

# The Allegro 5 Library

## Reference Manual

© 2008 — 2011



---

# Contents

<b>Contents</b>	<b>iii</b>
<b>1 Getting started guide</b>	<b>1</b>
1.1 Introduction . . . . .	1
1.2 Structure of the library and its addons . . . . .	1
1.3 The main function . . . . .	1
1.4 Initialisation . . . . .	2
1.5 Opening a window . . . . .	2
1.6 Display an image . . . . .	2
1.7 Changing the drawing target . . . . .	2
1.8 Event queues and input . . . . .	2
1.9 Displaying some text . . . . .	3
1.10 Drawing primitives . . . . .	3
1.11 Blending . . . . .	3
1.12 Sound . . . . .	3
1.13 Not the end . . . . .	3
<b>2 Configuration files</b>	<b>5</b>
2.1 ALLEGRO_CONFIG . . . . .	5
2.2 ALLEGRO_CONFIG_SECTION . . . . .	6
2.3 ALLEGRO_CONFIG_ENTRY . . . . .	6
2.4 al_create_config . . . . .	6
2.5 al_destroy_config . . . . .	6
2.6 al_load_config_file . . . . .	6
2.7 al_load_config_file_f . . . . .	6
2.8 al_save_config_file . . . . .	6
2.9 al_save_config_file_f . . . . .	7
2.10 al_add_config_section . . . . .	7
2.11 al_remove_config_section . . . . .	7
2.12 al_add_config_comment . . . . .	7
2.13 al_get_config_value . . . . .	7
2.14 al_set_config_value . . . . .	7
2.15 al_remove_config_key . . . . .	8
2.16 al_get_first_config_section . . . . .	8
2.17 al_get_next_config_section . . . . .	8
2.18 al_get_first_config_entry . . . . .	8
2.19 al_get_next_config_entry . . . . .	9
2.20 al_merge_config . . . . .	9
2.21 al_merge_config_into . . . . .	9
<b>3 Displays</b>	<b>11</b>
3.1 Display creation . . . . .	11
3.1.1 ALLEGRO_DISPLAY . . . . .	11
3.1.2 al_create_display . . . . .	11
3.1.3 al_destroy_display . . . . .	11

3.1.4	<code>al_get_new_display_flags</code>	12
3.1.5	<code>al_set_new_display_flags</code>	12
3.1.6	<code>al_get_new_display_option</code>	13
3.1.7	<code>al_set_new_display_option</code>	13
3.1.8	<code>al_reset_new_display_options</code>	15
3.1.9	<code>al_get_new_window_position</code>	15
3.1.10	<code>al_set_new_window_position</code>	15
3.1.11	<code>al_get_new_display_refresh_rate</code>	16
3.1.12	<code>al_set_new_display_refresh_rate</code>	16
3.2	Display operations	16
3.2.1	<code>al_get_display_event_source</code>	16
3.2.2	<code>al_get_backbuffer</code>	16
3.2.3	<code>al_flip_display</code>	16
3.2.4	<code>al_update_display_region</code>	17
3.2.5	<code>al_wait_for_vsync</code>	17
3.3	Display size and position	17
3.3.1	<code>al_get_display_width</code>	17
3.3.2	<code>al_get_display_height</code>	17
3.3.3	<code>al_resize_display</code>	17
3.3.4	<code>al_acknowledge_resize</code>	18
3.3.5	<code>al_get_window_position</code>	18
3.3.6	<code>al_set_window_position</code>	18
3.3.7	<code>al_get_window_constraints</code>	18
3.3.8	<code>al_set_window_constraints</code>	18
3.4	Display settings	19
3.4.1	<code>al_get_display_flags</code>	19
3.4.2	<code>al_set_display_flag</code>	19
3.4.3	<code>al_toggle_display_flag</code>	19
3.4.4	<code>al_get_display_option</code>	19
3.4.5	<code>al_set_display_option</code>	19
3.4.6	<code>al_get_display_format</code>	20
3.4.7	<code>al_get_display_orientation</code>	20
3.4.8	<code>al_get_display_refresh_rate</code>	20
3.4.9	<code>al_set_window_title</code>	20
3.4.10	<code>al_set_display_icon</code>	20
3.4.11	<code>al_set_display_icons</code>	20
3.5	Drawing halts	21
3.5.1	<code>al_acknowledge_drawing_halt</code>	21
3.5.2	<code>al_acknowledge_drawing_resume</code>	21
3.6	Screensaver	21
3.6.1	<code>al_inhibit_screensaver</code>	21
<b>4</b>	<b>Event system and events</b>	<b>23</b>
4.1	<code>ALLEGRO_EVENT</code>	23
4.1.1	<code>ALLEGRO_EVENT_JOYSTICK_AXIS</code>	23
4.1.2	<code>ALLEGRO_EVENT_JOYSTICK_BUTTON_DOWN</code>	24
4.1.3	<code>ALLEGRO_EVENT_JOYSTICK_BUTTON_UP</code>	24
4.1.4	<code>ALLEGRO_EVENT_JOYSTICK_CONFIGURATION</code>	24
4.1.5	<code>ALLEGRO_EVENT_KEY_DOWN</code>	24
4.1.6	<code>ALLEGRO_EVENT_KEY_UP</code>	24
4.1.7	<code>ALLEGRO_EVENT_KEY_CHAR</code>	24
4.1.8	<code>ALLEGRO_EVENT_MOUSE_AXES</code>	25
4.1.9	<code>ALLEGRO_EVENT_MOUSE_BUTTON_DOWN</code>	26
4.1.10	<code>ALLEGRO_EVENT_MOUSE_BUTTON_UP</code>	26
4.1.11	<code>ALLEGRO_EVENT_MOUSE_WARPED</code>	26
4.1.12	<code>ALLEGRO_EVENT_MOUSE_ENTER_DISPLAY</code>	26
4.1.13	<code>ALLEGRO_EVENT_MOUSE_LEAVE_DISPLAY</code>	27

4.1.14	ALLEGRO_EVENT_TIMER	27
4.1.15	ALLEGRO_EVENT_DISPLAY_EXPOSE	27
4.1.16	ALLEGRO_EVENT_DISPLAY_RESIZE	27
4.1.17	ALLEGRO_EVENT_DISPLAY_CLOSE	28
4.1.18	ALLEGRO_EVENT_DISPLAY_LOST	28
4.1.19	ALLEGRO_EVENT_DISPLAY_FOUND	28
4.1.20	ALLEGRO_EVENT_DISPLAY_SWITCH_OUT	28
4.1.21	ALLEGRO_EVENT_DISPLAY_SWITCH_IN	28
4.1.22	ALLEGRO_EVENT_DISPLAY_ORIENTATION	29
4.1.23	ALLEGRO_EVENT_DISPLAY_HALT_DRAWING	29
4.1.24	ALLEGRO_EVENT_DISPLAY_RESUME_DRAWING	29
4.1.25	ALLEGRO_EVENT_DISPLAY_CONNECTED	30
4.1.26	ALLEGRO_EVENT_DISPLAY_DISCONNECTED	30
4.1.27	ALLEGRO_EVENT_TOUCH_BEGIN	30
4.1.28	ALLEGRO_EVENT_TOUCH_END	30
4.1.29	ALLEGRO_EVENT_TOUCH_MOVE	30
4.1.30	ALLEGRO_EVENT_TOUCH_CANCEL	30
4.2	ALLEGRO_USER_EVENT	31
4.3	ALLEGRO_EVENT_QUEUE	31
4.4	ALLEGRO_EVENT_SOURCE	32
4.5	ALLEGRO_EVENT_TYPE	32
4.6	ALLEGRO_GET_EVENT_TYPE	32
4.7	ALLEGRO_EVENT_TYPE_IS_USER	32
4.8	al_create_event_queue	33
4.9	al_destroy_event_queue	33
4.10	al_register_event_source	33
4.11	al_unregister_event_source	33
4.12	al_pause_event_queue	33
4.13	al_is_event_queue_paused	34
4.14	al_is_event_queue_empty	34
4.15	al_get_next_event	34
4.16	al_peek_next_event	34
4.17	al_drop_next_event	34
4.18	al_flush_event_queue	34
4.19	al_wait_for_event	35
4.20	al_wait_for_event_timed	35
4.21	al_wait_for_event_until	35
4.22	al_init_user_event_source	35
4.23	al_destroy_user_event_source	36
4.24	al_emit_user_event	36
4.25	al_unref_user_event	37
4.26	al_get_event_source_data	37
4.27	al_set_event_source_data	37
<b>5</b>	<b>File I/O</b>	<b>39</b>
5.1	ALLEGRO_FILE	39
5.2	ALLEGRO_FILE_INTERFACE	39
5.3	ALLEGRO_SEEK	40
5.4	al_fopen	40
5.5	al_fopen_interface	40
5.6	al_fopen_slice	40
5.7	al_fclose	41
5.8	al_fread	41
5.9	al_fwrite	41
5.10	al_fflush	41
5.11	al_ftell	42
5.12	al_fseek	42

5.13	al_feof . . . . .	42
5.14	al_ferror . . . . .	42
5.15	al_ferrmsg . . . . .	43
5.16	al_fclearerr . . . . .	43
5.17	al_fungetc . . . . .	43
5.18	al_fsize . . . . .	43
5.19	al_fgetc . . . . .	43
5.20	al_fputc . . . . .	43
5.21	al_fprintf . . . . .	44
5.22	al_vfprintf . . . . .	44
5.23	al_fread16le . . . . .	44
5.24	al_fread16be . . . . .	44
5.25	al_fwrite16le . . . . .	44
5.26	al_fwrite16be . . . . .	45
5.27	al_fread32le . . . . .	45
5.28	al_fread32be . . . . .	45
5.29	al_fwrite32le . . . . .	45
5.30	al_fwrite32be . . . . .	45
5.31	al_fgets . . . . .	46
5.32	al_fget_ustr . . . . .	46
5.33	al_fputs . . . . .	46
5.34	Standard I/O specific routines . . . . .	47
5.34.1	al_fopen_fd . . . . .	47
5.34.2	al_make_temp_file . . . . .	47
5.35	Alternative file streams . . . . .	47
5.35.1	al_set_new_file_interface . . . . .	47
5.35.2	al_set_standard_file_interface . . . . .	47
5.35.3	al_get_new_file_interface . . . . .	48
5.35.4	al_create_file_handle . . . . .	48
5.35.5	al_get_file_userdata . . . . .	48
<b>6</b>	<b>Fixed point math routines</b> . . . . .	<b>49</b>
6.1	al_fixed . . . . .	49
6.2	al_itofix . . . . .	49
6.3	al_fixtoi . . . . .	50
6.4	al_fixfloor . . . . .	50
6.5	al_fixceil . . . . .	50
6.6	al_ftofix . . . . .	51
6.7	al_fixtof . . . . .	51
6.8	al_fixmul . . . . .	51
6.9	al_fixdiv . . . . .	52
6.10	al_fixadd . . . . .	52
6.11	al_fixsub . . . . .	53
6.12	Fixed point trig . . . . .	53
6.12.1	al_fixtorad_r . . . . .	53
6.12.2	al_radtofix_r . . . . .	54
6.12.3	al_fixsin . . . . .	54
6.12.4	al_fixcos . . . . .	54
6.12.5	al_fixtan . . . . .	55
6.12.6	al_fixasin . . . . .	55
6.12.7	al_fixacos . . . . .	56
6.12.8	al_fixatan . . . . .	56
6.12.9	al_fixatan2 . . . . .	56
6.12.10	al_fixsqrt . . . . .	57
6.12.11	al_fixhypot . . . . .	57
<b>7</b>	<b>File system routines</b> . . . . .	<b>59</b>
7.1	ALLEGRO_FS_ENTRY . . . . .	59

7.2	ALLEGRO_FILE_MODE . . . . .	59
7.3	al_create_fs_entry . . . . .	59
7.4	al_destroy_fs_entry . . . . .	59
7.5	al_get_fs_entry_name . . . . .	60
7.6	al_update_fs_entry . . . . .	60
7.7	al_get_fs_entry_mode . . . . .	60
7.8	al_get_fs_entry_atime . . . . .	60
7.9	al_get_fs_entry_ctime . . . . .	60
7.10	al_get_fs_entry_mtime . . . . .	60
7.11	al_get_fs_entry_size . . . . .	61
7.12	al_fs_entry_exists . . . . .	61
7.13	al_remove_fs_entry . . . . .	61
7.14	al_filename_exists . . . . .	61
7.15	al_remove_filename . . . . .	61
7.16	Directory functions . . . . .	61
7.16.1	al_open_directory . . . . .	61
7.16.2	al_read_directory . . . . .	62
7.16.3	al_close_directory . . . . .	62
7.16.4	al_get_current_directory . . . . .	62
7.16.5	al_change_directory . . . . .	62
7.16.6	al_make_directory . . . . .	62
7.16.7	al_open_fs_entry . . . . .	62
7.16.8	ALLEGRO_FOR_EACH_FS_ENTRY_RESULT . . . . .	63
7.16.9	al_for_each_fs_entry . . . . .	63
7.17	Alternative filesystem functions . . . . .	63
7.17.1	ALLEGRO_FS_INTERFACE . . . . .	64
7.17.2	al_set_fs_interface . . . . .	64
7.17.3	al_set_standard_fs_interface . . . . .	64
7.17.4	al_get_fs_interface . . . . .	64
<b>8</b>	<b>Fullscreen modes</b> . . . . .	<b>65</b>
8.1	ALLEGRO_DISPLAY_MODE . . . . .	65
8.2	al_get_display_mode . . . . .	65
8.3	al_get_num_display_modes . . . . .	65
<b>9</b>	<b>Graphics routines</b> . . . . .	<b>67</b>
9.1	Colors . . . . .	67
9.1.1	ALLEGRO_COLOR . . . . .	67
9.1.2	al_map_rgb . . . . .	67
9.1.3	al_map_rgb_f . . . . .	67
9.1.4	al_map_rgba . . . . .	67
9.1.5	al_map_rgba_f . . . . .	68
9.1.6	al_unmap_rgb . . . . .	68
9.1.7	al_unmap_rgb_f . . . . .	68
9.1.8	al_unmap_rgba . . . . .	68
9.1.9	al_unmap_rgba_f . . . . .	69
9.2	Locking and pixel formats . . . . .	69
9.2.1	ALLEGRO_LOCKED_REGION . . . . .	69
9.2.2	ALLEGRO_PIXEL_FORMAT . . . . .	69
9.2.3	al_get_pixel_size . . . . .	71
9.2.4	al_get_pixel_format_bits . . . . .	71
9.2.5	al_get_pixel_block_size . . . . .	71
9.2.6	al_get_pixel_block_width . . . . .	72
9.2.7	al_get_pixel_block_height . . . . .	72
9.2.8	al_lock_bitmap . . . . .	72
9.2.9	al_lock_bitmap_region . . . . .	73
9.2.10	al_unlock_bitmap . . . . .	73
9.2.11	al_lock_bitmap_blocked . . . . .	73

9.2.12	al_lock_bitmap_region_blocked . . . . .	73
9.3	Bitmap creation . . . . .	74
9.3.1	ALLEGRO_BITMAP . . . . .	74
9.3.2	al_create_bitmap . . . . .	74
9.3.3	al_create_sub_bitmap . . . . .	74
9.3.4	al_clone_bitmap . . . . .	75
9.3.5	al_convert_bitmap . . . . .	75
9.3.6	al_convert_bitmaps . . . . .	75
9.3.7	al_destroy_bitmap . . . . .	75
9.3.8	al_get_new_bitmap_flags . . . . .	75
9.3.9	al_get_new_bitmap_format . . . . .	76
9.3.10	al_set_new_bitmap_flags . . . . .	76
9.3.11	al_add_new_bitmap_flag . . . . .	77
9.3.12	al_set_new_bitmap_format . . . . .	77
9.4	Bitmap properties . . . . .	77
9.4.1	al_get_bitmap_flags . . . . .	77
9.4.2	al_get_bitmap_format . . . . .	77
9.4.3	al_get_bitmap_height . . . . .	77
9.4.4	al_get_bitmap_width . . . . .	78
9.4.5	al_get_pixel . . . . .	78
9.4.6	al_is_bitmap_locked . . . . .	78
9.4.7	al_is_compatible_bitmap . . . . .	78
9.4.8	al_is_sub_bitmap . . . . .	78
9.4.9	al_get_parent_bitmap . . . . .	78
9.5	Drawing operations . . . . .	79
9.5.1	al_clear_to_color . . . . .	79
9.5.2	al_clear_depth_buffer . . . . .	79
9.5.3	al_draw_bitmap . . . . .	79
9.5.4	al_draw_tinted_bitmap . . . . .	80
9.5.5	al_draw_bitmap_region . . . . .	80
9.5.6	al_draw_tinted_bitmap_region . . . . .	80
9.5.7	al_draw_pixel . . . . .	81
9.5.8	al_draw_rotated_bitmap . . . . .	81
9.5.9	al_draw_tinted_rotated_bitmap . . . . .	82
9.5.10	al_draw_scaled_rotated_bitmap . . . . .	82
9.5.11	al_draw_tinted_scaled_rotated_bitmap . . . . .	82
9.5.12	al_draw_tinted_scaled_rotated_bitmap_region . . . . .	82
9.5.13	al_draw_scaled_bitmap . . . . .	83
9.5.14	al_draw_tinted_scaled_bitmap . . . . .	83
9.5.15	al_get_target_bitmap . . . . .	83
9.5.16	al_put_pixel . . . . .	84
9.5.17	al_put_blended_pixel . . . . .	84
9.5.18	al_set_target_bitmap . . . . .	84
9.5.19	al_set_target_backbuffer . . . . .	85
9.5.20	al_get_current_display . . . . .	85
9.6	Blending modes . . . . .	85
9.6.1	al_get_blender . . . . .	85
9.6.2	al_get_separate_blender . . . . .	85
9.6.3	al_set_blender . . . . .	85
9.6.4	al_set_separate_blender . . . . .	87
9.7	Clipping . . . . .	87
9.7.1	al_get_clipping_rectangle . . . . .	87
9.7.2	al_set_clipping_rectangle . . . . .	88
9.7.3	al_reset_clipping_rectangle . . . . .	88
9.8	Graphics utility functions . . . . .	88
9.8.1	al_convert_mask_to_alpha . . . . .	88
9.9	Deferred drawing . . . . .	88



9.9.1	al_hold_bitmap_drawing . . . . .	88
9.9.2	al_is_bitmap_drawing_held . . . . .	88
9.10	Image I/O . . . . .	89
9.10.1	al_register_bitmap_loader . . . . .	89
9.10.2	al_register_bitmap_saver . . . . .	89
9.10.3	al_register_bitmap_loader_f . . . . .	89
9.10.4	al_register_bitmap_saver_f . . . . .	89
9.10.5	al_load_bitmap . . . . .	90
9.10.6	al_load_bitmap_flags . . . . .	90
9.10.7	al_load_bitmap_f . . . . .	91
9.10.8	al_load_bitmap_flags_f . . . . .	92
9.10.9	al_save_bitmap . . . . .	92
9.10.10	al_save_bitmap_f . . . . .	92
9.11	Render State . . . . .	93
9.11.1	ALLEGRO_RENDER_STATE . . . . .	93
9.11.2	ALLEGRO_RENDER_FUNCTION . . . . .	93
9.11.3	ALLEGRO_WRITE_MASK_FLAGS . . . . .	93
9.11.4	al_set_render_state . . . . .	94
<b>10</b>	<b>Haptic routines</b>	<b>95</b>
10.1	ALLEGRO_HAPTIC . . . . .	95
10.2	ALLEGRO_HAPTIC_CONSTANTS . . . . .	95
10.3	ALLEGRO_HAPTIC_EFFECT . . . . .	96
10.4	ALLEGRO_HAPTIC_EFFECT_ID . . . . .	98
10.5	al_install_haptic . . . . .	98
10.6	al_uninstall_haptic . . . . .	99
10.7	al_is_haptic_installed . . . . .	99
10.8	al_is_mouse_haptic . . . . .	99
10.9	al_is_keyboard_haptic . . . . .	99
10.10	al_is_display_haptic . . . . .	99
10.11	al_is_joystick_haptic . . . . .	99
10.12	al_is_touch_input_haptic . . . . .	100
10.13	al_get_haptic_from_mouse . . . . .	100
10.14	al_get_haptic_from_keyboard . . . . .	100
10.15	al_get_haptic_from_display . . . . .	100
10.16	al_get_haptic_from_joystick . . . . .	100
10.17	al_get_haptic_from_touch_input . . . . .	100
10.18	al_release_haptic . . . . .	101
10.19	al_get_haptic_active . . . . .	101
10.20	al_get_haptic_capabilities . . . . .	101
10.21	al_is_haptic_capable . . . . .	101
10.22	al_set_haptic_gain . . . . .	101
10.23	al_get_haptic_gain . . . . .	102
10.24	al_set_haptic_autocenter . . . . .	102
10.25	al_get_haptic_autocenter . . . . .	102
10.26	al_get_num_haptic_effects . . . . .	102
10.27	al_is_haptic_effect_ok . . . . .	102
10.28	al_upload_haptic_effect . . . . .	103
10.29	al_play_haptic_effect . . . . .	103
10.30	al_upload_and_play_haptic_effect . . . . .	103
10.31	al_stop_haptic_effect . . . . .	103
10.32	al_is_haptic_effect_playing . . . . .	104
10.33	al_get_haptic_effect_duration . . . . .	104
10.34	al_release_haptic_effect . . . . .	104
10.35	al_rumble_haptic . . . . .	104
<b>11</b>	<b>Joystick routines</b>	<b>105</b>
11.1	ALLEGRO_JOYSTICK . . . . .	105

11.2	ALLEGRO_JOYSTICK_STATE . . . . .	105
11.3	ALLEGRO_JOYFLAGS . . . . .	105
11.4	al_install_joystick . . . . .	106
11.5	al_uninstall_joystick . . . . .	106
11.6	al_is_joystick_installed . . . . .	106
11.7	al_reconfigure_joysticks . . . . .	106
11.8	al_get_num_joysticks . . . . .	106
11.9	al_get_joystick . . . . .	107
11.10	al_release_joystick . . . . .	107
11.11	al_get_joystick_active . . . . .	107
11.12	al_get_joystick_name . . . . .	107
11.13	al_get_joystick_stick_name . . . . .	107
11.14	al_get_joystick_axis_name . . . . .	107
11.15	al_get_joystick_button_name . . . . .	108
11.16	al_get_joystick_stick_flags . . . . .	108
11.17	al_get_joystick_num_sticks . . . . .	108
11.18	al_get_joystick_num_axes . . . . .	108
11.19	al_get_joystick_num_buttons . . . . .	108
11.20	al_get_joystick_state . . . . .	108
11.21	al_get_joystick_event_source . . . . .	108
<b>12</b>	<b>Keyboard routines</b>	<b>109</b>
12.1	ALLEGRO_KEYBOARD_STATE . . . . .	109
12.2	Key codes . . . . .	109
12.3	Keyboard modifier flags . . . . .	111
12.4	al_install_keyboard . . . . .	111
12.5	al_is_keyboard_installed . . . . .	111
12.6	al_uninstall_keyboard . . . . .	112
12.7	al_get_keyboard_state . . . . .	112
12.8	al_key_down . . . . .	112
12.9	al_keycode_to_name . . . . .	112
12.10	al_set_keyboard_leds . . . . .	112
12.11	al_get_keyboard_event_source . . . . .	112
<b>13</b>	<b>Memory management routines</b>	<b>113</b>
13.1	al_malloc . . . . .	113
13.2	al_free . . . . .	113
13.3	al_realloc . . . . .	113
13.4	al_calloc . . . . .	114
13.5	al_malloc_with_context . . . . .	114
13.6	al_free_with_context . . . . .	114
13.7	al_realloc_with_context . . . . .	114
13.8	al_calloc_with_context . . . . .	114
13.9	ALLEGRO_MEMORY_INTERFACE . . . . .	114
13.10	al_set_memory_interface . . . . .	115
<b>14</b>	<b>Miscellaneous routines</b>	<b>117</b>
14.1	ALLEGRO_PI . . . . .	117
14.2	al_run_main . . . . .	117
<b>15</b>	<b>Monitors</b>	<b>119</b>
15.1	ALLEGRO_MONITOR_INFO . . . . .	119
15.2	al_get_new_display_adapter . . . . .	119
15.3	al_set_new_display_adapter . . . . .	119
15.4	al_get_monitor_info . . . . .	120
15.5	al_get_num_video_adapters . . . . .	120
<b>16</b>	<b>Mouse routines</b>	<b>121</b>

16.1	ALLEGRO_MOUSE_STATE . . . . .	121
16.2	al_install_mouse . . . . .	121
16.3	al_is_mouse_installed . . . . .	121
16.4	al_uninstall_mouse . . . . .	121
16.5	al_get_mouse_num_axes . . . . .	122
16.6	al_get_mouse_num_buttons . . . . .	122
16.7	al_get_mouse_state . . . . .	122
16.8	al_get_mouse_state_axis . . . . .	122
16.9	al_mouse_button_down . . . . .	122
16.10	al_set_mouse_xy . . . . .	123
16.11	al_set_mouse_z . . . . .	123
16.12	al_set_mouse_w . . . . .	123
16.13	al_set_mouse_axis . . . . .	123
16.14	al_get_mouse_event_source . . . . .	123
16.15	Mouse cursors . . . . .	123
16.15.1	al_create_mouse_cursor . . . . .	123
16.15.2	al_destroy_mouse_cursor . . . . .	124
16.15.3	al_set_mouse_cursor . . . . .	124
16.15.4	al_set_system_mouse_cursor . . . . .	124
16.15.5	al_get_mouse_cursor_position . . . . .	125
16.15.6	al_hide_mouse_cursor . . . . .	125
16.15.7	al_show_mouse_cursor . . . . .	125
16.15.8	al_grab_mouse . . . . .	125
16.15.9	al_ungrab_mouse . . . . .	125
<b>17</b>	<b>Path structures</b>	<b>127</b>
17.1	al_create_path . . . . .	127
17.2	al_create_path_for_directory . . . . .	127
17.3	al_destroy_path . . . . .	127
17.4	al_clone_path . . . . .	127
17.5	al_join_paths . . . . .	128
17.6	al_rebase_path . . . . .	128
17.7	al_get_path_drive . . . . .	128
17.8	al_get_path_num_components . . . . .	128
17.9	al_get_path_component . . . . .	128
17.10	al_get_path_tail . . . . .	129
17.11	al_get_path_filename . . . . .	129
17.12	al_get_path_basename . . . . .	129
17.13	al_get_path_extension . . . . .	129
17.14	al_set_path_drive . . . . .	129
17.15	al_append_path_component . . . . .	129
17.16	al_insert_path_component . . . . .	130
17.17	al_replace_path_component . . . . .	130
17.18	al_remove_path_component . . . . .	130
17.19	al_drop_path_tail . . . . .	130
17.20	al_set_path_filename . . . . .	130
17.21	al_set_path_extension . . . . .	130
17.22	al_path_cstr . . . . .	131
17.23	al_make_path_canonical . . . . .	131
<b>18</b>	<b>State</b>	<b>133</b>
18.1	ALLEGRO_STATE . . . . .	133
18.2	ALLEGRO_STATE_FLAGS . . . . .	133
18.3	al_restore_state . . . . .	134
18.4	al_store_state . . . . .	134
18.5	al_get_errno . . . . .	134
18.6	al_set_errno . . . . .	134

<b>19</b>	<b>System routines</b>	<b>135</b>
19.1	al_install_system . . . . .	135
19.2	al_init . . . . .	135
19.3	al_uninstall_system . . . . .	135
19.4	al_is_system_installed . . . . .	135
19.5	al_get_allegro_version . . . . .	136
19.6	al_get_standard_path . . . . .	136
19.7	al_set_exe_name . . . . .	137
19.8	al_set_app_name . . . . .	137
19.9	al_set_org_name . . . . .	137
19.10	al_get_app_name . . . . .	137
19.11	al_get_org_name . . . . .	138
19.12	al_get_system_config . . . . .	138
19.13	al_register_assert_handler . . . . .	138
19.14	al_register_trace_handler . . . . .	138
<b>20</b>	<b>Threads</b>	<b>139</b>
20.1	ALLEGRO_THREAD . . . . .	139
20.2	ALLEGRO_MUTEX . . . . .	139
20.3	ALLEGRO_COND . . . . .	139
20.4	al_create_thread . . . . .	139
20.5	al_start_thread . . . . .	140
20.6	al_join_thread . . . . .	140
20.7	al_set_thread_should_stop . . . . .	140
20.8	al_get_thread_should_stop . . . . .	140
20.9	al_destroy_thread . . . . .	140
20.10	al_run_detached_thread . . . . .	140
20.11	al_create_mutex . . . . .	141
20.12	al_create_mutex_recursive . . . . .	141
20.13	al_lock_mutex . . . . .	141
20.14	al_unlock_mutex . . . . .	141
20.15	al_destroy_mutex . . . . .	141
20.16	al_create_cond . . . . .	142
20.17	al_destroy_cond . . . . .	142
20.18	al_wait_cond . . . . .	142
20.19	al_wait_cond_until . . . . .	142
20.20	al_broadcast_cond . . . . .	143
20.21	al_signal_cond . . . . .	143
<b>21</b>	<b>Time routines</b>	<b>145</b>
21.1	ALLEGRO_TIMEOUT . . . . .	145
21.2	al_get_time . . . . .	145
21.3	al_current_time . . . . .	145
21.4	al_init_timeout . . . . .	145
21.5	al_rest . . . . .	145
<b>22</b>	<b>Timer routines</b>	<b>147</b>
22.1	ALLEGRO_TIMER . . . . .	147
22.2	ALLEGRO_USECS_TO_SECS . . . . .	147
22.3	ALLEGRO_MSECS_TO_SECS . . . . .	147
22.4	ALLEGRO_BPS_TO_SECS . . . . .	147
22.5	ALLEGRO_BPM_TO_SECS . . . . .	147
22.6	al_create_timer . . . . .	148
22.7	al_start_timer . . . . .	148
22.8	al_stop_timer . . . . .	148
22.9	al_get_timer_started . . . . .	148
22.10	al_destroy_timer . . . . .	148
22.11	al_get_timer_count . . . . .	148

22.12	al_set_timer_count . . . . .	149
22.13	al_add_timer_count . . . . .	149
22.14	al_get_timer_speed . . . . .	149
22.15	al_set_timer_speed . . . . .	149
22.16	al_get_timer_event_source . . . . .	149
<b>23</b>	<b>Touch input</b>	<b>151</b>
23.1	ALLEGRO_TOUCH_INPUT . . . . .	151
23.2	ALLEGRO_TOUCH_INPUT_MAX_TOUCH_COUNT . . . . .	151
23.3	ALLEGRO_TOUCH_STATE . . . . .	151
23.4	ALLEGRO_TOUCH_INPUT_STATE . . . . .	152
23.5	ALLEGRO_MOUSE_EMULATION_MODE . . . . .	152
23.6	al_install_touch_input . . . . .	152
23.7	al_uninstall_touch_input . . . . .	152
23.8	al_is_touch_input_installed . . . . .	153
23.9	al_get_touch_input_state . . . . .	153
23.10	al_set_mouse_emulation_mode . . . . .	153
23.11	al_get_mouse_emulation_mode . . . . .	153
23.12	al_get_touch_input_event_source . . . . .	153
23.13	al_get_touch_input_mouse_emulation_event_source . . . . .	153
<b>24</b>	<b>Transformations</b>	<b>155</b>
24.1	ALLEGRO_TRANSFORM . . . . .	156
24.2	al_copy_transform . . . . .	156
24.3	al_use_transform . . . . .	156
24.4	al_get_current_transform . . . . .	156
24.5	al_get_current_inverse_transform . . . . .	157
24.6	al_invert_transform . . . . .	157
24.7	al_check_inverse . . . . .	157
24.8	al_identity_transform . . . . .	158
24.9	al_build_transform . . . . .	158
24.10	al_translate_transform . . . . .	158
24.11	al_rotate_transform . . . . .	159
24.12	al_scale_transform . . . . .	159
24.13	al_transform_coordinates . . . . .	159
24.14	al_compose_transform . . . . .	159
24.15	al_orthographic_transform . . . . .	160
24.16	al_perspective_transform . . . . .	160
24.17	al_translate_transform_3d . . . . .	160
24.18	al_scale_transform_3d . . . . .	160
24.19	al_rotate_transform_3d . . . . .	161
24.20	al_get_projection_transform . . . . .	161
24.21	al_set_projection_transform . . . . .	161
24.22	al_horizontal_shear_transform . . . . .	161
24.23	al_vertical_shear_transform . . . . .	162
<b>25</b>	<b>UTF-8 string routines</b>	<b>163</b>
25.1	About UTF-8 string routines . . . . .	163
25.2	UTF-8 string types . . . . .	164
25.2.1	ALLEGRO_USTR . . . . .	164
25.2.2	ALLEGRO_USTR_INFO . . . . .	164
25.3	Creating and destroying strings . . . . .	164
25.3.1	al_ustr_new . . . . .	164
25.3.2	al_ustr_new_from_buffer . . . . .	165
25.3.3	al_ustr_newf . . . . .	165
25.3.4	al_ustr_free . . . . .	165
25.3.5	al_cstr . . . . .	165
25.3.6	al_ustr_to_buffer . . . . .	166

25.3.7	<code>al_ustr_dup</code> . . . . .	166
25.3.8	<code>al_ustr_dup</code> . . . . .	166
25.3.9	<code>al_ustr_dup_substr</code> . . . . .	166
25.4	Predefined strings . . . . .	166
25.4.1	<code>al_ustr_empty_string</code> . . . . .	166
25.5	Creating strings by referencing other data . . . . .	166
25.5.1	<code>al_ref_ustr</code> . . . . .	166
25.5.2	<code>al_ref_buffer</code> . . . . .	167
25.5.3	<code>al_ref_ustr</code> . . . . .	167
25.6	Sizes and offsets . . . . .	167
25.6.1	<code>al_ustr_size</code> . . . . .	167
25.6.2	<code>al_ustr_length</code> . . . . .	167
25.6.3	<code>al_ustr_offset</code> . . . . .	167
25.6.4	<code>al_ustr_next</code> . . . . .	168
25.6.5	<code>al_ustr_prev</code> . . . . .	168
25.7	Getting code points . . . . .	168
25.7.1	<code>al_ustr_get</code> . . . . .	168
25.7.2	<code>al_ustr_get_next</code> . . . . .	168
25.7.3	<code>al_ustr_prev_get</code> . . . . .	169
25.8	Inserting into strings . . . . .	169
25.8.1	<code>al_ustr_insert</code> . . . . .	169
25.8.2	<code>al_ustr_insert_ustr</code> . . . . .	169
25.8.3	<code>al_ustr_insert_chr</code> . . . . .	169
25.9	Appending to strings . . . . .	169
25.9.1	<code>al_ustr_append</code> . . . . .	169
25.9.2	<code>al_ustr_append_ustr</code> . . . . .	170
25.9.3	<code>al_ustr_append_chr</code> . . . . .	170
25.9.4	<code>al_ustr_appendf</code> . . . . .	170
25.9.5	<code>al_ustr_vappendf</code> . . . . .	170
25.10	Removing parts of strings . . . . .	170
25.10.1	<code>al_ustr_remove_chr</code> . . . . .	170
25.10.2	<code>al_ustr_remove_range</code> . . . . .	170
25.10.3	<code>al_ustr_truncate</code> . . . . .	171
25.10.4	<code>al_ustr_ltrim_ws</code> . . . . .	171
25.10.5	<code>al_ustr_rtrim_ws</code> . . . . .	171
25.10.6	<code>al_ustr_trim_ws</code> . . . . .	171
25.11	Assigning one string to another . . . . .	171
25.11.1	<code>al_ustr_assign</code> . . . . .	171
25.11.2	<code>al_ustr_assign_substr</code> . . . . .	171
25.11.3	<code>al_ustr_assign_ustr</code> . . . . .	172
25.12	Replacing parts of string . . . . .	172
25.12.1	<code>al_ustr_set_chr</code> . . . . .	172
25.12.2	<code>al_ustr_replace_range</code> . . . . .	172
25.13	Searching . . . . .	172
25.13.1	<code>al_ustr_find_chr</code> . . . . .	172
25.13.2	<code>al_ustr_rfind_chr</code> . . . . .	172
25.13.3	<code>al_ustr_find_set</code> . . . . .	173
25.13.4	<code>al_ustr_find_set_ustr</code> . . . . .	173
25.13.5	<code>al_ustr_find_cset</code> . . . . .	173
25.13.6	<code>al_ustr_find_cset_ustr</code> . . . . .	173
25.13.7	<code>al_ustr_find_str</code> . . . . .	173
25.13.8	<code>al_ustr_find_ustr</code> . . . . .	173
25.13.9	<code>al_ustr_rfind_str</code> . . . . .	174
25.13.10	<code>al_ustr_rfind_ustr</code> . . . . .	174
25.13.11	<code>al_ustr_find_replace</code> . . . . .	174
25.13.12	<code>al_ustr_find_replace_ustr</code> . . . . .	174
25.14	Comparing . . . . .	174

25.14.1	al_ustr_equal . . . . .	174
25.14.2	al_ustr_compare . . . . .	174
25.14.3	al_ustr_ncompare . . . . .	175
25.14.4	al_ustr_has_prefix . . . . .	175
25.14.5	al_ustr_has_prefix_cstr . . . . .	175
25.14.6	al_ustr_has_suffix . . . . .	175
25.14.7	al_ustr_has_suffix_cstr . . . . .	175
25.15	UTF-16 conversion . . . . .	175
25.15.1	al_ustr_new_from_utf16 . . . . .	175
25.15.2	al_ustr_size_utf16 . . . . .	175
25.15.3	al_ustr_encode_utf16 . . . . .	176
25.16	Low-level UTF-8 routines . . . . .	176
25.16.1	al_utf8_width . . . . .	176
25.16.2	al_utf8_encode . . . . .	176
25.17	Low-level UTF-16 routines . . . . .	176
25.17.1	al_utf16_width . . . . .	176
25.17.2	al_utf16_encode . . . . .	176
<b>26</b>	<b>Platform-specific functions</b>	<b>177</b>
26.1	Windows . . . . .	177
26.1.1	al_get_win_window_handle . . . . .	177
26.1.2	al_win_add_window_callback . . . . .	177
26.1.3	al_win_remove_window_callback . . . . .	177
26.2	Mac OS X . . . . .	177
26.2.1	al_osx_get_window . . . . .	178
26.3	iPhone . . . . .	178
26.3.1	al_iphone_override_screen_scale . . . . .	178
26.3.2	al_iphone_set_statusbar_orientation . . . . .	178
26.3.3	al_iphone_get_last_shake_time . . . . .	178
26.3.4	al_iphone_get_battery_level . . . . .	178
26.3.5	al_iphone_get_screen_scale . . . . .	179
26.3.6	al_iphone_get_view . . . . .	179
26.3.7	al_iphone_get_window . . . . .	179
26.4	Android . . . . .	179
26.4.1	al_android_set_apk_file_interface . . . . .	179
26.4.2	al_android_get_os_version . . . . .	179
<b>27</b>	<b>Direct3D integration</b>	<b>181</b>
27.1	al_get_d3d_device . . . . .	181
27.2	al_get_d3d_system_texture . . . . .	181
27.3	al_get_d3d_video_texture . . . . .	181
27.4	al_have_d3d_non_pow2_texture_support . . . . .	181
27.5	al_have_d3d_non_square_texture_support . . . . .	181
27.6	al_get_d3d_texture_size . . . . .	182
27.7	al_get_d3d_texture_position . . . . .	182
27.8	al_is_d3d_device_lost . . . . .	182
27.9	al_set_d3d_device_release_callback . . . . .	182
27.10	al_set_d3d_device_restore_callback . . . . .	183
<b>28</b>	<b>OpenGL integration</b>	<b>185</b>
28.1	al_get_opengl_extension_list . . . . .	185
28.2	al_get_opengl_proc_address . . . . .	185
28.3	al_get_opengl_texture . . . . .	186
28.4	al_get_opengl_texture_size . . . . .	186
28.5	al_get_opengl_texture_position . . . . .	186
28.6	al_get_opengl_fbo . . . . .	186
28.7	al_remove_opengl_fbo . . . . .	187
28.8	al_have_opengl_extension . . . . .	187

