled_ellipse

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 arc, pass `<= 0` to draw hairline arc

See also: al_calculate_arc, al_draw_elliptical_arc

al_draw_elliptical_arc

```
void al_draw_elliptical_arc(float cx, float cy, float rx, float ry, float start_theta,
   float delta_theta, ALLEGRO_COLOR color, float thickness)
```

Draws an elliptical arc.

Parameters:

- cx, cy - Center of the arc
- rx, ry - Radii 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 arc, pass `<= 0` to draw hairline arc

Since: 5.0.6, 5.1.0

See also: al_calculate_arc, al_draw_arc

al_calculate_spline

```
void al_calculate_spline(float* dest, int stride, float points[8],
   float thickness, int num_segments)
```

Calculates a Bézier 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

See also: al_draw_spline, al_calculate_arc, al_calculate_ribbon

al_draw_spline

```
void al_draw_spline(float points[8], ALLEGRO_COLOR color, float thickness)
```

Draws a Bézier 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

See also: al_calculate_spline

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

See also: al_draw_ribbon, al_calculate_arc, al_calculate_spline

al_draw_ribbon

```
void al_draw_ribbon(const float *points, int points_stride, ALLEGRO_COLOR color,
   float thickness, int num_segments)
```

Draws a series of straight lines given an array of points. The ribbon will go through all of the passed points.

Parameters:

- points - An array of coordinate pairs (x and y) for each point
- color - Color of the spline
- thickness - Thickness of the spline, pass `<= 0` to draw hairline spline

See also: al_calculate_ribbon

0.34.3 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.

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.

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 NULL, the vertices are assumed to be of the ALLEGRO_VERTEX type
- start - Start index of the subset of the vertex buffer to draw
- end - One past the last index of subset of the vertex buffer to draw
- type - Primitive type to draw

Returns: Number of primitives drawn

For example to draw a textured triangle you could use:

```
ALLEGRO_VERTEX v[] = {
    {.x = 128, .y = 0, .z = 0, .u = 128, .v = 0},
    {.x = 0, .y = 256, .z = 0, .u = 0, .v = 256},
    {.x = 256, .y = 256, .z = 0, .u = 256, .v = 256}};
al_draw_prim(v, NULL, texture, 0, 3, ALLEGRO_PRIM_TRIANGLE_LIST);
```

See also: ALLEGRO_VERTEX, ALLEGRO_PRIM_TYPE, ALLEGRO_VERTEX_DECL, al_draw_indexed_prim

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, ALLEGRO_PRIM_TYPE, ALLEGRO_VERTEX_DECL, al_draw_prim

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, ALLEGRO_VERTEX_DECL, al_destroy_vertex_decl

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, ALLEGRO_VERTEX_DECL, al_create_vertex_decl

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.

See also: al_draw_triangle

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.

0.34.4 Structures and types

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. Note that when you must initialize all members of this struct when you're using it. One exception to this rule are the u and v variables which can be left uninitialized when you are not using textures.

Fields:

- x, y, z - Position of the vertex
- color - ALLEGRO_COLOR structure, storing the color of the vertex
- u, v - Texture coordinates measured in pixels

See also: ALLEGRO_PRIM_ATTR

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, al_destroy_vertex_decl, ALLEGRO_VERTEX_ELEMENT

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 (passing this as vertex declaration to al_draw_prim would be identical to passing NULL):

```
/* On compilers without the offsetof keyword you need to obtain the
 * offset with sizeof and make sure to account for packing.
 */
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(ALLEGRO_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, ALLEGRO_VERTEX_DECL, ALLEGRO_PRIM_STORAGE

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

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_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.

See also: ALLEGRO_VERTEX_DECL, ALLEGRO_PRIM_STORAGE

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

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.

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.