28.9	al_get_opengl_version . . . . .	187
28.10	al_get_opengl_variant . . . . .	187
28.11	al_set_current_opengl_context . . . . .	188
28.12	OpenGL configuration . . . . .	188
<b>29</b>	<b>Audio addon</b>	<b>189</b>
29.1	Audio types . . . . .	189
29.1.1	ALLEGRO_AUDIO_DEPTH . . . . .	189
29.1.2	ALLEGRO_AUDIO_PAN_NONE . . . . .	189
29.1.3	ALLEGRO_CHANNEL_CONF . . . . .	190
29.1.4	ALLEGRO_MIXER . . . . .	190
29.1.5	ALLEGRO_MIXER_QUALITY . . . . .	190
29.1.6	ALLEGRO_PLAYMODE . . . . .	190
29.1.7	ALLEGRO_AUDIO_EVENT_TYPE . . . . .	190
29.1.8	ALLEGRO_SAMPLE_ID . . . . .	190
29.1.9	ALLEGRO_SAMPLE . . . . .	191
29.1.10	ALLEGRO_SAMPLE_INSTANCE . . . . .	191
29.1.11	ALLEGRO_AUDIO_STREAM . . . . .	191
29.1.12	ALLEGRO_VOICE . . . . .	192
29.2	Setting up audio . . . . .	192
29.2.1	al_install_audio . . . . .	192
29.2.2	al_uninstall_audio . . . . .	192
29.2.3	al_is_audio_installed . . . . .	192
29.2.4	al_reserve_samples . . . . .	192
29.3	Misc audio functions . . . . .	193
29.3.1	al_get_allegro_audio_version . . . . .	193
29.3.2	al_get_audio_depth_size . . . . .	193
29.3.3	al_get_channel_count . . . . .	193
29.3.4	al_fill_silence . . . . .	193
29.4	Voice functions . . . . .	193
29.4.1	al_create_voice . . . . .	193
29.4.2	al_destroy_voice . . . . .	193
29.4.3	al_detach_voice . . . . .	194
29.4.4	al_attach_audio_stream_to_voice . . . . .	194
29.4.5	al_attach_mixer_to_voice . . . . .	194
29.4.6	al_attach_sample_instance_to_voice . . . . .	194
29.4.7	al_get_voice_frequency . . . . .	194
29.4.8	al_get_voice_channels . . . . .	194
29.4.9	al_get_voice_depth . . . . .	195
29.4.10	al_get_voice_playing . . . . .	195
29.4.11	al_set_voice_playing . . . . .	195
29.4.12	al_get_voice_position . . . . .	195
29.4.13	al_set_voice_position . . . . .	195
29.5	Sample functions . . . . .	195
29.5.1	al_create_sample . . . . .	195
29.5.2	al_destroy_sample . . . . .	196
29.5.3	al_play_sample . . . . .	196
29.5.4	al_stop_sample . . . . .	196
29.5.5	al_stop_samples . . . . .	196
29.5.6	al_get_sample_channels . . . . .	197
29.5.7	al_get_sample_depth . . . . .	197
29.5.8	al_get_sample_frequency . . . . .	197
29.5.9	al_get_sample_length . . . . .	197
29.5.10	al_get_sample_data . . . . .	197
29.6	Sample instance functions . . . . .	197
29.6.1	al_create_sample_instance . . . . .	197
29.6.2	al_destroy_sample_instance . . . . .	197



29.6.3	al_play_sample_instance . . . . .	198
29.6.4	al_stop_sample_instance . . . . .	198
29.6.5	al_get_sample_instance_channels . . . . .	198
29.6.6	al_get_sample_instance_depth . . . . .	198
29.6.7	al_get_sample_instance_frequency . . . . .	198
29.6.8	al_get_sample_instance_length . . . . .	198
29.6.9	al_set_sample_instance_length . . . . .	198
29.6.10	al_get_sample_instance_position . . . . .	199
29.6.11	al_set_sample_instance_position . . . . .	199
29.6.12	al_get_sample_instance_speed . . . . .	199
29.6.13	al_set_sample_instance_speed . . . . .	199
29.6.14	al_get_sample_instance_gain . . . . .	199
29.6.15	al_set_sample_instance_gain . . . . .	199
29.6.16	al_get_sample_instance_pan . . . . .	199
29.6.17	al_set_sample_instance_pan . . . . .	200
29.6.18	al_get_sample_instance_time . . . . .	200
29.6.19	al_get_sample_instance_playmode . . . . .	200
29.6.20	al_set_sample_instance_playmode . . . . .	200
29.6.21	al_get_sample_instance_playing . . . . .	200
29.6.22	al_set_sample_instance_playing . . . . .	200
29.6.23	al_get_sample_instance_attached . . . . .	201
29.6.24	al_detach_sample_instance . . . . .	201
29.6.25	al_get_sample . . . . .	201
29.6.26	al_set_sample . . . . .	201
29.7	Mixer functions . . . . .	201
29.7.1	al_create_mixer . . . . .	201
29.7.2	al_destroy_mixer . . . . .	202
29.7.3	al_get_default_mixer . . . . .	202
29.7.4	al_set_default_mixer . . . . .	202
29.7.5	al_restore_default_mixer . . . . .	202
29.7.6	al_attach_mixer_to_mixer . . . . .	202
29.7.7	al_attach_sample_instance_to_mixer . . . . .	203
29.7.8	al_attach_audio_stream_to_mixer . . . . .	203
29.7.9	al_get_mixer_frequency . . . . .	203
29.7.10	al_set_mixer_frequency . . . . .	203
29.7.11	al_get_mixer_channels . . . . .	203
29.7.12	al_get_mixer_depth . . . . .	203
29.7.13	al_get_mixer_gain . . . . .	203
29.7.14	al_set_mixer_gain . . . . .	204
29.7.15	al_get_mixer_quality . . . . .	204
29.7.16	al_set_mixer_quality . . . . .	204
29.7.17	al_get_mixer_playing . . . . .	204
29.7.18	al_set_mixer_playing . . . . .	204
29.7.19	al_get_mixer_attached . . . . .	204
29.7.20	al_detach_mixer . . . . .	204
29.7.21	al_set_mixer_postprocess_callback . . . . .	205
29.8	Stream functions . . . . .	205
29.8.1	al_create_audio_stream . . . . .	205
29.8.2	al_destroy_audio_stream . . . . .	206
29.8.3	al_get_audio_stream_event_source . . . . .	206
29.8.4	al_drain_audio_stream . . . . .	206
29.8.5	al_rewind_audio_stream . . . . .	206
29.8.6	al_get_audio_stream_frequency . . . . .	206
29.8.7	al_get_audio_stream_channels . . . . .	206
29.8.8	al_get_audio_stream_depth . . . . .	206
29.8.9	al_get_audio_stream_length . . . . .	207
29.8.10	al_get_audio_stream_speed . . . . .	207

29.8.11	al_set_audio_stream_speed . . . . .	207
29.8.12	al_get_audio_stream_gain . . . . .	207
29.8.13	al_set_audio_stream_gain . . . . .	207
29.8.14	al_get_audio_stream_pan . . . . .	207
29.8.15	al_set_audio_stream_pan . . . . .	207
29.8.16	al_get_audio_stream_playing . . . . .	208
29.8.17	al_set_audio_stream_playing . . . . .	208
29.8.18	al_get_audio_stream_playmode . . . . .	208
29.8.19	al_set_audio_stream_playmode . . . . .	208
29.8.20	al_get_audio_stream_attached . . . . .	208
29.8.21	al_detach_audio_stream . . . . .	208
29.8.22	al_get_audio_stream_played_samples . . . . .	208
29.8.23	al_get_audio_stream_fragment . . . . .	209
29.8.24	al_set_audio_stream_fragment . . . . .	209
29.8.25	al_get_audio_stream_fragments . . . . .	209
29.8.26	al_get_available_audio_stream_fragments . . . . .	209
29.8.27	al_seek_audio_stream_secs . . . . .	209
29.8.28	al_get_audio_stream_position_secs . . . . .	210
29.8.29	al_get_audio_stream_length_secs . . . . .	210
29.8.30	al_set_audio_stream_loop_secs . . . . .	210
29.9	Audio file I/O . . . . .	210
29.9.1	al_register_sample_loader . . . . .	210
29.9.2	al_register_sample_loader_f . . . . .	210
29.9.3	al_register_sample_saver . . . . .	211
29.9.4	al_register_sample_saver_f . . . . .	211
29.9.5	al_register_audio_stream_loader . . . . .	211
29.9.6	al_register_audio_stream_loader_f . . . . .	211
29.9.7	al_load_sample . . . . .	212
29.9.8	al_load_sample_f . . . . .	212
29.9.9	al_load_audio_stream . . . . .	212
29.9.10	al_load_audio_stream_f . . . . .	213
29.9.11	al_save_sample . . . . .	213
29.9.12	al_save_sample_f . . . . .	213
29.10	Audio events . . . . .	213
29.10.1	al_get_audio_event_source . . . . .	214
29.11	Audio recording . . . . .	214
29.11.1	ALLEGRO_AUDIO_RECORDER . . . . .	214
29.11.2	ALLEGRO_AUDIO_RECORDER_EVENT . . . . .	214
29.11.3	ALLEGRO_EVENT_AUDIO_RECORDER_FRAGMENT . . . . .	214
29.11.4	al_create_audio_recorder . . . . .	215
29.11.5	al_start_audio_recorder . . . . .	215
29.11.6	al_stop_audio_recorder . . . . .	215
29.11.7	al_is_audio_recorder_recording . . . . .	216
29.11.8	al_get_audio_recorder_event . . . . .	216
29.11.9	al_get_audio_recorder_event_source . . . . .	216
29.11.10	al_destroy_audio_recorder . . . . .	216
<b>30</b>	<b>Audio codecs addon</b>	<b>217</b>
30.1	al_init_acodec_addon . . . . .	217
30.2	al_get_allegro_acodec_version . . . . .	217
<b>31</b>	<b>Color addon</b>	<b>219</b>
31.1	al_color_cmyk . . . . .	219
31.2	al_color_cmyk_to_rgb . . . . .	219
31.3	al_color_hsl . . . . .	219
31.4	al_color_hsl_to_rgb . . . . .	219
31.5	al_color_hsv . . . . .	220
31.6	al_color_hsv_to_rgb . . . . .	220

31.7	<code>al_color_html</code>	220
31.8	<code>al_color_html_to_rgb</code>	220
31.9	<code>al_color_rgb_to_html</code>	221
31.10	<code>al_color_name</code>	221
31.11	<code>al_color_name_to_rgb</code>	221
31.12	<code>al_color_rgb_to_cmyk</code>	222
31.13	<code>al_color_rgb_to_hsl</code>	222
31.14	<code>al_color_rgb_to_hsv</code>	222
31.15	<code>al_color_rgb_to_name</code>	222
31.16	<code>al_color_rgb_to_yuv</code>	223
31.17	<code>al_color_yuv</code>	223
31.18	<code>al_color_yuv_to_rgb</code>	223
31.19	<code>al_get_allegro_color_version</code>	223
<b>32</b>	<b>Font addons</b>	<b>225</b>
32.1	General font routines	225
32.1.1	<code>ALLEGRO_FONT</code>	225
32.1.2	<code>al_init_font_addon</code>	225
32.1.3	<code>al_shutdown_font_addon</code>	225
32.1.4	<code>al_load_font</code>	225
32.1.5	<code>al_destroy_font</code>	226
32.1.6	<code>al_register_font_loader</code>	226
32.1.7	<code>al_get_font_line_height</code>	226
32.1.8	<code>al_get_font_ascent</code>	227
32.1.9	<code>al_get_font_descent</code>	227
32.1.10	<code>al_get_text_width</code>	227
32.1.11	<code>al_get_ustr_width</code>	227
32.1.12	<code>al_draw_text</code>	227
32.1.13	<code>al_draw_ustr</code>	228
32.1.14	<code>al_draw_justified_text</code>	228
32.1.15	<code>al_draw_justified_ustr</code>	228
32.1.16	<code>al_draw_textf</code>	228
32.1.17	<code>al_draw_justified_textf</code>	228
32.1.18	<code>al_get_text_dimensions</code>	229
32.1.19	<code>al_get_ustr_dimensions</code>	229
32.1.20	<code>al_get_allegro_font_version</code>	229
32.1.21	<code>al_get_font_ranges</code>	229
32.2	Multiline text drawing	230
32.2.1	<code>al_draw_multiline_text</code>	230
32.2.2	<code>al_draw_multiline_ustr</code>	230
32.2.3	<code>al_draw_multiline_textf</code>	230
32.2.4	<code>al_do_multiline_text</code>	231
32.2.5	<code>al_do_multiline_ustr</code>	231
32.3	Bitmap fonts	231
32.3.1	<code>al_grab_font_from_bitmap</code>	231
32.3.2	<code>al_load_bitmap_font</code>	232
32.3.3	<code>al_load_bitmap_font_flags</code>	232
32.3.4	<code>al_create_builtin_font</code>	233
32.4	TTF fonts	233
32.4.1	<code>al_init_ttf_addon</code>	233
32.4.2	<code>al_shutdown_ttf_addon</code>	233
32.4.3	<code>al_load_ttf_font</code>	233
32.4.4	<code>al_load_ttf_font_f</code>	234
32.4.5	<code>al_load_ttf_font_stretch</code>	234
32.4.6	<code>al_load_ttf_font_stretch_f</code>	234
32.4.7	<code>al_get_allegro_ttf_version</code>	235
<b>33</b>	<b>Image I/O addon</b>	<b>237</b>

33.1	al_init_image_addon . . . . .	237
33.2	al_shutdown_image_addon . . . . .	237
33.3	al_get_allegro_image_version . . . . .	237
<b>34</b>	<b>Main addon</b>	<b>239</b>
<b>35</b>	<b>Memfile interface</b>	<b>241</b>
35.1	al_open_memfile . . . . .	241
35.2	al_get_allegro_memfile_version . . . . .	241
<b>36</b>	<b>Native dialogs support</b>	<b>243</b>
36.1	ALLEGRO_FILECHOOSER . . . . .	243
36.2	ALLEGRO_TEXTLOG . . . . .	243
36.3	al_init_native_dialog_addon . . . . .	243
36.4	al_shutdown_native_dialog_addon . . . . .	243
36.5	al_create_native_file_dialog . . . . .	244
36.6	al_show_native_file_dialog . . . . .	244
36.7	al_get_native_file_dialog_count . . . . .	245
36.8	al_get_native_file_dialog_path . . . . .	245
36.9	al_destroy_native_file_dialog . . . . .	245
36.10	al_show_native_message_box . . . . .	245
36.11	al_open_native_text_log . . . . .	246
36.12	al_close_native_text_log . . . . .	246
36.13	al_append_native_text_log . . . . .	246
36.14	al_get_native_text_log_event_source . . . . .	247
36.15	al_get_allegro_native_dialog_version . . . . .	247
36.16	Menus . . . . .	247
36.16.1	ALLEGRO_MENU . . . . .	248
36.16.2	ALLEGRO_MENU_INFO . . . . .	248
36.16.3	al_create_menu . . . . .	249
36.16.4	al_create_popup_menu . . . . .	249
36.16.5	al_build_menu . . . . .	249
36.16.6	al_append_menu_item . . . . .	249
36.16.7	al_insert_menu_item . . . . .	249
36.16.8	al_remove_menu_item . . . . .	250
36.16.9	al_clone_menu . . . . .	250
36.16.10	al_clone_menu_for_popup . . . . .	250
36.16.11	al_destroy_menu . . . . .	250
36.16.12	al_get_menu_item_caption . . . . .	251
36.16.13	al_set_menu_item_caption . . . . .	251
36.16.14	al_get_menu_item_flags . . . . .	251
36.16.15	al_set_menu_item_flags . . . . .	251
36.16.16	al_toggle_menu_item_flags . . . . .	251
36.16.17	al_get_menu_item_icon . . . . .	252
36.16.18	al_set_menu_item_icon . . . . .	252
36.16.19	al_find_menu . . . . .	252
36.16.20	al_find_menu_item . . . . .	252
36.16.21	al_get_default_menu_event_source . . . . .	252
36.16.22	al_enable_menu_event_source . . . . .	253
36.16.23	al_disable_menu_event_source . . . . .	253
36.16.24	al_get_display_menu . . . . .	253
36.16.25	al_set_display_menu . . . . .	253
36.16.26	al_popup_menu . . . . .	253
36.16.27	al_remove_display_menu . . . . .	254
<b>37</b>	<b>PhysicsFS integration</b>	<b>255</b>
37.1	al_set_physfs_file_interface . . . . .	255
37.2	al_get_allegro_physfs_version . . . . .	255

<b>38 Primitives addon</b>	<b>257</b>
38.1 General	257
38.1.1 al_get_allegro_primitives_version	257
38.1.2 al_init_primitives_addon	257
38.1.3 al_shutdown_primitives_addon	257
38.2 High level drawing routines	257
38.2.1 Pixel-precise output	258
38.2.2 al_draw_line	259
38.2.3 al_draw_triangle	260
38.2.4 al_draw_filled_triangle	260
38.2.5 al_draw_rectangle	260
38.2.6 al_draw_filled_rectangle	260
38.2.7 al_draw_rounded_rectangle	261
38.2.8 al_draw_filled_rounded_rectangle	261
38.2.9 al_calculate_arc	261
38.2.10 al_draw_pieslice	262
38.2.11 al_draw_filled_pieslice	263
38.2.12 al_draw_ellipse	263
38.2.13 al_draw_filled_ellipse	263
38.2.14 al_draw_circle	264
38.2.15 al_draw_filled_circle	264
38.2.16 al_draw_arc	264
38.2.17 al_draw_elliptical_arc	265
38.2.18 al_calculate_spline	265
38.2.19 al_draw_spline	265
38.2.20 al_calculate_ribbon	266
38.2.21 al_draw_ribbon	266
38.3 Low level drawing routines	266
38.3.1 al_draw_prim	267
38.3.2 al_draw_indexed_prim	267
38.3.3 al_draw_vertex_buffer	268
38.3.4 al_draw_indexed_buffer	268
38.3.5 al_draw_soft_triangle	268
38.3.6 al_draw_soft_line	269
38.4 Custom vertex declaration routines	269
38.4.1 al_create_vertex_decl	269
38.4.2 al_destroy_vertex_decl	270
38.5 Vertex buffer routines	270
38.5.1 al_create_vertex_buffer	270
38.5.2 al_destroy_vertex_buffer	271
38.5.3 al_lock_vertex_buffer	271
38.5.4 al_unlock_vertex_buffer	271
38.5.5 al_get_vertex_buffer_size	271
38.6 Index buffer routines	271
38.6.1 al_create_index_buffer	271
38.6.2 al_destroy_index_buffer	272
38.6.3 al_lock_index_buffer	272
38.6.4 al_unlock_index_buffer	272
38.6.5 al_get_index_buffer_size	273
38.7 Polygon routines	273
38.7.1 al_draw_polyline	273
38.7.2 al_draw_polygon	273
38.7.3 al_draw_filled_polygon	274
38.7.4 al_draw_filled_polygon_with_holes	274
38.7.5 al_triangulate_polygon	275
38.8 Structures and types	275
38.8.1 ALLEGRO_VERTEX	275

38.8.2	ALLEGRO_VERTEX_DECL . . . . .	276
38.8.3	ALLEGRO_VERTEX_ELEMENT . . . . .	276
38.8.4	ALLEGRO_PRIM_TYPE . . . . .	276
38.8.5	ALLEGRO_PRIM_ATTR . . . . .	277
38.8.6	ALLEGRO_PRIM_STORAGE . . . . .	277
38.8.7	ALLEGRO_VERTEX_CACHE_SIZE . . . . .	278
38.8.8	ALLEGRO_PRIM_QUALITY . . . . .	279
38.8.9	ALLEGRO_LINE_JOIN . . . . .	279
38.8.10	ALLEGRO_LINE_CAP . . . . .	279
38.8.11	ALLEGRO_VERTEX_BUFFER . . . . .	280
38.8.12	ALLEGRO_INDEX_BUFFER . . . . .	280
38.8.13	ALLEGRO_PRIM_BUFFER_FLAGS . . . . .	280
<b>39</b>	<b>Shader routines</b>	<b>281</b>
39.1	ALLEGRO_SHADER . . . . .	281
39.2	ALLEGRO_SHADER_TYPE . . . . .	281
39.3	ALLEGRO_SHADER_PLATFORM . . . . .	282
39.4	al_create_shader . . . . .	282
39.5	al_attach_shader_source . . . . .	282
39.6	al_attach_shader_source_file . . . . .	284
39.7	al_build_shader . . . . .	284
39.8	al_get_shader_log . . . . .	284
39.9	al_get_shader_platform . . . . .	284
39.10	al_use_shader . . . . .	284
39.11	al_destroy_shader . . . . .	285
39.12	al_set_shader_sampler . . . . .	285
39.13	al_set_shader_matrix . . . . .	285
39.14	al_set_shader_int . . . . .	286
39.15	al_set_shader_float . . . . .	286
39.16	al_set_shader_bool . . . . .	286
39.17	al_set_shader_int_vector . . . . .	286
39.18	al_set_shader_float_vector . . . . .	287
39.19	al_get_default_shader_source . . . . .	287
<b>40</b>	<b>Video streaming addon</b>	<b>289</b>
40.1	ALLEGRO_VIDEO_EVENT_TYPE . . . . .	289
40.2	al_open_video . . . . .	289
40.3	al_close_video . . . . .	289
40.4	al_start_video . . . . .	289
40.5	al_start_video_with_voice . . . . .	290
40.6	al_get_video_event_source . . . . .	290
40.7	al_pause_video . . . . .	290
40.8	al_is_video_paused . . . . .	290
40.9	al_get_video_aspect_ratio . . . . .	290
40.10	al_get_video_audio_rate . . . . .	290
40.11	al_get_video_fps . . . . .	290
40.12	al_get_video_width . . . . .	291
40.13	al_get_video_height . . . . .	291
40.14	al_get_video_frame . . . . .	291
40.15	al_get_video_position . . . . .	291
40.16	al_seek_video . . . . .	291

## Getting started guide

### 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](http://Allegro.cc).

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

### 1.2 Structure of the library and its addons

Allegro 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.

### 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.

### 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`.

### 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`.

### 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`.

### 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`.

### 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.



## 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`.

## 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.

## 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.

## 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.

## 1.13 Not the end

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

Enjoy!



## 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 as 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);
```

### 2.1 ALLEGRO\_CONFIG

```
typedef struct ALLEGRO_CONFIG ALLEGRO_CONFIG;
```

An abstract configuration structure.

### 2.2 ALLEGRO\_CONFIG\_SECTION

```
typedef struct ALLEGRO_CONFIG_SECTION ALLEGRO_CONFIG_SECTION;
```

An opaque structure used for iterating across sections in a configuration structure.

See also: [al\\_get\\_first\\_config\\_section](#), [al\\_get\\_next\\_config\\_section](#)

### 2.3 ALLEGRO\_CONFIG\_ENTRY

```
typedef struct ALLEGRO_CONFIG_ENTRY ALLEGRO_CONFIG_ENTRY;
```

An opaque structure used for iterating across entries in a configuration section.

See also: [al\\_get\\_first\\_config\\_entry](#), [al\\_get\\_next\\_config\\_entry](#)

### 2.4 al\_create\_config

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

Create an empty configuration structure.

See also: [al\\_load\\_config\\_file](#), [al\\_destroy\\_config](#)

### 2.5 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](#)

### 2.6 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](#)

### 2.7 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](#)

### 2.8 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](#)

## 2.9 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](#)

## 2.10 al\_add\_config\_section

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

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

## 2.11 al\_remove\_config\_section

```
bool al_remove_config_section(ALLEGRO_CONFIG *config, char const *section)
```

Remove a section of a configuration.

Returns true if the section was removed, or false if the section did not exist.

Since: 5.1.5

## 2.12 al\_add\_config\_comment

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

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

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

See also: [al\\_add\\_config\\_section](#)

## 2.13 al\_get\_config\_value

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

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

See also: [al\\_set\\_config\\_value](#)

## 2.14 al\_set\_config\_value

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

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

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

See also: [al\\_get\\_config\\_value](#)

### 2.15 al\_remove\_config\_key

```
bool al_remove_config_key(ALLEGRO_CONFIG *config, char const *section,  
    char const *key)
```

Remove a key and its associated value in a section of a configuration.

Returns true if the entry was removed, or false if the entry did not exist.

Since: 5.1.5

### 2.16 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](#)

### 2.17 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](#)

### 2.18 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](#)

## 2.19 `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`.

## 2.20 `al_merge_config`

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

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

See also: [al\\_merge\\_config\\_into](#)

## 2.21 `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](#)





## Displays

These functions are declared in the main Allegro header file:

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

### 3.1 Display creation

#### 3.1.1 ALLEGRO\_DISPLAY

```
typedef struct ALLEGRO_DISPLAY ALLEGRO_DISPLAY;
```

An opaque type representing an open display or window.

#### 3.1.2 `al_create_display`

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

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

Returns NULL on error.

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](#)

#### 3.1.3 `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](#)

### 3.1.4 `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](#)

### 3.1.5 `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\_PROGRAMMABLE\_PIPELINE**

Require a programmable graphics pipeline. This flag is required to use [ALLEGRO\\_SHADER](#) objects. Since: 5.1.6

**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: 5.0.7, 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.

**ALLEGRO\_GTK\_TOPLEVEL**

Create a GTK toplevel window for the display, on X. This flag is conditionally defined by the native dialog addon. You must call [al\\_init\\_native\\_dialog\\_addon](#) for it to succeed.

[ALLEGRO\\_GTK\\_TOPLEVEL](#) is incompatible with [ALLEGRO\\_FULLSCREEN](#). Since: 5.1.5

0 can be used for default values.

See also: [al\\_set\\_new\\_display\\_option](#), [al\\_get\\_display\\_option](#), [al\\_set\\_display\\_option](#)

**3.1.6 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](#).

**3.1.7 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\\_DONT CARE](#) - 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.

**ALLEGRO\_AUTO\_CONVERT\_BITMAPS**

This is on by default. It causes any existing memory bitmaps with the ALLEGRO\_CONVERT\_BITMAP flag to be converted to a display bitmap of the newly created display with the option set.

Since: 5.1.0

**ALLEGRO\_SUPPORTED\_ORIENTATIONS**

This is a bit-combination of the orientations supported by the application. The orientations are the same as for [al\\_get\\_display\\_orientation](#) with the additional possibilities:

- ALLEGRO\_DISPLAY\_ORIENTATION\_PORTRAIT
- ALLEGRO\_DISPLAY\_ORIENTATION\_LANDSCAPE
- ALLEGRO\_DISPLAY\_ORIENTATION\_ALL

PORTRAIT means only the two portrait orientations are supported, LANDSCAPE means only the two landscape orientations and ALL allows all four orientations. When the orientation changes between a portrait and a landscape orientation the display needs to be resized. This is done by sending an [ALLEGRO\\_EVENT\\_DISPLAY\\_RESIZE](#) message which should be handled by calling [al\\_acknowledge\\_resize](#).

Since: 5.1.0

See also: [al\\_set\\_new\\_display\\_flags](#)

**3.1.8 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.

**3.1.9 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](#)

**3.1.10 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](#)

### 3.1.11 `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](#)

### 3.1.12 `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](#)

## 3.2 Display operations

### 3.2.1 `al_get_display_event_source`

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

Retrieve the associated event source.

### 3.2.2 `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 transformation 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.

### 3.2.3 `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](#)

### 3.2.4 `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. If this is not supported, this function falls back to the behavior of [al\\_flip\\_display](#). You can query the support for this function using `al_get_display_option(display, ALLEGRO_UPDATE_DISPLAY_REGION)`.

See also: [al\\_flip\\_display](#), [al\\_get\\_display\\_option](#)

### 3.2.5 `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](#)

## 3.3 Display size and position

### 3.3.1 `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](#)

### 3.3.2 `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](#)

### 3.3.3 `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](#)

### 3.3.4 `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](#)

### 3.3.5 `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](#)

### 3.3.6 `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](#)

### 3.3.7 `al_get_window_constraints`

```
bool al_get_window_constraints(ALLEGRO_DISPLAY *display,  
    int *min_w, int *min_h, int *max_w, int *max_h)
```

Gets the constraints for a non-fullscreen resizable display.

Since: 5.1.0

See also: [al\\_set\\_window\\_constraints](#)

### 3.3.8 `al_set_window_constraints`

```
bool al_set_window_constraints(ALLEGRO_DISPLAY *display,  
    int min_w, int min_h, int max_w, int max_h)
```

Constrains a non-fullscreen resizable display. The constraints are a hint only, and are not necessarily respected by the window environment. A value of 0 for any of the parameters indicates no constraint for that parameter.

Since: 5.1.0

See also: [al\\_get\\_window\\_constraints](#)



## 3.4 Display settings

### 3.4.1 `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_set_display_flag`

### 3.4.2 `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: 5.0.7, 5.1.2

See also: `al_set_new_display_flags`, `al_get_display_flags`

### 3.4.3 `al_toggle_display_flag`

Deprecated synonym for `al_set_display_flag`.

### 3.4.4 `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`

### 3.4.5 `al_set_display_option`

```
void al_set_display_option(ALLEGRO_DISPLAY *display, int option, int value)
```

Change an option that was previously set for a display. After displays are created, they take on the options set with `al_set_new_display_option`. Calling `al_set_new_display_option` subsequently only changes options for newly created displays, and doesn't touch the options of already created displays. `al_set_display_option` allows changing some of these values. Not all display options can be changed or changing them will have no effect. Changing options other than those listed below is undefined.

- `ALLEGRO_SUPPORTED_ORIENTATIONS` - This can be changed to allow new or restrict previously enabled orientations of the screen/device. See `al_set_new_display_option` for more information on this option.

Since: 5.1.5

See also: `al_set_new_display_option`

#### 3.4.6 `al_get_display_format`

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

Gets the pixel format of the display.

See also: [ALLEGRO\\_PIXEL\\_FORMAT](#)

#### 3.4.7 `al_get_display_orientation`

```
int al_get_display_orientation(ALLEGRO_DISPLAY* display)
```

Return the display orientation, which can be one of the following:

- `ALLEGRO_DISPLAY_ORIENTATION_UNKNOWN`
- `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`

Since: 5.1.0

#### 3.4.8 `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](#)

#### 3.4.9 `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\\_icons](#)

#### 3.4.10 `al_set_display_icon`

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

Changes the icon associated with the display (window). Same as [al\\_set\\_display\\_icons](#) with one icon.

See also: [al\\_set\\_display\\_icons](#), [al\\_set\\_window\\_title](#)

#### 3.4.11 `al_set_display_icons`

```
void al_set_display_icons(ALLEGRO_DISPLAY *display,  
    int num_icons, ALLEGRO_BITMAP *icons[])
```

Changes the icons associated with the display (window). Multiple icons can be provided for use in different contexts, e.g. window frame, taskbar, alt-tab popup. The number of icons must be at least one.

*Note:* If the underlying OS requires an icon of a size not provided then one of the bitmaps will be scaled up or down to the required size. The choice of bitmap is implementation dependent.

Since: 5.0.9, 5.1.5

See also: [al\\_set\\_display\\_icon](#), [al\\_set\\_window\\_title](#)

## 3.5 Drawing halts

### 3.5.1 `al_acknowledge_drawing_halt`

```
void al_acknowledge_drawing_halt(ALLEGRO_DISPLAY *display)
```

Call this in response to the [ALLEGRO\\_EVENT\\_DISPLAY\\_HALT\\_DRAWING](#) event. This is currently necessary for Android and iOS as you are not allowed to draw to your display while it is not being shown. If you do not call this function to let the operating system know that you have stopped drawing or if you call it too late the application likely will be considered misbehaving and get terminated.

Since: 5.1.0

See also: [ALLEGRO\\_EVENT\\_DISPLAY\\_HALT\\_DRAWING](#)

### 3.5.2 `al_acknowledge_drawing_resume`

```
void al_acknowledge_drawing_resume(ALLEGRO_DISPLAY *display)
```

Call this in response to the [ALLEGRO\\_EVENT\\_DISPLAY\\_RESUME\\_DRAWING](#) event.

Since: 5.1.1

See also: [ALLEGRO\\_EVENT\\_DISPLAY\\_RESUME\\_DRAWING](#)

## 3.6 Screensaver

### 3.6.1 `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.



## Event system and events

These functions are declared in the main Allegro header file:

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

### 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.

#### 4.1.1 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.

### 4.1.2 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.

### 4.1.3 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.

### 4.1.4 ALLEGRO\_EVENT\_JOYSTICK\_CONFIGURATION

A joystick was plugged in or unplugged. See [al\\_reconfigure\\_joysticks](#) for details.

### 4.1.5 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 pressed on the keyboard. Look for ALLEGRO\_EVENT\_KEY\_CHAR events for character input.

### 4.1.6 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.

### 4.1.7 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.

**4.1.8 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.pressure (float)**

Pressure, ranging from 0.0 to 1.0.

**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](#).

#### 4.1.9 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.pressure (float)**

Pressure, ranging from 0.0 to 1.0.

**mouse.display (ALLEGRO\_DISPLAY \*)**

The display which had mouse focus.

#### 4.1.10 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.pressure (float)**

Pressure, ranging from 0.0 to 1.0.

**mouse.display (ALLEGRO\_DISPLAY \*)**

The display which had mouse focus.

#### 4.1.11 ALLEGRO\_EVENT\_MOUSE\_WARPED

[`al\_set\_mouse\_xy`](#) was called to move the mouse. This event is identical to `ALLEGRO_EVENT_MOUSE_AXES` otherwise.

#### 4.1.12 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.



#### 4.1.13 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.

#### 4.1.14 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.

#### 4.1.15 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.

#### 4.1.16 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.

You should normally respond to these events by calling `al_acknowledge_resize`. Note that further resize events may be generated by the time you process the event, so these fields may hold outdated information.

### 4.1.17 ALLEGRO\_EVENT\_DISPLAY\_CLOSE

The close button of the window has been pressed.

**display.source** (ALLEGRO\_DISPLAY \*)

The display which was closed.

### 4.1.18 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.

*Note:* This event merely means that the display was lost, that is, DirectX suddenly lost the contents of all video bitmaps. In particular, you can keep calling drawing functions – they just most likely won’t do anything. If Allegro’s restoration of the bitmaps works well for you then no further action is required when you receive this event.

**display.source** (ALLEGRO\_DISPLAY \*)

The display which was lost.

### 4.1.19 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.

### 4.1.20 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.

### 4.1.21 ALLEGRO\_EVENT\_DISPLAY\_SWITCH\_IN

The window is the active one again.

**display.source** (ALLEGRO\_DISPLAY \*)

The display which was switched into.

#### 4.1.22 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](#)

#### 4.1.23 ALLEGRO\_EVENT\_DISPLAY\_HALT\_DRAWING

When a display receives this event it should stop doing any drawing and then call [al\\_acknowledge\\_drawing\\_halt](#) immediately.

This is currently only relevant for Android and iOS. It will be sent when the application is switched to background mode, in addition to [ALLEGRO\\_EVENT\\_DISPLAY\\_SWITCH\\_OUT](#). The latter may also be sent in situations where the application is not active but still should continue drawing, for example when a popup is displayed in front of it.

*Note:* This event means that the next time you call a drawing function, your program will crash. So you must stop drawing and you must immediately reply with [al\\_acknowledge\\_drawing\\_halt](#). Allegro sends this event because it cannot handle this automatically. Your program might be doing the drawing in a different thread from the event handling, in which case the drawing thread needs to be signaled to stop drawing before acknowledging this event.

Since: 5.1.0

See also: [ALLEGRO\\_EVENT\\_DISPLAY\\_RESUME\\_DRAWING](#)

#### 4.1.24 ALLEGRO\_EVENT\_DISPLAY\_RESUME\_DRAWING

When a display receives this event, it may resume drawing again, and it must call [al\\_acknowledge\\_drawing\\_resume](#) immediately.

This is currently only relevant for Android and iOS. The event will be sent when an application returns from background mode and is allowed to draw to the display again, in addition to [ALLEGRO\\_EVENT\\_DISPLAY\\_SWITCH\\_IN](#). The latter event may also be sent in a situation where the application is already active, for example when a popup in front of it closes.

*Note:* Unlike [ALLEGRO\\_EVENT\\_DISPLAY\\_FOUND](#) it is not necessary to reload any bitmaps when you receive this event.

Since: 5.1.0

See also: [ALLEGRO\\_EVENT\\_DISPLAY\\_HALT\\_DRAWING](#)

### 4.1.25 ALLEGRO\_EVENT\_DISPLAY\_CONNECTED

This event is sent when a physical display is connected to the device Allegro runs on. Currently, on most platforms, Allegro supports only a single physical display. However, on iOS, a secondary physical display is supported.

**display.source** ([ALLEGRO\\_DISPLAY \\*](#))

The display which was connected.

Since: 5.1.1

### 4.1.26 ALLEGRO\_EVENT\_DISPLAY\_DISCONNECTED

This event is sent when a physical display is disconnected from the device Allegro runs on. Currently, on most platforms, Allegro supports only a single physical display. However, on iOS, a secondary physical display is supported.

**display.source** ([ALLEGRO\\_DISPLAY \\*](#))

The display which was disconnected.

### 4.1.27 ALLEGRO\_EVENT\_TOUCH\_BEGIN

The touch input device registered a new touch.

**touch.display** ([ALLEGRO\\_DISPLAY](#))

The display which was touched. **touch.id** (int)

An identifier for this touch. If supported by the device it will stay the same for events from the same finger until the touch ends. **touch.x** (float)

The x coordinate of the touch in pixels. **touch.y** (float)

The y coordinate of the touch in pixels. **touch.dx** (float)

Movement speed in pixels in x direction. **touch.dy** (float)

Movement speed in pixels in y direction. **touch.primary** (bool)

Whether this is the only/first touch or an additional touch.

Since: 5.1.0

### 4.1.28 ALLEGRO\_EVENT\_TOUCH\_END

A touch ended.

Has the same fields as [ALLEGRO\\_EVENT\\_TOUCH\\_BEGIN](#).

Since: 5.1.0

### 4.1.29 ALLEGRO\_EVENT\_TOUCH\_MOVE

The position of a touch changed.

Has the same fields as [ALLEGRO\\_EVENT\\_TOUCH\\_BEGIN](#).

Since: 5.1.0

### 4.1.30 ALLEGRO\_EVENT\_TOUCH\_CANCEL

A touch was cancelled. This is device specific but could for example mean that a finger moved off the border of the device or moved so fast that it could not be tracked any longer.

Has the same fields as [ALLEGRO\\_EVENT\\_TOUCH\\_BEGIN](#).

Since: 5.1.0

See also: [ALLEGRO\\_EVENT\\_SOURCE](#), [ALLEGRO\\_EVENT\\_TYPE](#), [ALLEGRO\\_USER\\_EVENT](#), [ALLEGRO\\_GET\\_EVENT\\_TYPE](#)

## 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](#)

## 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](#)

## 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](#)

## 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](#)

## 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](#)

## 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](#).

## 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](#)

## 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](#)

## 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](#)

## 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](#)

## 4.12 `al_pause_event_queue`

```
void al_pause_event_queue(ALLEGRO_EVENT_QUEUE *queue, bool pause)
```

Pause or resume accepting new events into the event queue. Events already in the queue are unaffected.

While a queue is paused, any events which would be entered into the queue are simply ignored. This is an alternative to unregistering then re-registering all event sources from the event queue, if you just need to prevent events piling up in the queue for a while.

See also: [al\\_is\\_event\\_queue\\_paused](#)

Since: 5.1.0

### 4.13 `al_is_event_queue_paused`

```
bool al_is_event_queue_paused(const ALLEGRO_EVENT_QUEUE *queue)
```

Return true if the event queue is paused.

See also: [al\\_pause\\_event\\_queue](#)

Since: 5.1.0

### 4.14 `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](#)

### 4.15 `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](#)

### 4.16 `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](#)

### 4.17 `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](#)

### 4.18 `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](#)



## 4.19 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](#)

## 4.20 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.

`secs` 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](#)

## 4.21 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](#)

## 4.22 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](#)

### 4.23 `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](#)

### 4.24 `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](#)

## 4.25 `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`

## 4.26 `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`

## 4.27 `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`



These functions are declared in the main Allegro header file:

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

## 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.

## 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);
bool       (*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);
int        (*fi_ferror)(ALLEGRO_FILE *f);
const char * (*fi_ferrmsg)(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.

### 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](#)

### 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](#).

### 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](#)

### 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](#)

## 5.7 `al_fclose`

```
bool al_fclose(ALLEGRO_FILE *f)
```

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

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

## 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](#)

## 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](#)

## 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](#)

### 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](#)

### 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](#)

### 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](#)

### 5.14 `al_ferror`

```
int al_ferror(ALLEGRO_FILE *f)
```

Returns non-zero if the error indicator is set on the given file, i.e. there was some sort of previous error. The error code may be system or file interface specific.

See also: [al\\_feof](#), [al\\_fclearerr](#), [al\\_ferrmsg](#)



## 5.15 `al_ferrmsg`

```
const char *al_ferrmsg(ALLEGRO_FILE *f)
```

Return a message string with details about the last error that occurred on the given file handle. The returned string is empty if there was no error, or if the file interface does not provide more information.

See also: [al\\_fclearerr](#), [al\\_ferror](#)

## 5.16 `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](#)

## 5.17 `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](#)

## 5.18 `al_fsize`

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

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

## 5.19 `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](#)

## 5.20 `al_fputc`

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

Write a single byte to the given file. The byte written is the value of `c` cast to an unsigned char.

Parameters:

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

Returns the written byte (cast back to an `int`) on success, or `EOF` on error.

### 5.21 `al_fprintf`

```
int al_fprintf(ALLEGRO_FILE *pfile, const char *format, ...)
```

Writes to a file with stdio “printf”-like formatting. Returns the number of bytes written, or a negative number on error.

See also: [al\\_vfprintf](#)

### 5.22 `al_vfprintf`

```
int al_vfprintf(ALLEGRO_FILE *pfile, const char *format, va_list args)
```

Like `al_fprintf` but takes a `va_list`. Useful for creating your own variations of formatted printing. Returns the number of bytes written, or a negative number on error.

See also: [al\\_fprintf](#)

### 5.23 `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](#)

### 5.24 `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](#)

### 5.25 `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](#)

## 5.26 `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](#)

## 5.27 `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](#)

## 5.28 `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](#)

## 5.29 `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](#)

## 5.30 `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](#)

### 5.31 `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](#)

### 5.32 `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](#)

### 5.33 `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](#)

## 5.34 Standard I/O specific routines

### 5.34.1 `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`

### 5.34.2 `al_make_temp_file`

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

Make a temporary randomly named file given a filename ‘template’.

‘template’ is a string giving the format of the generated filename and should include one or more capital Xs. The Xs are replaced with random alphanumeric characters, produced using a simple pseudo-random number generator only. 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.

## 5.35 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.

### 5.35.1 `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`.

### 5.35.2 `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`

### 5.35.3 `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`.

### 5.35.4 `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`

### 5.35.5 `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`.

## Fixed point math routines

These functions are declared in the main Allegro header file:

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

### 6.1 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.

### 6.2 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](#).

### 6.3 [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](#).

### 6.4 [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](#).

### 6.5 [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](#).

## 6.6 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](#)

## 6.7 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](#).

## 6.8 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](#).

## 6.9 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](#).

## 6.10 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](#).

## 6.11 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](#).

## 6.12 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.

### 6.12.1 al\_fixtorad\_r

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

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

Example:

```
al_fixed rad_angle, binary_angle;

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

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

See also: [al\\_fixmul](#), [al\\_radtofix\\_r](#).

### 6.12.2 al\_radtofix\_r

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

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

Example:

```
al_fixed rad_angle, binary_angle;
...
binary_angle = al_fixmul(rad_angle, radtofix_r);
```

See also: [al\\_fixmul](#), [al\\_fixtorad\\_r](#).

### 6.12.3 al\_fixsin

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

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

Example:

```
al_fixed angle;
int result;

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

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

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

### 6.12.4 al\_fixcos

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

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

Example:

```

al_fixed angle;
float result;

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

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

```

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

### 6.12.5 al\_fixtan

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

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

Example:

```

al_fixed angle, res_a, res_b;
float dif;

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

```

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

### 6.12.6 al\_fixasin

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

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

Example:

```

float angle;
al_fixed val;

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

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

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

```

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

### 6.12.7 `al_fixacos`

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

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

Example:

```
al_fixed result;

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

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

### 6.12.8 `al_fixatan`

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

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

Example:

```
al_fixed result;

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

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

### 6.12.9 `al_fixatan2`

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

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

Example:

```
al_fixed result;

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

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

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

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

#### 6.12.10 `al_fixsqrt`

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

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

#### 6.12.11 `al_fixhypot`

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

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





## 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).

### 7.1 ALLEGRO\_FS\_ENTRY

```
typedef struct ALLEGRO_FS_ENTRY ALLEGRO_FS_ENTRY;
```

Opaque filesystem entry object. Represents a file or a directory (check with [al\\_get\\_fs\\_entry\\_mode](#)). There are no user accessible member variables.

### 7.2 ALLEGRO\_FILE\_MODE

```
typedef enum ALLEGRO_FILE_MODE
```

Filesystem modes/types

- ALLEGRO\_FILEMODE\_READ - Readable
- ALLEGRO\_FILEMODE\_WRITE - Writable
- ALLEGRO\_FILEMODE\_EXECUTE - Executable
- ALLEGRO\_FILEMODE\_HIDDEN - Hidden
- ALLEGRO\_FILEMODE\_ISFILE - Regular file
- ALLEGRO\_FILEMODE\_ISDIR - Directory

### 7.3 al\_create\_fs\_entry

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

Creates an [ALLEGRO\\_FS\\_ENTRY](#) object pointing to path on the filesystem. 'path' can be a file or a directory and must not be NULL.

### 7.4 al\_destroy\_fs\_entry

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

Destroys a fs entry handle. The file or directory represented by it is not destroyed. If the entry was opened, it is closed before being destroyed.

Does nothing if passed NULL.

## 7.5 `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 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.

Note: prior to 5.1.5 it was written: "... the path will not be an absolute path if the entry wasn't created from an absolute path". This is no longer true.

## 7.6 `al_update_fs_entry`

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

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

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

See also: `al_get_errno`, `al_get_fs_entry_atime`, `al_get_fs_entry_ctime`, `al_get_fs_entry_mode`

## 7.7 `al_get_fs_entry_mode`

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

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

See also: `al_get_errno`, `ALLEGRO_FILE_MODE`

## 7.8 `al_get_fs_entry_atime`

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

Returns the time in seconds 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`

## 7.9 `al_get_fs_entry_ctime`

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

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

See also: `al_get_fs_entry_atime`, `al_get_fs_entry_mtime`, `al_update_fs_entry`

## 7.10 `al_get_fs_entry_mtime`

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

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

See also: `al_get_fs_entry_atime`, `al_get_fs_entry_ctime`, `al_update_fs_entry`

## 7.11 `al_get_fs_entry_size`

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

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

See also: [al\\_update\\_fs\\_entry](#)

## 7.12 `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 occurred. Error is indicated in Allegro's `errno`.

See also: [al\\_filename\\_exists](#)

## 7.13 `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's `errno`.

See also: [al\\_filename\\_exists](#)

## 7.14 `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](#)

## 7.15 `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](#)

## 7.16 Directory functions

### 7.16.1 `al_open_directory`

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

Opens a directory entry object. You must call this before using [al\\_read\\_directory](#) 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](#)

### 7.16.2 `al_read_directory`

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

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

Returns NULL if there are no more entries or if an error occurs. Call [al\\_destroy\\_fs\\_entry](#) on the returned entry when you are done with it.

This function will ignore any files or directories named `.` or `..` which may exist on certain platforms and may signify the current and the parent directory.

See also: [al\\_open\\_directory](#), [al\\_close\\_directory](#)

### 7.16.3 `al_close_directory`

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

Closes a previously opened directory entry object.

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

See also: [al\\_open\\_directory](#), [al\\_read\\_directory](#)

### 7.16.4 `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](#)

### 7.16.5 `al_change_directory`

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

Changes the current working directory to 'path'.

Returns true on success, false on error.

### 7.16.6 `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](#)

### 7.16.7 `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](#)

### 7.16.8 ALLEGRO\_FOR\_EACH\_FS\_ENTRY\_RESULT

```
typedef enum ALLEGRO_FOR_EACH_FS_ENTRY_RESULT {
```

Return values for the callbacks of `al_for_each_fs_entry` and for that function itself.

- `ALLEGRO_FOR_EACH_FS_ENTRY_ERROR` - An error occurred.
- `ALLEGRO_FOR_EACH_FS_ENTRY_OK` - Continue normally and recurse into directories.
- `ALLEGRO_FOR_EACH_FS_ENTRY_SKIP` - Continue but do NOT recursively descend.
- `ALLEGRO_FOR_EACH_FS_ENTRY_STOP` - Stop iterating and return.

See also: `al_for_each_fs_entry`

Since: 5.1.9

### 7.16.9 al\_for\_each\_fs\_entry

```
int al_for_each_fs_entry(ALLEGRO_FS_ENTRY *dir,
                        int (*callback)(ALLEGRO_FS_ENTRY *dir, void *extra),
                        void *extra)
```

This function takes the `ALLEGRO_FS_ENTRY` `dir`, which should represent a directory, and looks for any other file system entries that are in it. This function will then call the callback function `callback` once for every filesystem entry in the directory `dir`.

The callback `callback` must be of type `int callback(ALLEGRO_FS_ENTRY * entry, void * extra)`. The callback will be called with a pointer to an `ALLEGRO_FS_ENTRY` that matches one file or directory in `dir`, and the pointer passed in the `extra` parameter to `al_for_each_fs_entry`.

When `callback` returns `ALLEGRO_FOR_EACH_FS_ENTRY_STOP` or `ALLEGRO_FOR_EACH_FS_ENTRY_ERROR`, iteration will stop immediately and `al_for_each_fs_entry` will return the value the callback returned.

When `callback` returns `ALLEGRO_FOR_EACH_FS_ENTRY_OK` iteration will continue normally, and if the `[ALLEGRO_FS_ENTRY]` parameter of `callback` is a directory, `al_for_each_fs_entry` will call itself on that directory. Therefore the function will recursively descend into that directory.

However, when `callback` returns `ALLEGRO_FOR_EACH_FS_ENTRY_SKIP` iteration will continue, but `al_for_each_fs_entry` will NOT recurse into the `[ALLEGRO_FS_ENTRY]` parameter of `callback` even if it is a directory.

This function will skip any files or directories named `.` or `..` which may exist on certain platforms and may signify the current and the parent directory. The callback will not be called for files or directories with such a name.

Returns `ALLEGRO_FOR_EACH_FS_ENTRY_OK` if successful, or `ALLEGRO_FOR_EACH_FS_ENTRY_ERROR` if something went wrong in processing the directory. In that case it will use `al_set_errno` to indicate the type of error which occurred. This function returns `ALLEGRO_FOR_EACH_FS_ENTRY_STOP` in case iteration was stopped by making `callback` return that value. In this case, `al_set_errno` will not be used.

See also: `ALLEGRO_FOR_EACH_FS_ENTRY_RESULT`

Since: 5.1.9

## 7.17 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).

### 7.17.1 ALLEGRO\_FS\_INTERFACE

```
typedef struct ALLEGRO_FS_INTERFACE ALLEGRO_FS_INTERFACE;
```

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

```
ALLEGRO_FS_ENTRY * fs_create_entry    (const char *path);
void               fs_destroy_entry   (ALLEGRO_FS_ENTRY *e);
const 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);
```

### 7.17.2 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](#).

### 7.17.3 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](#).

### 7.17.4 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](#).

## Fullscreen modes

These functions are declared in the main Allegro header file:

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

### 8.1 ALLEGRO\_DISPLAY\_MODE

```
typedef struct ALLEGRO_DISPLAY_MODE
```

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

```
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](#)

### 8.2 al\_get\_display\_mode

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

Retrieves a fullscreen 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](#)

### 8.3 al\_get\_num\_display\_modes

```
int al_get_num_display_modes(void)
```

Get the number of available fullscreen display modes for the current set of display parameters. This will use the values set with [al\\_set\\_new\\_display\\_refresh\\_rate](#), 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](#)





## Graphics routines

These functions are declared in the main Allegro header file:

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

### 9.1 Colors

#### 9.1.1 ALLEGRO\_COLOR

```
typedef struct ALLEGRO_COLOR ALLEGRO_COLOR;
```

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

#### 9.1.2 al\_map\_rgb

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

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

See also: `al_map_rgba`, `al_map_rgba_f`, `al_map_rgb_f`

#### 9.1.3 al\_map\_rgb\_f

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

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

See also: `al_map_rgba`, `al_map_rgb`, `al_map_rgba_f`

#### 9.1.4 al\_map\_rgba

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

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

By default Allegro uses pre-multiplied alpha for transparent blending of bitmaps and primitives (see `al_load_bitmap_flags` for a discussion of that feature). This means that if you want to tint a bitmap or primitive to be transparent you need to multiply the color components by the alpha components when you pass them to this function. For example:

```
int r = 255;
int g = 0;
int b = 0;
int a = 127;
ALLEGRO_COLOR c = al_map_rgba(r * a / 255, g * a / 255, b * a / 255, a);
/* Draw the bitmap tinted red and half-transparent. */
al_draw_tinted_bitmap(bmp, c, 0, 0, 0);
```

See also: [al\\_map\\_rgb](#), [al\\_map\\_rgba\\_f](#), [al\\_map\\_rgb\\_f](#)

### 9.1.5 al\_map\_rgba\_f

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

Convert r, g, b, a (ranging from 0.0f-1.0f) into an [ALLEGRO\\_COLOR](#).

By default Allegro uses pre-multiplied alpha for transparent blending of bitmaps and primitives (see [al\\_load\\_bitmap\\_flags](#) for a discussion of that feature). This means that if you want to tint a bitmap or primitive to be transparent you need to multiply the color components by the alpha components when you pass them to this function. For example:

```
float r = 1;
float g = 0;
float b = 0;
float a = 0.5;
ALLEGRO_COLOR c = al_map_rgba_f(r * a, g * a, b * a, a);
/* Draw the bitmap tinted red and half-transparent. */
al_draw_tinted_bitmap(bmp, c, 0, 0, 0);
```

See also: [al\\_map\\_rgba](#), [al\\_map\\_rgb](#), [al\\_map\\_rgb\\_f](#)

### 9.1.6 al\_unmap\_rgb

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

Retrieves components of an [ALLEGRO\\_COLOR](#), ignoring alpha. Components will range from 0-255.

See also: [al\\_unmap\\_rgba](#), [al\\_unmap\\_rgba\\_f](#), [al\\_unmap\\_rgb\\_f](#)

### 9.1.7 al\_unmap\_rgb\_f

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

Retrieves components of an [ALLEGRO\\_COLOR](#), ignoring alpha. Components will range from 0.0f-1.0f.

See also: [al\\_unmap\\_rgba](#), [al\\_unmap\\_rgb](#), [al\\_unmap\\_rgba\\_f](#)

### 9.1.8 al\_unmap\_rgba

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

Retrieves components of an [ALLEGRO\\_COLOR](#). Components will range from 0-255.

See also: [al\\_unmap\\_rgb](#), [al\\_unmap\\_rgba\\_f](#), [al\\_unmap\\_rgb\\_f](#)

### 9.1.9 al\_unmap\_rgba\_f

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

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

See also: [al\\_unmap\\_rgba](#), [al\\_unmap\\_rgb](#), [al\\_unmap\\_rgb\\_f](#)

## 9.2 Locking and pixel formats

### 9.2.1 ALLEGRO\_LOCKED\_REGION

```
typedef struct ALLEGRO_LOCKED_REGION ALLEGRO_LOCKED_REGION;
```

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

```
typedef struct ALLEGRO_LOCKED_REGION {
    void *data;
    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. For blocked formats, this points to the leftmost block of the first row of blocks.
- *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). For blocked formats, 'row' refers to the row of blocks, not of pixels.
- *pixel\_size* is the number of bytes used to represent a single block of pixels for the pixel format of this locked region. For most formats (and historically, this used to be true for all formats), this is just the size of a single pixel, but for blocked pixel formats this value is different.

See also: [al\\_lock\\_bitmap](#), [al\\_lock\\_bitmap\\_region](#), [al\\_unlock\\_bitmap](#), [ALLEGRO\\_PIXEL\\_FORMAT](#)

### 9.2.2 ALLEGRO\_PIXEL\_FORMAT

```
typedef enum ALLEGRO_PIXEL_FORMAT
```

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

*Note:*

The pixel format is independent of endianness. That is, in the above example you can always get the red component with

```
(pixel & 0x00ff0000) >> 16
```

But you can *not* rely on this code:

```
*(pixel + 2)
```

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

Also note that Allegro's naming is different from OpenGL naming here, where a format of GL\_RGBA8 merely defines the component order and the exact layout including endianness treatment is specified separately. Usually GL\_RGBA8 will correspond to ALLEGRO\_PIXEL\_ABGR\_8888 though on little endian systems, so care must be taken (note the reversal of RGBA <-> ABGR).

The only exception to this ALLEGRO\_PIXEL\_FORMAT\_ABGR\_8888\_LE which will always have the components as 4 bytes corresponding to red, green, blue and alpha, in this order, independent of the endianness.

Some of the pixel formats represent compressed bitmap formats. Compressed bitmaps take up less space in the GPU memory than bitmaps with regular (uncompressed) pixel formats. This smaller footprint means that you can load more resources into GPU memory, and they will be drawn somewhat faster. The compression is lossy, however, so it is not appropriate for all graphical styles: it tends to work best for images with smooth color gradations. It is possible to compress bitmaps at runtime by passing the appropriate bitmap format in `al_set_new_bitmap_format` and then creating, loading, cloning or converting a non-compressed bitmap. This, however, is not recommended as the compression quality differs between different GPU drivers. It is recommended to compress these bitmaps ahead of time using external tools and then load them compressed.

Unlike regular pixel formats, compressed pixel formats are not laid out in memory one pixel row at a time. Instead, the bitmap is subdivided into rectangular blocks of pixels that are then laid out in block rows. This means that regular locking functions cannot use compressed pixel formats as the destination format. Instead, you can use the blocked versions of the bitmap locking functions which do support these formats.

It is not recommended to use compressed bitmaps as target bitmaps, as that operation cannot be hardware accelerated. Due to proprietary algorithms used, it is typically impossible to create compressed memory bitmaps.

- 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
- `ALLEGRO_PIXEL_FORMAT_SINGLE_CHANNEL_8` - A single 8-bit channel. A pixel value maps onto the red channel when displayed, but it is undefined how it maps onto green, blue and alpha channels. When drawing to bitmaps of this format, only the red channel is taken into account. Allegro may have to use fallback methods to render to bitmaps of this format. This pixel format is mainly intended for storing the color indices of an indexed (paletted) image, usually in conjunction with a pixel shader that maps indices to RGBA values. Since 5.1.2.
- `ALLEGRO_PIXEL_FORMAT_COMPRESSED_RGBA_DXT1` - Compressed using the DXT1 compression algorithm. Each 4x4 pixel block is encoded in 64 bytes, resulting in 6-8x compression ratio. Only a single bit of alpha per pixel is supported. Since 5.1.9.
- `ALLEGRO_PIXEL_FORMAT_COMPRESSED_RGBA_DXT3` - Compressed using the DXT3 compression algorithm. Each 4x4 pixel block is encoded in 128 bytes, resulting in 4x compression ratio. This format supports sharp alpha transitions. Since 5.1.9.
- `ALLEGRO_PIXEL_FORMAT_COMPRESSED_RGBA_DXT5` - Compressed using the DXT5 compression algorithm. Each 4x4 pixel block is encoded in 128 bytes, resulting in 4x compression ratio. This format supports smooth alpha transitions. Since 5.1.9.

See also: [al\\_set\\_new\\_bitmap\\_format](#), [al\\_get\\_bitmap\\_format](#)

### 9.2.3 `al_get_pixel_size`

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

Return the number of bytes that a pixel of the given format occupies. For blocked pixel formats (e.g. compressed formats), this returns 0.

See also: [ALLEGRO\\_PIXEL\\_FORMAT](#), [al\\_get\\_pixel\\_format\\_bits](#)

### 9.2.4 `al_get_pixel_format_bits`

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

Return the number of bits that a pixel of the given format occupies. For blocked pixel formats (e.g. compressed formats), this returns 0.

See also: [ALLEGRO\\_PIXEL\\_FORMAT](#), [al\\_get\\_pixel\\_size](#)

### 9.2.5 `al_get_pixel_block_size`

```
int al_get_pixel_block_size(int format)
```

Return the number of bytes that a block of pixels with this format occupies.

Since: 5.1.9.

See also: [ALLEGRO\\_PIXEL\\_FORMAT](#), [al\\_get\\_pixel\\_block\\_width](#), [al\\_get\\_pixel\\_block\\_height](#)

### 9.2.6 `al_get_pixel_block_width`

```
int al_get_pixel_block_width(int format)
```

Return the width of the the pixel block for this format.

Since: 5.1.9.

See also: [ALLEGRO\\_PIXEL\\_FORMAT](#), [al\\_get\\_pixel\\_block\\_size](#), [al\\_get\\_pixel\\_block\\_height](#)

### 9.2.7 `al_get_pixel_block_height`

```
int al_get_pixel_block_height(int format)
```

Return the height of the the pixel block for this format.

Since: 5.1.9.

See also: [ALLEGRO\\_PIXEL\\_FORMAT](#), [al\\_get\\_pixel\\_block\\_size](#), [al\\_get\\_pixel\\_block\\_width](#)

### 9.2.8 `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. This function also returns NULL if the format is a compressed format.

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. If the bitmap format is compressed, using `ALLEGRO_PIXEL_FORMAT_ANY` will choose an implementation defined non-compressed format.

*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](#), [al\\_lock\\_bitmap\\_blocked](#), [al\\_lock\\_bitmap\\_region\\_blocked](#)

### 9.2.9 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 video bitmap, only that area of the texture will be updated when it is unlocked. Locking only the region you intend to modify will be faster than locking the whole bitmap.

*Note:* Using the `ALLEGRO_LOCK_WRITEONLY` with a blocked pixel format (i.e. formats for which [al\\_get\\_pixel\\_block\\_width](#) or [al\\_get\\_pixel\\_block\\_height](#) do not return 1) requires you to have the region be aligned to the block width for optimal performance. If it is not, then the function will have to lock the region with the `ALLEGRO_LOCK_READWRITE` instead in order to pad this region with valid data.

See also: [ALLEGRO\\_LOCKED\\_REGION](#), [ALLEGRO\\_PIXEL\\_FORMAT](#), [al\\_unlock\\_bitmap](#)

### 9.2.10 al\_unlock\_bitmap

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

Unlock a previously locked bitmap or bitmap region. If the bitmap is a video 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](#), [al\\_lock\\_bitmap\\_blocked](#), [al\\_lock\\_bitmap\\_region\\_blocked](#)

### 9.2.11 al\_lock\_bitmap\_blocked

```
ALLEGRO_LOCKED_REGION *al_lock_bitmap_blocked(ALLEGRO_BITMAP *bitmap,
    int flags)
```

Like [al\\_lock\\_bitmap](#), but allows locking bitmaps with a blocked pixel format (i.e. a format for which [al\\_get\\_pixel\\_block\\_width](#) or [al\\_get\\_pixel\\_block\\_height](#) do not return 1) in that format. To that end, this function also does not allow format conversion. For bitmap formats with a block size of 1, this function is identical to calling `al_lock_bitmap(bmp, al_get_bitmap_format(bmp), flags)`.

*Note:* Currently there are no drawing functions that work when the bitmap is locked with a compressed format. [al\\_get\\_pixel](#) will also not work.

Since: 5.1.9

See also: [al\\_lock\\_bitmap](#), [al\\_lock\\_bitmap\\_region\\_blocked](#)

### 9.2.12 al\_lock\_bitmap\_region\_blocked

```
ALLEGRO_LOCKED_REGION *al_lock_bitmap_region_blocked(ALLEGRO_BITMAP *bitmap,
    int x_block, int y_block, int width_block, int height_block, int flags)
```

Like [al\\_lock\\_bitmap\\_blocked](#), but allows locking a sub-region, for performance. Unlike [al\\_lock\\_bitmap\\_region](#) the region specified in terms of blocks and not pixels.

Since: 5.1.9

See also: [al\\_lock\\_bitmap\\_region](#), [al\\_lock\\_bitmap\\_blocked](#)

## 9.3 Bitmap creation

### 9.3.1 ALLEGRO\_BITMAP

```
typedef struct ALLEGRO_BITMAP ALLEGRO_BITMAP;
```

Abstract type representing a bitmap (2D image).

### 9.3.2 al\_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. The minimum size that will work on all platforms is 32 by 32.

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.

If you create a bitmap without ALLEGRO\_MEMORY\_BITMAP set but there is no current display, a temporary memory bitmap will be created instead. You can later convert all such bitmap to video bitmap and assign to a display by calling [al\\_convert\\_bitmaps](#).

See also: [al\\_set\\_new\\_bitmap\\_format](#), [al\\_set\\_new\\_bitmap\\_flags](#), [al\\_clone\\_bitmap](#), [al\\_create\\_sub\\_bitmap](#), [al\\_convert\\_bitmaps](#), [al\\_destroy\\_bitmap](#)

### 9.3.3 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](#)



### 9.3.4 `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_convert_bitmap`

### 9.3.5 `al_convert_bitmap`

```
void al_convert_bitmap(ALLEGRO_BITMAP *bitmap)
```

Converts the bitmap to the current bitmap flags and format. The bitmap will be as if it was created anew with `al_create_bitmap` but retain its contents. All of this bitmap's sub-bitmaps are also converted.

If this bitmap is a sub-bitmap, then it, its parent and all the sibling sub-bitmaps are also converted.

Since: 5.1.0

See also: `al_create_bitmap`, `al_set_new_bitmap_format`, `al_set_new_bitmap_flags`, `al_clone_bitmap`

### 9.3.6 `al_convert_bitmaps`

```
void al_convert_bitmaps(void)
```

If you create a bitmap when there is no current display (for example because you have not called `al_create_display` in the current thread) and are using the `ALLEGRO_CONVERT_BITMAP` bitmap flag (which is set by default) then the bitmap will be created successfully, but as a memory bitmap. This function converts all such bitmaps to proper video bitmaps belonging to the current display.

Note that video bitmaps get automatically converted back to memory bitmaps when the last display is destroyed.

Since: 5.1.0

See also: `al_convert_bitmap`, `al_create_bitmap`

### 9.3.7 `al_destroy_bitmap`

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

Destroys the given bitmap, freeing all resources used by it. This function does nothing if the bitmap argument is NULL.

As a convenience, if the calling thread is currently targets the bitmap then the bitmap will be untargeted first. The new target bitmap is unspecified. (since: 5.0.10, 5.1.6)

Otherwise, it is an error to destroy a bitmap while it (or a sub-bitmap) is the target bitmap of any thread.

See also: `al_create_bitmap`

### 9.3.8 `al_get_new_bitmap_flags`

```
int al_get_new_bitmap_flags(void)
```

Returns the flags used for newly created bitmaps.

See also: `al_set_new_bitmap_flags`

### 9.3.9 `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`

### 9.3.10 `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\_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\_VIDEO\_BITMAP**

Creates a bitmap that resides in the video card memory. These types of bitmaps receive the greatest benefit from hardware acceleration.

*Note:* Creating a video bitmap will fail if there is no current display or the current display driver cannot create the bitmap. The latter will happen if for example the format or dimensions are not supported.

*Note:* Bitmaps created with this flag will be converted to memory bitmaps when the last display is destroyed. In most cases it is therefore easier to use the `ALLEGRO_CONVERT_BITMAP` flag instead.

#### **ALLEGRO\_CONVERT\_BITMAP**

This is the default. It will try to create a video bitmap and if that fails create a memory bitmap. Bitmaps created with this flag when there is no active display will be converted to video bitmaps next time a display is created. They also will remain video bitmaps if the last display is destroyed and then another is created again. Since 5.1.0.

*Note:* You can combine this flag with `ALLEGRO_MEMORY_BITMAP` or `ALLEGRO_VIDEO_BITMAP` to force the initial type (and fail in the latter case if no video bitmap can be created) - but usually neither of those combinations is very useful.

You can use the display option `ALLEGRO_AUTO_CONVERT_BITMAPS` to control which displays will try to auto-convert bitmaps.

#### **ALLEGRO\_FORCE\_LOCKING**

Does nothing since 5.1.8. Kept for backwards compatibility only.

#### **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.

See also: [al\\_get\\_new\\_bitmap\\_flags](#), [al\\_get\\_bitmap\\_flags](#)

**9.3.11 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](#)

**9.3.12 al\_set\_new\_bitmap\_format**

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

Sets the pixel format ([ALLEGRO\\_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](#)

**9.4 Bitmap properties****9.4.1 al\_get\_bitmap\_flags**

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

Return the flags used to create the bitmap.

See also: [al\\_set\\_new\\_bitmap\\_flags](#)

**9.4.2 al\_get\_bitmap\_format**

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

Returns the pixel format of a bitmap.

See also: [ALLEGRO\\_PIXEL\\_FORMAT](#), [al\\_set\\_new\\_bitmap\\_flags](#)

**9.4.3 al\_get\_bitmap\_height**

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

Returns the height of a bitmap in pixels.

#### 9.4.4 `al_get_bitmap_width`

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

Returns the width of a bitmap in pixels.

#### 9.4.5 `al_get_pixel`

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

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

See also: [ALLEGRO\\_COLOR](#), [al\\_put\\_pixel](#), [al\\_lock\\_bitmap](#)

#### 9.4.6 `al_is_bitmap_locked`

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

Returns whether or not a bitmap is already locked.

See also: [al\\_lock\\_bitmap](#), [al\\_lock\\_bitmap\\_region](#), [al\\_unlock\\_bitmap](#)

#### 9.4.7 `al_is_compatible_bitmap`

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

D3D and OpenGL allow sharing a texture in a way so it can be used for multiple windows. Each [ALLEGRO\\_BITMAP](#) created with [al\\_create\\_bitmap](#) 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 (bitting in the current display will be slow).

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

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

#### 9.4.8 `al_is_sub_bitmap`

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

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

See also: [al\\_create\\_sub\\_bitmap](#), [al\\_get\\_parent\\_bitmap](#)

#### 9.4.9 `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. This function always returns the real bitmap, and never a sub-bitmap. This might NOT match what was passed to [al\\_create\\_sub\\_bitmap](#). Consider this code, for instance:

```
ALLEGRO_BITMAP* a = al_create_bitmap(512, 512);
ALLEGRO_BITMAP* b = al_create_sub_bitmap(a, 128, 128, 256, 256);
ALLEGRO_BITMAP* c = al_create_sub_bitmap(b, 64, 64, 128, 128);
ASSERT(al_get_parent_bitmap(b) == a && al_get_parent_bitmap(c) == a);
```

The assertion will pass because only a is a real bitmap, and both b and c are its sub-bitmaps.

Since: 5.0.6, 5.1.2

See also: [al\\_create\\_sub\\_bitmap](#), [al\\_is\\_sub\\_bitmap](#)

## 9.5 Drawing operations

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

### 9.5.1 al\_clear\_to\_color

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

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

See also: [ALLEGRO\\_COLOR](#), [al\\_set\\_clipping\\_rectangle](#), [al\\_clear\\_depth\\_buffer](#)

### 9.5.2 al\_clear\_depth\_buffer

```
void al_clear_depth_buffer(float z)
```

Clear the depth buffer (confined by the clipping rectangle) to the given value. A depth buffer is only available if it was requested with [al\\_set\\_new\\_display\\_option](#) and the requirement could be met by the [al\\_create\\_display](#) call creating the current display. Operations involving the depth buffer are also affected by [al\\_set\\_render\\_state](#).

Since: 5.1.2

See also: [al\\_clear\\_to\\_color](#), [al\\_set\\_clipping\\_rectangle](#), [al\\_set\\_render\\_state](#), [al\\_set\\_new\\_display\\_option](#)

### 9.5.3 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](#)

### 9.5.4 `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(0.5, 0.5, 0.5, 0.5), x, y, 0);
```

The above will draw the bitmap 50% transparently (r/g/b values need to be pre-multiplied with the alpha component with the default blend mode).

```
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.

See [al\\_draw\\_bitmap](#) for a note on restrictions on which bitmaps can be drawn where.

See also: [al\\_draw\\_bitmap](#)

### 9.5.5 `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 [al\\_draw\\_bitmap](#) for a note on restrictions on which bitmaps can be drawn where.

See also: [al\\_draw\\_bitmap](#), [al\\_draw\\_scaled\\_bitmap](#), [al\\_draw\\_rotated\\_bitmap](#), [al\\_draw\\_scaled\\_rotated\\_bitmap](#)

### 9.5.6 `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 [al\\_draw\\_bitmap](#) for a note on restrictions on which bitmaps can be drawn where.

See also: [al\\_draw\\_tinted\\_bitmap](#)

### 9.5.7 `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 (that is, the pixel's position is transformed, but its size is unaffected - it remains a pixel). 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 add-on.

- 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 add-on documentation for details on how to control exactly where the pixel is drawn.

See also: `ALLEGRO_COLOR`, `al_put_pixel`

### 9.5.8 `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 (radians)
- 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 `al_draw_bitmap` for a note on restrictions on which bitmaps can be drawn where.

See also: `al_draw_bitmap`, `al_draw_bitmap_region`, `al_draw_scaled_bitmap`, `al_draw_scaled_rotated_bitmap`

### 9.5.9 `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 [al\\_draw\\_bitmap](#) for a note on restrictions on which bitmaps can be drawn where.

See also: [al\\_draw\\_tinted\\_bitmap](#)

### 9.5.10 `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 (radians)
- flags - same as for [al\\_draw\\_bitmap](#)

See [al\\_draw\\_bitmap](#) for a note on restrictions on which bitmaps can be drawn where.

See also: [al\\_draw\\_bitmap](#), [al\\_draw\\_bitmap\\_region](#), [al\\_draw\\_scaled\\_bitmap](#), [al\\_draw\\_rotated\\_bitmap](#)

### 9.5.11 `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 [al\\_draw\\_bitmap](#) for a note on restrictions on which bitmaps can be drawn where.

See also: [al\\_draw\\_tinted\\_bitmap](#)

### 9.5.12 `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);
```

See [al\\_draw\\_bitmap](#) for a note on restrictions on which bitmaps can be drawn where.

Since: 5.0.6, 5.1.0

See also: [al\\_draw\\_tinted\\_bitmap](#)

### 9.5.13 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 [al\\_draw\\_bitmap](#) for a note on restrictions on which bitmaps can be drawn where.

See also: [al\\_draw\\_bitmap](#), [al\\_draw\\_bitmap\\_region](#), [al\\_draw\\_rotated\\_bitmap](#), [al\\_draw\\_scaled\\_rotated\\_bitmap](#),

### 9.5.14 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 [al\\_draw\\_bitmap](#) for a note on restrictions on which bitmaps can be drawn where.

See also: [al\\_draw\\_tinted\\_bitmap](#)

### 9.5.15 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](#)

### 9.5.16 `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](#)

### 9.5.17 `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](#)

### 9.5.18 `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](#)

### 9.5.19 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](#)

### 9.5.20 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](#)

## 9.6 Blending modes

### 9.6.1 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](#)

### 9.6.2 al\_get\_separate\_blender

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

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](#)

### 9.6.3 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):  $s = s.r, s.g, s.b, s.a$ . And this color is drawn to a destination, which already has a color:  $d = d.r, d.g, d.b, d.a$ .

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

- ALLEGRO\_ADD

```
r = d.r * df.r + s.r * sf.r
g = d.g * df.g + s.g * sf.g
b = d.b * df.b + s.b * sf.b
a = d.a * df.a + s.a * sf.a
```

- ALLEGRO\_DEST\_MINUS\_SRC

```
r = d.r * df.r - s.r * sf.r
g = d.g * df.g - s.g * sf.g
b = d.b * df.b - s.b * sf.b
a = d.a * df.a - s.a * sf.a
```

- ALLEGRO\_SRC\_MINUS\_DEST

```
r = s.r * sf.r - d.r * df.r
g = s.g * sf.g - d.g * df.g
b = s.b * sf.b - d.b * df.b
a = s.a * sf.a - d.a * df.a
```

Valid values for the factors sf and df passed to this function are

- ALLEGRO\_ZERO

```
f = 0, 0, 0, 0
```

- ALLEGRO\_ONE

```
f = 1, 1, 1, 1
```

- ALLEGRO\_ALPHA

```
f = s.a, s.a, s.a, s.a
```

- ALLEGRO\_INVERSE\_ALPHA

```
f = 1 - s.a, 1 - s.a, 1 - s.a, 1 - s.a
```

- ALLEGRO\_SRC\_COLOR (since: 5.0.10, 5.1.0)

```
f = s.r, s.g, s.b, s.a
```

- ALLEGRO\_DEST\_COLOR (since: 5.0.10, 5.1.8)

```
f = d.r, d.g, d.b, d.a
```

- ALLEGRO\_INVERSE\_SRC\_COLOR (since: 5.0.10, 5.1.0)

```
f = 1 - s.r, 1 - s.g, 1 - s.b, 1 - s.a
```

- ALLEGRO\_INVERSE\_DEST\_COLOR (since: 5.0.10, 5.1.8)

```
f = 1 - d.r, 1 - d.g, 1 - d.b, 1 - d.a
```

Blending examples:

So for example, to restore the default of using premultiplied alpha blending, you would use:

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

As formula:

```

r = d.r * (1 - s.a) + s.r * 1
g = d.g * (1 - s.a) + s.g * 1
b = d.b * (1 - s.a) + s.b * 1
a = d.a * (1 - s.a) + s.a * 1

```

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);
```

Multiplying source and destination components

```
al_set_blender(ALLEGRO_ADD, ALLEGRO_DEST_COLOR, ALLEGRO_ZERO)
```

As formula:

```

r = d.r * 0 + s.r * d.r
g = d.g * 0 + s.g * d.g
b = d.b * 0 + s.b * d.b
a = d.a * 0 + s.a * d.a

```

See also: [al\\_set\\_separate\\_blender](#), [al\\_get\\_blender](#)

#### 9.6.4 al\_set\_separate\_blender

```

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

```

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](#)

## 9.7 Clipping

### 9.7.1 al\_get\_clipping\_rectangle

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

Gets the clipping rectangle of the target bitmap.

See also: [al\\_set\\_clipping\\_rectangle](#)

### 9.7.2 `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](#)

### 9.7.3 `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

## 9.8 Graphics utility functions

### 9.8.1 `al_convert_mask_to_alpha`

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

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

See also: [ALLEGRO\\_COLOR](#)

## 9.9 Deferred drawing

### 9.9.1 `al_hold_bitmap_drawing`

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

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

### 9.9.2 `al_is_bitmap_drawing_held`

```
bool al_is_bitmap_drawing_held(void)
```

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

See also: [al\\_hold\\_bitmap\\_drawing](#)

## 9.10 Image I/O

### 9.10.1 `al_register_bitmap_loader`

```
bool al_register_bitmap_loader(const char *extension,
                             ALLEGRO_BITMAP *(*loader)(const char *filename, int flags))
```

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](#)

### 9.10.2 `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](#)

### 9.10.3 `al_register_bitmap_loader_f`

```
bool al_register_bitmap_loader_f(const char *extension,
                                 ALLEGRO_BITMAP *(*loader_f)(ALLEGRO_FILE *fp, int flags))
```

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](#)

### 9.10.4 `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](#)

### 9.10.5 `al_load_bitmap`

```
ALLEGRO_BITMAP *al_load_bitmap(const char *filename)
```

Loads an image file into a new `ALLEGRO_BITMAP`. The file type is determined by the extension.

Returns NULL on error.

This is the same as calling `al_load_bitmap_flags` with a flags parameter of 0.

*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_flags`, `al_load_bitmap_f`, `al_register_bitmap_loader`, `al_set_new_bitmap_format`, `al_set_new_bitmap_flags`, `al_init_image_addon`

### 9.10.6 `al_load_bitmap_flags`

```
ALLEGRO_BITMAP *al_load_bitmap_flags(const char *filename, int flags)
```

Loads an image file into a new `ALLEGRO_BITMAP`. The file type is determined by the extension.

Returns NULL on error.

The flags parameter may be a combination of the following constants:

#### `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.

### ALLEGRO\_KEEP\_INDEX

Load the palette indices of 8-bit .bmp and .pcx files instead of the rgb colors. Since 5.1.0.

### ALLEGRO\_KEEP\_BITMAP\_FORMAT

Force the resulting `ALLEGRO_BITMAP` to use the same format as the file.

*This is not yet honoured.*

*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.

Since: 5.1.0

See also: [al\\_load\\_bitmap](#)

## 9.10.7 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 a new `ALLEGRO_BITMAP`. The file type is determined by the passed 'ident' parameter, which is a file name extension including the leading dot. This is the same as calling `al_load_bitmap_flags_f` with 0 for the flags parameter.

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\\_flags\\_f](#), [al\\_load\\_bitmap](#), [al\\_register\\_bitmap\\_loader\\_f](#), [al\\_init\\_image\\_addon](#)

### 9.10.8 `al_load_bitmap_flags_f`

```
ALLEGRO_BITMAP *al_load_bitmap_flags_f(ALLEGRO_FILE *fp, const char *ident,
int flags)
```

Loads an image from an `ALLEGRO_FILE` stream into a new `ALLEGRO_BITMAP`. The file type is determined by the passed ‘ident’ parameter, which is a file name extension including the leading dot. The flags parameter is the same as for [al\\_load\\_bitmap\\_flags](#).

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.

Since: 5.1.0

See also: [al\\_load\\_bitmap\\_f](#), [al\\_load\\_bitmap\\_flags](#)

### 9.10.9 `al_save_bitmap`

```
bool al_save_bitmap(const char *filename, ALLEGRO_BITMAP *bitmap)
```

Saves an `ALLEGRO_BITMAP` to an image file. The file type is determined by the extension.

Returns true on success, false on error.

*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](#)

### 9.10.10 `al_save_bitmap_f`

```
bool al_save_bitmap_f(ALLEGRO_FILE *fp, const char *ident,
ALLEGRO_BITMAP *bitmap)
```

Saves an `ALLEGRO_BITMAP` to an `ALLEGRO_FILE` stream. The file type is determined by the passed ‘ident’ parameter, which is a file name extension including the leading dot.

Returns true on success, false on error. 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](#)

## 9.11 Render State

### 9.11.1 ALLEGRO\_RENDER\_STATE

```
typedef enum ALLEGRO_RENDER_STATE {
```

Possible render states which can be set with [al\\_set\\_render\\_state](#):

#### ALLEGRO\_ALPHA\_TEST

If this is set to 1, the values of `ALLEGRO_ALPHA_FUNCTION` and `ALLEGRO_ALPHA_TEST_VALUE` define a comparison function which is performed for each pixel. Only if it evaluates to true the pixel is written. Otherwise no subsequent processing (like depth test or blending) is performed.

#### ALLEGRO\_ALPHA\_FUNCTION

One of [ALLEGRO\\_RENDER\\_FUNCTION](#), only used when `ALLEGRO_ALPHA_TEST` is 1.

#### ALLEGRO\_ALPHA\_TEST\_VALUE

Only used when `ALLEGRO_ALPHA_TEST` is 1.

#### ALLEGRO\_WRITE\_MASK

This determines how the framebuffer and depthbuffer are updated whenever a pixel is written (in case alpha and/or depth testing is enabled, after all such enabled tests succeed). Depth values are only written if `ALLEGRO_DEPTH_TEST` is 1, in addition to the write mask flag being set.

#### ALLEGRO\_DEPTH\_TEST

If this is set to 1, compare the depth value of any newly written pixels with the depth value already in the buffer, according to `ALLEGRO_DEPTH_FUNCTION`. Allegro primitives with no explicit z coordinate will write a value of 0 into the depth buffer.

#### ALLEGRO\_DEPTH\_FUNCTION

One of [ALLEGRO\\_RENDER\\_FUNCTION](#), only used when `ALLEGRO_DEPTH_TEST` is 1.

Since: 5.1.2

See also: [al\\_set\\_render\\_state](#), [ALLEGRO\\_RENDER\\_FUNCTION](#), [ALLEGRO\\_WRITE\\_MASK\\_FLAGS](#)

### 9.11.2 ALLEGRO\_RENDER\_FUNCTION

```
typedef enum ALLEGRO_RENDER_FUNCTION {
```

Possible functions are:

- `ALLEGRO_RENDER_NEVER`
- `ALLEGRO_RENDER_ALWAYS`
- `ALLEGRO_RENDER_LESS`
- `ALLEGRO_RENDER_EQUAL`
- `ALLEGRO_RENDER_LESS_EQUAL`
- `ALLEGRO_RENDER_GREATER`
- `ALLEGRO_RENDER_NOT_EQUAL`
- `ALLEGRO_RENDER_GREATER_EQUAL`

Since: 5.1.2

See also: [al\\_set\\_render\\_state](#)

### 9.11.3 ALLEGRO\_WRITE\_MASK\_FLAGS

```
typedef enum ALLEGRO_WRITE_MASK_FLAGS {
```

Each enabled bit means the corresponding value is written, a disabled bit means it is not.

- `ALLEGRO_MASK_RED`
- `ALLEGRO_MASK_GREEN`
- `ALLEGRO_MASK_BLUE`
- `ALLEGRO_MASK_ALPHA`
- `ALLEGRO_MASK_DEPTH`
- `ALLEGRO_MASK_RGB` - Same as `RED` | `GREEN` | `BLUE`.
- `ALLEGRO_MASK_RGBA` - Same as `RGB` | `ALPHA`.

Since: 5.1.2

See also: [al\\_set\\_render\\_state](#)

### 9.11.4 `al_set_render_state`

```
void al_set_render_state(ALLEGRO_RENDER_STATE state, int value)
```

Set one of several render attributes; see [ALLEGRO\\_RENDER\\_STATE](#) for details.

This function does nothing if the target bitmap is a memory bitmap.

Since: 5.1.2

See also: [ALLEGRO\\_RENDER\\_STATE](#), [ALLEGRO\\_RENDER\\_FUNCTION](#),  
[ALLEGRO\\_WRITE\\_MASK\\_FLAGS](#)

## Haptic routines

Haptic functions support force feedback and vibration on input devices. These functions are declared in the main Allegro header file:

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

Currently force feedback is fully supported on Linux and on Windows for DirectInput compatible devices. There is also minimal support for Android. It is not yet supported on OSX, iOS, or on Windows for XInput compatible devices.

### 10.1 ALLEGRO\_HAPTIC

```
typedef struct ALLEGRO_HAPTIC ALLEGRO_HAPTIC;
```

This is an abstract data type representing a haptic device that supports force feedback or vibration.

Since: 5.1.8

See also: [al\\_get\\_haptic\\_from\\_joystick](#)

### 10.2 ALLEGRO\_HAPTIC\_CONSTANTS

```
enum ALLEGRO_HAPTIC_CONSTANTS
```

This enum contains flags that are used to define haptic effects and capabilities. If the flag is set in the return value of [al\\_get\\_haptic\\_capabilities](#), it means the device supports the given effect. The value of these flags should be set into a [ALLEGRO\\_HAPTIC\\_EFFECT](#) struct to determine what kind of haptic effect should be caused when it is played.

- ALLEGRO\_HAPTIC\_RUMBLE - simple vibration effects
- ALLEGRO\_HAPTIC\_PERIODIC - periodic, wave-form effects
- ALLEGRO\_HAPTIC\_CONSTANT - constant effects
- ALLEGRO\_HAPTIC\_SPRING - spring effects
- ALLEGRO\_HAPTIC\_FRICTION - friction effects
- ALLEGRO\_HAPTIC\_DAMPER - damper effects
- ALLEGRO\_HAPTIC\_INERTIA - inertia effects
- ALLEGRO\_HAPTIC\_RAMP - ramp effects
- ALLEGRO\_HAPTIC\_SQUARE - square wave periodic effect
- ALLEGRO\_HAPTIC\_TRIANGLE - triangle wave periodic effect
- ALLEGRO\_HAPTIC\_SINE - sine wave periodic effect
- ALLEGRO\_HAPTIC\_SAW\_UP - upwards saw wave periodic effect
- ALLEGRO\_HAPTIC\_SAW\_DOWN - downwards saw wave periodic effect
- ALLEGRO\_HAPTIC\_CUSTOM - custom wave periodic effect

- ALLEGRO\_HAPTIC\_GAIN - the haptic device supports gain setting
- ALLEGRO\_HAPTIC\_ANGLE - the haptic device supports angle coordinates
- ALLEGRO\_HAPTIC\_RADIUS - the haptic device supports radius coordinates
- ALLEGRO\_HAPTIC\_AZIMUTH - the haptic device supports azimuth coordinates

Since: 5.1.8

See also: [al\\_get\\_haptic\\_capabilities](#), [ALLEGRO\\_HAPTIC\\_EFFECT](#)

### 10.3 ALLEGRO\_HAPTIC\_EFFECT

**struct** ALLEGRO\_HAPTIC\_EFFECT

This struct models a particular haptic or vibration effect. It needs to be filled correctly in and uploaded to a haptic device, before the device can play it back.

*Fields:*

**type** The type of the haptic effect. May be one of the ALLEGRO\_HAPTIC\_CONSTANTS constants between or equal to ALLEGRO\_HAPTIC\_RUMBLE and ALLEGRO\_HAPTIC\_RAMP.

- If type is set to ALLEGRO\_HAPTIC\_RUMBLE, then the effect is a simple “rumble” or vibration effect that shakes the device. In some cases, such as on a mobile platform, the whole device may shake.
- If type is set to ALLEGRO\_HAPTIC\_PERIODIC, the effect is a shake or vibration of which the intensity is a periodic wave form.
- If type is set to ALLEGRO\_HAPTIC\_CONSTANT, the effect is a constant pressure, motion or push-back in a certain direction of the axes of the device.
- If type is set to ALLEGRO\_HAPTIC\_SPRING, the effect is a springy kind of resistance against motion of the axes of the haptic device.
- If type is set to ALLEGRO\_HAPTIC\_FRICTION, the effect is a friction kind of resistance against motion of the axes of the haptic device.
- If type is set to ALLEGRO\_HAPTIC\_DAMPER, the effect is a damper kind of resistance against motion of the axes of the haptic device.
- If type is set to ALLEGRO\_HAPTIC\_INERTIA, the effect causes inertia or slowness of motions on the axes of the haptic device.
- If type is set to ALLEGRO\_HAPTIC\_RAMP, the effect causes a pressure or push-back that ramps up or down depending on the position of the axis.

#### direction

The direction of location in 3D space where the effect should be played. Allegro haptic devices model directions in 3D space using spherical coordinates. However, the haptic device may not support localized effects, or may not support all coordinate components.

In Allegro’s coordinate system, the value in `direction.angle` determines the planar angle between the effect and the direction of the user who holds the device, expressed in radians. This angle increases clockwise away from the user. So, an effect with an angle 0.0 takes place in the direction of the user of the haptic device, an angle of  $\pi/2$  is to the left of the user, an angle of  $\pi$  means the direction away from the user, and an angle of  $3\pi/2$  means to the right of the user.

If [al\\_get\\_haptic\\_capabilities](#) has the flag ALLEGRO\_HAPTIC\_ANGLE set, then setting `direction.angle` is supported. Otherwise, it is unsupported, and you should set it to 0.

The value in `direction.radius` is a relative value between 0.0 and 1.0 that determines the relative distance from the center of the haptic device at which the effect will play back. A value of 0 means that the effect should play back at the center of the device. A value of 1.0 means that the effect should play back away from the center as far as is possible.

If `al_get_haptic_capabilities` has the flag `ALLEGRO_HAPTIC_RADIUS` set, then setting `direction.radius` is supported. Otherwise, it is unsupported, and you should set it to 0.

The value in `direction.azimuth` determines the elevation angle between the effect and the plane in which the user is holding the device, expressed in radians. An effect with an azimuth 0.0 plays back in the plane in which the user is holding the device, an azimuth  $+\pi/2$  means the effect plays back vertically above the user plane, and an azimuth  $-\pi/2$  means the effect plays back vertically below the user plane.

If `al_get_haptic_capabilities` has the flag `ALLEGRO_HAPTIC_AZIMUTH` set, then setting `direction.azimuth` is supported. Otherwise, it is unsupported, and you should set it to 0.

### replay

Determines how the effect should be played back. `replay.length` is the duration in seconds of the effect, and `replay.delay` is the time in seconds that the effect playback should be delayed when playback is started with `al_play_haptic_effect`.

### data

Determines in detail the parameters of the haptic effect to play back.

If type is set to `ALLEGRO_HAPTIC_RUMBLE`, then `data.rumble.strong_magnitude` must be set to a relative magnitude between 0.0 and 1.0 to determine how intensely the “large” rumble motor of the haptic device will vibrate, and `data.rumble.weak_magnitude` must be set to relative magnitude between 0.0 and 1.0 to determine how intensely the “weak” rumble motor of the haptic device will vibrate. Not all devices have a “weak” motor, in which case the value set in `data.rumble.weak_magnitude` will be ignored.

If type is set to `ALLEGRO_HAPTIC_PERIODIC`, then `data.periodic.waveform` must be set to one of `ALLEGRO_HAPTIC_SQUARE`, `ALLEGRO_HAPTIC_TRIANGLE`, `ALLEGRO_HAPTIC_SINE`, `ALLEGRO_HAPTIC_SAW_UP`, `ALLEGRO_HAPTIC_SAW_DOWN`, `ALLEGRO_HAPTIC_CUSTOM`. This will then determine the wave form of the vibration effect that will be played on the haptic device.

In these cases, `data.periodic.period` must be set to the period in seconds of the wave form. The field `data.periodic.magnitude` must be set to the relative magnitude of intensity between -1.0 and 1.0 at which the wave form of the vibration will be played back. The field `data.periodic.offset` must be filled in with the offset from origin in seconds of the wave form of vibration, and the field `data.periodic.phase` is the phase of the wave form of vibration in seconds.

If `data.periodic.waveform` is set to `ALLEGRO_HAPTIC_CUSTOM`, then `data.periodic.custom_data` must point to an array of `data.periodic.custom_len` doubles, each with values between -1.0 and 1.0. This value array will determine the shape of the wave form of the haptic effect. `ALLEGRO_HAPTIC_CUSTOM` is not supported on some platforms, so use `al_get_haptic_capabilities` to check if it’s available. If not, then it’s a good idea play back a non-custom wave effect instead as a substitute.

If type is set to `ALLEGRO_HAPTIC_CONSTANT`, then `data.constant.level` must be set to a relative intensity value between 0.0 and 1.0 to determine the intensity of the effect.

If type is set to any of `ALLEGRO_HAPTIC_SPRING`, `ALLEGRO_HAPTIC_FRICTION`, `ALLEGRO_HAPTIC_DAMPER`, `ALLEGRO_HAPTIC_INERTIA`, `ALLEGRO_HAPTIC_RAMP`, then the `data.condition` struct should be filled in. To explain this better, it’s best to keep in mind that this kind of effects is most useful for steering-wheel kind of devices, where resistance or inertia should be applied when turning the wheel of the device a certain distance to the right or the left.

The field `data.condition.right_saturation` must be filled in with a relative magnitude between -1.0 and 1.0 to determine the the intensity of resistance or inertia on the “right” side of the axis. Likewise, `data.condition.left_saturation` must be filled in with a relative magnitude between -1.0 and 1.0 to determine the the intensity of resistance or inertia on the “left” side of the axis.

The field `data.condition.deadband` must be filled in with a relative value between 0.0 and 1.0, to determine the relative width of the “dead band” of the haptic effect. As long as the axis of the haptic device remains in the “dead band” area, the effect will not be applied. A value of 0.0

means there is no dead band, and a value of 1.0 means it applied over the whole range of the axis in question.

The field `data.condition.center` must be filled in with a relative value between -1.0 and 1.0, to determine the relative position of the “center” of the effect around which the dead band is centered. It should be set to 0.0 in case the center should not be shifted.

The field `data.condition.right_coef` and `data.condition.right_left_coef` must be filled in with a relative coefficient, that will determine how quickly the effect ramps up on the right and left side. If set to 1.0, then the effect will be immediately at full intensity when outside of the dead band. If set to 0.0 the effect will not be felt at all.

If type is set to `ALLEGRO_HAPTIC_RAMP`, then `data.ramp.start_level` should be set to a relative magnitude value between -1.0 and 1.0 to determine the initial intensity of the haptic effect. The field `data.ramp.end_level` should be set to a relative magnitude value between -1.0 and 1.0 to determine the final intensity of the haptic effect at the end of playback.

If type is set to any of `ALLEGRO_HAPTIC_PERIODIC`, `ALLEGRO_HAPTIC_CONSTANT`, `ALLEGRO_HAPTIC_RAMP`, then `data.envelope` determines the “envelope” of the effect. That is, it determines the duration and intensity for the ramp-up attack or “fade in” and the ramp-down “fade out” of the effect.

In these cases the field `data.envelope.attack_level` must be set to a relative value between 0.0 and 1.0 that determines the intensity the effect should have when it starts playing after `replay.delay` seconds have passed since the playback started. The field `data.envelope.attack_length` must be set to the time in seconds that the effect should ramp up to the maximum intensity as set by the other parameters of the effect. If `data.envelope.attack_length` is 0, then the effect will play immediately at full intensity.

The field `data.envelope.fade_level` must be set to a relative value between 0.0 and 1.0 that determines the intensity the effect should have when at the moment it stops playing after `replay.length + replay.delay` seconds have passed since the playback of the effect started. The field `data.envelope.fade_length` must be set to the time in seconds that the effect should fade out before it finished playing. If `data.envelope.fade_length` is 0, then the effect will not fade out.

If you don’t want to use an envelope, then set all four fields of `data.envelope` to 0.0. The effect will then play back at full intensity throughout its playback.

Since: 5.1.8

## 10.4 ALLEGRO\_HAPTIC\_EFFECT\_ID

```
typedef struct ALLEGRO_HAPTIC_EFFECT_ID ALLEGRO_HAPTIC_EFFECT_ID;
```

This struct is used as a handle to control playback of a haptic effect. The struct should be considered opaque. Its implementation is visible merely to allow allocation by the users of the Allegro library.

Since: 5.1.8

## 10.5 al\_install\_haptic

```
bool al_install_haptic(void)
```

Installs the haptic (force feedback) device subsystem. This must be called before using any other haptic-related functions. Returns true if the haptics subsystem could be initialized correctly, false if not.

For portability you should first open a display before calling `al_install_haptic`. On some platforms, such as DirectInput under Windows, `al_install_haptic` will only work if at least one active display is available. This display must stay available until `al_uninstall_haptic` is called.

If you need to close and reopen your active display, e.g. then you should call `al_uninstall_haptic` before closing the display, and `al_install_haptic` after opening it again.



on Windows and DirectInput

Since: 5.1.8

## 10.6 al\_uninstall\_haptic

```
void al_uninstall_haptic(void)
```

Uninstalls the haptic device subsystem. This is useful since on some platforms haptic effects are bound to the active display.

If you need to close and reopen your active display, e.g. then you should call `al_uninstall_haptic` before closing the display, and `al_install_haptic` after opening it again.

Since: 5.1.8

## 10.7 al\_is\_haptic\_installed

```
bool al_is_haptic_installed(void)
```

Returns true if the haptic device subsystem is installed, false if not.

Since: 5.1.8

## 10.8 al\_is\_mouse\_haptic

```
bool al_is_mouse_haptic(ALLEGRO_MOUSE *dev)
```

Returns true if the mouse has haptic capabilities, false if not.

Since: 5.1.8

## 10.9 al\_is\_keyboard\_haptic

```
bool al_is_keyboard_haptic(ALLEGRO_KEYBOARD *dev)
```

Returns true if the keyboard has haptic capabilities, false if not.

Since: 5.1.8

## 10.10 al\_is\_display\_haptic

```
bool al_is_display_haptic(ALLEGRO_DISPLAY *dev)
```

Returns true if the display has haptic capabilities, false if not. To be more precise, this is mainly meant for force feedback that shakes a hand held device, such as a phone or a tablet.

Since: 5.1.8

## 10.11 al\_is\_joystick\_haptic

```
bool al_is_joystick_haptic(ALLEGRO_JOYSTICK *dev)
```

Returns true if the joystick has haptic capabilities, false if not.

Since: 5.1.8

### 10.12 `al_is_touch_input_haptic`

Returns true if the touch input device has haptic capabilities, false if not.

Since: 5.1.8

### 10.13 `al_get_haptic_from_mouse`

```
ALLEGRO_HAPTIC *al_get_haptic_from_mouse(ALLEGRO_MOUSE *dev)
```

If the mouse has haptic capabilities, returns the associated haptic device handle. Otherwise returns NULL.

Since: 5.1.8

### 10.14 `al_get_haptic_from_keyboard`

```
ALLEGRO_HAPTIC *al_get_haptic_from_keyboard(ALLEGRO_KEYBOARD *dev)
```

If the keyboard has haptic capabilities, returns the associated haptic device handle. Otherwise returns NULL.

Since: 5.1.8

### 10.15 `al_get_haptic_from_display`

```
ALLEGRO_HAPTIC *al_get_haptic_from_display(ALLEGRO_DISPLAY *dev)
```

If the display has haptic capabilities, returns the associated haptic device handle. Otherwise returns NULL.

Since: 5.1.8

### 10.16 `al_get_haptic_from_joystick`

```
ALLEGRO_HAPTIC *al_get_haptic_from_joystick(ALLEGRO_JOYSTICK *dev)
```

If the joystick has haptic capabilities, returns the associated haptic device handle. Otherwise returns NULL. It's necessary to call this again every time the joystick configuration changes, such as through hot plugging. In that case, the old haptic device must be released using [al\\_release\\_haptic](#).

Since: 5.1.8

### 10.17 `al_get_haptic_from_touch_input`

```
ALLEGRO_HAPTIC *al_get_haptic_from_touch_input(ALLEGRO_TOUCH_INPUT *dev)
```

If the touch input device has haptic capabilities, returns the associated haptic device handle. Otherwise returns NULL.

Since: 5.1.8

## 10.18 `al_release_haptic`

```
bool al_release_haptic(ALLEGRO_HAPTIC *haptic)
```

Releases the haptic device and it's resources when it's not needed anymore. Should also be used in case the joystick configuration changed, such as when a joystick is hot plugged. This function also automatically releases all haptic effects that are still uploaded to the device and that have not been released manually using [al\\_release\\_haptic\\_effect](#).

Since: 5.1.8

## 10.19 `al_get_haptic_active`

```
bool al_get_haptic_active(ALLEGRO_HAPTIC *hap)
```

Returns true if the haptic device can currently be used, false if not.

Since: 5.1.8

## 10.20 `al_get_haptic_capabilities`

```
int al_get_haptic_capabilities(ALLEGRO_HAPTIC *hap)
```

Returns an integer with or'ed values from [ALLEGRO\\_HAPTIC\\_CONSTANTS](#), which, if set, indicate that the haptic device supports the given feature.

Since: 5.1.8

## 10.21 `al_is_haptic_capable`

```
bool al_is_haptic_capable(ALLEGRO_HAPTIC * hap, int query) {
```

Returns true if the haptic device is supports the feature indicated by the query parameter, false if the feature is not supported. The query parameter must be one of the values of [ALLEGRO\\_HAPTIC\\_CONSTANTS](#).

Since: 5.1.9

See also: [al\\_get\\_haptic\\_capabilities](#)

## 10.22 `al_set_haptic_gain`

```
bool al_set_haptic_gain(ALLEGRO_HAPTIC *hap, double gain)
```

Sets the gain of the haptic device if supported. Gain is much like volume for sound, it is as if every effect's intensity is multiplied by it. Gain is a value between 0.0 and 1.0. Returns true if set sucessfully, false if not. Only works if [al\\_get\\_haptic\\_capabilities](#) returns a value that has `[ALLEGRO_HAPTIC_GAIN]` set. If not, this function returns false, and all effects will be played without any gain influence.

Since: 5.1.8

### 10.23 `al_get_haptic_gain`

```
double al_get_haptic_gain(ALLEGRO_HAPTIC *hap)
```

Returns the current gain of the device. Gain is much like volume for sound, it is as if every effect's intensity is multiplied by it. Gain is a value between 0.0 and 1.0. Only works correctly if `al_get_haptic_capabilities` returns a value that has `[ALLEGRO_HAPTIC_GAIN]` set. If this is not set, this function will simply return 1.0 and all effects will be played without any gain influence.

Since: 5.1.8

### 10.24 `al_set_haptic_autocenter`

```
bool al_set_haptic_autocenter(ALLEGRO_HAPTIC *hap, double intensity)
```

Turns on or off the automatic centering feature of the haptic device if supported. Depending on the device automatic centering may ensure that the axes of the device are centered again automatically after playing a haptic effect. The intensity parameter should be passed with a value between 0.0 and 1.0. The value 0.0 means automatic centering is disabled, and 1.0 means full strength automatic centering. Any value in between those two extremes will result in partial automatic centering. Some platforms do not support partial automatic centering. If that is the case, a value of less than 0.5 will turn it off, while a value equal to or higher to 0.5 will turn it on. Returns true if set successfully, false if not. Can only work if `al_get_haptic_capabilities` returns a value that has `[ALLEGRO_HAPTIC_AUTOCENTER]` set. If not, this function returns false.

Since: 5.1.9

### 10.25 `al_get_haptic_autocenter`

```
double al_get_haptic_autocenter(ALLEGRO_HAPTIC *hap)
```

Returns the current automatic centering intensity of the device. Depending on the device automatic centering may ensure that the axes of the device are centered again automatically after playing a haptic effect. The return value can be between 0.0 and 1.0. The value 0.0 means automatic centering is disabled, and 1.0 means automatic centering is enabled at full strength. Any value in between those two extremes means partial automatic centering is enabled. Some platforms do not support partial automatic centering. If that is the case, a value of less than 0.5 means it is turned off, while a value equal to or higher to 0.5 means it is turned on. Can only work if `al_get_haptic_capabilities` returns a value that has `[ALLEGRO_HAPTIC_AUTOCENTER]` set. If not, this function returns 0.0.

Since: 5.1.9

### 10.26 `al_get_num_haptic_effects`

```
int al_get_num_haptic_effects(ALLEGRO_HAPTIC *hap)
```

Returns the maximum amount of haptic effects that can be uploaded to the device. This depends on the operating system, driver, platform and the device itself. This can return a value as low as 1.

Since: 5.1.8

### 10.27 `al_is_haptic_effect_ok`

```
bool al_is_haptic_effect_ok(ALLEGRO_HAPTIC *hap, ALLEGRO_HAPTIC_EFFECT *effect)
```

Returns true if the haptic device can play the haptic effect as given, false if not. The haptic effect must have been filled in completely and correctly.

Since: 5.1.8

## 10.28 al\_upload\_haptic\_effect

```
bool al_upload_haptic_effect(ALLEGRO_HAPTIC *hap,
    ALLEGRO_HAPTIC_EFFECT *effect, ALLEGRO_HAPTIC_EFFECT_ID *id)
```

Uploads the haptic effect to the device. The haptic effect must have been filled in completely and correctly. You must also pass in a pointer to a user allocated [ALLEGRO\\_HAPTIC\\_EFFECT\\_ID](#). This id can be used to control playback of the effect. Returns true if the effect was successfully uploaded, false if not.

The function [al\\_get\\_num\\_haptic\\_effects](#) returns how many effects can be uploaded to the device at the same time.

The same haptic effect can be uploaded several times, as long as care is taken to pass in a different [ALLEGRO\\_HAPTIC\\_EFFECT\\_ID](#).

Since: 5.1.8

## 10.29 al\_play\_haptic\_effect

```
bool al_play_haptic_effect(ALLEGRO_HAPTIC_EFFECT_ID *id, int loop)
```

Plays back a previously uploaded haptic effect. The play\_id must be a valid [ALLEGRO\\_HAPTIC\\_EFFECT\\_ID](#) obtained from [al\\_upload\\_haptic\\_effect](#), [al\\_upload\\_and\\_play\\_haptic\\_effect](#) or [al\\_rumble\\_haptic](#).

The haptic effect will be played back loop times in sequence. If loop is less than or equal to 1, then the effect will be played once only.

This function returns immediately and doesn't wait for the playback to finish. It returns true if the playback was started successfully or false if not.

Since: 5.1.8

## 10.30 al\_upload\_and\_play\_haptic\_effect

```
bool al_upload_and_play_haptic_effect(ALLEGRO_HAPTIC *hap,
    ALLEGRO_HAPTIC_EFFECT *effect, int loop, ALLEGRO_HAPTIC_EFFECT_ID *id)
```

Uploads and immediately plays back the haptic effect to the device. Returns true if the upload and playback were successful, false if either failed.

In case false is returned, the haptic effect will be automatically released as if [al\\_release\\_haptic\\_effect](#) had been called, so there is no need to call it again manually in this case. However, if true is returned, it is necessary to call [al\\_release\\_haptic\\_effect](#) when the effect isn't needed anymore, to prevent the amount of available effects on the haptic device from running out.

Since: 5.1.8

See also: [al\\_upload\\_haptic\\_effect](#), [al\\_play\\_haptic\\_effect](#)

## 10.31 al\_stop\_haptic\_effect

```
bool al_stop_haptic_effect(ALLEGRO_HAPTIC_EFFECT_ID *id)
```

Stops playing a previously uploaded haptic effect. The play\_id must be a valid [ALLEGRO\\_HAPTIC\\_EFFECT\\_ID](#) obtained from [al\\_upload\\_haptic\\_effect](#), [al\\_upload\\_and\\_play\\_haptic\\_effect](#) or [al\\_rumble\\_haptic](#).

Since: 5.1.8

### 10.32 `al_is_haptic_effect_playing`

```
bool al_is_haptic_effect_playing(ALLEGRO_HAPTIC_EFFECT_ID *id)
```

Returns true if the haptic effect is currently playing. Returns false if the effect has been stopped or if it finished playing, or if it has not been played yet. The `play_id` must be a valid `ALLEGRO_HAPTIC_EFFECT_ID` obtained from `al_upload_haptic_effect`, `al_upload_and_play_haptic_effect` or `al_rumble_haptic`.

Since: 5.1.8

### 10.33 `al_get_haptic_effect_duration`

```
double al_get_haptic_effect_duration(ALLEGRO_HAPTIC_EFFECT *effect)
```

Returns the estimated duration in seconds of a single loop of the given haptic effect. The effect's `effect.replay` must have been filled in correctly before using this function.

Since: 5.1.9

### 10.34 `al_release_haptic_effect`

```
bool al_release_haptic_effect(ALLEGRO_HAPTIC_EFFECT_ID *id)
```

Releases a previously uploaded haptic effect from the device it has been uploaded to, allowing for other effects to be uploaded. The `play_id` must be a valid `ALLEGRO_HAPTIC_EFFECT_ID` obtained from `al_upload_haptic_effect`, `al_upload_and_play_haptic_effect` or `al_rumble_haptic`.

This function is called automatically when you call `al_release_haptic` on a `ALLEGRO_HAPTIC` for all effects that are still uploaded to the device. Therefore this function is most useful if you want to upload and release haptic effects dynamically, for example as a way to circumvent the limit imposed by `al_get_num_haptic_effects`.

Since: 5.1.8

### 10.35 `al_rumble_haptic`

```
bool al_rumble_haptic(ALLEGRO_HAPTIC *hap,  
    double intensity, double duration, ALLEGRO_HAPTIC_EFFECT_ID *id)
```

Uploads a simple rumble effect to the haptic device and starts playback immediately. The parameter `intensity` is a relative magnitude between 0.0 and 1.0 that determines the intensity of the rumble effect. The `duration` determines the duration of the effect in seconds.

You must also pass in a pointer to a user allocated `ALLEGRO_HAPTIC_EFFECT_ID`. It is stored a reference to be used to control playback of the effect. Returns true if the rumble effect was successfully uploaded and started, false if not.

In case false is returned, the rumble effect will be automatically released as if `al_release_haptic_effect` had been called, so there is no need to call it again manually in this case. However, if true is returned, it is necessary to call `al_release_haptic_effect` when the effect isn't needed anymore, to prevent the amount of available effects on the haptic device from running out.

Since: 5.1.8

## Joystick routines

These functions are declared in the main Allegro header file:

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

On Windows there are two joystick drivers, a DirectInput one and an Xinput one. If support for XInput was compiled in, then it can be enabled by calling `al_set_config_value(al_get_system_config(), "joystick", "driver", "xinput")` before calling `al_install_joystick`, or by setting the same option in the `allegro5.cfg` configuration file. The Xinput and DirectInput drivers are mutually exclusive. The haptics subsystem will use the same drivers as the joystick system does.

### 11.1 ALLEGRO\_JOYSTICK

```
typedef struct ALLEGRO_JOYSTICK ALLEGRO_JOYSTICK;
```

This is an abstract data type representing a physical joystick.

See also: [al\\_get\\_joystick](#)

### 11.2 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](#)

### 11.3 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](#)

## 11.4 `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](#)

## 11.5 `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](#)

## 11.6 `al_is_joystick_installed`

```
bool al_is_joystick_installed(void)
```

Returns true if [al\\_install\\_joystick](#) was called successfully.

## 11.7 `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`

## 11.8 `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](#)



## 11.9 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](#)

## 11.10 al\_release\_joystick

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

This function currently does nothing.

See also: [al\\_get\\_joystick](#)

## 11.11 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](#)

## 11.12 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](#)

## 11.13 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](#)

## 11.14 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](#)

### 11.15 `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](#)

### 11.16 `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](#)

### 11.17 `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](#)

### 11.18 `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](#)

### 11.19 `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](#)

### 11.20 `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](#)

### 11.21 `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.

## Keyboard routines

These functions are declared in the main Allegro header file:

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

### 12.1 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](#).

### 12.2 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

/* Since: 5.1.1 */
/* Android only for now */
ALLEGRO_KEY_BACK

/* Since: 5.1.2 */
/* Android only for now */
```

```
ALLEGRO_KEY_VOLUME_UP
ALLEGRO_KEY_VOLUME_DOWN

/* Since: 5.1.6 */
/* Android only for now */
ALLEGRO_KEY_SEARCH
ALLEGRO_KEY_DPAD_CENTER
ALLEGRO_KEY_BUTTON_X
ALLEGRO_KEY_BUTTON_Y
ALLEGRO_KEY_DPAD_UP
ALLEGRO_KEY_DPAD_DOWN
ALLEGRO_KEY_DPAD_LEFT
ALLEGRO_KEY_DPAD_RIGHT
ALLEGRO_KEY_SELECT
ALLEGRO_KEY_START
ALLEGRO_KEY_L1
ALLEGRO_KEY_R1
```

## 12.3 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.

## 12.4 `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](#)

## 12.5 `al_is_keyboard_installed`

```
bool al_is_keyboard_installed(void)
```

Returns true if [al\\_install\\_keyboard](#) was called successfully.

## 12.6 `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](#)

## 12.7 `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](#)

## 12.8 `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](#)

## 12.9 `al_keycode_to_name`

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

Converts the given keycode to a description of the key.

## 12.10 `al_set_keyboard_leds`

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

Overrides the state of the keyboard LED indicators. Set leds to a combination of the keyboard modifier flags to enable the corresponding LED indicators ([ALLEGRO\\_KEYMOD\\_NUMLOCK](#), [ALLEGRO\\_KEYMOD\\_CAPSLOCK](#) and [ALLEGRO\\_KEYMOD\\_SCROLLLOCK](#) are supported) or to -1 to return to default behavior. False is returned if the current keyboard driver cannot set LED indicators.

## 12.11 `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.

## Memory management routines

These functions are declared in the main Allegro header file:

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

### 13.1 `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](#)

### 13.2 `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](#)

### 13.3 `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](#)

### 13.4 `al_malloc`

```
#define al_malloc(c, n) \  
    (al_malloc_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\\_malloc\\_with\\_context](#)

### 13.5 `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 overridden with [al\\_set\\_memory\\_interface](#),

Generally you should use the [al\\_malloc](#) macro.

### 13.6 `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 overridden with [al\\_set\\_memory\\_interface](#).

Generally you should use the [al\\_free](#) macro.

### 13.7 `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 overridden with [al\\_set\\_memory\\_interface](#),

Generally you should use the [al\\_realloc](#) macro.

### 13.8 `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 overridden with [al\\_set\\_memory\\_interface](#),

Generally you should use the [al\\_calloc](#) macro.

### 13.9 `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](#)

## 13.10 `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](#)



## Miscellaneous routines

These functions are declared in the main Allegro header file:

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

### 14.1 ALLEGRO\_PI

```
#define ALLEGRO_PI 3.14159265358979323846
```

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

### 14.2 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.



These functions are declared in the main Allegro header file:

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

### 15.1 ALLEGRO\_MONITOR\_INFO

```
typedef struct ALLEGRO_MONITOR_INFO
```

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

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

See also: [al\\_get\\_monitor\\_info](#)

### 15.2 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](#)

### 15.3 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](#)

### 15.4 `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](#)

### 15.5 `al_get_num_video_adapters`

```
int al_get_num_video_adapters(void)
```

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

See also: [al\\_get\\_monitor\\_info](#)

## Mouse routines

These functions are declared in the main Allegro header file:

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

### 16.1 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.
- pressure - pressure, ranging from 0.0 to 1.0

See also: [al\\_get\\_mouse\\_state](#), [al\\_get\\_mouse\\_state\\_axis](#), [al\\_mouse\\_button\\_down](#)

### 16.2 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.

### 16.3 al\_is\_mouse\_installed

```
bool al_is_mouse_installed(void)
```

Returns true if [al\\_install\\_mouse](#) was called successfully.

### 16.4 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.

## 16.5 `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](#)

## 16.6 `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](#)

## 16.7 `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](#)

## 16.8 `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](#)

## 16.9 `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](#)



## 16.10 `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](#)

## 16.11 `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](#)

## 16.12 `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](#)

## 16.13 `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](#)

## 16.14 `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.

## 16.15 Mouse cursors

### 16.15.1 `al_create_mouse_cursor`

```
ALLEGRO_MOUSE_CURSOR *al_create_mouse_cursor(ALLEGRO_BITMAP *bmp,  
int x_focus, int y_focus)
```

Create a mouse cursor from the bitmap provided.

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

See also: [al\\_set\\_mouse\\_cursor](#), [al\\_destroy\\_mouse\\_cursor](#)

### 16.15.2 `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](#)

### 16.15.3 `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](#)

### 16.15.4 `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](#)

### 16.15.5 `al_get_mouse_cursor_position`

```
bool al_get_mouse_cursor_position(int *ret_x, int *ret_y)
```

On platforms where this information is available, this function returns the global location of the mouse cursor, relative to the desktop. You should not normally use this function, as the information is not useful except for special scenarios as moving a window.

Returns true on success, false on failure.

### 16.15.6 `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](#)

### 16.15.7 `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](#)

### 16.15.8 `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](#)

### 16.15.9 `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](#)



## 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.

### 17.1 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](#)

### 17.2 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](#)

### 17.3 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](#)

### 17.4 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](#)

## 17.5 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](#)

## 17.6 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](#)

## 17.7 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.

## 17.8 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](#)

## 17.9 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](#)

## 17.10 `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.

## 17.11 `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](#)

## 17.12 `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](#)

## 17.13 `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](#)

## 17.14 `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](#)

## 17.15 `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](#)

### 17.16 `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:  $0 \leq i \leq \text{al\_get\_path\_num\_components}(\text{path})$ .

See also: [al\\_append\\_path\\_component](#), [al\\_replace\\_path\\_component](#), [al\\_remove\\_path\\_component](#)

### 17.17 `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](#)

### 17.18 `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](#)

### 17.19 `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](#)

### 17.20 `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](#)

### 17.21 `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](#)



## 17.22 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.

## 17.23 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.



These functions are declared in the main Allegro header file:

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

### 18.1 ALLEGRO\_STATE

```
typedef struct ALLEGRO_STATE ALLEGRO_STATE;
```

Opaque type which is passed to [al\\_store\\_state/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
    current shader
```

In general, the only real global state is the active system driver. All other global state is per-thread, so if your application has multiple separate threads they never will interfere with each other. (Except if there are objects accessed by multiple threads of course. Usually you want to minimize that though and for the remaining cases use synchronization primitives described in the threads section or events described in the events section to control inter-thread communication.)

### 18.2 ALLEGRO\_STATE\_FLAGS

```
typedef enum ALLEGRO_STATE_FLAGS
```

Flags which can be passed to [al\\_store\\_state/al\\_restore\\_state](#) as bit combinations. See [al\\_store\\_state](#) for the list of flags.

### 18.3 `al_restore_state`

```
void al_restore_state(ALLEGRO_STATE const *state)
```

Restores part of the state of the current thread from the given `ALLEGRO_STATE` object.

See also: `al_store_state`, `ALLEGRO_STATE_FLAGS`

### 18.4 `al_store_state`

```
void al_store_state(ALLEGRO_STATE *state, int flags)
```

Stores part of the state of the current thread in the given `ALLEGRO_STATE` 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`

### 18.5 `al_get_errno`

```
int al_get_errno(void)
```

Some Allegro functions will set an error number as well as returning an error code. Call this function to retrieve the last error number set for the calling thread.

### 18.6 `al_set_errno`

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

Set the error number for the calling thread.

## System routines

These functions are declared in the main Allegro header file:

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

### 19.1 `al_install_system`

```
bool al_install_system(int version, int (*atexit_ptr)(void (*)(void)))
```

Initialize the Allegro system. No other Allegro functions can be called before this (with one or two exceptions).

The version field should always be set to `ALLEGRO_VERSION_INT`.

If `atexit_ptr` is non-NULL, and if hasn't been done already, `al_uninstall_system` will be registered as an `atexit` function.

Returns true if Allegro was successfully initialized by this function call (or already was initialized previously), false if Allegro cannot be used.

See also: [al\\_init](#)

### 19.2 `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](#)

### 19.3 `al_uninstall_system`

```
void al_uninstall_system(void)
```

Closes down the Allegro system.

Note: `al_uninstall_system()` can be called without a corresponding `al_install_system` call, e.g. from `atexit()`.

### 19.4 `al_is_system_installed`

```
bool al_is_system_installed(void)
```

Returns true if Allegro is initialized, otherwise returns false.

## 19.5 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.

## 19.6 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`

## 19.7 `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`

## 19.8 `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`

## 19.9 `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`

## 19.10 `al_get_app_name`

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

Returns the global application name string.

See also: `al_set_app_name`

### 19.11 `al_get_org_name`

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

Returns the global organization name string.

See also: [al\\_set\\_org\\_name](#)

### 19.12 `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. The returned configuration should not be destroyed with [al\\_destroy\\_config](#). This is mainly used for configuring Allegro and its addons.

### 19.13 `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

### 19.14 `al_register_trace_handler`

```
void al_register_trace_handler(void (*handler)(char const *))
```

Register a callback which is called whenever Allegro writes something to its log files. The default logging to `allegro.log` is disabled while this callback is active. Pass NULL to revert to the default logging.

This function may be called prior to `al_install_system`.

See the example `allegro5.cfg` for documentation on how to configure the used debug channels, logging levels and trace format.

Note that logging is disabled in release mode by default and needs to be enabled in the build system - the callback will never be called otherwise.

See also: [Configuration files](#)

Since: 5.1.5



## 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>
```

### 20.1 ALLEGRO\_THREAD

```
typedef struct ALLEGRO_THREAD ALLEGRO_THREAD;
```

An opaque structure representing a thread.

### 20.2 ALLEGRO\_MUTEX

```
typedef struct ALLEGRO_MUTEX ALLEGRO_MUTEX;
```

An opaque structure representing a mutex.

### 20.3 ALLEGRO\_COND

```
typedef struct ALLEGRO_COND ALLEGRO_COND;
```

An opaque structure representing a condition variable.

### 20.4 al\_create\_thread

```
ALLEGRO_THREAD *al_create_thread(  
    void *(*proc)(ALLEGRO_THREAD *thread, void *arg), void *arg)
```

Spawn a new thread which begins executing proc. The new thread is passed its own thread handle and the value arg.

Returns a pointer to the thread on success. Otherwise, returns NULL if there was an error.

See also: [al\\_start\\_thread](#), [al\\_join\\_thread](#).

## 20.5 `al_start_thread`

```
void al_start_thread(ALLEGRO_THREAD *thread)
```

When a thread is created, it is initially in a suspended state. Calling `al_start_thread` will start its actual execution.

Starting a thread which has already been started does nothing.

See also: [al\\_create\\_thread](#).

## 20.6 `al_join_thread`

```
void al_join_thread(ALLEGRO_THREAD *thread, void **ret_value)
```

Wait for the thread to finish executing. This implicitly calls `al_set_thread_should_stop` 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](#).

## 20.7 `al_set_thread_should_stop`

```
void al_set_thread_should_stop(ALLEGRO_THREAD *thread)
```

Set the flag to indicate thread should stop. Returns immediately.

See also: [al\\_join\\_thread](#), [al\\_get\\_thread\\_should\\_stop](#).

## 20.8 `al_get_thread_should_stop`

```
bool al_get_thread_should_stop(ALLEGRO_THREAD *thread)
```

Check if another thread is waiting for thread to stop. Threads which run in a loop should check this periodically and act on it when convenient.

Returns true if another thread has called `al_join_thread` 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.

## 20.9 `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](#).

## 20.10 `al_run_detached_thread`

```
void al_run_detached_thread(void *(*proc)(void *arg), void *arg)
```

Runs the passed function in its own thread, with `arg` passed to it as only parameter. This is similar to calling `al_create_thread`, `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.

## 20.11 `al_create_mutex`

```
ALLEGRO_MUTEX *al_create_mutex(void)
```

Create the mutex object (a mutual exclusion device). The mutex may or may not support “recursive” locking.

Returns the mutex on success or NULL on error.

See also: [al\\_create\\_mutex\\_recursive](#).

## 20.12 `al_create_mutex_recursive`

```
ALLEGRO_MUTEX *al_create_mutex_recursive(void)
```

Create the mutex object (a mutual exclusion device), with support for “recursive” locking. That is, the mutex will count the number of times it has been locked by the same thread. If the caller tries to acquire a lock on the mutex when it already holds the lock then the count is incremented. The mutex is only unlocked when the thread releases the lock on the mutex an equal number of times, i.e. the count drops down to zero.

See also: [al\\_create\\_mutex](#).

## 20.13 `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`.

## 20.14 `al_unlock_mutex`

```
void al_unlock_mutex(ALLEGRO_MUTEX *mutex)
```

Release the lock on mutex if the calling thread holds the lock on it.

If the calling thread doesn't hold the lock, or if the mutex is not locked, undefined behaviour results.

See also: [al\\_lock\\_mutex](#).

## 20.15 `al_destroy_mutex`

```
void al_destroy_mutex(ALLEGRO_MUTEX *mutex)
```

Free the resources used by the mutex. The mutex should be unlocked. Destroying a locked mutex results in undefined behaviour.

Does nothing if mutex is NULL.

## 20.16 `al_create_cond`

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

Create a condition variable.

Returns the condition value on success or NULL on error.

## 20.17 `al_destroy_cond`

```
void al_destroy_cond(ALLEGRO_COND *cond)
```

Destroy a condition variable.

Destroying a condition variable which has threads block on it results in undefined behaviour.

Does nothing if cond is NULL.

## 20.18 `al_wait_cond`

```
void al_wait_cond(ALLEGRO_COND *cond, ALLEGRO_MUTEX *mutex)
```

On entering this function, mutex must be locked by the calling thread. The function will atomically release mutex and block on cond. The function will return when cond is “signalled”, acquiring the lock on the mutex in the process.

Example of proper use:

```
al_lock_mutex(mutex);
while (something_not_true) {
    al_wait_cond(cond, mutex);
}
do_something();
al_unlock_mutex(mutex);
```

The mutex should be locked before checking the condition, and should be rechecked `al_wait_cond` 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](#).

## 20.19 `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](#)

## 20.20 `al_broadcast_cond`

```
void al_broadcast_cond(ALLEGRO_COND *cond)
```

Unblock all threads currently waiting on a condition variable. That is, broadcast that some condition which those threads were waiting for has become true.

See also: [al\\_signal\\_cond](#).

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

## 20.21 `al_signal_cond`

```
void al_signal_cond(ALLEGRO_COND *cond)
```

Unblock at least one thread waiting on a condition variable.

Generally you should use [al\\_broadcast\\_cond](#) but [al\\_signal\\_cond](#) may be more efficient when it's applicable.

See also: [al\\_broadcast\\_cond](#).



---

Time routines

These functions are declared in the main Allegro header file:

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

### 21.1 ALLEGRO\_TIMEOUT

```
typedef struct ALLEGRO_TIMEOUT ALLEGRO_TIMEOUT;
```

Represent a timeout value. The size of the structure is known so can be statically allocated. The contents are private.

See also: [al\\_init\\_timeout](#)

### 21.2 al\_get\_time

```
double al_get_time(void)  
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.

### 21.3 al\_current\_time

Alternate spelling of [al\\_get\\_time](#).

### 21.4 al\_init\_timeout

```
void al_init_timeout(ALLEGRO_TIMEOUT *timeout, double seconds)  
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](#)

### 21.5 al\_rest

```
void al_rest(double seconds)  
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.



---

Timer routines

These functions are declared in the main Allegro header file:

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

### 22.1 ALLEGRO\_TIMER

```
typedef struct ALLEGRO_TIMER ALLEGRO_TIMER;
```

This is an abstract data type representing a timer object.

### 22.2 ALLEGRO\_USECS\_TO\_SECS

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

Convert microseconds to seconds.

### 22.3 ALLEGRO\_MSECS\_TO\_SECS

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

Convert milliseconds to seconds.

### 22.4 ALLEGRO\_BPS\_TO\_SECS

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

Convert beats per second to seconds.

### 22.5 ALLEGRO\_BPM\_TO\_SECS

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

Convert beats per minute to seconds.

## 22.6 `al_create_timer`

```
ALLEGRO_TIMER *al_create_timer(double speed_secs)
```

Allocates and initializes a 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.

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](#)

## 22.7 `al_start_timer`

```
void al_start_timer(ALLEGRO_TIMER *timer)
```

Start the timer specified. From then, the timer’s counter will increment at a constant rate, and it will begin generating events. Starting a timer that is already started does nothing.

See also: [al\\_stop\\_timer](#), [al\\_get\\_timer\\_started](#)

## 22.8 `al_stop_timer`

```
void al_stop_timer(ALLEGRO_TIMER *timer)
```

Stop the timer specified. The timer’s counter will stop incrementing and it will stop generating events. Stopping a timer that is already stopped does nothing.

See also: [al\\_start\\_timer](#), [al\\_get\\_timer\\_started](#)

## 22.9 `al_get_timer_started`

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

Return true if the timer specified is currently started.

## 22.10 `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](#)

## 22.11 `al_get_timer_count`

```
int64_t al_get_timer_count(const ALLEGRO_TIMER *timer)
```

Return the timer’s counter value. The timer can be started or stopped.

See also: [al\\_set\\_timer\\_count](#)

## 22.12 `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](#)

## 22.13 `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](#)

## 22.14 `al_get_timer_speed`

```
double al_get_timer_speed(const ALLEGRO_TIMER *timer)
```

Return the timer's speed, in seconds. (The same value passed to [al\\_create\\_timer](#) or [al\\_set\\_timer\\_speed](#).)

See also: [al\\_set\\_timer\\_speed](#)

## 22.15 `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 occurred precisely at the last tick.

*speed\_secs* has exactly the same meaning as with [al\\_create\\_timer](#).

See also: [al\\_get\\_timer\\_speed](#)

## 22.16 `al_get_timer_event_source`

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

Retrieve the associated event source.



## Touch input

These functions are declared in the main Allegro header file:

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

### 23.1 ALLEGRO\_TOUCH\_INPUT

```
typedef struct ALLEGRO_TOUCH_INPUT ALLEGRO_TOUCH_INPUT;
```

An abstract data type representing a physical touch screen or touch pad.

Since: 5.1.0

### 23.2 ALLEGRO\_TOUCH\_INPUT\_MAX\_TOUCH\_COUNT

```
#define ALLEGRO_TOUCH_INPUT_MAX_TOUCH_COUNT 16
```

The maximum amount of simultaneous touches that can be detected.

Since: 5.1.0

### 23.3 ALLEGRO\_TOUCH\_STATE

```
typedef struct ALLEGRO_TOUCH_STATE ALLEGRO_TOUCH_STATE;
```

This is a structure that is used to hold a “snapshot” of a touch at a particular instant.

Public fields (read only):

- id - identifier of the touch. If the touch is valid, this is positive.
- x - touch x position
- y - touch y position
- dx - touch relative x position
- dy - touch relative y position
- primary - TRUE if this touch is the primary one (usually the first one).
- display - The [ALLEGRO\\_DISPLAY](#) that was touched.

Since: 5.1.0

## 23.4 ALLEGRO\_TOUCH\_INPUT\_STATE

```
typedef struct ALLEGRO_TOUCH_INPUT_STATE ALLEGRO_TOUCH_INPUT_STATE;
```

This is a structure that holds a snapshot of all simultaneous touches at a particular instant.

Public fields (read only):

- touches - an array of [ALLEGRO\\_TOUCH\\_STATE](#)

Since: 5.1.0

## 23.5 ALLEGRO\_MOUSE\_EMULATION\_MODE

```
typedef enum ALLEGRO_MOUSE_EMULATION_MODE
```

Type of mouse emulation to apply.

**ALLEGRO\_MOUSE\_EMULATION\_NONE**

Disables mouse emulation.

**ALLEGRO\_MOUSE\_EMULATION\_TRANSPARENT**

Enables transparent mouse emulation.

**ALLEGRO\_MOUSE\_EMULATION\_INCLUSIVE**

Enable inclusive mouse emulation.

**ALLEGRO\_MOUSE\_EMULATION\_EXCLUSIVE**

Enables exclusive mouse emulation.

**ALLEGRO\_MOUSE\_EMULATION\_5\_0\_x**

Enables mouse emulation that is backwards compatible with Allegro 5.0.x.

Since: 5.1.0

## 23.6 al\_install\_touch\_input

```
bool al_install_touch_input(void)
```

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

Since: 5.1.0

See also: [al\\_uninstall\\_touch\\_input](#)

## 23.7 al\_uninstall\_touch\_input

```
void al_uninstall_touch_input(void)
```

Uninstalls the active touch input driver. If no touch input driver was active, this function does nothing.

This function is automatically called when Allegro is shut down.

Since: 5.1.0

See also: [al\\_install\\_touch\\_input](#)

## 23.8 `al_is_touch_input_installed`

```
bool al_is_touch_input_installed(void)
```

Returns true if `al_install_touch_input` was called successfully.

Since: 5.1.0

## 23.9 `al_get_touch_input_state`

```
void al_get_touch_input_state(ALLEGRO_TOUCH_INPUT_STATE *ret_state)
```

Gets the current touch input state. The touch information is copied into the `ALLEGRO_TOUCH_INPUT_STATE` you provide to this function.

Since: 5.1.0

## 23.10 `al_set_mouse_emulation_mode`

```
void al_set_mouse_emulation_mode(int mode)
```

Sets the kind of mouse emulation for the touch input subsystem to perform.

Since: 5.1.0

See also: `ALLEGRO_MOUSE_EMULATION_MODE`, `al_get_mouse_emulation_mode`.

## 23.11 `al_get_mouse_emulation_mode`

```
int al_get_mouse_emulation_mode(void)
```

Returns the kind of mouse emulation which the touch input subsystem is set to perform.

Since: 5.1.0

See also: `ALLEGRO_MOUSE_EMULATION_MODE`, `al_set_mouse_emulation_mode`.

## 23.12 `al_get_touch_input_event_source`

```
ALLEGRO_EVENT_SOURCE *al_get_touch_input_event_source(void)
```

Returns the global touch input event source. This event source generates touch input events.

Since: 5.1.0

See also: `ALLEGRO_EVENT_SOURCE`, `al_register_event_source`

## 23.13 `al_get_touch_input_mouse_emulation_event_source`

```
ALLEGRO_EVENT_SOURCE *al_get_touch_input_mouse_emulation_event_source(void)
```

Returns the global touch input event source for emulated mouse events. This event source generates emulated mouse events that are based on touch events.

See also: `ALLEGRO_EVENT_SOURCE`, `al_register_event_source`

Since: 5.1.0





## 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 know 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);
```

## 24.1 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

## 24.2 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

## 24.3 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](#)

## 24.4 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.

## 24.5 `al_get_current_inverse_transform`

```
const ALLEGRO_TRANSFORM *al_get_current_inverse_transform(void)
```

Returns the inverse of the current transformation of the target bitmap. If there is no target bitmap, this function returns NULL.

This is similar to calling `al_invert_transform(al_get_current_transform())` but the result of this function is cached.

Since: 5.1.0

## 24.6 `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](#)

## 24.7 `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 be 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](#)

## 24.8 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](#)

## 24.9 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, rotate, scale, 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 in radians

*Note:* this function was previously documented to be equivalent to a different (and more useful) order of operations: identity, scale, rotate, translate.

See also: [al\\_translate\\_transform](#), [al\\_rotate\\_transform](#), [al\\_scale\\_transform](#), [al\\_compose\\_transform](#)

## 24.10 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](#)

## 24.11 `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 in radians

See also: [al\\_translate\\_transform](#), [al\\_scale\\_transform](#), [al\\_build\\_transform](#)

## 24.12 `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](#)

## 24.13 `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](#)

## 24.14 `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`

See also: [al\\_translate\\_transform](#), [al\\_rotate\\_transform](#), [al\\_scale\\_transform](#)

### 24.15 `al_orthographic_transform`

```
void al_orthographic_transform(ALLEGRO_TRANSFORM *trans,  
    float left, float top, float n,  
    float right, float bottom, float f)
```

Combines the given transformation with an orthographic transformation which maps the screen rectangle to the given left/top and right/bottom coordinates. near/far is the z-buffer range, if there is no z-buffer you can set it to -1/1.

Since: 5.1.3

See also: [al\\_set\\_projection\\_transform](#), [al\\_perspective\\_transform](#)

### 24.16 `al_perspective_transform`

```
void al_perspective_transform(ALLEGRO_TRANSFORM *trans,  
    float left, float top, float n,  
    float right, float bottom, float f)
```

Like [al\\_orthographic\\_transform](#) but honors perspective. If everything is at a z-position of -near it will look the same as with an orthographic transformation.

To use a specific horizontal field of view you can use the relation:

$$\tan(\text{hfov} / 2) = (\text{right} - \text{left}) / 2 / \text{near}$$

Since: 5.1.3

See also: [al\\_set\\_projection\\_transform](#), [al\\_orthographic\\_transform](#)

### 24.17 `al_translate_transform_3d`

```
void al_translate_transform_3d(ALLEGRO_TRANSFORM *trans, float x, float y,  
    float z)
```

Combines the given transformation with a transformation which translates coordinates by the given vector.

Since: 5.1.3

See also: [al\\_set\\_projection\\_transform](#)

### 24.18 `al_scale_transform_3d`

```
void al_scale_transform_3d(ALLEGRO_TRANSFORM *trans, float sx, float sy,  
    float sz)
```

Combines the given transformation with a transformation which scales coordinates by the given vector.

Since: 5.1.3

See also: [al\\_set\\_projection\\_transform](#)

## 24.19 al\_rotate\_transform\_3d

```
void al_rotate_transform_3d(ALLEGRO_TRANSFORM *trans,  
    float x, float y, float z, float angle)
```

Combines the given transformation with a transformation which rotates coordinates around the given vector by the given angle in radians.

Since: 5.1.3

See also: [al\\_set\\_projection\\_transform](#)

## 24.20 al\_get\_projection\_transform

```
ALLEGRO_TRANSFORM *al_get_projection_transform(ALLEGRO_DISPLAY *display)
```

Returns the projection transformation of the display as set with [al\\_set\\_projection\\_transform](#).

Since: 5.1.0

See also: [al\\_set\\_projection\\_transform](#)

## 24.21 al\_set\_projection\_transform

```
void al_set_projection_transform(ALLEGRO_DISPLAY *display, ALLEGRO_TRANSFORM *t)
```

Replaces the projection transformation of the given display.

*Note:* Unlike the regular 2D transformations, the projection transformation affects all drawing operations done to the current display. This means for example that it is not affected by calls to `al_set_target_bitmap` (if the bitmap belongs to the same display).

*Note:* Drawing to memory bitmaps is not affected by the projection transformation.

Since: 5.1.0

See also: [al\\_get\\_projection\\_transform](#)

## 24.22 al\_horizontal\_shear\_transform

```
void al_horizontal_shear_transform(ALLEGRO_TRANSFORM* trans, float theta)
```

Apply a horizontal shear to the transform

*Parameters:*

- trans - Transformation to alter
- theta - Rotation angle in radians

Since: 5.1.7

See also: [al\\_vertical\\_shear\\_transform](#)

### 24.23 `al_vertical_shear_transform`

```
void al_vertical_shear_transform(ALLEGRO_TRANSFORM* trans, float theta)
```

Apply a vertical shear to the transform

*Parameters:*

- `trans` - Transformation to alter
- `theta` - Rotation angle in radians

Since: 5.1.7

See also: [al\\_horizontal\\_shear\\_transform](#)



## UTF-8 string routines

These functions are declared in the main Allegro header file:

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

### 25.1 About UTF-8 string routines

Some parts of the Allegro API, such as the font routines, expect Unicode strings encoded in UTF-8. The following 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.

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. Unlike legacy character sets, the set of code points is open ended and more are assigned with time.

Clearly it is impossible represent each code point with a 8-bit byte (limited to 256 code points) or even a 16-bit integer (limited to 65536 code points). It is possible to store code points in a 32-bit integers but it is space inefficient, and not actually that useful (at least, when handling the full complexity of Unicode; Allegro only does the very basics). There exist different Unicode Transformation Formats for encoding code points into smaller *code units*. The most important transformation formats are UTF-8 and UTF-16.

UTF-8 is a *variable-length encoding* which encodes each code point to between one and four 8-bit bytes each. UTF-8 has many nice properties, but the main advantages are that it is backwards compatible with C strings, and ASCII characters (code points in the range 0-127) are encoded in UTF-8 exactly as they would be in ASCII.

UTF-16 is another variable-length encoding, but encodes each code point to one or two 16-bit words each. It is, of course, not compatible with traditional C strings. Allegro does not generally use UTF-16 strings.

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 bytes	0xC3, 0xA5	0x6C	0x00
UTF-16LE bytes	0xE5, 0x00	0x6C, 0x00	0x00, 0x00

You can see the aforementioned properties of UTF-8. The first code point U+00E5 (“å”) is outside of the ASCII range (0-127) so is encoded to multiple code units – it requires two bytes. U+006C (“l”) and U+0000 (NUL) both exist in the ASCII range so take exactly one byte each, 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 point by either one or two 16-bit code units (two or four bytes). The representation of each 16-bit code unit depends on the byte order; here we have demonstrated little endian.

Both UTF-8 and UTF-16 are self-synchronising. Starting from any offset within a string, it is efficient to find the beginning of the previous or next code point.

Not all sequences of bytes or 16-bit words are valid UTF-8 and UTF-16 strings respectively. UTF-8 also has an additional problem of overlong forms, where a code point value is encoded using more bytes than is strictly necessary. This is invalid and needs to be guarded against.

In the following “ustr” functions, be careful whether a function takes code unit (byte) or code point indices. In general, all position parameters are in code unit offsets. This may be surprising, but if you think about, is required for good performance. (It also means some functions will work even if they do not contain UTF-8, since they only care about storing bytes, so you may actually store arbitrary data in the ALLEGRO\_USTRs.)

For actual text processing, where you want to specify positions with code point indices, you should use [al\\_ustr\\_offset](#) to find the code unit offset position. However, most of the time you would probably just work with byte offsets.

## 25.2 UTF-8 string types

### 25.2.1 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.

### 25.2.2 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 also: [al\\_ref\\_cstr](#), [al\\_ref\\_buffer](#) and [al\\_ref\\_ustr](#).

## 25.3 Creating and destroying strings

### 25.3.1 al\_ustr\_new

```
ALLEGRO_USTR *al_ustr_new(const char *s)
```

Create a new string containing a copy of the C-style string *s*. The string must eventually be freed with [al\\_ustr\\_free](#).

See also: [al\\_ustr\\_new\\_from\\_buffer](#), [al\\_ustr\\_newf](#), [al\\_ustr\\_dup](#), [al\\_ustr\\_new\\_from\\_utf16](#)

### 25.3.2 `al_ustr_new_from_buffer`

```
ALLEGRO_USTR *al_ustr_new_from_buffer(const char *s, size_t size)
```

Create a new string containing a copy of the buffer pointed to by `s` of the given size in bytes. The string must eventually be freed with `al_ustr_free`.

See also: [al\\_ustr\\_new](#)

### 25.3.3 `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_USTR`s. 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 is only usable for ASCII characters (value  $\leq 127$ ) 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](#)

### 25.3.4 `al_ustr_free`

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

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

See also: [al\\_ustr\\_new](#), [al\\_ustr\\_new\\_from\\_buffer](#), [al\\_ustr\\_newf](#)

### 25.3.5 `al_cstr`

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

Get a `char *` pointer to the data in a string. This pointer will only be valid while the `ALLEGRO_USTR` object is not modified and not destroyed. The pointer may be passed to functions expecting C-style strings, with the following caveats:

- `ALLEGRO_USTR`s are allowed to contain embedded NUL (`'\0'`) bytes. That means `al_ustr_size(u)` and `strlen(al_cstr(u))` may not agree.
- An `ALLEGRO_USTR` may be created in such a way that it is not NUL terminated. A string which is dynamically allocated will always be NUL terminated, but a string which references the middle of another string or region of memory will *not* be NUL terminated.
- If the `ALLEGRO_USTR` references another string, the returned C string will point into the referenced string. Again, no NUL terminator will be added to the referenced string.

See also: [al\\_ustr\\_to\\_buffer](#), [al\\_cstr\\_dup](#)

### 25.3.6 `al_ustr_to_buffer`

```
void al_ustr_to_buffer(const ALLEGRO_USTR *us, char *buffer, int size)
```

Write the contents of the string into a pre-allocated buffer of the given size in bytes. The result will always be NUL terminated, so a maximum of `size - 1` bytes will be copied.

See also: [al\\_cstr](#), [al\\_cstr\\_dup](#)

### 25.3.7 `al_cstr_dup`

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

Create a NUL (`'\0'`) terminated copy of the string. Any embedded NUL bytes will still be presented in the returned string. The new string must eventually be freed with [al\\_free](#).

If an error occurs NULL is returned.

See also: [al\\_cstr](#), [al\\_ustr\\_to\\_buffer](#), [al\\_free](#)

### 25.3.8 `al_ustr_dup`

```
ALLEGRO_USTR *al_ustr_dup(const ALLEGRO_USTR *us)
```

Return a duplicate copy of a string. The new string will need to be freed with [al\\_ustr\\_free](#).

See also: [al\\_ustr\\_dup\\_substr](#), [al\\_ustr\\_free](#)

### 25.3.9 `al_ustr_dup_substr`

```
ALLEGRO_USTR *al_ustr_dup_substr(const ALLEGRO_USTR *us, int start_pos,
int end_pos)
```

Return a new copy of a string, containing its contents in the byte interval `[start_pos, end_pos)`. The new string will be NUL terminated and will need to be freed with [al\\_ustr\\_free](#).

If necessary, use [al\\_ustr\\_offset](#) to find the byte offsets for a given code point that you are interested in.

See also: [al\\_ustr\\_dup](#), [al\\_ustr\\_free](#)

## 25.4 Predefined strings

### 25.4.1 `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.

## 25.5 Creating strings by referencing other data

### 25.5.1 `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](#)

### 25.5.2 al\_ref\_buffer

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

Create a string that references the storage of an underlying buffer. The size of the buffer is given in bytes. You can use it to reference only part of a string or an arbitrary region of memory.

The string is valid while the underlying memory buffer is valid.

See also: [al\\_ref\\_cstr](#), [al\\_ref\\_ustr](#)

### 25.5.3 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 `ALLEGRO_USTR` string. The information about the string (e.g. its size) is stored in the structure pointed to by the `info` parameter. The new 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)`. Both are byte offsets.

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](#)

## 25.6 Sizes and offsets

### 25.6.1 al\_ustr\_size

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

Return the size of the string in bytes. This is equal to the number of code points in the string if the string is empty or contains only 7-bit ASCII characters.

See also: [al\\_ustr\\_length](#)

### 25.6.2 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](#)

### 25.6.3 al\_ustr\_offset

```
int al_ustr_offset(const ALLEGRO_USTR *us, int index)
```

Return the byte offset (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](#)

### 25.6.4 `al_ustr_next`

```
bool al_ustr_next(const ALLEGRO_USTR *us, int *pos)
```

Find the byte offset of the next code point in string, beginning at \*pos. \*pos does not have to be at the beginning of a code point.

Returns true on success, and the 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, and \*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](#)

### 25.6.5 `al_ustr_prev`

```
bool al_ustr_prev(const ALLEGRO_USTR *us, int *pos)
```

Find the byte offset of the previous code point in string, before \*pos. \*pos does not have to be at the beginning of a code point. Returns true on success, then value pointed to by pos will be updated to the found offset. Otherwise returns false if \*pos was already at the end of the string, then \*pos is unmodified.

This function just looks for an appropriate byte; it doesn't check if found offset is the beginning of a valid code point. If you are working with possibly invalid UTF-8 strings then it could skip over some invalid bytes.

See also: [al\\_ustr\\_next](#)

## 25.7 Getting code points

### 25.7.1 `al_ustr_get`

```
int32_t al_ustr_get(const ALLEGRO_USTR *ub, int pos)
```

Return the code point in us beginning at byte offset 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](#)

### 25.7.2 `al_ustr_get_next`

```
int32_t al_ustr_get_next(const ALLEGRO_USTR *us, int *pos)
```

Find the code point in us beginning at byte offset \*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](#)

### 25.7.3 `al_ustr_prev_get`

```
int32_t al_ustr_prev_get(const ALLEGRO_USTR *us, int *pos)
```

Find the beginning of a code point before byte offset `*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](#)

## 25.8 Inserting into strings

### 25.8.1 `al_ustr_insert`

```
bool al_ustr_insert(ALLEGRO_USTR *us1, int pos, const ALLEGRO_USTR *us2)
```

Insert `us2` into `us1` beginning at byte offset `pos`. `pos` cannot be less than 0. If `pos` is past the end of `us1` then the space between the end of the string and `pos` will be padded with NUL (`'\0'`) bytes.

If required, use [al\\_ustr\\_offset](#) to find the byte offset for a given code point index.

Returns true on success, false on error.

See also: [al\\_ustr\\_insert\\_cstr](#), [al\\_ustr\\_insert\\_chr](#), [al\\_ustr\\_append](#), [al\\_ustr\\_offset](#)

### 25.8.2 `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 at byte offset `pos`.

See also: [al\\_ustr\\_insert](#), [al\\_ustr\\_insert\\_chr](#)

### 25.8.3 `al_ustr_insert_chr`

```
size_t al_ustr_insert_chr(ALLEGRO_USTR *us, int pos, int32_t c)
```

Insert a code point into `us` beginning at byte offset `pos`. `pos` cannot be less than 0. If `pos` is past the end of `us` then the space between the end of the string and `pos` will be padded with NUL (`'\0'`) bytes.

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

See also: [al\\_ustr\\_insert](#), [al\\_ustr\\_insert\\_cstr](#)

## 25.9 Appending to strings

### 25.9.1 `al_ustr_append`

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

Append `us2` to the end of `us1`.

Returns true on success, false on error.

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](#)

### 25.9.2 `al_ustr_append_cstr`

```
bool al_ustr_append_cstr(ALLEGRO_USTR *us, const char *s)
```

Append C-style string `s` to the end of `us`.

Returns true on success, false on error.

See also: [al\\_ustr\\_append](#)

### 25.9.3 `al_ustr_append_chr`

```
size_t al_ustr_append_chr(ALLEGRO_USTR *us, int32_t c)
```

Append a code point to the end of `us`.

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

See also: [al\\_ustr\\_append](#)

### 25.9.4 `al_ustr_appendf`

```
bool al_ustr_appendf(ALLEGRO_USTR *us, const char *fmt, ...)
```

This function appends formatted output to the string `us`. `fmt` is a printf-style format string. See [al\\_ustr\\_newf](#) about the “%s” and “%c” specifiers.

Returns true on success, false on error.

See also: [al\\_ustr\\_vappendf](#), [al\\_ustr\\_append](#)

### 25.9.5 `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](#), [al\\_ustr\\_append](#)

## 25.10 Removing parts of strings

### 25.10.1 `al_ustr_remove_chr`

```
bool al_ustr_remove_chr(ALLEGRO_USTR *us, int pos)
```

Remove the code point beginning at byte offset `pos`. Returns true on success. If `pos` is out of range or `pos` is not the beginning of a valid code point, returns false leaving the string unmodified.

Use [al\\_ustr\\_offset](#) to find the byte offset for a code-points offset.

See also: [al\\_ustr\\_remove\\_range](#)

### 25.10.2 `al_ustr_remove_range`

```
bool al_ustr_remove_range(ALLEGRO_USTR *us, int start_pos, int end_pos)
```

Remove the interval `[start_pos, end_pos)` from a string. `start_pos` and `end_pos` are byte offsets. Both may 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](#)



### 25.10.3 `al_ustr_truncate`

```
bool al_ustr_truncate(ALLEGRO_USTR *us, int start_pos)
```

Truncate a portion of a string at byte offset `start_pos` onwards. `start_pos` can be past the end of the string (has no effect) but cannot be less than 0.

Returns true on success, false on error.

See also: [al\\_ustr\\_remove\\_range](#), [al\\_ustr\\_ltrim\\_ws](#), [al\\_ustr\\_rtrim\\_ws](#), [al\\_ustr\\_trim\\_ws](#)

### 25.10.4 `al_ustr_ltrim_ws`

```
bool al_ustr_ltrim_ws(ALLEGRO_USTR *us)
```

Remove leading whitespace characters from a string, as defined by the C function `isspace()`.

Returns true on success, or false on error.

See also: [al\\_ustr\\_rtrim\\_ws](#), [al\\_ustr\\_trim\\_ws](#)

### 25.10.5 `al_ustr_rtrim_ws`

```
bool al_ustr_rtrim_ws(ALLEGRO_USTR *us)
```

Remove trailing (“right”) whitespace characters from a string, as defined by the C function `isspace()`.

Returns true on success, or false on error.

See also: [al\\_ustr\\_ltrim\\_ws](#), [al\\_ustr\\_trim\\_ws](#)

### 25.10.6 `al_ustr_trim_ws`

```
bool al_ustr_trim_ws(ALLEGRO_USTR *us)
```

Remove both leading and trailing whitespace characters from a string.

Returns true on success, or false on error.

See also: [al\\_ustr\\_ltrim\\_ws](#), [al\\_ustr\\_rtrim\\_ws](#)

## 25.11 Assigning one string to another

### 25.11.1 `al_ustr_assign`

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

Overwrite the string `us1` with another string `us2`. Returns true on success, false on error.

See also: [al\\_ustr\\_assign\\_substr](#), [al\\_ustr\\_assign\\_cstr](#)

### 25.11.2 `al_ustr_assign_substr`

```
bool al_ustr_assign_substr(ALLEGRO_USTR *us1, const ALLEGRO_USTR *us2,
    int start_pos, int end_pos)
```

Overwrite the string `us1` with the contents of `us2` in the byte interval `[start_pos, end_pos)`. The end points will be clamped 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](#)

### 25.11.3 `al_ustr_assign_cstr`

```
bool al_ustr_assign_cstr(ALLEGRO_USTR *us1, const char *s)
```

Overwrite the string `us` with the contents of the C-style string `s`. Returns true on success, false on error.

See also: [al\\_ustr\\_assign\\_substr](#), [al\\_ustr\\_assign\\_cstr](#)

## 25.12 Replacing parts of string

### 25.12.1 `al_ustr_set_chr`

```
size_t al_ustr_set_chr(ALLEGRO_USTR *us, int start_pos, int32_t c)
```

Replace the code point beginning at byte offset `pos` with `c`. `pos` cannot be less than 0. If `pos` is past the end of `us1` then the space between the end of the string and `pos` will be padded with NUL (`'\0'`) bytes. If `pos` is not the start of a valid code point, that is an error and the string will be unmodified.

On success, returns the number of bytes written, i.e. the offset to the following code point. On error, returns 0.

See also: [al\\_ustr\\_replace\\_range](#)

### 25.12.2 `al_ustr_replace_range`

```
bool al_ustr_replace_range(ALLEGRO_USTR *us1, int start_pos1, int end_pos1,
                           const ALLEGRO_USTR *us2)
```

Replace the part of `us1` in the byte interval `[start_pos, end_pos)` with the contents of `us2`. `start_pos` cannot be less than 0. If `start_pos` is past the end of `us1` then the space between the end of the string and `start_pos` will be padded with NUL (`'\0'`) bytes.

Use [al\\_ustr\\_offset](#) to find the byte offsets.

Returns true on success, false on error.

See also: [al\\_ustr\\_set\\_chr](#)

## 25.13 Searching

### 25.13.1 `al_ustr_find_chr`

```
int al_ustr_find_chr(const ALLEGRO_USTR *us, int start_pos, int32_t c)
```

Search for the encoding of code point `c` in `us` from byte offset `start_pos` (inclusive).

Returns the position where it is found or -1 if it is not found.

See also: [al\\_ustr\\_rfind\\_chr](#)

### 25.13.2 `al_ustr_rfind_chr`

```
int al_ustr_rfind_chr(const ALLEGRO_USTR *us, int end_pos, int32_t c)
```

Search for the encoding of code point `c` in `us` backwards from byte offset `end_pos` (exclusive). Returns the position where it is found or -1 if it is not found.

See also: [al\\_ustr\\_find\\_chr](#)

### 25.13.3 `al_ustr_find_set`

```
int al_ustr_find_set(const ALLEGRO_USTR *us, int start_pos,
                    const ALLEGRO_USTR *accept)
```

This function finds the first code point in `us`, beginning from byte offset `start_pos`, that matches any code point in `accept`. Returns the position if a code point was found. Otherwise returns -1.

See also: [al\\_ustr\\_find\\_set\\_cstr](#), [al\\_ustr\\_find\\_cset](#)

### 25.13.4 `al_ustr_find_set_cstr`

```
int al_ustr_find_set_cstr(const ALLEGRO_USTR *us, int start_pos,
                         const char *accept)
```

Like [al\\_ustr\\_find\\_set](#) but takes a C-style string for `accept`.

See also: [al\\_ustr\\_find\\_set](#), [al\\_ustr\\_find\\_cset\\_cstr](#)

### 25.13.5 `al_ustr_find_cset`

```
int al_ustr_find_cset(const ALLEGRO_USTR *us, int start_pos,
                     const ALLEGRO_USTR *reject)
```

This function finds the first code point in `us`, beginning from byte offset `start_pos`, that does *not* match any code point in `reject`. In other words it finds a code point in the complementary set of `reject`. Returns the byte position of that code point, if any. Otherwise returns -1.

See also: [al\\_ustr\\_find\\_cset\\_cstr](#), [al\\_ustr\\_find\\_set](#)

### 25.13.6 `al_ustr_find_cset_cstr`

```
int al_ustr_find_cset_cstr(const ALLEGRO_USTR *us, int start_pos,
                          const char *reject)
```

Like [al\\_ustr\\_find\\_cset](#) but takes a C-style string for `reject`.

See also: [al\\_ustr\\_find\\_cset](#), [al\\_ustr\\_find\\_set\\_cstr](#)

### 25.13.7 `al_ustr_find_str`

```
int al_ustr_find_str(const ALLEGRO_USTR *haystack, int start_pos,
                    const ALLEGRO_USTR *needle)
```

Find the first occurrence of string `needle` in `haystack`, beginning from byte offset `pos` (inclusive). Return the byte offset of the occurrence if it is found, otherwise return -1.

See also: [al\\_ustr\\_find\\_cstr](#), [al\\_ustr\\_rfind\\_str](#), [al\\_ustr\\_find\\_replace](#)

### 25.13.8 `al_ustr_find_cstr`

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

Like [al\\_ustr\\_find\\_str](#) but takes a C-style string for `needle`.

See also: [al\\_ustr\\_find\\_str](#), [al\\_ustr\\_rfind\\_cstr](#)

### 25.13.9 `al_ustr_rfind_str`

```
int al_ustr_rfind_str(const ALLEGRO_USTR *haystack, int end_pos,  
    const ALLEGRO_USTR *needle)
```

Find the last occurrence of string `needle` in `haystack` before byte offset `end_pos` (exclusive). Return the byte offset of the occurrence if it is found, otherwise return -1.

See also: [al\\_ustr\\_rfind\\_cstr](#), [al\\_ustr\\_find\\_str](#)

### 25.13.10 `al_ustr_rfind_cstr`

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

Like [al\\_ustr\\_rfind\\_str](#) but takes a C-style string for `needle`.

See also: [al\\_ustr\\_rfind\\_str](#), [al\\_ustr\\_find\\_cstr](#)

### 25.13.11 `al_ustr_find_replace`

```
bool al_ustr_find_replace(ALLEGRO_USTR *us, int start_pos,  
    const ALLEGRO_USTR *find, const ALLEGRO_USTR *replace)
```

Replace all occurrences of `find` in `us` with `replace`, beginning at byte offset `start_pos`. The `find` string must be non-empty. Returns true on success, false on error.

See also: [al\\_ustr\\_find\\_replace\\_cstr](#)

### 25.13.12 `al_ustr_find_replace_cstr`

```
bool al_ustr_find_replace_cstr(ALLEGRO_USTR *us, int start_pos,  
    const char *find, const char *replace)
```

Like [al\\_ustr\\_find\\_replace](#) but takes C-style strings for `find` and `replace`.

## 25.14 Comparing

### 25.14.1 `al_ustr_equal`

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

Return true iff the two strings are equal. This function is more efficient than [al\\_ustr\\_compare](#) so is preferable if ordering is not important.

See also: [al\\_ustr\\_compare](#)

### 25.14.2 `al_ustr_compare`

```
int al_ustr_compare(const ALLEGRO_USTR *us1, const ALLEGRO_USTR *us2)
```

This function compares `us1` and `us2` by code point values. Returns zero if the strings are equal, a positive number if `us1` comes after `us2`, else a negative number.

This does *not* take into account locale-specific sorting rules. For that you will need to use another library.

See also: [al\\_ustr\\_ncompare](#), [al\\_ustr\\_equal](#)

### 25.14.3 `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.

See also: `al_ustr_compare`, `al_ustr_equal`

### 25.14.4 `al_ustr_has_prefix`

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

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

See also: `al_ustr_has_prefix_cstr`, `al_ustr_has_suffix`

### 25.14.5 `al_ustr_has_prefix_cstr`

```
bool al_ustr_has_prefix_cstr(const ALLEGRO_USTR *us1, const char *s2)
```

Returns true iff `us1` begins with `s2`.

See also: `al_ustr_has_prefix`, `al_ustr_has_suffix_cstr`

### 25.14.6 `al_ustr_has_suffix`

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

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

See also: `al_ustr_has_suffix_cstr`, `al_ustr_has_prefix`

### 25.14.7 `al_ustr_has_suffix_cstr`

```
bool al_ustr_has_suffix_cstr(const ALLEGRO_USTR *us1, const char *s2)
```

Returns true iff `us1` ends with `s2`.

See also: `al_ustr_has_suffix`, `al_ustr_has_prefix_cstr`

## 25.15 UTF-16 conversion

### 25.15.1 `al_ustr_new_from_utf16`

```
ALLEGRO_USTR *al_ustr_new_from_utf16(uint16_t const *s)
```

Create a new string containing a copy of the 0-terminated string `s` which must be encoded as UTF-16. The string must eventually be freed with `al_ustr_free`.

See also: `al_ustr_new`

### 25.15.2 `al_ustr_size_utf16`

```
size_t al_ustr_size_utf16(const ALLEGRO_USTR *us)
```

Returns the number of bytes required to encode the string in UTF-16 (including the terminating 0).

Usually called before `al_ustr_encode_utf16` to determine the size of the buffer to allocate.

See also: `al_ustr_size`

### 25.15.3 `al_ustr_encode_utf16`

```
size_t al_ustr_encode_utf16(const ALLEGRO_USTR *us, uint16_t *s,  
    size_t n)
```

Encode the string into the given buffer, in UTF-16. Returns the number of bytes written. There are never more than `n` bytes written. The minimum size to encode the complete string can be queried with [al\\_ustr\\_size\\_utf16](#). 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](#)

## 25.16 Low-level UTF-8 routines

### 25.16.1 `al_utf8_width`

```
size_t al_utf8_width(int c)
```

Returns the number of bytes that would be occupied by the specified code point when encoded in UTF-8. This is between 1 and 4 bytes for legal code point values. Otherwise returns 0.

See also: [al\\_utf8\\_encode](#), [al\\_utf16\\_width](#)

### 25.16.2 `al_utf8_encode`

```
size_t al_utf8_encode(char s[], int32_t c)
```

Encode the specified code point to UTF-8 into the buffer `s`. The buffer must have enough space to hold the encoding, which takes between 1 and 4 bytes. This routine will refuse to encode code points above 0x10FFFF.

Returns the number of bytes written, which is the same as that returned by [al\\_utf8\\_width](#).

See also: [al\\_utf16\\_encode](#)

## 25.17 Low-level UTF-16 routines

### 25.17.1 `al_utf16_width`

```
size_t al_utf16_width(int c)
```

Returns the number of bytes that would be occupied by the specified code point when encoded in UTF-16. This is either 2 or 4 bytes for legal code point values. Otherwise returns 0.

See also: [al\\_utf16\\_encode](#), [al\\_utf8\\_width](#)

### 25.17.2 `al_utf16_encode`

```
size_t al_utf16_encode(uint16_t s[], int32_t c)
```

Encode the specified code point to UTF-16 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](#)

## Platform-specific functions

### 26.1 Windows

These functions are declared in the following header file:

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

#### 26.1.1 `al_get_win_window_handle`

```
HWND al_get_win_window_handle(ALLEGRO_DISPLAY *display)
```

Returns the handle to the window that the passed display is using.

#### 26.1.2 `al_win_add_window_callback`

```
bool al_win_add_window_callback(ALLEGRO_DISPLAY *display,
    bool (*callback)(ALLEGRO_DISPLAY *, UINT, WPARAM, LPARAM, void *), void *userdata)
```

The specified callback function will intercept the window's message before Allegro processes it. If the callback function consumes the event, then it should return true. In that case, Allegro will not do anything with the event.

The userdata pointer can be used to supply additional context to the callback function.

The callbacks are executed in the same order they were added.

Returns true if the callback was added.

Since: 5.1.2

#### 26.1.3 `al_win_remove_window_callback`

```
bool al_win_remove_window_callback(ALLEGRO_DISPLAY *display,
    bool (*callback)(ALLEGRO_DISPLAY *, UINT, WPARAM, LPARAM, void *), void *userdata)
```

Removes the callback that was previously registered with `al_win_add_window_callback`. The userdata pointer must be the same as what was used during the registration of the callback.

Returns true if the callback was removed.

Since: 5.1.2

### 26.2 Mac OS X

These functions are declared in the following header file:

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

### 26.2.1 `al_osx_get_window`

```
NSWindow* al_osx_get_window(ALLEGRO_DISPLAY *display)
```

Retrieves the `NSWindow` handle associated with the Allegro display.

Since: 5.0.8, 5.1.3

## 26.3 iPhone

These functions are declared in the following header file:

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

### 26.3.1 `al_iphone_override_screen_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`.

### 26.3.2 `al_iphone_set_statusbar_orientation`

```
void al_iphone_set_statusbar_orientation(int o)
```

Sets the orientation of the status bar, which can be one of the following:

- `ALLEGRO_IPHONE_STATUSBAR_ORIENTATION_PORTRAIT`
- `ALLEGRO_IPHONE_STATUSBAR_ORIENTATION_PORTRAIT_UPSIDE_DOWN`
- `ALLEGRO_IPHONE_STATUSBAR_ORIENTATION_LANDSCAPE_RIGHT`
- `ALLEGRO_IPHONE_STATUSBAR_ORIENTATION_LANDSCAPE_LEFT`

Since: 5.1.0

### 26.3.3 `al_iphone_get_last_shake_time`

```
double al_iphone_get_last_shake_time(void)
```

Returns the time as reported by `al_get_time` when the device was last shaken.

Since: 5.1.0

### 26.3.4 `al_iphone_get_battery_level`

```
float al_iphone_get_battery_level(void)
```

Returns the level of the charge of the battery as a number between 0.0 (empty) and 1.0 (full). If the device is currently being charged or connected to a power supply, this function will also return 1.0 regardless of the real battery charge level.

Since: 5.1.0



### 26.3.5 `al_iphone_get_screen_scale`

Gets the current scaling factor of the screen.

Since: 5.1.0

See also: [al\\_iphone\\_override\\_screen\\_scale](#)

### 26.3.6 `al_iphone_get_view`

```
UIView *al_iphone_get_view(ALLEGRO_DISPLAY *display)
```

Retrieves the `UIView*` (`EAGLView*`) associated with the Allegro display.

Since: 5.1.0

### 26.3.7 `al_iphone_get_window`

```
UIWindow *al_iphone_get_window(ALLEGRO_DISPLAY *display)
```

Retrieves the `UIWindow*` associated with the Allegro display.

Since: 5.1.0

## 26.4 Android

These functions are declared in the following header file:

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

### 26.4.1 `al_android_set_apk_file_interface`

```
void al_android_set_apk_file_interface(void)
```

This function will set up a custom `ALLEGRO_FILE_INTERFACE` that makes all future calls of `al_fopen` read from the applications's APK file.

*Note:* Currently, access to the APK file after calling this function is read only.

Since: 5.1.2

### 26.4.2 `al_android_get_os_version`

```
const char *al_android_get_os_version(void)
```

Returns a pointer to a static buffer that contains the version string of the Android platform that the calling Allegro program is running on.

Since: 5.1.2



## Direct3D integration

These functions are declared in the following header file:

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

### 27.1 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.

### 27.2 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.

### 27.3 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.

### 27.4 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 supports NPOT textures, false otherwise.

### 27.5 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 supports non-square textures, false otherwise.

## 27.6 `al_get_d3d_texture_size`

```
bool al_get_d3d_texture_size(ALLEGRO_BITMAP *bitmap, int *width, int *height)
```

Retrieves the size of the Direct3D texture used for the bitmap.

Returns true on success, false on failure. Zero width and height are returned if the bitmap is not a Direct3D bitmap.

Since: 5.1.0

See also: [al\\_get\\_d3d\\_texture\\_position](#)

## 27.7 `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

See also: [al\\_get\\_d3d\\_texture\\_size](#)

## 27.8 `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

## 27.9 `al_set_d3d_device_release_callback`

```
void al_set_d3d_device_release_callback(  
    void (*callback)(ALLEGRO_DISPLAY *display))
```

The callback will be called whenever a D3D device is reset (minimize, toggle fullscreen window, etc). In the callback you should release any d3d resources you have created yourself. The callback receives the affected display as a parameter.

Pass NULL to disable the callback.

Since: 5.1.0

## 27.10 `al_set_d3d_device_restore_callback`

```
void al_set_d3d_device_restore_callback(  
    void (*callback)(ALLEGRO_DISPLAY *display))
```

The callback will be called whenever a D3D device that has been reset is restored. In the callback you should restore any d3d resources you have created yourself. The callback receives the affected display as a parameter.

Pass NULL to disable the callback.

Since: 5.1.0



## OpenGL integration

These functions are declared in the following header file:

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

### 28.1 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.

### 28.2 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.

### 28.3 `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);
```

### 28.4 `al_get_opengl_texture_size`

```
bool 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.

Returns true on success, false on failure. Zero width and height are returned if the bitmap is not an OpenGL bitmap.

See also: [al\\_get\\_opengl\\_texture\\_position](#)

### 28.5 `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.

See also: [al\\_get\\_opengl\\_texture\\_size](#)

### 28.6 `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`

## 28.7 `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`

## 28.8 `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.

## 28.9 `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`

## 28.10 `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`

### 28.11 `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 it if 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`.

### 28.12 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).

## Audio addon

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

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

### 29.1 Audio types

#### 29.1.1 ALLEGRO\_AUDIO\_DEPTH

```
enum ALLEGRO_AUDIO_DEPTH
```

Sample depth and type, and signedness. Mixers only use 32-bit signed float (-1..+1), or 16-bit signed integers. The unsigned value is a bit-flag applied to the depth value.

- `ALLEGRO_AUDIO_DEPTH_INT8`
- `ALLEGRO_AUDIO_DEPTH_INT16`
- `ALLEGRO_AUDIO_DEPTH_INT24`
- `ALLEGRO_AUDIO_DEPTH_FLOAT32`
- `ALLEGRO_AUDIO_DEPTH_UNSIGNED`

For convenience:

- `ALLEGRO_AUDIO_DEPTH_UINT8`
- `ALLEGRO_AUDIO_DEPTH_UINT16`
- `ALLEGRO_AUDIO_DEPTH_UINT24`

#### 29.1.2 ALLEGRO\_AUDIO\_PAN\_NONE

```
#define ALLEGRO_AUDIO_PAN_NONE (-1000.0f)
```

A special value for the pan property of samples and audio streams. Use this value to disable panning on samples and audio streams, and play them without attenuation implied by panning support.

`ALLEGRO_AUDIO_PAN_NONE` is different from a pan value of 0.0 (centered) because, when panning is enabled, we try to maintain a constant sound power level as a sample is panned from left to right. A sound coming out of one speaker should sound as loud as it does when split over two speakers. As a consequence, a sample with pan value 0.0 will be 3 dB softer than the original level.

(Please correct us if this is wrong.)

### 29.1.3 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

### 29.1.4 ALLEGRO\_MIXER

**typedef struct** ALLEGRO\_MIXER ALLEGRO\_MIXER;

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

### 29.1.5 ALLEGRO\_MIXER\_QUALITY

**enum** ALLEGRO\_MIXER\_QUALITY

- ALLEGRO\_MIXER\_QUALITY\_POINT - point sampling
- ALLEGRO\_MIXER\_QUALITY\_LINEAR - linear interpolation
- ALLEGRO\_MIXER\_QUALITY\_CUBIC - cubic interpolation (since: 5.0.8, 5.1.4)

### 29.1.6 ALLEGRO\_PLAYMODE

**enum** ALLEGRO\_PLAYMODE

Sample and stream playback mode.

- ALLEGRO\_PLAYMODE\_ONCE
- ALLEGRO\_PLAYMODE\_LOOP
- ALLEGRO\_PLAYMODE\_BIDIR

### 29.1.7 ALLEGRO\_AUDIO\_EVENT\_TYPE

Defines types of audio events that can be retrieved from the audio event source. Not all audio drivers will generate these events.

- ALLEGRO\_EVENT\_AUDIO\_ROUTE\_CHANGE
- ALLEGRO\_EVENT\_AUDIO\_INTERRUPT
- ALLEGRO\_EVENT\_AUDIO\_END\_INTERRUPT

Since: 5.1.0

### 29.1.8 ALLEGRO\_SAMPLE\_ID

**typedef struct** ALLEGRO\_SAMPLE\_ID ALLEGRO\_SAMPLE\_ID;

An ALLEGRO\_SAMPLE\_ID represents a sample being played via [al\\_play\\_sample](#). It can be used to later stop the sample with [al\\_stop\\_sample](#).

### 29.1.9 ALLEGRO\_SAMPLE

```
typedef struct ALLEGRO_SAMPLE ALLEGRO_SAMPLE;
```

An `ALLEGRO_SAMPLE` object stores the data necessary for playing pre-defined digital audio. It holds information pertaining to data length, frequency, channel configuration, etc. You can have an `ALLEGRO_SAMPLE` object playing multiple times simultaneously. The object holds a user-specified PCM data buffer, of the format the object is created with.

See also: [ALLEGRO\\_SAMPLE\\_INSTANCE](#)

### 29.1.10 ALLEGRO\_SAMPLE\_INSTANCE

```
typedef struct ALLEGRO_SAMPLE_INSTANCE ALLEGRO_SAMPLE_INSTANCE;
```

An `ALLEGRO_SAMPLE_INSTANCE` object represents a playable instance of a predefined sound effect. It holds information pertaining to the looping mode, loop start/end points, playing position, etc. An instance uses the data from an [ALLEGRO\\_SAMPLE](#) 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](#)

### 29.1.11 ALLEGRO\_AUDIO\_STREAM

```
typedef struct ALLEGRO_AUDIO_STREAM ALLEGRO_AUDIO_STREAM;
```

An `ALLEGRO_AUDIO_STREAM` object is used to stream generated audio to the sound device, in real-time. This is done by reading from a buffer, which is split into a number of fragments. Whenever a fragment has finished playing, the user can refill it with new data.

As with [ALLEGRO\\_SAMPLE\\_INSTANCE](#) 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.

### 29.1.12 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](#)

## 29.2 Setting up audio

### 29.2.1 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](#)

### 29.2.2 al\_uninstall\_audio

```
void al_uninstall_audio(void)
```

Uninstalls the audio subsystem.

See also: [al\\_install\\_audio](#)

### 29.2.3 al\_is\_audio\_installed

```
bool al_is_audio_installed(void)
```

Returns true if [al\\_install\\_audio](#) was called previously and returned successfully.

### 29.2.4 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](#)

## 29.3 Misc audio functions

### 29.3.1 `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](#).

### 29.3.2 `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](#).

### 29.3.3 `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](#).

### 29.3.4 `al_fill_silence`

```
void al_fill_silence(void *buf, unsigned int samples,
    ALLEGRO_AUDIO_DEPTH depth, ALLEGRO_CHANNEL_CONF chan_conf)
```

Fill a buffer with silence, for the given format and channel configuration. The buffer must have enough space for the given number of samples, and be properly aligned.

Since: 5.1.8

## 29.4 Voice functions

### 29.4.1 `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](#)

### 29.4.2 `al_destroy_voice`

```
void al_destroy_voice(ALLEGRO_VOICE *voice)
```

Destroys the voice and deallocates it from the digital driver. Does nothing if the voice is NULL.

See also: [al\\_create\\_voice](#)

### 29.4.3 `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](#)

### 29.4.4 `al_attach_audio_stream_to_voice`

```
bool al_attach_audio_stream_to_voice(ALLEGRO_AUDIO_STREAM *stream,  
    ALLEGRO_VOICE *voice)
```

Attaches an audio stream to a voice. The same rules as [al\\_attach\\_sample\\_instance\\_to\\_voice](#) 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](#)

### 29.4.5 `al_attach_mixer_to_voice`

```
bool al_attach_mixer_to_voice(ALLEGRO_MIXER *mixer, ALLEGRO_VOICE *voice)
```

Attaches a mixer to a voice. The same rules as [al\\_attach\\_sample\\_instance\\_to\\_voice](#) apply, with the exception of the depth requirement.

Returns true on success, false on failure.

See also: [al\\_detach\\_voice](#)

### 29.4.6 `al_attach_sample_instance_to_voice`

```
bool al_attach_sample_instance_to_voice(ALLEGRO_SAMPLE_INSTANCE *spl,  
    ALLEGRO_VOICE *voice)
```

Attaches a sample to a voice, and allows it to play. The sample's volume and loop mode will be ignored, and it must have the same frequency and depth (including signed-ness) as the voice. This function may fail if the selected driver doesn't support preloading sample data.

At this time, we don't recommend attaching samples directly to voices. Use a mixer in between.

Returns true on success, false on failure.

See also: [al\\_detach\\_voice](#)

### 29.4.7 `al_get_voice_frequency`

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

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

### 29.4.8 `al_get_voice_channels`

```
ALLEGRO_CHANNEL_CONF al_get_voice_channels(const ALLEGRO_VOICE *voice)
```

Return the channel configuration of the voice.

See also: [ALLEGRO\\_CHANNEL\\_CONF](#).



### 29.4.9 al\_get\_voice\_depth

```
ALLEGRO_AUDIO_DEPTH al_get_voice_depth(const ALLEGRO_VOICE *voice)
```

Return the audio depth of the voice.

See also: [ALLEGRO\\_AUDIO\\_DEPTH](#).

### 29.4.10 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](#)

### 29.4.11 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](#)

### 29.4.12 al\_get\_voice\_position

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

When the voice has a non-streaming object attached to it, e.g. a sample, returns the voice's current sample position. Otherwise, returns zero.

See also: [al\\_set\\_voice\\_position](#).

### 29.4.13 al\_set\_voice\_position

```
bool al_set_voice_position(ALLEGRO_VOICE *voice, unsigned int val)
```

Set the voice position. This can only work if the voice has a non-streaming object attached to it, e.g. a sample instance.

Returns true on success, false on failure.

See also: [al\\_get\\_voice\\_position](#).

## 29.5 Sample functions

### 29.5.1 al\_create\_sample

```
ALLEGRO_SAMPLE *al_create_sample(void *buf, unsigned int samples,
    unsigned int freq, ALLEGRO_AUDIO_DEPTH depth,
    ALLEGRO_CHANNEL_CONF chan_conf, bool free_buf)
```

Create a sample data structure from the supplied buffer. If `free_buf` is true then the buffer will be freed 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:

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

See also: [al\\_destroy\\_sample](#), [ALLEGRO\\_AUDIO\\_DEPTH](#), [ALLEGRO\\_CHANNEL\\_CONF](#)

### 29.5.2 [al\\_destroy\\_sample](#)

```
void al_destroy_sample(ALLEGRO_SAMPLE *spl)
```

Free the sample data structure. If it was created with the `free_buf` parameter set to true, then the buffer will be freed 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](#)

### 29.5.3 [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](#).

### 29.5.4 [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](#)

### 29.5.5 [al\\_stop\\_samples](#)

```
void al_stop_samples(void)
```

Stop all samples started by [al\\_play\\_sample](#).

See also: [al\\_stop\\_sample](#)

### 29.5.6 `al_get_sample_channels`

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

Return the channel configuration.

See also: [ALLEGRO\\_CHANNEL\\_CONF](#), [al\\_get\\_sample\\_depth](#), [al\\_get\\_sample\\_frequency](#), [al\\_get\\_sample\\_length](#), [al\\_get\\_sample\\_data](#)

### 29.5.7 `al_get_sample_depth`

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

Return the audio depth.

See also: [ALLEGRO\\_AUDIO\\_DEPTH](#), [al\\_get\\_sample\\_channels](#), [al\\_get\\_sample\\_frequency](#), [al\\_get\\_sample\\_length](#), [al\\_get\\_sample\\_data](#)

### 29.5.8 `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](#)

### 29.5.9 `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](#)

### 29.5.10 `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](#)

## 29.6 Sample instance functions

### 29.6.1 `al_create_sample_instance`

```
ALLEGRO_SAMPLE_INSTANCE *al_create_sample_instance(ALLEGRO_SAMPLE *sample_data)
```

Creates a sample stream, using the supplied data. This must be attached to a voice or mixer before it can be played. The argument may be NULL. You can then set the data later with [al\\_set\\_sample](#).

See also: [al\\_destroy\\_sample\\_instance](#)

### 29.6.2 `al_destroy_sample_instance`

```
void al_destroy_sample_instance(ALLEGRO_SAMPLE_INSTANCE *spl)
```

Detaches the sample stream from anything it may be attached to and frees it (the sample data is *not* freed!).

See also: [al\\_create\\_sample\\_instance](#)

### 29.6.3 `al_play_sample_instance`

```
bool al_play_sample_instance(ALLEGRO_SAMPLE_INSTANCE *spl)
```

Play an instance of a sample data. Returns true on success, false on failure.

See also: [al\\_stop\\_sample\\_instance](#)

### 29.6.4 `al_stop_sample_instance`

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

Stop an sample instance playing.

See also: [al\\_play\\_sample\\_instance](#)

### 29.6.5 `al_get_sample_instance_channels`

```
ALLEGRO_CHANNEL_CONF al_get_sample_instance_channels(  
    const ALLEGRO_SAMPLE_INSTANCE *spl)
```

Return the channel configuration.

See also: [ALLEGRO\\_CHANNEL\\_CONF](#).

### 29.6.6 `al_get_sample_instance_depth`

```
ALLEGRO_AUDIO_DEPTH al_get_sample_instance_depth(const ALLEGRO_SAMPLE_INSTANCE *spl)
```

Return the audio depth.

See also: [ALLEGRO\\_AUDIO\\_DEPTH](#).

### 29.6.7 `al_get_sample_instance_frequency`

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

Return the frequency of the sample instance.

### 29.6.8 `al_get_sample_instance_length`

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

Return the length of the sample instance in sample values.

See also: [al\\_set\\_sample\\_instance\\_length](#), [al\\_get\\_sample\\_instance\\_time](#)

### 29.6.9 `al_set_sample_instance_length`

```
bool al_set_sample_instance_length(ALLEGRO_SAMPLE_INSTANCE *spl,  
    unsigned int val)
```

Set the length of the sample instance in sample values.

Return true on success, false on failure. Will fail if the sample instance is currently playing.

See also: [al\\_get\\_sample\\_instance\\_length](#)

**29.6.10 al\_get\_sample\_instance\_position**

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

Get the playback position of a sample instance.

See also: [al\\_set\\_sample\\_instance\\_position](#)

**29.6.11 al\_set\_sample\_instance\_position**

```
bool al_set_sample_instance_position(ALLEGRO_SAMPLE_INSTANCE *spl,
    unsigned int val)
```

Set the playback position of a sample instance.

Returns true on success, false on failure.

See also: [al\\_get\\_sample\\_instance\\_position](#)

**29.6.12 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](#)

**29.6.13 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](#)

**29.6.14 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](#)

**29.6.15 al\_set\_sample\_instance\_gain**

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

Set the playback gain.

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

See also: [al\\_get\\_sample\\_instance\\_gain](#)

**29.6.16 al\_get\_sample\_instance\_pan**

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

Get the pan value.

See also: [al\\_set\\_sample\\_instance\\_pan](#).

### 29.6.17 `al_set_sample_instance_pan`

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

Set the pan value on a sample instance. A value of -1.0 means to play the sample only through the left speaker; +1.0 means only through the right speaker; 0.0 means the sample is centre balanced. A special value `ALLEGRO_AUDIO_PAN_NONE` disables panning and plays the sample at its original level. This will be louder than a pan value of 0.0.

Note: panning samples with more than two channels doesn't work yet.

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\\_pan](#), [ALLEGRO\\_AUDIO\\_PAN\\_NONE](#)

### 29.6.18 `al_get_sample_instance_time`

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

Return the length of the sample instance in seconds, assuming a playback speed of 1.0.

See also: [al\\_get\\_sample\\_instance\\_length](#)

### 29.6.19 `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](#)

### 29.6.20 `al_set_sample_instance_playmode`

```
bool al_set_sample_instance_playmode(ALLEGRO_SAMPLE_INSTANCE *spl,  
    ALLEGRO_PLAYMODE val)
```

Set the playback mode.

Returns true on success, false on failure.

See also: [ALLEGRO\\_PLAYMODE](#), [al\\_get\\_sample\\_instance\\_playmode](#)

### 29.6.21 `al_get_sample_instance_playing`

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

Return true if the sample instance is in the playing state. This may be true even if the instance is not attached to anything.

See also: [al\\_set\\_sample\\_instance\\_playing](#)

### 29.6.22 `al_set_sample_instance_playing`

```
bool al_set_sample_instance_playing(ALLEGRO_SAMPLE_INSTANCE *spl, bool val)
```

Change whether the sample instance is playing.

The instance does not need to be attached to anything (since: 5.1.8).

Returns true on success, false on failure.

See also: [al\\_get\\_sample\\_instance\\_playing](#)

### 29.6.23 `al_get_sample_instance_attached`

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

Return whether the sample instance is attached to something.

See also: [al\\_attach\\_sample\\_instance\\_to\\_mixer](#), [al\\_attach\\_sample\\_instance\\_to\\_voice](#), [al\\_detach\\_sample\\_instance](#)

### 29.6.24 `al_detach_sample_instance`

```
bool al_detach_sample_instance(ALLEGRO_SAMPLE_INSTANCE *spl)
```

Detach the sample instance from whatever it's attached to, if anything.

Returns true on success.

See also: [al\\_attach\\_sample\\_instance\\_to\\_mixer](#), [al\\_attach\\_sample\\_instance\\_to\\_voice](#), [al\\_get\\_sample\\_instance\\_attached](#)

### 29.6.25 `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](#)

### 29.6.26 `al_set_sample`

```
bool al_set_sample(ALLEGRO_SAMPLE_INSTANCE *spl, ALLEGRO_SAMPLE *data)
```

Change the sample data that a sample instance plays. This can be quite an involved process.

First, the sample is stopped if it is not already.

Next, if data is NULL, the sample is detached from its parent (if any).

If data is not NULL, the sample may be detached and reattached to its parent (if any). This is not necessary if the old sample data and new sample data have the same frequency, depth and channel configuration. Reattaching may not always succeed.

On success, the sample remains stopped. The playback position and loop end points are reset to their default values. The loop mode remains unchanged.

Returns true on success, false on failure. On failure, the sample will be stopped and detached from its parent.

See also: [al\\_get\\_sample](#)

## 29.7 Mixer functions

### 29.7.1 `al_create_mixer`

```
ALLEGRO_MIXER *al_create_mixer(unsigned int freq,
    ALLEGRO_AUDIO_DEPTH depth, ALLEGRO_CHANNEL_CONF chan_conf)
```

Creates a mixer stream, to attach sample streams or other mixers to. It will mix into a buffer at the requested frequency and channel count.

The only supported audio depths are `ALLEGRO_AUDIO_DEPTH_FLOAT32` and `ALLEGRO_AUDIO_DEPTH_INT16` (not yet complete).

Returns true on success, false on error.

See also: [al\\_destroy\\_mixer](#), `ALLEGRO_AUDIO_DEPTH`, `ALLEGRO_CHANNEL_CONF`

### 29.7.2 `al_destroy_mixer`

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

Destroys the mixer stream.

See also: [al\\_create\\_mixer](#)

### 29.7.3 `al_get_default_mixer`

```
ALLEGRO_MIXER *al_get_default_mixer(void)
```

Return the default mixer, or NULL if one has not been set. Although different configurations of mixers and voices can be used, in most cases a single mixer attached to a voice is what you want. The default mixer is used by [al\\_play\\_sample](#).

See also: [al\\_reserve\\_samples](#), [al\\_play\\_sample](#), [al\\_set\\_default\\_mixer](#), [al\\_restore\\_default\\_mixer](#)

### 29.7.4 `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](#)

### 29.7.5 `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](#).

### 29.7.6 `al_attach_mixer_to_mixer`

```
bool al_attach_mixer_to_mixer(ALLEGRO_MIXER *stream, ALLEGRO_MIXER *mixer)
```

Attaches the mixer passed as the first argument onto the mixer passed as the second argument. 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.

It is invalid to attach a mixer to itself.

See also: [al\\_detach\\_mixer](#).



### 29.7.7 `al_attach_sample_instance_to_mixer`

```
bool al_attach_sample_instance_to_mixer(ALLEGRO_SAMPLE_INSTANCE *spl,  
    ALLEGRO_MIXER *mixer)
```

Attach a sample instance to a mixer. The instance must not already be attached to anything.

Returns true on success, false on failure.

See also: [al\\_detach\\_sample\\_instance](#).

### 29.7.8 `al_attach_audio_stream_to_mixer`

```
bool al_attach_audio_stream_to_mixer(ALLEGRO_AUDIO_STREAM *stream, ALLEGRO_MIXER *mixer)
```

Attach a stream to a mixer.

Returns true on success, false on failure.

See also: [al\\_detach\\_audio\\_stream](#).

### 29.7.9 `al_get_mixer_frequency`

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

Return the mixer frequency.

See also: [al\\_set\\_mixer\\_frequency](#)

### 29.7.10 `al_set_mixer_frequency`

```
bool al_set_mixer_frequency(ALLEGRO_MIXER *mixer, unsigned int val)
```

Set the mixer frequency. This will only work if the mixer is not attached to anything.

Returns true on success, false on failure.

See also: [al\\_get\\_mixer\\_frequency](#)

### 29.7.11 `al_get_mixer_channels`

```
ALLEGRO_CHANNEL_CONF al_get_mixer_channels(const ALLEGRO_MIXER *mixer)
```

Return the mixer channel configuration.

See also: [ALLEGRO\\_CHANNEL\\_CONF](#).

### 29.7.12 `al_get_mixer_depth`

```
ALLEGRO_AUDIO_DEPTH al_get_mixer_depth(const ALLEGRO_MIXER *mixer)
```

Return the mixer audio depth.

See also: [ALLEGRO\\_AUDIO\\_DEPTH](#).

### 29.7.13 `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](#).

### 29.7.14 `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](#)

### 29.7.15 `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](#)

### 29.7.16 `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](#)

### 29.7.17 `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](#).

### 29.7.18 `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](#).

### 29.7.19 `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](#)

### 29.7.20 `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](#).

### 29.7.21 `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.

## 29.8 Stream functions

### 29.8.1 `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:

$$\text{delay} = \text{fragment\_count} * \text{frag\_samples} / \text{freq}$$

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.

### 29.8.2 `al_destroy_audio_stream`

```
void al_destroy_audio_stream(ALLEGRO_AUDIO_STREAM *stream)
```

Destroy an audio stream which was created with `al_create_audio_stream` 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`.

### 29.8.3 `al_get_audio_stream_event_source`

```
ALLEGRO_EVENT_SOURCE *al_get_audio_stream_event_source(  
    ALLEGRO_AUDIO_STREAM *stream)
```

Retrieve the associated event source.

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

### 29.8.4 `al_drain_audio_stream`

```
void al_drain_audio_stream(ALLEGRO_AUDIO_STREAM *stream)
```

You should call this to finalise an audio stream that you will no longer be feeding, to wait for all pending buffers to finish playing. The stream's playing state will change to false.

See also: `al_destroy_audio_stream`

### 29.8.5 `al_rewind_audio_stream`

```
bool al_rewind_audio_stream(ALLEGRO_AUDIO_STREAM *stream)
```

Set the streaming file playing position to the beginning. Returns true on success. Currently this can only be called on streams created with `al_load_audio_stream`, `al_load_audio_stream_f` and the format-specific functions underlying those functions.

### 29.8.6 `al_get_audio_stream_frequency`

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

Return the stream frequency.

### 29.8.7 `al_get_audio_stream_channels`

```
ALLEGRO_CHANNEL_CONF al_get_audio_stream_channels(  
    const ALLEGRO_AUDIO_STREAM *stream)
```

Return the stream channel configuration.

See also: `ALLEGRO_CHANNEL_CONF`.

### 29.8.8 `al_get_audio_stream_depth`

```
ALLEGRO_AUDIO_DEPTH al_get_audio_stream_depth(  
    const ALLEGRO_AUDIO_STREAM *stream)
```

Return the stream audio depth.

See also: `ALLEGRO_AUDIO_DEPTH`.

**29.8.9 al\_get\_audio\_stream\_length**

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

Return the stream length in samples.

**29.8.10 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](#).

**29.8.11 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](#).

**29.8.12 al\_get\_audio\_stream\_gain**

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

Return the playback gain.

See also: [al\\_set\\_audio\\_stream\\_gain](#).

**29.8.13 al\_set\_audio\_stream\_gain**

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

Set the playback gain.

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

See also: [al\\_get\\_audio\\_stream\\_gain](#).

**29.8.14 al\_get\_audio\_stream\_pan**

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

Get the pan value.

See also: [al\\_set\\_audio\\_stream\\_pan](#).

**29.8.15 al\_set\_audio\_stream\_pan**

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

Set the pan value on an audio stream. A value of -1.0 means to play the stream only through the left speaker; +1.0 means only through the right speaker; 0.0 means the sample is centre balanced. A special value `ALLEGRO_AUDIO_PAN_NONE` disables panning and plays the stream at its original level. This will be louder than a pan value of 0.0.

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\\_pan](#), `ALLEGRO_AUDIO_PAN_NONE`

### 29.8.16 `al_get_audio_stream_playing`

```
bool al_get_audio_stream_playing(const ALLEGRO_AUDIO_STREAM *stream)
```

Return true if the stream is playing.

See also: [al\\_set\\_audio\\_stream\\_playing](#).

### 29.8.17 `al_set_audio_stream_playing`

```
bool al_set_audio_stream_playing(ALLEGRO_AUDIO_STREAM *stream, bool val)
```

Change whether the stream is playing.

Returns true on success, false on failure.

See also: [al\\_get\\_audio\\_stream\\_playing](#)

### 29.8.18 `al_get_audio_stream_playmode`

```
ALLEGRO_PLAYMODE al_get_audio_stream_playmode(  
    const ALLEGRO_AUDIO_STREAM *stream)
```

Return the playback mode.

See also: [ALLEGRO\\_PLAYMODE](#), [al\\_set\\_audio\\_stream\\_playmode](#).

### 29.8.19 `al_set_audio_stream_playmode`

```
bool al_set_audio_stream_playmode(ALLEGRO_AUDIO_STREAM *stream,  
    ALLEGRO_PLAYMODE val)
```

Set the playback mode.

Returns true on success, false on failure.

See also: [ALLEGRO\\_PLAYMODE](#), [al\\_get\\_audio\\_stream\\_playmode](#).

### 29.8.20 `al_get_audio_stream_attached`

```
bool al_get_audio_stream_attached(const ALLEGRO_AUDIO_STREAM *stream)
```

Return whether the stream is attached to something.

See also: [al\\_attach\\_audio\\_stream\\_to\\_mixer](#), [al\\_attach\\_audio\\_stream\\_to\\_voice](#), [al\\_detach\\_audio\\_stream](#).

### 29.8.21 `al_detach_audio_stream`

```
bool al_detach_audio_stream(ALLEGRO_AUDIO_STREAM *stream)
```

Detach the stream from whatever it's attached to, if anything.

See also: [al\\_attach\\_audio\\_stream\\_to\\_mixer](#), [al\\_attach\\_audio\\_stream\\_to\\_voice](#), [al\\_get\\_audio\\_stream\\_attached](#).

### 29.8.22 `al_get_audio_stream_played_samples`

```
uint64_t al_get_audio_stream_played_samples(const ALLEGRO_AUDIO_STREAM *stream)
```

Get the number of samples consumed by the parent since the audio stream was started.

Since: 5.1.8

### 29.8.23 `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`

### 29.8.24 `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.

See also: `al_get_audio_stream_fragment`

### 29.8.25 `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.

See also: `al_get_available_audio_stream_fragments`

### 29.8.26 `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_get_audio_stream_fragments`

### 29.8.27 `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`

### 29.8.28 `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`

### 29.8.29 `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`

### 29.8.30 `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.

## 29.9 Audio file I/O

### 29.9.1 `al_register_sample_loader`

```
bool al_register_sample_loader(const char *ext,  
ALLEGRO_SAMPLE *(*loader)(const char *filename))
```

Register a handler for `al_load_sample`. 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`

### 29.9.2 `al_register_sample_loader_f`

```
bool al_register_sample_loader_f(const char *ext,  
ALLEGRO_SAMPLE *(*loader)(ALLEGRO_FILE* fp))
```

Register a handler for `al_load_sample_f`. 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`



### 29.9.3 `al_register_sample_saver`

```
bool al_register_sample_saver(const char *ext,
                             bool (*saver)(const char *filename, ALLEGRO_SAMPLE *spl))
```

Register a handler for `al_save_sample`. 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](#)

### 29.9.4 `al_register_sample_saver_f`

```
bool al_register_sample_saver_f(const char *ext,
                                bool (*saver)(ALLEGRO_FILE* fp, ALLEGRO_SAMPLE *spl))
```

Register a handler for `al_save_sample_f`. 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](#)

### 29.9.5 `al_register_audio_stream_loader`

```
bool al_register_audio_stream_loader(const char *ext,
                                     ALLEGRO_AUDIO_STREAM *(*stream_loader)(const char *filename,
                                     size_t buffer_count, unsigned int samples))
```

Register a handler for `al_load_audio_stream`. 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](#)

### 29.9.6 `al_register_audio_stream_loader_f`

```
bool al_register_audio_stream_loader_f(const char *ext,
                                       ALLEGRO_AUDIO_STREAM *(*stream_loader)(ALLEGRO_FILE* fp,
                                       size_t buffer_count, unsigned int samples))
```

Register a handler for `al_load_audio_stream_f`. 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](#)

### 29.9.7 `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](#)

### 29.9.8 `al_load_sample_f`

```
ALLEGRO_SAMPLE *al_load_sample_f(ALLEGRO_FILE* fp, const char *ident)
```

Loads an audio file from an `ALLEGRO_FILE` stream into an `ALLEGRO_SAMPLE`. The file type is determined by the passed 'ident' parameter, which is a file name extension including the leading dot.

Note that this stores the entire file in memory at once, which may be time consuming. To read the file as it is needed, use [al\\_load\\_audio\\_stream\\_f](#).

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](#)

### 29.9.9 `al_load_audio_stream`

```
ALLEGRO_AUDIO_STREAM *al_load_audio_stream(const char *filename,  
size_t buffer_count, unsigned int samples)
```

Loads an audio file from disk as it is needed.

Unlike regular streams, the one returned by this function need not be fed by the user; the library will automatically read more of the file as it is needed. The stream will contain *buffer\_count* buffers with *samples* samples.

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](#)

### 29.9.10 `al_load_audio_stream_f`

```
ALLEGRO_AUDIO_STREAM *al_load_audio_stream_f(ALLEGRO_FILE* fp, const char *ident,
size_t buffer_count, unsigned int samples)
```

Loads an audio file from [ALLEGRO\\_FILE](#) stream as it is needed.

Unlike regular streams, the one returned by this function need not be fed by the user; the library will automatically read more of the file as it is needed. The stream will contain *buffer\_count* buffers with *samples* samples.

The file type is determined by the passed ‘ident’ parameter, which is a file name extension including the leading dot.

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](#)

### 29.9.11 `al_save_sample`

```
bool al_save_sample(const char *filename, ALLEGRO_SAMPLE *spl)
```

Writes a sample into a file. Currently, wav is the only supported format, and the extension must be “.wav”.

Returns true on success, false on error.

*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](#)

### 29.9.12 `al_save_sample_f`

```
bool al_save_sample_f(ALLEGRO_FILE *fp, const char *ident, ALLEGRO_SAMPLE *spl)
```

Writes a sample into a [ALLEGRO\\_FILE](#) 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](#)

## 29.10 Audio events

Audio events are all user events and so must be handled as such, mainly calling [al\\_unref\\_user\\_event](#) on them.

See also: [al\\_unref\\_user\\_event](#)

### 29.10.1 `al_get_audio_event_source`

Get the audio event source. `al_install_audio` must be called prior to calling this function.

This emits events related to the core sound system. For events specific to streams or recorders, you must use their own unique event sources.

Since: 5.1.0

See also: `ALLEGRO_AUDIO_EVENT_TYPE`, `al_get_audio_stream_event_source`, `al_get_audio_recorder_event_source`

## 29.11 Audio recording

Allegro's audio recording routines give you real-time access to raw, uncompressed audio input streams. Since Allegro hides all of the platform specific implementation details with its own buffering, it will add a small amount of latency. However, for most applications that small overhead will not adversely affect performance.

Recording is supported by the ALSA, AudioQueue, DirectSound8, and PulseAudio drivers. Enumerating or choosing other recording devices is not yet supported.

### 29.11.1 `ALLEGRO_AUDIO_RECORDER`

```
typedef struct ALLEGRO_AUDIO_RECORDER ALLEGRO_AUDIO_RECORDER;
```

An opaque datatype that represents a recording device.

Since: 5.1.1

### 29.11.2 `ALLEGRO_AUDIO_RECORDER_EVENT`

Structure that holds the audio recorder event data. Every event type will contain:

- `.source`: pointer to the audio recorder

The following will be available depending on the event type:

- `.buffer`: pointer to buffer containing the audio samples
- `.samples`: number of samples (not bytes) that are available

Since 5.1.1

See also: `al_get_audio_recorder_event`

### 29.11.3 `ALLEGRO_EVENT_AUDIO_RECORDER_FRAGMENT`

Sent after a user-specified number of samples have been recorded.

You must always check the values for the buffer and samples as they are not guaranteed to be exactly what was originally specified.

Since: 5.1.1

### 29.11.4 `al_create_audio_recorder`

```
ALLEGRO_AUDIO_RECORDER *al_create_audio_recorder(size_t fragment_count,
    unsigned int samples, unsigned int frequency,
    ALLEGRO_AUDIO_DEPTH depth, ALLEGRO_CHANNEL_CONF chan_conf)
```

Creates an audio recorder using the system's default recording device. (So if the returned device does not work, try updating the system's default recording device.)

Allegro will internally buffer several seconds of captured audio with minimal latency. (XXX: These settings need to be exposed via config or API calls.) Audio will be copied out of that private buffer into a fragment buffer of the size specified by the samples parameter. Whenever a new fragment is ready an event will be generated.

The total size of the fragment buffer is `fragment_count * samples * bytes_per_sample`. It is treated as a circular, never ending buffer. If you do not process the information fast enough, it will be overrun. Because of that, even if you only ever need to process one small fragment at a time, you should still use a large enough value for `fragment_count` to hold a few seconds of audio.

frequency is the number of samples per second to record. Common values are:

- 8000 - telephone quality speech
- 11025
- 22050
- 44100 - CD quality music (if 16-bit, stereo)

For maximum compatibility, use a depth of `ALLEGRO_AUDIO_DEPTH_UINT8` or `ALLEGRO_AUDIO_DEPTH_INT16`, and a single (mono) channel.

The recorder will not record until you start it with `al_start_audio_recorder`.

On failure, returns NULL.

Since: 5.1.1

### 29.11.5 `al_start_audio_recorder`

```
bool al_start_audio_recorder(ALLEGRO_AUDIO_RECORDER *r)
```

Begin recording into the fragment buffer. Once a complete fragment has been captured (as specified in `al_create_audio_recorder`), an `ALLEGRO_EVENT_AUDIO_RECORDER_FRAGMENT` event will be triggered.

Returns true if it was able to begin recording.

Since: 5.1.1

### 29.11.6 `al_stop_audio_recorder`

```
void al_stop_audio_recorder(ALLEGRO_AUDIO_RECORDER *r)
```

Stop capturing audio data. Note that the audio recorder is still active and consuming resources, so if you are finished recording you should destroy it with `al_destroy_audio_recorder`.

You may still receive a few events after you call this function as the device flushes the buffer.

If you restart the recorder, it will begin recording at the beginning of the next fragment buffer.

Since: 5.1.1

**29.11.7 al\_is\_audio\_recorder\_recording**

```
bool al_is_audio_recorder_recording(ALLEGRO_AUDIO_RECORDER *r)
```

Returns true if the audio recorder is currently capturing data and generating events.

Since: 5.1.1

**29.11.8 al\_get\_audio\_recorder\_event**

```
ALLEGRO_AUDIO_RECORDER_EVENT *al_get_audio_recorder_event(ALLEGRO_EVENT *event)
```

Returns the event as an [ALLEGRO\\_AUDIO\\_RECORDER\\_EVENT](#).

Since: 5.1.1

**29.11.9 al\_get\_audio\_recorder\_event\_source**

```
ALLEGRO_EVENT_SOURCE *al_get_audio_recorder_event_source(ALLEGRO_AUDIO_RECORDER *r)
```

Returns the event source for the recorder that generates the various recording events.

Since: 5.1.1

**29.11.10 al\_destroy\_audio\_recorder**

```
void al_destroy_audio_recorder(ALLEGRO_AUDIO_RECORDER *r)
```

Destroys the audio recorder and frees all resources associated with it. It is safe to destroy a recorder that is playing.

You may receive events after the recorder has been destroyed. They must be ignored, as the fragment buffer will no longer be valid.

Since: 5.1.1

## Audio codecs addon

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

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

### 30.1 `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`.

*Limitations:*

- Saving is only supported for wav files.
- Wav file loader currently only supports 8/16 bit little endian PCM files. 16 bits are used when saving wav files. Use flac files if more precision is required.
- Module files (`.it`, `.mod`, `.s3m`, `.xm`) are often composed with streaming in mind, and sometimes cannot be easily rendered into a finite length sample. Therefore they cannot be loaded with `al_load_sample`/`al_load_sample_f` and must be streamed with `al_load_audio_stream` or `al_load_audio_stream_f`.

Return true on success.

### 30.2 `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`.





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

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

### 31.1 `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](#)

### 31.2 `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](#)

### 31.3 `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](#)

### 31.4 `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](#)

### 31.5 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](#)

### 31.6 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](#)

### 31.7 al\_color\_html

```
ALLEGRO_COLOR al_color_html(char const *string)
```

Interprets an HTML-style hex number (e.g. #00faff) as a color. The accepted format is the same as [al\\_color\\_html\\_to\\_rgb](#).

Returns the interpreted color, or `al_map_rgba(0, 0, 0, 0)` if the string could not be parsed.

*Note:* the behaviour on invalid strings is different from Allegro 5.0.x.

See also: [al\\_color\\_html\\_to\\_rgb](#), [al\\_color\\_rgb\\_to\\_html](#)

### 31.8 al\_color\_html\_to\_rgb

```
bool al_color_html_to_rgb(char const *string,  
    float *red, float *green, float *blue)
```

Interprets an HTML-style hex number (e.g. #00faff) as a color. The only accepted formats are “#RRGGBB” and “RRGGBB” where R, G, B are hexadecimal digits [0-9A-Fa-f].

Returns true on success, false on failure. On failure all components are set to zero.

*Note:* the behaviour on invalid strings is different from Allegro 5.0.x.

See also: [al\\_color\\_html](#), [al\\_color\\_rgb\\_to\\_html](#)

## 31.9 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 pointer to a buffer of at least 8 bytes, into which the result will be written (including the NUL terminator).

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](#)

## 31.10 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.

## 31.11 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, avajowwhite, 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](#)

### 31.12 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:

$$\begin{aligned}C &= 1 - R \\M &= 1 - G \\Y &= 1 - B \\K &= 0\end{aligned}$$

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](#)

### 31.13 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](#)

### 31.14 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](#)

### 31.15 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](#)

### 31.16 `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](#)

### 31.17 `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](#)

### 31.18 `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](#)

### 31.19 `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](#).



## Font addons

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

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

### 32.1 General font routines

#### 32.1.1 ALLEGRO\_FONT

```
typedef struct ALLEGRO_FONT ALLEGRO_FONT;
```

A handle identifying any kind of font. Usually you will create it with `al_load_font` 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`.

#### 32.1.2 al\_init\_font\_addon

```
bool 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).

Returns true on success, false on failure. On the 5.0 branch, this function has no return value. You may wish to avoid checking the return value if your code needs to be compatible with Allegro 5.0. Currently, the function only ever true.

See also: `al_init_image_addon`, `al_init_ttf_addon`, `al_shutdown_font_addon`

#### 32.1.3 al\_shutdown\_font\_addon

```
void al_shutdown_font_addon(void)
```

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

See also: `al_init_font_addon`

#### 32.1.4 al\_load\_font

```
ALLEGRO_FONT *al_load_font(char const *filename, int size, int flags)
```

Loads a font from disk. This will use `al_load_bitmap_font_flags` if you pass the name of a known bitmap format, or else `al_load_ttf_font`.

The flags parameter is passed through to either of those functions. Bitmap and TTF fonts are also 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_load_bitmap_font_flags`, `al_load_ttf_font`

### 32.1.5 `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`

### 32.1.6 `al_register_font_loader`

```
bool al_register_font_loader(char const *extension,  
    ALLEGRO_FONT *(*load_font)(char const *filename, int size, int flags))
```

Informs Allegro of a new font file type, telling it how to load files of this format.

The extension should include the leading dot (‘.’) character. It will be matched case-insensitively.

The `load_font` argument may be NULL to unregister an entry.

Returns true on success, false on error. Returns false if unregistering an entry that doesn’t exist.

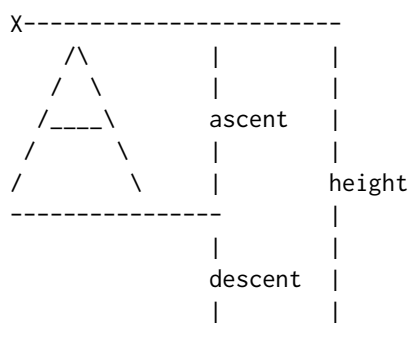
See also: `al_init_font_addon`

### 32.1.7 `al_get_font_line_height`

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

Returns the usual height of a line of text in the specified font. For bitmap fonts this is simply the height of all glyph bitmaps. For truetype fonts it is whatever the font file specifies. In particular, some special glyphs may be higher than the height returned here.

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.



See also: `al_get_text_width`, `al_get_text_dimensions`



### 32.1.8 `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](#)

### 32.1.9 `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](#)

### 32.1.10 `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](#)

### 32.1.11 `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](#)

### 32.1.12 `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 the target bitmap 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.

It can also be combined with this flag:

- `ALLEGRO_ALIGN_INTEGER` - Always draw text aligned to an integer pixel position. This is formerly the default behaviour. Since: 5.0.8, 5.1.4

This function does not support newline characters (`\n`), but you can use [al\\_draw\\_multiline\\_text](#) for multi line text output.

See also: [al\\_draw\\_ustr](#), [al\\_draw\\_textf](#), [al\\_draw\\_justified\\_text](#), [al\\_draw\\_multiline\\_text](#).

### 32.1.13 `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_multiline_ustr`

### 32.1.14 `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 justifying the text would exceed *diff* pixels, or the string contains less than two words, then the string will be drawn left aligned.

The flags parameter can be 0 or one of the following flags:

- `ALLEGRO_ALIGN_INTEGER` - Draw text aligned to integer pixel positions. Since: 5.0.8, 5.1.5

See also: `al_draw_justified_textf`, `al_draw_justified_ustr`

### 32.1.15 `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`.

### 32.1.16 `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`

### 32.1.17 `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`.

### 32.1.18 `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`

### 32.1.19 `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`

### 32.1.20 `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`.

### 32.1.21 `al_get_font_ranges`

```
int al_get_font_ranges(ALLEGRO_FONT *f, int ranges_count, int *ranges)
```

Gets information about all glyphs contained in a font, as a list of ranges. Ranges have the same format as with `al_grab_font_from_bitmap`.

`ranges_count` is the maximum number of ranges that will be returned.

`ranges` should be an array with room for `ranges_count * 2` elements. The even integers are the first unicode point in a range, the odd integers the last unicode point in a range.

Returns the number of ranges contained in the font (even if it is bigger than `ranges_count`).

Since: 5.1.4

See also: `al_grab_font_from_bitmap`

## 32.2 Multiline text drawing

### 32.2.1 `al_draw_multiline_text`

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

Like `al_draw_text`, but this function supports drawing multiple lines of text. It will break text in lines based on its contents and the `max_width` parameter. The lines are then layed out vertically depending on the `line_height` parameter and drawn each as if `al_draw_text` was called on them.

A newline `\n` in the text will cause a “hard” line break after its occurrence in the string. The text after a hard break is placed on a new line. Carriage return `\r` is not supported, will not cause a line break, and will likely be drawn as a square or a space depending on the font.

The `max_width` parameter controls the maximum desired width of the lines. This function will try to introduce a “soft” line break after the longest possible series of words that will fit in `max_length` when drawn with the given font. A “soft” line break can occur on either a space ‘ ’ or `tab` character.

However, it is possible that `max_width` is too small, or the words in text are too long to fit `max_width` when drawn with font. In that case, the word that is too wide will simply be drawn completely on a line by itself. If you don’t want the text that overflows `max_width` to be visible, then use `al_set_clipping_rectangle` to clip it off and hide it.

The function `al_draw_multiline_text` will draw each of the lines using the font, `x`, color and flags parameters as they were passed to this function. It supports the `ALLEGRO_ALIGN_LEFT`, `ALLEGRO_ALIGN_CENTRE`, `ALLEGRO_ALIGN_RIGHT` and `ALLEGRO_ALIGN_INTEGER` value for flags.

The lines that `al_draw_multiline_text` breaks the text into will be drawn starting at `y` and with a vertical distance between them of `line_height`. If `line_height` is zero (0), then the value returned by `al_get_font_line_height` on font will be used as the default value instead.

For complex or custom text layout that `al_draw_multiline_text` does not provide, or for calculating the size of what this function will draw without actually drawing it, you can use `al_do_multiline_text`.

Since: 5.1.9

See also: `al_do_multiline_text`, `al_draw_multiline_text`, `al_draw_multiline_textf`

### 32.2.2 `al_draw_multiline_ustr`

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

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

Since: 5.1.9

See also: `al_draw_multiline_text`, `al_draw_multiline_textf`, `al_do_multiline_text`

### 32.2.3 `al_draw_multiline_textf`

```
void al_draw_multiline_textf(const ALLEGRO_FONT *font,
    ALLEGRO_COLOR color, float x, float y, float max_width, float line_height,
    int flags, const char *format, ...)
```

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

Since: 5.1.9

See also: [al\\_draw\\_multiline\\_text](#), [al\\_draw\\_multiline\\_ustr](#), [al\\_do\\_multiline\\_text](#)

### 32.2.4 al\_do\_multiline\_text

This function processes the text and splits it into lines as [al\\_draw\\_multiline\\_text](#) would, and then calls the callback cb once for every line. This is useful for custom drawing of multiline text, or for calculating the size of multiline text ahead of time. See the documentation of [al\\_draw\\_multiline\\_text](#) for the explanation of the splitting algorithm.

For every line that this function splits text into, the callback cb will be called once. This callback cb must be of type `bool callback(int line_num, const char *line, int size, void *extra)`.

The callback cb will be called with the number of the line starting from zero and counting up in `line_num`, a pointer to the beginning character of the line in `line`, the size of the line in `size`, and the pointer passed in the `extra` parameter to [al\\_do\\_multiline\\_text](#).

Note that while the `line` pointer points to the beginning of the line, however, `line` is NOT guaranteed to be a NUL terminated string. Instead, `line` is simply a pointer, normally to a character in text, or to an empty string in case of an empty line. If you need a NUL terminated string, you will have to copy `line` to a buffer and NUL terminate it yourself.

The pointer `line` is not guaranteed to be valid after the callback cb has returned. So in case you still need the contents of `line` after that, you must also make a copy of it yourself. In case of an empty line the parameter `size` to the callback will be zero and `line` will point to an empty string.

When cb returns true, [al\\_do\\_multiline\\_text](#) will continue on to the next line. And when cb returns false, this function will stop immediately and not call cb anymore.

Since: 5.1.9

See also: [al\\_draw\\_multiline\\_text](#)

### 32.2.5 al\_do\_multiline\_ustr

```
void al_do_multiline_ustr(const ALLEGRO_FONT *font, float max_width,
    const ALLEGRO_USTR *ustr,
    bool (*cb)(int line_num, const ALLEGRO_USTR * line, void *extra),
    void *extra)
```

Like [al\\_do\\_multiline\\_text](#), but using `ALLEGRO_USTR` instead of a NUL-terminated char array for text.

Since: 5.1.9

See also: [al\\_draw\\_multiline\\_ustr](#)

## 32.3 Bitmap fonts

### 32.3.1 al\_grab\_font\_from\_bitmap

```
ALLEGRO_FONT *al_grab_font_from_bitmap(ALLEGRO_BITMAP *bmp,
    int ranges_n, const 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](#)

### 32.3.2 al\_load\_bitmap\_font

```
ALLEGRO_FONT *al_load_bitmap_font(const char *fname)
```

Load a bitmap font from. It does this by first calling [al\\_load\\_bitmap\\_flags](#) 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](#).

See also: [al\\_load\\_bitmap\\_font\\_flags](#), [al\\_load\\_font](#), [al\\_load\\_bitmap\\_flags](#)

### 32.3.3 al\_load\_bitmap\_font\_flags

```
ALLEGRO_FONT *al_load_bitmap_font_flags(const char *fname, int flags)
```

Like [al\\_load\\_bitmap\\_font](#) but additionally takes a flags parameter which is a bitfield containing a combination of the following:

`ALLEGRO_NO_PREMULTIPLIED_ALPHA` : The same meaning as for [al\\_load\\_bitmap\\_flags](#).

See also: [al\\_load\\_bitmap\\_font](#), [al\\_load\\_bitmap\\_flags](#)

### 32.3.4 `al_create_builtin_font`

```
ALLEGRO_FONT *al_create_builtin_font(void)
```

Creates a monochrome bitmap font (8x8 pixels per character).

This font is primarily intended to be used for displaying information in environments or during early runtime states where no external font data is available or loaded (e.g. for debugging).

The builtin font contains the following unicode character ranges:

```
0x0020 to 0x007F (ASCII)
0x00A1 to 0x00FF (Latin 1)
0x0100 to 0x017F (Extended A)
0x20AC to 0x20AC (euro currency symbol)
```

Returns NULL on an error.

The font memory must be freed the same way as for any other font, using `al_destroy_font`.

Since: 5.0.8, 5.1.3

See also: `al_load_bitmap_font`, `al_destroy_font`

## 32.4 TTF fonts

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

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

### 32.4.1 `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`.

Returns true on success, false on failure.

### 32.4.2 `al_shutdown_ttf_addon`

```
void al_shutdown_ttf_addon(void)
```

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

### 32.4.3 `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](#)

#### 32.4.4 `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.

#### 32.4.5 `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.

Returns `NULL` if 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](#)

#### 32.4.6 `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](#)



### 32.4.7 `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`.



## Image I/O addon

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

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

### 33.1 `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, DDS, 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.

The DDS format is only supported to load from, and only if the DDS file contains textures compressed in the DXT1, DXT3 and DXT5 formats. Note that when loading a DDS file, the created bitmap will always be a video bitmap and will have the pixel format matching the format in the file.

### 33.2 `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.

### 33.3 `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`.



## 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`.



## 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>
```

### 35.1 `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. The file is always read from/written to in binary mode, which means that no newline translation is performed.

It should be closed with `al_fclose`. After the file is closed, you are responsible for freeing the memory (if needed).

### 35.2 `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`.





## Native dialogs support

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

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

### 36.1 ALLEGRO\_FILECHOOSER

```
typedef struct ALLEGRO_FILECHOOSER ALLEGRO_FILECHOOSER;
```

Opaque handle to a native file dialog.

### 36.2 ALLEGRO\_TEXTLOG

```
typedef struct ALLEGRO_TEXTLOG ALLEGRO_TEXTLOG;
```

Opaque handle to a text log window.

### 36.3 al\_init\_native\_dialog\_addon

```
bool al_init_native_dialog_addon(void)
```

Initialise the native dialog addon.

Returns true on success, false on error.

Since: 5.0.9, 5.1.0

*Note:* Prior to Allegro 5.1.0 native dialog functions could be called without explicit initialisation, but that is now deprecated. Future functionality may require explicit initialisation. An exception is [al\\_show\\_native\\_message\\_box](#), which may be useful to show an error message if Allegro fails to initialise.

See also: [al\\_shutdown\\_native\\_dialog\\_addon](#)

### 36.4 al\_shutdown\_native\_dialog\_addon

```
void al_shutdown_native_dialog_addon(void)
```

Shut down the native dialog addon.

Since: 5.0.9, 5.1.5

See also: [al\\_init\\_native\\_dialog\\_addon](#)

## 36.5 `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.

## 36.6 `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.

### 36.7 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.

### 36.8 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.

### 36.9 al\_destroy\_native\_file\_dialog

```
void al_destroy_native_file_dialog(ALLEGRO_FILECHOOSER *dialog)
```

Frees up all resources used by the file dialog.

### 36.10 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:

```
int 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
);
```

### 36.11 `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](#)

### 36.12 `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](#)

### 36.13 `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.

### 36.14 `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.

### 36.15 `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`.

### 36.16 Menus

Menus are implemented on Windows, X and OS X. Menus on X are implemented with GTK, and have a special requirement: you must set the `ALLEGRO_GTK_TOPLEVEL` display flag prior to creating the display which will have menus attached.

A menu can be attached to a single display window or popped up as a context menu. If you wish to use the same menu on multiple displays or use a sub-menu as a context menu, you must make a copy via `al_clone_menu` or `al_clone_menu_for_popup`.

Each menu item can be given an ID of any 16-bit integer greater than zero. When a user clicks on a menu item, an event will be generated only if it has an ID. This ID should be unique per menu; if you duplicate IDs, then there will be no way for you to determine exactly which item generated the event.

There are many functions that take `pos` as a parameter used for locating a particular menu item. In those cases, it represents one of two things: an ID or zero-based index. Any value greater than zero will always be treated as an ID. Anything else (including zero) will be considered an index based on the absolute value. In other words, 0 is the first menu item, -1 is the second menu item, -2 is the third menu item, and so on.

The event type is `ALLEGRO_EVENT_MENU_CLICK`. It contains three fields:

```
c event.user.data1 = id; event.user.data2 = (intptr_t) display; event.user.data3 = (intptr_t) menu;
```

The display and menu may be NULL if it was not possible to tell exactly which item generated the event.

A basic example:

```
#define FILE_EXIT_ID 1

ALLEGRO_MENU *menu = al_create_menu();
ALLEGRO_MENU *file_menu = al_create_menu();
al_append_menu_item(file_menu, "Exit", FILE_EXIT_ID, 0, NULL, NULL);
al_append_menu_item(menu, "File", 0, 0, NULL, file_menu);
al_set_display_menu(display, menu);

al_register_event_source(queue, al_get_default_menu_event_source());
al_wait_for_event(queue, &event);
```

```
if (event.type == ALLEGRO_EVENT_MENU_CLICK) {  
    if (event.user.data1 == FILE_EXIT_ID) {  
        exit_program();  
    }  
}
```

Because there is no “DISPLAY\_DESTROYED” event, you must call `al_set_display_menu(display, NULL)` before destroying any display with a menu attached, to avoid leaking resources.

### 36.16.1 ALLEGRO\_MENU

```
typedef struct ALLEGRO_MENU ALLEGRO_MENU;
```

An opaque data type that represents a menu that contains menu items. Each of the menu items may optionally include a sub-menu.

### 36.16.2 ALLEGRO\_MENU\_INFO

```
typedef struct ALLEGRO_MENU_INFO {
```

A structure that defines how to create a complete menu system. For standard menu items, the following format is used:

```
{ caption, id, flags, icon }
```

For special items, these macros are helpful:

```
ALLEGRO_START_OF_MENU(caption, id)  
ALLEGRO_MENU_SEPARATOR  
ALLEGRO_END_OF_MENU
```

A well-defined menu will begin with `ALLEGRO_START_OF_MENU`, contain one or more menu items, and end with `ALLEGRO_END_OF_MENU`. A menu may contain sub-menus. An example:

```
ALLEGRO_MENU_INFO menu_info[] = {  
    ALLEGRO_START_OF_MENU("&File", 1),  
    { "&Open", 2, 0, NULL },  
    ALLEGRO_START_OF_MENU("Open &Recent...", 3),  
    { "Recent 1", 4, 0, NULL },  
    { "Recent 2", 5, 0, NULL },  
    ALLEGRO_END_OF_MENU,  
    ALLEGRO_MENU_SEPARATOR,  
    { "E&xit", 6, 0, NULL },  
    ALLEGRO_END_OF_MENU,  
    ALLEGRO_START_OF_MENU("&Help", 7),  
    { "&About", 8, 0, NULL },  
    ALLEGRO_END_OF_MENU,  
    ALLEGRO_END_OF_MENU  
};  
  
ALLEGRO_MENU *menu = al_build_menu(menu_info);
```

If you prefer, you can build the menu without the structure by using `al_create_menu` and `al_insert_menu_item`.

See also: [al\\_build\\_menu](#)

### 36.16.3 `al_create_menu`

```
ALLEGRO_MENU *al_create_menu(void)
```

Creates a menu container that can hold menu items.

Returns NULL on failure.

Since: 5.1.0

See also: [al\\_create\\_popup\\_menu](#), [al\\_build\\_menu](#)

### 36.16.4 `al_create_popup_menu`

```
ALLEGRO_MENU *al_create_popup_menu(void)
```

Creates a menu container for popup menus. Only the root (outermost) menu should be created with this function. Sub menus of popups should be created with [al\\_create\\_menu](#).

Returns NULL on failure.

Since: 5.1.0

See also: [al\\_create\\_menu](#), [al\\_build\\_menu](#)

### 36.16.5 `al_build_menu`

```
ALLEGRO_MENU *al_build_menu(ALLEGRO_MENU_INFO *info)
```

Builds a menu based on the specifications of a sequence of `ALLEGRO_MENU_INFO` elements.

Returns a pointer to the root `ALLEGRO_MENU`, or NULL on failure. To gain access to the other menus and items, you will need to search for them using [al\\_find\\_menu\\_item](#).

Since: 5.1.0

See also: [ALLEGRO\\_MENU\\_INFO](#), [al\\_create\\_menu](#), [al\\_create\\_popup\\_menu](#)

### 36.16.6 `al_append_menu_item`

```
int al_append_menu_item(ALLEGRO_MENU *parent, char const *title, uint16_t id,
    int flags, ALLEGRO_BITMAP *icon, ALLEGRO_MENU *submenu)
```

Appends a menu item to the end of the menu. See [al\\_insert\\_menu\\_item](#) for more information.

Since: 5.1.0

See also: [al\\_insert\\_menu\\_item](#), [al\\_remove\\_menu\\_item](#)

### 36.16.7 `al_insert_menu_item`

```
int al_insert_menu_item(ALLEGRO_MENU *parent, int pos, char const *title,
    uint16_t id, int flags, ALLEGRO_BITMAP *icon, ALLEGRO_MENU *submenu)
```

Inserts a menu item at the spot specified. See the introductory text for a detailed explanation of how the `pos` parameter is interpreted.

The parent menu can be a popup menu or a regular menu. To underline one character in the title, prefix it with an ampersand.

The flags can be any combination of:

#### **`ALLEGRO_MENU_ITEM_DISABLED`**

The item is “grayed out” and cannot be selected.

**ALLEGRO\_MENU\_ITEM\_CHECKBOX**

The item is a check box. This flag can only be set at the time the menu is created. If a check box is clicked, it will automatically be toggled.

**ALLEGRO\_MENU\_ITEM\_CHECKED**

The item is checked. If set, `ALLEGRO_MENU_ITEM_CHECKBOX` will automatically be set as well.

The icon is not yet supported.

The submenu parameter indicates that this item contains a child menu. The child menu must have previously been created with `al_create_menu`, and is not associated with any other menu.

Returns true on success.

Since: 5.1.0

See also: [al\\_append\\_menu\\_item](#), [al\\_remove\\_menu\\_item](#)

**36.16.8 al\_remove\_menu\_item**

```
bool al_remove_menu_item(ALLEGRO_MENU *menu, int pos)
```

Removes the specified item from the menu. If the item contains a sub-menu, it too is destroyed. Any references to it are invalidated. If you want to preserve that sub-menu, you should first make a copy with [al\\_clone\\_menu](#).

This is safe to call on a menu that is currently being displayed.

Returns true if an item was removed.

Since: 5.1.0

See also: [al\\_append\\_menu\\_item](#), [al\\_insert\\_menu\\_item](#), [al\\_destroy\\_menu](#)

**36.16.9 al\_clone\_menu**

```
ALLEGRO_MENU *al_clone_menu(ALLEGRO_MENU *menu)
```

Makes a copy of a menu so that it can be reused on another display. The menu being cloned can be anything: a regular menu, a popup menu, or a sub-menu.

Returns the cloned menu.

Since: 5.1.0

See also: [al\\_clone\\_menu\\_for\\_popup](#)

**36.16.10 al\_clone\_menu\_for\_popup**

```
ALLEGRO_MENU *al_clone_menu_for_popup(ALLEGRO_MENU *menu)
```

Exactly like [al\\_clone\\_menu](#), except that the copy is for a popup menu.

Since: 5.1.0

See also: [al\\_clone\\_menu](#)

**36.16.11 al\_destroy\_menu**

```
void al_destroy_menu(ALLEGRO_MENU *menu)
```

Destroys an entire menu, including its sub-menus. Any references to it or a sub-menu is no longer valid. It is safe to call this on a menu that is currently being displayed.

Since: 5.1.0

See also: [al\\_remove\\_menu\\_item](#)



### 36.16.12 `al_get_menu_item_caption`

```
const char *al_get_menu_item_caption(ALLEGRO_MENU *menu, int pos)
```

Returns the caption associated with the menu item. It is valid as long as the caption is not modified.

Returns NULL if the item was not found.

Since: 5.1.0

See also: [al\\_set\\_menu\\_item\\_caption](#)

### 36.16.13 `al_set_menu_item_caption`

```
void al_set_menu_item_caption(ALLEGRO_MENU *menu, int pos, const char *caption)
```

Updates the menu item caption with the new caption. This will invalidate any previous calls to [al\\_get\\_menu\\_item\\_caption](#).

Since: 5.1.0

See also: [al\\_get\\_menu\\_item\\_caption](#)

### 36.16.14 `al_get_menu_item_flags`

```
int al_get_menu_item_flags(ALLEGRO_MENU *menu, int pos)
```

Returns the currently set flags. See [al\\_insert\\_menu\\_item](#) for a description of the available flags.

Returns -1 if the item was not found.

Since: 5.1.0

See also: [al\\_set\\_menu\\_item\\_flags](#), [al\\_toggle\\_menu\\_item\\_flags](#)

### 36.16.15 `al_set_menu_item_flags`

```
void al_set_menu_item_flags(ALLEGRO_MENU *menu, int pos, int flags)
```

Updates the menu item's flags. See [al\\_insert\\_menu\\_item](#) for a description of the available flags.

Since: 5.1.0

See also: [al\\_get\\_menu\\_item\\_flags](#), [al\\_toggle\\_menu\\_item\\_flags](#)

### 36.16.16 `al_toggle_menu_item_flags`

```
int al_toggle_menu_item_flags(ALLEGRO_MENU *menu, int pos, int flags)
```

Toggles the specified menu item's flags. See [al\\_insert\\_menu\\_item](#) for a description of the available flags.

Returns a bitfield of only the specified flags that are set after the toggle. A flag that was not toggled will not be returned, even if it is set. Returns -1 if the id is invalid.

Since: 5.1.0

See also: [al\\_get\\_menu\\_item\\_flags](#), [al\\_set\\_menu\\_item\\_flags](#)

### 36.16.17 `al_get_menu_item_icon`

```
ALLEGRO_BITMAP *al_get_menu_item_icon(ALLEGRO_MENU *menu, int pos)
```

Returns the icon associated with the menu. It is safe to draw to the returned bitmap, but you must call `al_set_menu_item_icon` in order for the changes to be applied.

Returns NULL if the item was not found or if it has no icon.

Since: 5.1.0

See also: [al\\_set\\_menu\\_item\\_icon](#)

### 36.16.18 `al_set_menu_item_icon`

```
void al_set_menu_item_icon(ALLEGRO_MENU *menu, int pos, ALLEGRO_BITMAP *icon)
```

Sets the icon for the specified menu item. The menu assumes ownership of the `ALLEGRO_BITMAP` and may invalidate the pointer, so you must clone it if you wish to continue using it.

If a video bitmap is passed, it will automatically be converted to a memory bitmap, so it is preferable to pass a memory bitmap.

Since: 5.1.0

See also: [al\\_get\\_menu\\_item\\_icon](#), [al\\_clone\\_bitmap](#)

### 36.16.19 `al_find_menu`

```
ALLEGRO_MENU *al_find_menu(ALLEGRO_MENU *haystack, uint16_t id)
```

Searches in the haystack menu for any submenu with the given id. (Note that this only represents a literal ID, and cannot be used as an index.)

Returns the menu, if found. Otherwise returns NULL.

Since: 5.1.0

See also: [al\\_find\\_menu\\_item](#)

### 36.16.20 `al_find_menu_item`

```
bool al_find_menu_item(ALLEGRO_MENU *haystack, uint16_t id, ALLEGRO_MENU **menu, int *index)
```

Searches in the haystack menu for an item with the given id. (Note that this only represents a literal ID, and cannot be used as an index.)

If menu and index are not NULL, they will be set as the parent menu containing the item and the zero-based index of the item. (If the menu item was not found, then their values are undefined.)

Returns true if the menu item was found.

Since: 5.1.0

See also: [al\\_find\\_menu](#)

### 36.16.21 `al_get_default_menu_event_source`

```
ALLEGRO_EVENT_SOURCE *al_get_default_menu_event_source(void)
```

Returns the default event source used for menu clicks. If a menu was not given its own event source via `al_enable_menu_event_source`, then it will use this default source.

Since: 5.1.0

See also: [al\\_register\\_event\\_source](#), [al\\_enable\\_menu\\_event\\_source](#), [al\\_disable\\_menu\\_event\\_source](#)

**36.16.22 al\_enable\_menu\_event\_source**

```
ALLEGRO_EVENT_SOURCE *al_enable_menu_event_source(ALLEGRO_MENU *menu)
```

Enables a unique event source for this menu. It and all of its sub-menus will use this event source. (It is safe to call this multiple times on the same menu.)

Returns the event source.

Since: 5.1.0

See also: [al\\_register\\_event\\_source](#), [al\\_get\\_default\\_menu\\_event\\_source](#), [al\\_disable\\_menu\\_event\\_source](#)

**36.16.23 al\_disable\_menu\_event\_source**

```
void al_disable_menu_event_source(ALLEGRO_MENU *menu)
```

Disables a unique event source for the menu, causing it to use the default event source.

Since: 5.1.0

See also: [al\\_get\\_default\\_menu\\_event\\_source](#), [al\\_enable\\_menu\\_event\\_source](#)

**36.16.24 al\_get\_display\_menu**

```
ALLEGRO_MENU *al_get_display_menu(ALLEGRO_DISPLAY *display)
```

Returns the menu associated with the display, or NULL if it does not have a menu.

Since: 5.1.0

See also: [al\\_set\\_display\\_menu](#)

**36.16.25 al\_set\_display\_menu**

```
bool al_set_display_menu(ALLEGRO_DISPLAY *display, ALLEGRO_MENU *menu)
```

Associates the menu with the display and shows it. If there was a previous menu associated with the display, it will be destroyed. If you don't want that to happen, you should first remove the menu with [al\\_remove\\_display\\_menu](#).

If the menu is already attached to a display, it will not be attached to the new display. If menu is NULL, the current menu will still be destroyed.

Note that attaching a menu may cause the window as available to your application to be resized! You should listen for a resize event, check how much space was lost, and resize the window accordingly if you want to maintain your window's prior size.

Returns true if successful.

Since: 5.1.0

See also: [al\\_create\\_menu](#), [al\\_remove\\_display\\_menu](#)

**36.16.26 al\_popup\_menu**

```
bool al_popup_menu(ALLEGRO_MENU *popup, ALLEGRO_DISPLAY *display)
```

Displays a context menu next to the mouse cursor. The menu must have been created with [al\\_create\\_popup\\_menu](#). It generates events just like a regular display menu does. It is possible that the menu will be canceled without any selection being made.

The display parameter indicates which window the menu is associated with (when you process the menu click event), but does not actually affect where the menu is located on the screen.

Returns true if the context menu was displayed.

Since: 5.1.0

See also: [al\\_create\\_popup\\_menu](#)

### 36.16.27 **al\_remove\_display\_menu**

```
ALLEGRO_MENU *al_remove_display_menu(ALLEGRO_DISPLAY *display)
```

Detaches the menu associated with the display and returns it. The menu can then be used on a different display.

If you simply want to destroy the active menu, you can call [al\\_set\\_display\\_menu](#) with a NULL menu.

Since: 5.1.0

See also: [al\\_set\\_display\\_menu](#)

## 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>
```

### 37.1 `al_set_physfs_file_interface`

```
void al_set_physfs_file_interface(void)
```

This function sets *both* the `ALLEGRO_FILE_INTERFACE` and `ALLEGRO_FS_INTERFACE` for the calling thread.

Subsequent calls to `al_fopen` on the calling thread will be handled by `PHYSFS_open()`. Operations on the files returned by `al_fopen` will then be performed through PhysicsFS. Calls to the Allegro filesystem functions, such as `al_read_directory` or `al_create_fs_entry`, on the calling thread will be diverted to PhysicsFS.

To remember and restore another file I/O backend, you can use `al_store_state/al_restore_state`.

*Note:* due to an oversight, this function differs from `al_set_new_file_interface` and `al_set_standard_file_interface` which only alter the current `ALLEGRO_FILE_INTERFACE`.

*Note:* PhysFS does not support the text-mode reading and writing, which means that Windows-style newlines will not be preserved.

See also: `al_set_new_file_interface`.

### 37.2 `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`.



## Primitives addon

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

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

### 38.1 General

#### 38.1.1 `al_get_allegro_primitives_version`

```
uint32_t al_get_allegro_primitives_version(void)
```

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

#### 38.1.2 `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](#)

#### 38.1.3 `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](#)

### 38.2 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.

### 38.2.1 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:

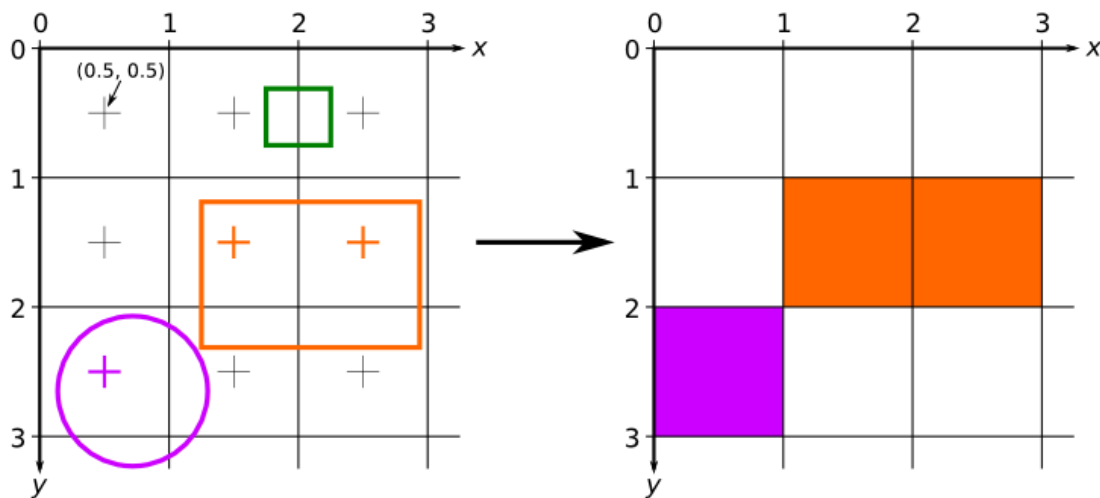


Figure 38.1: Diagram showing a how pixel output is calculated by the GPU given the mathematical description of several shapes.

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);
```

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



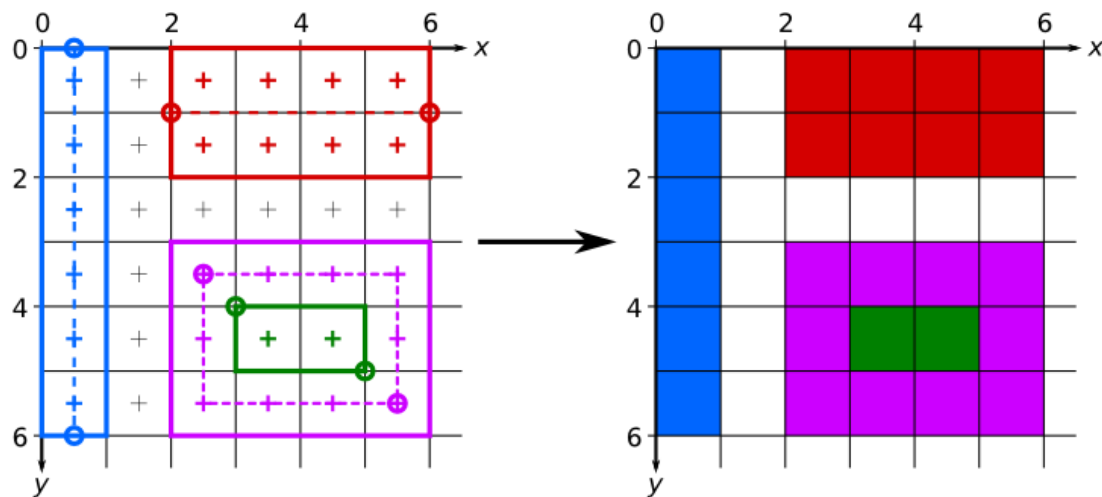


Figure 38.2: Diagram showing a practical example of pixel output resulting from the invocation of several primitives addon functions.

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 a thickness of 1, as depicted in the diagram, this means that an offset of 0.5 is needed for both sets of endpoint coordinates to exactly line up with the pixels of the display raster.

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.

### 38.2.2 `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](#)

### 38.2.3 `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](#)

### 38.2.4 `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](#)

### 38.2.5 `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](#)

### 38.2.6 `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](#)

### 38.2.7 `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](#)

### 38.2.8 `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](#)

### 38.2.9 `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_points)
```

When `thickness <= 0` this function computes positions of `num_points` regularly spaced points on an elliptical arc. When `thickness > 0` this function computes two sets of points, obtained as follows: the first set is obtained by taking the points computed in the `thickness <= 0` case and shifting them by `thickness / 2` outward, in a direction perpendicular to the arc curve. The second set is the same, but shifted `thickness / 2` inward relative to the arc. The two sets of points are interleaved in the destination buffer (i.e. the first pair of points will be collinear with the arc center, the first point of the pair will be farther from the center than the second point; the next pair will also be collinear, but at a different angle and so on).

The destination buffer `dest` is interpreted as a set of regularly spaced pairs of floats, each pair holding the coordinates of the corresponding point on the arc. The two floats in the pair are adjacent, and the distance (in bytes) between the addresses of the first float in two successive pairs is `stride`. For example, if you have a tightly packed array of floats with no spaces between pairs, then `stride` will be exactly `2 * sizeof(float)`.

Example with `thickness <= 0`:

```
const int num_points = 4;
float points[num_points][2];
al_calculate_arc(&points[0][0], 2 * sizeof(float), 0, 0, 10, 10, 0, ALLEGRO_PI / 2, 0, num_points);

assert((int)points[0][0] == 10);
assert((int)points[0][1] == 0);

assert((int)points[num_points - 1][0] == 0);
assert((int)points[num_points - 1][1] == 10);
```

Example with thickness > 0:

```
const int num_points = 4;
float points[num_points * 2][2];
al_calculate_arc(&points[0][0], 2 * sizeof(float), 0, 0, 10, 10, 0, ALLEGRO_PI / 2, 2, num_points);

assert((int)points[0][0] == 11);
assert((int)points[0][1] == 0);
assert((int)points[1][0] == 9);
assert((int)points[1][1] == 0);

assert((int)points[(num_points - 1) * 2][0] == 0);
assert((int)points[(num_points - 1) * 2][1] == 11);
assert((int)points[(num_points - 1) * 2 + 1][0] == 0);
assert((int)points[(num_points - 1) * 2 + 1][1] == 9);
```

*Parameters:*

- dest - The destination buffer
- stride - Distance (in bytes) between starts of successive pairs of points
- 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](#)

### 38.2.10 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](#)

### 38.2.11 `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](#)

### 38.2.12 `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](#)

### 38.2.13 `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](#)

### 38.2.14 `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](#)

### 38.2.15 `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](#)

### 38.2.16 `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](#)

### 38.2.17 `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](#)

### 38.2.18 `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](#)

### 38.2.19 `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](#)

### 38.2.20 `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  $\leq 0$ , 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](#)

### 38.2.21 `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  $\leq 0$  to draw hairline spline

See also: [al\\_calculate\\_ribbon](#)

## 38.3 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 the following 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.

Some platforms also dictate a minimum texture size, which means that textures smaller than that size will not tile properly. The minimum size that will work on all platforms is 32 by 32.

A note about pixel coordinates. In OpenGL the texture coordinate (0, 0) refers to the top left corner of the pixel. This confuses some drivers, because due to rounding errors the actual pixel sampled might be the pixel to the top and/or left of the (0, 0) pixel. To make this error less likely it is advisable to offset the texture coordinates you pass to the `al_draw_prim` by (0.5, 0.5) if you need precise pixel control. E.g. to refer to pixel (5, 10) you'd set the u and v to 5.5 and 10.5 respectively.

See also: [Pixel-precise output](#)



### 38.3.1 `al_draw_prim`

```
int al_draw_prim(const void* vtxs, const ALLEGRO_VERTEX_DECL* decl,
    ALLEGRO_BITMAP* texture, int start, int end, int type)
```

Draws a subset of the passed vertex array.

*Parameters:*

- texture - Texture to use, pass 0 to use only color shaded primitives
- 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 array to draw
- end - One past the last index of subset of the vertex array to draw
- type - A member of the `ALLEGRO_PRIM_TYPE` enumeration, specifying what kind of primitive to draw

*Returns:* Number of primitives drawn

For example to draw a textured triangle you could use:

```
ALLEGRO_COLOR white = al_map_rgb_f(1, 1, 1);
ALLEGRO_VERTEX v[] = {
    {.x = 128, .y = 0, .z = 0, .color = white, .u = 128, .v = 0},
    {.x = 0, .y = 256, .z = 0, .color = white, .u = 0, .v = 256},
    {.x = 256, .y = 256, .z = 0, .color = white, .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`

### 38.3.2 `al_draw_indexed_prim`

```
int al_draw_indexed_prim(const void* vtxs, const ALLEGRO_VERTEX_DECL* decl,
    ALLEGRO_BITMAP* texture, const int* indices, int num_vtx, int type)
```

Draws a subset of the passed vertex array. This function uses an index array to specify which vertices to use.

*Parameters:*

- texture - Texture to use, pass 0 to use only shaded primitives
- 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 array
- num\_vtx - Number of indices from the indices array you want to draw
- type - A member of the `ALLEGRO_PRIM_TYPE` enumeration, specifying what kind of primitive to draw

*Returns:* Number of primitives drawn

See also: `ALLEGRO_VERTEX`, `ALLEGRO_PRIM_TYPE`, `ALLEGRO_VERTEX_DECL`, `al_draw_prim`

### 38.3.3 al\_draw\_vertex\_buffer

```
int al_draw_vertex_buffer(ALLEGRO_VERTEX_BUFFER* vertex_buffer,  
    ALLEGRO_BITMAP* texture, int start, int end, int type)
```

Draws a subset of the passed vertex buffer. The vertex buffer must not be locked. Additionally, to draw onto memory bitmaps or with memory bitmap textures the vertex buffer must support reading (i.e. it must be created with the `ALLEGRO_PRIM_BUFFER_READWRITE`).

*Parameters:*

- `vertex_buffer` - Vertex buffer to draw
- `texture` - Texture to use, pass 0 to use only color shaded primitives
- `start` - Start index of the subset of the vertex buffer to draw
- `end` - One past the last index of the subset of the vertex buffer to draw
- `type` - A member of the `ALLEGRO_PRIM_TYPE` enumeration, specifying what kind of primitive to draw

*Returns:* Number of primitives drawn

Since: 5.1.3

See also: [ALLEGRO\\_VERTEX\\_BUFFER](#), [ALLEGRO\\_PRIM\\_TYPE](#)

### 38.3.4 al\_draw\_indexed\_buffer

```
int al_draw_indexed_buffer(ALLEGRO_VERTEX_BUFFER* vertex_buffer,  
    ALLEGRO_BITMAP* texture, ALLEGRO_INDEX_BUFFER* index_buffer,  
    int start, int end, int type)
```

Draws a subset of the passed vertex buffer. This function uses an index buffer to specify which vertices to use. Both buffers must not be locked. Additionally, to draw onto memory bitmaps or with memory bitmap textures both buffers must support reading (i.e. they must be created with the `ALLEGRO_PRIM_BUFFER_READWRITE`).

*Parameters:*

- `vertex_buffer` - Vertex buffer to draw
- `texture` - Texture to use, pass 0 to use only color shaded primitives
- `index_buffer` - Index buffer to use
- `start` - Start index of the subset of the vertex buffer to draw
- `end` - One past the last index of the subset of the vertex buffer to draw
- `type` - A member of the `ALLEGRO_PRIM_TYPE` enumeration, specifying what kind of primitive to draw. Note that `ALLEGRO_PRIM_LINE_LOOP` and `ALLEGRO_PRIM_POINT_LIST` are not supported.

*Returns:* Number of primitives drawn

Since: 5.1.8

See also: [ALLEGRO\\_VERTEX\\_BUFFER](#), [ALLEGRO\\_INDEX\\_BUFFER](#), [ALLEGRO\\_PRIM\\_TYPE](#)

### 38.3.5 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 delimited 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](#)

### 38.3.6 `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.

See also: [al\\_draw\\_line](#)

## 38.4 Custom vertex declaration routines

### 38.4.1 `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](#)

### 38.4.2 `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](#)

## 38.5 Vertex buffer routines

### 38.5.1 `al_create_vertex_buffer`

```
ALLEGRO_VERTEX_BUFFER* al_create_vertex_buffer(ALLEGRO_VERTEX_DECL* decl,  
const void* initial_data, int num_vertices, int flags)
```

Creates a vertex buffer. Can return NULL if the buffer could not be created (e.g. the system only supports write-only buffers).

*Note:*

This is an advanced feature, often unsupported on lower-end video cards. Be extra mindful of this function failing and make arrangements for fallback drawing functionality or a nice error message for users with such lower-end cards.

*Parameters:*

- decl - Vertex type that this buffer will hold. 0 implies that this buffer will hold [ALLEGRO\\_VERTEX](#) vertices
- initial\_data - Memory buffer to copy from to initialize the vertex buffer. Can be NULL, in which case the buffer is uninitialized.
- num\_vertices - Number of vertices the buffer will hold
- flags - A combination of the [ALLEGRO\\_PRIM\\_BUFFER\\_FLAGS](#) flags specifying how this buffer will be created. Passing 0 is the same as passing `ALLEGRO_PRIM_BUFFER_STATIC`.

Since: 5.1.3

See also: [ALLEGRO\\_VERTEX\\_BUFFER](#), [al\\_destroy\\_vertex\\_buffer](#)

### 38.5.2 `al_destroy_vertex_buffer`

```
void al_destroy_vertex_buffer(ALLEGRO_VERTEX_BUFFER* buffer)
```

Destroys a vertex buffer. Does nothing if passed 0.

Since: 5.1.3

See also: [ALLEGRO\\_VERTEX\\_BUFFER](#), [al\\_create\\_vertex\\_buffer](#)

### 38.5.3 `al_lock_vertex_buffer`

```
void* al_lock_vertex_buffer(ALLEGRO_VERTEX_BUFFER* buffer, int offset,
    int length, int flags)
```

Locks a vertex buffer so you can access its data. Will return 0 if the parameters are invalid, if reading is requested from a write only buffer and if the buffer is already locked.

*Parameters:*

- `buffer` - Vertex buffer to lock
- `offset` - Vertex index of the start of the locked range
- `length` - How many vertices to lock
- `flags` - `ALLEGRO_LOCK_READONLY`, `ALLEGRO_LOCK_WRITEONLY` or `ALLEGRO_LOCK_READWRITE`

Since: 5.1.3

See also: [ALLEGRO\\_VERTEX\\_BUFFER](#), [al\\_unlock\\_vertex\\_buffer](#)

### 38.5.4 `al_unlock_vertex_buffer`

```
void al_unlock_vertex_buffer(ALLEGRO_VERTEX_BUFFER* buffer)
```

Unlocks a previously locked vertex buffer.

Since: 5.1.3

See also: [ALLEGRO\\_VERTEX\\_BUFFER](#), [al\\_lock\\_vertex\\_buffer](#)

### 38.5.5 `al_get_vertex_buffer_size`

```
int al_get_vertex_buffer_size(ALLEGRO_VERTEX_BUFFER* buffer)
```

Returns the size of the vertex buffer

Since: 5.1.8

See also: [ALLEGRO\\_VERTEX\\_BUFFER](#)

## 38.6 Index buffer routines

### 38.6.1 `al_create_index_buffer`

```
ALLEGRO_INDEX_BUFFER* al_create_index_buffer(int index_size,
    const void* initial_data, int num_indices, int flags)
```

Creates a index buffer. Can return NULL if the buffer could not be created (e.g. the system only supports write-only buffers).

*Note:*

This is an advanced feature, often unsupported on lower-end video cards. Be extra mindful of this function failing and make arrangements for fallback drawing functionality or a nice error message for users with such lower-end cards.

*Parameters:*

- `index_size` - Size of the index in bytes. Supported sizes are 2 for short integers and 4 for integers
- `initial_data` - Memory buffer to copy from to initialize the index buffer. Can be NULL, in which case the buffer is uninitialized.
- `num_indices` - Number of indices the buffer will hold
- `flags` - A combination of the [ALLEGRO\\_PRIM\\_BUFFER\\_FLAGS](#) flags specifying how this buffer will be created. Passing 0 is the same as passing `ALLEGRO_PRIM_BUFFER_STATIC`.

Since: 5.1.8

See also: [ALLEGRO\\_INDEX\\_BUFFER](#), [al\\_destroy\\_index\\_buffer](#)

### 38.6.2 `al_destroy_index_buffer`

```
void al_destroy_index_buffer(ALLEGRO_INDEX_BUFFER* buffer)
```

Destroys a index buffer. Does nothing if passed 0.

Since: 5.1.8

See also: [ALLEGRO\\_INDEX\\_BUFFER](#), [al\\_create\\_index\\_buffer](#)

### 38.6.3 `al_lock_index_buffer`

```
void* al_lock_index_buffer(ALLEGRO_INDEX_BUFFER* buffer, int offset,  
                           int length, int flags)
```

Locks a index buffer so you can access its data. Will return 0 if the parameters are invalid, if reading is requested from a write only buffer and if the buffer is already locked.

*Parameters:*

- `buffer` - Index buffer to lock
- `offset` - Element index of the start of the locked range
- `length` - How many indices to lock
- `flags` - `ALLEGRO_LOCK_READONLY`, `ALLEGRO_LOCK_WRITEONLY` or `ALLEGRO_LOCK_READWRITE`

Since: 5.1.8

See also: [ALLEGRO\\_INDEX\\_BUFFER](#), [al\\_unlock\\_index\\_buffer](#)

### 38.6.4 `al_unlock_index_buffer`

```
void al_unlock_index_buffer(ALLEGRO_INDEX_BUFFER* buffer)
```

Unlocks a previously locked index buffer.

Since: 5.1.8

See also: [ALLEGRO\\_INDEX\\_BUFFER](#), [al\\_lock\\_index\\_buffer](#)

### 38.6.5 al\_get\_index\_buffer\_size

```
int al_get_index_buffer_size(ALLEGRO_INDEX_BUFFER* buffer)
```

Returns the size of the index buffer

Since: 5.1.8

See also: [ALLEGRO\\_INDEX\\_BUFFER](#)

## 38.7 Polygon routines

### 38.7.1 al\_draw\_polyline

```
void al_draw_polyline(const float* vertices, int vertex_stride,
    int vertex_count, int join_style, int cap_style,
    ALLEGRO_COLOR color, float thickness, float miter_limit)
```

Draw a series of line segments.

- vertex - Interleaved array of (x, y) vertex coordinates
- vertex\_stride - the number of bytes between pairs of vertices (the stride)
- vertex\_count - Number of vertices in the array
- join\_style - Member of [ALLEGRO\\_LINE\\_JOIN](#) specifying how to render the joins between line segments
- cap\_style - Member of [ALLEGRO\\_LINE\\_CAP](#) specifying how to render the end caps
- color - Color of the line
- thickness - Thickness of the line, pass  $\leq 0$  to draw hairline lines
- miter\_limit - Parameter for miter join style

The stride is normally  $2 * \text{sizeof}(\text{float})$  but may be more if the vertex coordinates are in an array of some structure type, e.g.

```
struct VertexInfo {
    float x;
    float y;
    int id;
};

void my_draw(struct VertexInfo verts[], int vertex_count, ALLEGRO_COLOR c)
{
    al_draw_polyline((float *)verts, sizeof(VertexInfo), vertex_count,
        ALLEGRO_LINE_JOIN_NONE, ALLEGRO_LINE_CAP_NONE, c, 1.0, 1.0);
}
```

The stride may also be negative if the vertices are stored in reverse order.

Since: 5.1.0

See also: [al\\_draw\\_polygon](#), [ALLEGRO\\_LINE\\_JOIN](#), [ALLEGRO\\_LINE\\_CAP](#)

### 38.7.2 al\_draw\_polygon

```
void al_draw_polygon(const float *vertices, int vertex_count,
    int join_style, ALLEGRO_COLOR color, float thickness, float miter_limit)
```

Draw an unfilled polygon. This is the same as passing [ALLEGRO\\_LINE\\_CAP\\_CLOSED](#) to [al\\_draw\\_polyline](#).

- vertex - Interleaved array of (x, y) vertex coordinates
- vertex\_count - Number of vertices in the array
- join\_style - Member of `ALLEGRO_LINE_JOIN` specifying how to render the joins between line segments
- color - Color of the line
- thickness - Thickness of the line, pass `<= 0` to draw hairline lines
- miter\_limit - Parameter for miter join style

Since: 5.1.0

See also: [al\\_draw\\_filled\\_polygon](#), [al\\_draw\\_polyline](#), [ALLEGRO\\_LINE\\_JOIN](#)

### 38.7.3 `al_draw_filled_polygon`

```
void al_draw_filled_polygon(const float *vertices, int vertex_count,
                           ALLEGRO_COLOR color)
```

Draw a filled, simple polygon. Simple means it does not have to be convex but must not be self-overlapping.

- vertices - Interleaved array of (x, y) vertex coordinates
- vertex\_count - Number of vertices in the array
- color - Color of the filled polygon

When the y-axis is facing downwards (the usual), the coordinates must be ordered anti-clockwise.

Since: 5.1.0

See also: [al\\_draw\\_polygon](#), [al\\_draw\\_filled\\_polygon\\_with\\_holes](#)

### 38.7.4 `al_draw_filled_polygon_with_holes`

```
void al_draw_filled_polygon_with_holes(const float *vertices,
                                       const int *vertex_counts, ALLEGRO_COLOR color)
```

Draws a filled simple polygon with zero or more other simple polygons subtracted from it - the holes. The holes cannot touch or intersect with the outline of the filled polygon.

- vertices - Interleaved array of (x, y) vertex coordinates for each of the polygons, including holes.
- vertex\_counts - Number of vertices for each polygon. The number of vertices in the filled polygon is given by `vertex_counts[0]` and must be at least three. Subsequent elements indicate the number of vertices in each hole. The array must be terminated with an element with value zero.
- color - Color of the filled polygon

All hole vertices must use the opposite order (clockwise with y down) of the polygon vertices. All hole vertices must be inside the main polygon and no hole may overlap the main polygon.

For example:

```
float vertices[] = {
    0,  0, // filled polygon, upper left corner
    0, 100, // filled polygon, lower left corner
    100, 100, // filled polygon, lower right corner
    100,  0, // filled polygon, upper right corner
    10,  10, // hole, upper left
    90,  10, // hole, upper right
    90,  90, // hole, lower right
    0, 0, // terminator
```



```
};
int vertex_counts[] = {
    4, // number of vertices for filled polygon
    3, // number of vertices for hole
    0  // terminator
};
```

There are 7 vertices: four for an outer square from (0, 0) to (100, 100) in anti-clockwise order, and three more for an inner triangle in clockwise order. The outer main polygon uses vertices 0 to 3 (inclusive) and the hole uses vertices 4 to 6 (inclusive).

Since: 5.1.0

See also: [al\\_draw\\_filled\\_polygon](#), [al\\_draw\\_filled\\_polygon\\_with\\_holes](#), [al\\_triangulate\\_polygon](#)

### 38.7.5 al\_triangulate\_polygon

```
bool al_triangulate_polygon(
    const float* vertices, size_t vertex_stride, const int* vertex_counts,
    void (*emit_triangle)(int, int, int, void*), void* userdata)
```

Divides a simple polygon into triangles, with zero or more other simple polygons subtracted from it - the holes. The holes cannot touch or intersect with the outline of the main polygon. Simple means the polygon does not have to be convex but must not be self-overlapping.

*Parameters:*

- `vertices` - Interleaved array of (x, y) vertex coordinates for each of the polygons, including holes.
- `vertex_stride` - distance (in bytes) between successive pairs of vertices in the array.
- `vertex_counts` - Number of vertices for each polygon. The number of vertices in the main polygon is given by `vertex_counts[0]` and must be at least three. Subsequent elements indicate the number of vertices in each hole. The array must be terminated with an element with value zero.
- `emit_triangle` - a function to be called for every set of three points that form a triangle. The function is passed the indexes of the points in `vertices` and `userdata`.
- `userdata` - arbitrary data to be passed to `emit_triangle`.

Since: 5.1.0

See also: [al\\_draw\\_filled\\_polygon\\_with\\_holes](#)

## 38.8 Structures and types

### 38.8.1 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 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 (float)
- `u, v` - Texture coordinates measured in pixels (float)
- `color` - [ALLEGRO\\_COLOR](#) structure, storing the color of the vertex

See also: [ALLEGRO\\_PRIM\\_ATTR](#)

### 38.8.2 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](#)

### 38.8.3 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\\_ATTR](#), [ALLEGRO\\_PRIM\\_STORAGE](#)

### 38.8.4 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

### 38.8.5 `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 only in `ALLEGRO_PRIM_SHORT_2`, `ALLEGRO_PRIM_FLOAT_2` and `ALLEGRO_PRIM_FLOAT_3`.
- `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.
- `ALLEGRO_PRIM_USER_ATTR` - A user specified attribute. You can use any storage for this attribute. You may have at most `ALLEGRO_PRIM_MAX_USER_ATTR` (currently 10) of these that you can specify by adding an index to the value of `ALLEGRO_PRIM_USER_ATTR`, e.g. the first user attribute is `ALLEGRO_PRIM_USER_ATTR + 0`, the second is `ALLEGRO_PRIM_USER_ATTR + 1` and so on.

To access these custom attributes from GLSL shaders you need to declare attributes that follow this nomenclature: `user_attr_#` where `#` is the index of the attribute.

To access these custom attributes from HLSL you need to declare a parameter with the following semantics: `TEXCOORD{# + 2}` where `#` is the index of the attribute. E.g. the first attribute can be accessed via `TEXCOORD2`, second via `TEXCOORD3` and so on.

Since: 5.1.6

See also: `ALLEGRO_VERTEX_DECL`, `ALLEGRO_PRIM_STORAGE`

### 38.8.6 `ALLEGRO_PRIM_STORAGE`

`typedef enum` `ALLEGRO_PRIM_STORAGE`

Enumerates the types of storage an attribute of a custom vertex may be stored in. Many of these can only be used for `ALLEGRO_PRIM_USER_ATTR` attributes and can only be accessed via shaders. Usually no matter what the storage is specified the attribute gets converted to single precision floating point when the shader is run. Despite that, it may be advantageous to use more dense storage formats (e.g. `ALLEGRO_PRIM_NORMALIZED_UBYTE_4` instead of `ALLEGRO_PRIM_FLOAT_4`) when bandwidth (amount of memory sent to the GPU) is an issue but precision is not.

- `ALLEGRO_PRIM_FLOAT_1` - A single float

Since: 5.1.6

- `ALLEGRO_PRIM_FLOAT_2` - A doublet of floats
- `ALLEGRO_PRIM_FLOAT_3` - A triplet of floats
- `ALLEGRO_PRIM_FLOAT_4` - A quad of floats  
Since: 5.1.6
- `ALLEGRO_PRIM_SHORT_2` - A doublet of shorts
- `ALLEGRO_PRIM_SHORT_4` - A quad of shorts  
Since: 5.1.6
- `ALLEGRO_PRIM_UBYTE_4` - A quad of unsigned bytes  
Since: 5.1.6
- `ALLEGRO_PRIM_NORMALIZED_SHORT_2` - A doublet of shorts. Before being sent to the shader, each component is divided by 32767. Each component of the resultant float doublet ranges between -1.0 and 1.0  
Since: 5.1.6
- `ALLEGRO_PRIM_NORMALIZED_SHORT_4` - A quad of shorts. Before being sent to the shader, each component is divided by 32767. Each component of the resultant float quad ranges between -1.0 and 1.0  
Since: 5.1.6
- `ALLEGRO_PRIM_NORMALIZED_UBYTE_4` - A quad of unsigned bytes. Before being sent to the shader, each component is divided by 255. Each component of the resultant float quad ranges between 0.0 and 1.0  
Since: 5.1.6
- `ALLEGRO_PRIM_NORMALIZED_USHORT_2` - A doublet of unsigned shorts. Before being sent to the shader, each component is divided by 65535. Each component of the resultant float doublet ranges between 0.0 and 1.0  
Since: 5.1.6
- `ALLEGRO_PRIM_NORMALIZED_USHORT_4` - A quad of unsigned shorts. Before being sent to the shader, each component is divided by 65535. Each component of the resultant float quad ranges between 0.0 and 1.0  
Since: 5.1.6
- `ALLEGRO_PRIM_HALF_FLOAT_2` - A doublet of half-precision floats. Note that this storage format is not supported on all platforms. `al_create_vertex_decl` will return NULL if you use it on those platforms  
Since: 5.1.6
- `ALLEGRO_PRIM_HALF_FLOAT_4` - A quad of half-precision floats. Note that this storage format is not supported on all platforms. `al_create_vertex_decl` will return NULL if you use it on those platforms.  
Since: 5.1.6

See also: `ALLEGRO_PRIM_ATTR`

### 38.8.7 `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.

### 38.8.8 ALLEGRO\_PRIM\_QUALITY

```
#define ALLEGRO_PRIM_QUALITY 10
```

Controls the quality of the approximation of curved primitives (e.g. circles). Curved primitives are drawn by approximating them with a sequence of line segments. By default, this roughly corresponds to error of less than half of a pixel.

### 38.8.9 ALLEGRO\_LINE\_JOIN

```
typedef enum ALLEGRO_LINE_JOIN
```

- ALLEGRO\_LINE\_JOIN\_NONE
- ALLEGRO\_LINE\_JOIN\_BEVEL
- ALLEGRO\_LINE\_JOIN\_ROUND
- ALLEGRO\_LINE\_JOIN\_MITER

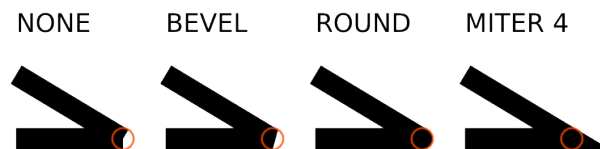


Figure 38.3: *ALLEGRO\_LINE\_JOIN* styles

See the picture for the difference.

The maximum miter length (relative to the line width) can be specified as parameter to the polygon functions.

Since: 5.1.0

See also: [al\\_draw\\_polygon](#)

### 38.8.10 ALLEGRO\_LINE\_CAP

```
typedef enum ALLEGRO_LINE_CAP
```

- ALLEGRO\_LINE\_CAP\_NONE
- ALLEGRO\_LINE\_CAP\_SQUARE
- ALLEGRO\_LINE\_CAP\_ROUND
- ALLEGRO\_LINE\_CAP\_TRIANGLE
- ALLEGRO\_LINE\_CAP\_CLOSED

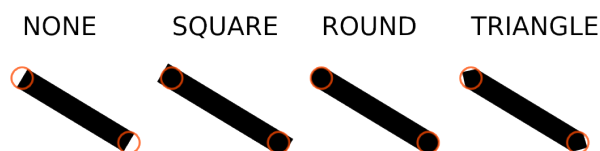


Figure 38.4: *ALLEGRO\_LINE\_CAP* styles

See the picture for the difference.

`ALLEGRO_LINE_CAP_CLOSED` is different from the others - it causes the polygon to have no caps. (And the [ALLEGRO\\_LINE\\_JOIN](#) style will determine how the vertex looks.)

Since: 5.1.0

See also: [al\\_draw\\_polygon](#)

### 38.8.11 ALLEGRO\_VERTEX\_BUFFER

```
typedef struct ALLEGRO_VERTEX_BUFFER ALLEGRO_VERTEX_BUFFER;
```

A GPU vertex buffer that you can use to store vertices on the GPU instead of uploading them afresh during every drawing operation.

Since: 5.1.3

See also: [al\\_create\\_vertex\\_buffer](#), [al\\_destroy\\_vertex\\_buffer](#)

### 38.8.12 ALLEGRO\_INDEX\_BUFFER

```
typedef struct ALLEGRO_INDEX_BUFFER ALLEGRO_INDEX_BUFFER;
```

A GPU index buffer that you can use to store indices of vertices in a vertex buffer on the GPU instead of uploading them afresh during every drawing operation.

Since: 5.1.8

See also: [al\\_create\\_index\\_buffer](#), [al\\_destroy\\_index\\_buffer](#)

### 38.8.13 ALLEGRO\_PRIM\_BUFFER\_FLAGS

```
typedef enum ALLEGRO_PRIM_BUFFER_FLAGS
```

Flags to specify how to create a vertex or an index buffer.

- `ALLEGRO_PRIM_BUFFER_STREAM` - Hints to the driver that the buffer is written to often, but used only a few times per frame
- `ALLEGRO_PRIM_BUFFER_STATIC` - Hints to the driver that the buffer is written to once and is used often
- `ALLEGRO_PRIM_BUFFER_DYNAMIC` - Hints to the driver that the buffer is written to often and is used often
- `ALLEGRO_PRIM_BUFFER_READWRITE` - Specifies that you want to be able read from this buffer. By default this is disabled for performance. Some platforms (like OpenGL ES) do not support reading from vertex buffers, so if you pass this flag to `al_create_vertex_buffer` or `al_create_index_buffer` the call will fail.

Since: 5.1.3

See also: [al\\_create\\_vertex\\_buffer](#), [al\\_create\\_index\\_buffer](#)

## Shader routines

These functions are declared in the main Allegro header file:

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

### 39.1 ALLEGRO\_SHADER

```
typedef struct ALLEGRO_SHADER ALLEGRO_SHADER;
```

An **ALLEGRO\_SHADER** is a program that runs on the GPU. It combines both a vertex and a pixel shader. (In OpenGL terms, an **ALLEGRO\_SHADER** is actually a *program* which has one or more *shaders* attached. This can be confusing.)

The source code for the underlying vertex or pixel shader can be provided either as GLSL or HLSL, depending on the value of **ALLEGRO\_SHADER\_PLATFORM** used when creating it.

Since: 5.1.0

### 39.2 ALLEGRO\_SHADER\_TYPE

```
typedef enum ALLEGRO_SHADER_TYPE ALLEGRO_SHADER_TYPE;
```

Used with **al\_attach\_shader\_source** and **al\_attach\_shader\_source\_file** to specify how to interpret the attached source.

#### ALLEGRO\_VERTEX\_SHADER

A vertex shader is executed for each vertex it is used with. The program will output exactly one vertex at a time.

When Allegro's graphics are being used then in addition to all vertices of primitives from the primitives addon, each drawn bitmap also consists of four vertices.

#### ALLEGRO\_PIXEL\_SHADER

A pixel shader is executed for each pixel it is used with. The program will output exactly one pixel at a time - either in the backbuffer or in the current target bitmap.

With Allegro's builtin graphics this means the shader is for example called for each destination pixel of the output of an **al\_draw\_bitmap** call.

A more accurate term for pixel shader would be fragment shader since one final pixel in the target bitmap is not necessarily composed of only a single output but of multiple fragments (for example when multi-sampling is being used).

Since: 5.1.0

### 39.3 ALLEGRO\_SHADER\_PLATFORM

```
typedef enum ALLEGRO_SHADER_PLATFORM ALLEGRO_SHADER_PLATFORM;
```

The underlying platform which the [ALLEGRO\\_SHADER](#) is built on top of, which dictates the language used to program the shader.

- [ALLEGRO\\_SHADER\\_AUTO](#)
- [ALLEGRO\\_SHADER\\_GLSL](#) - OpenGL Shading Language
- [ALLEGRO\\_SHADER\\_HLSL](#) - High Level Shader Language (for Direct3D)

Since: 5.1.0

### 39.4 al\_create\_shader

```
ALLEGRO_SHADER *al_create_shader(ALLEGRO_SHADER_PLATFORM platform)
```

Create a shader object.

The platform argument is one of the [ALLEGRO\\_SHADER\\_PLATFORM](#) values, and specifies the type of shader object to create, and which language is used to program the shader.

The shader platform must be compatible with the type of display that you will use the shader with. For example, you cannot create and use a HLSL shader on an OpenGL display, nor a GLSL shader on a Direct3D display.

The [ALLEGRO\\_SHADER\\_AUTO](#) value automatically chooses the appropriate platform for the display currently targeted by the calling thread; there must be such a display. It will create a GLSL shader for an OpenGL display, and a HLSL shader for a Direct3D display.

Returns the shader object on success. Otherwise, returns NULL.

Since: 5.1.0

See also: [al\\_attach\\_shader\\_source](#), [al\\_attach\\_shader\\_source\\_file](#), [al\\_build\\_shader](#), [al\\_use\\_shader](#), [al\\_destroy\\_shader](#), [al\\_get\\_shader\\_platform](#)

### 39.5 al\_attach\_shader\_source

```
bool al_attach_shader_source(ALLEGRO_SHADER *shader, ALLEGRO_SHADER_TYPE type,  
                             const char *source)
```

Attaches the shader's source code to the shader object and compiles it. Passing NULL deletes the underlying (OpenGL or DirectX) shader. See also [al\\_attach\\_shader\\_source\\_file](#) if you prefer to obtain your shader source from an external file.

If you do not use [ALLEGRO\\_PROGRAMMABLE\\_PIPELINE](#) Allegro's graphics functions will not use any shader specific functions themselves. In case of a system with no fixed function pipeline (like OpenGL ES 2 or OpenGL 3 or 4) this means Allegro's drawing functions cannot be used.

TODO: Is [ALLEGRO\\_PROGRAMMABLE\\_PIPELINE](#) set automatically in this case?

When [ALLEGRO\\_PROGRAMMABLE\\_PIPELINE](#) is used the following shader uniforms are provided by Allegro and can be accessed in your shaders:

#### [al\\_projview\\_matrix](#)

matrix for Allegro's orthographic projection multiplied by the [al\\_use\\_transform](#) matrix. The type is mat4 in GLSL, and float4x4 in HLSL.

#### [al\\_use\\_tex](#)

whether or not to use the bound texture. The type is bool in both GLSL and HLSL.



**al\_tex**

the texture if one is bound. The type is sampler2D in GLSL and texture in HLSL.

**al\_use\_tex\_matrix**

whether or not to use a texture matrix (used by the primitives addon). The type is bool in both GLSL and HLSL.

**al\_tex\_matrix**

the texture matrix (used by the primitives addon). Your shader should multiply the texture coordinates by this matrix. The type is mat4 in GLSL, and float4x4 in HLSL.

For GLSL shaders the vertex attributes are passed using the following variables:

**al\_pos**

vertex position attribute. Type is vec4.

**al\_texcoord**

vertex texture coordinate attribute. Type is vec2.

**al\_color**

vertex color attribute. Type is vec4.

For HLSL shaders the vertex attributes are passed using the following semantics:

**POSITION0**

vertex position attribute. Type is float4.

**TEXCOORD0**

vertex texture coordinate attribute. Type is float2.

**TEXCOORD1**

vertex color attribute. Type is float4.

Also, each shader variable has a corresponding macro name that can be used when defining the shaders using string literals. Don't use these macros with the other shader functions as that will lead to undefined behavior.

- ALLEGRO\_SHADER\_VAR\_PROJVIEW\_MATRIX for "al\_projview\_matrix"
- ALLEGRO\_SHADER\_VAR\_POS for "al\_pos"
- ALLEGRO\_SHADER\_VAR\_COLOR for "al\_color"
- ALLEGRO\_SHADER\_VAR\_TEXCOORD for "al\_texcoord"
- ALLEGRO\_SHADER\_VAR\_USE\_TEX for "al\_use\_tex"
- ALLEGRO\_SHADER\_VAR\_TEX for "al\_tex"
- ALLEGRO\_SHADER\_VAR\_USE\_TEX\_MATRIX for "al\_use\_tex\_matrix"
- ALLEGRO\_SHADER\_VAR\_TEX\_MATRIX for "al\_tex\_matrix"

Examine the output of [al\\_get\\_default\\_shader\\_source](#) for an example of how to use the above uniforms and attributes.

Returns true on access and false on error, in which case the error log is updated. The error log can be retrieved with [al\\_get\\_shader\\_log](#).

Since: 5.1.0

See also: [al\\_attach\\_shader\\_source\\_file](#), [al\\_build\\_shader](#), [al\\_get\\_default\\_shader\\_source](#), [al\\_get\\_shader\\_log](#)

### 39.6 `al_attach_shader_source_file`

```
bool al_attach_shader_source_file(ALLEGRO_SHADER *shader,  
    ALLEGRO_SHADER_TYPE type, const char *filename)
```

Like `al_attach_shader_source` but reads the source code for the shader from the named file.

Returns true on access and false on error, in which case the error log is updated. The error log can be retrieved with `al_get_shader_log`.

Since: 5.1.0

See also: `al_attach_shader_source`, `al_build_shader`, `al_get_shader_log`

### 39.7 `al_build_shader`

```
bool al_build_shader(ALLEGRO_SHADER *shader)
```

This is required before the shader can be used with `al_use_shader`. It should be called after successfully attaching the pixel and/or vertex shaders with `al_attach_shader_source` or `al_attach_shader_source_file`.

Returns true on access and false on error, in which case the error log is updated. The error log can be retrieved with `al_get_shader_log`.

Since: 5.1.6

See also: `al_use_shader`, `al_get_shader_log`

### 39.8 `al_get_shader_log`

```
const char *al_get_shader_log(ALLEGRO_SHADER *shader)
```

Return a read-only string containing the information log for a shader program. The log is updated by certain functions, such as `al_attach_shader_source` or `al_build_shader` when there is an error.

This function never returns NULL.

Since: 5.1.0

See also: `al_attach_shader_source`, `al_attach_shader_source_file`, `al_build_shader`

### 39.9 `al_get_shader_platform`

```
ALLEGRO_SHADER_PLATFORM al_get_shader_platform(ALLEGRO_SHADER *shader)
```

Returns the platform the shader was created with (either `ALLEGRO_SHADER_HLSL` or `ALLEGRO_SHADER_GLSL`).

Since: 5.1.6

See also: `al_create_shader`

### 39.10 `al_use_shader`

```
bool al_use_shader(ALLEGRO_SHADER *shader)
```

Uses the shader for subsequent drawing operations on the current target bitmap. Pass NULL to stop using any shader on the current target bitmap.

Returns true on success. Otherwise returns false, e.g. because the shader is incompatible with the target bitmap.

Since: 5.1.6

See also: [al\\_destroy\\_shader](#), [al\\_set\\_shader\\_sampler](#), [al\\_set\\_shader\\_matrix](#), [al\\_set\\_shader\\_int](#), [al\\_set\\_shader\\_float](#), [al\\_set\\_shader\\_bool](#), [al\\_set\\_shader\\_int\\_vector](#), [al\\_set\\_shader\\_float\\_vector](#)

## 39.11 `al_destroy_shader`

```
void al_destroy_shader(ALLEGRO_SHADER *shader)
```

Destroy a shader. Any bitmaps which currently use the shader will implicitly stop using the shader. In multi-threaded programs, be careful that no such bitmaps are being accessed by other threads at the time.

As a convenience, if the target bitmap of the calling thread is using the shader then the shader is implicitly unused before being destroyed.

This function does nothing if the shader argument is NULL.

Since: 5.1.0

See also: [al\\_create\\_shader](#)

## 39.12 `al_set_shader_sampler`

```
bool al_set_shader_sampler(const char *name,  
                           ALLEGRO_BITMAP *bitmap, int unit)
```

Sets a texture sampler uniform and texture unit of the current target bitmap's shader. The given bitmap must be a video bitmap.

Different samplers should use different units. The bitmap passed to Allegro's drawing functions uses the 0th unit, so if you're planning on using the `al_tex` variable in your pixel shader as well as another sampler, set the other sampler to use a unit different from 0. With the primitives add-on, it is possible to free up the 0th unit by passing NULL as the texture argument to the relevant drawing functions. In this case, you may set a sampler to use the 0th unit and thus not use `al_tex` (the `al_use_tex` variable will be set to false).

Returns true on success. Otherwise returns false, e.g. if the uniform by that name does not exist in the shader.

Since: 5.1.0

See also: [al\\_use\\_shader](#)

## 39.13 `al_set_shader_matrix`

```
bool al_set_shader_matrix(const char *name,  
                           ALLEGRO_TRANSFORM *matrix)
```

Sets a matrix uniform of the current target bitmap's shader.

Returns true on success. Otherwise returns false, e.g. if the uniform by that name does not exist in the shader.

Since: 5.1.0

See also: [al\\_use\\_shader](#)

### 39.14 `al_set_shader_int`

```
bool al_set_shader_int(const char *name, int i)
```

Sets an integer uniform of the current target bitmap's shader.

Returns true on success. Otherwise returns false, e.g. if the uniform by that name does not exist in the shader.

Since: 5.1.0

See also: [al\\_use\\_shader](#)

### 39.15 `al_set_shader_float`

```
bool al_set_shader_float(const char *name, float f)
```

Sets a float uniform of the target bitmap's shader.

Returns true on success. Otherwise returns false, e.g. if the uniform by that name does not exist in the shader.

Since: 5.1.0

See also: [al\\_use\\_shader](#)

### 39.16 `al_set_shader_bool`

```
bool al_set_shader_bool(const char *name, bool b)
```

Sets a boolean uniform of the target bitmap's shader.

Returns true on success. Otherwise returns false, e.g. if the uniform by that name does not exist in the shader.

Since: 5.1.6

See also: [al\\_use\\_shader](#)

### 39.17 `al_set_shader_int_vector`

```
bool al_set_shader_int_vector(const char *name,  
                             int num_components, int *i, int num_elems)
```

Sets an integer vector array uniform of the current target bitmap's shader. The 'num\_components' parameter can take one of the values 1, 2, 3 or 4. If it is 1 then an array of 'num\_elems' integer elements is added. Otherwise each added array element is assumed to be a vector with 2, 3 or 4 components in it.

For example, if you have a GLSL uniform declared as uniform ivec3 flowers[4] or an HLSL uniform declared as uniform int3 flowers[4], then you'd use this function from your code like so:

```
int flowers[4][3] =  
{  
    {1, 2, 3},  
    {4, 5, 6},  
    {7, 8, 9},  
    {2, 5, 7}  
};  
  
al_set_shader_int_vector("flowers", 3, (int*)flowers, 4);
```

Returns true on success. Otherwise returns false, e.g. if the uniform by that name does not exist in the shader.

Since: 5.1.0

See also: [al\\_set\\_shader\\_float\\_vector](#), [al\\_use\\_shader](#)

### 39.18 `al_set_shader_float_vector`

```
bool al_set_shader_float_vector(const char *name,
                               int num_components, float *f, int num_elems)
```

Same as [al\\_set\\_shader\\_int\\_vector](#) except all values are float instead of int.

Since: 5.1.0

See also: [al\\_set\\_shader\\_int\\_vector](#), [al\\_use\\_shader](#)

### 39.19 `al_get_default_shader_source`

```
char const *al_get_default_shader_source(ALLEGRO_SHADER_PLATFORM platform,
                                          ALLEGRO_SHADER_TYPE type)
```

Returns a string containing the source code to Allegro's default vertex or pixel shader appropriate for the passed platform. The `ALLEGRO_SHADER_AUTO` value means GLSL is used if OpenGL is being used otherwise HLSL. `ALLEGRO_SHADER_AUTO` requires that there is a current display set on the calling thread. This function can return NULL if Allegro was built without support for shaders of the selected platform.

Since: 5.1.6

See also: [al\\_attach\\_shader\\_source](#)



## Video streaming addon

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

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

Currently we have an `ffmpeg` backend and a `Ogg` backend (Theora + Vorbis). The choice is fixed at compile time. See <http://ffmpeg.org/> and <http://xiph.org/> for installation instructions, licensing information and supported video formats.

### 40.1 ALLEGRO\_VIDEO\_EVENT\_TYPE

```
enum ALLEGRO_VIDEO_EVENT_TYPE
```

- `ALLEGRO_EVENT_VIDEO_FRAME_ALLOC`
- `ALLEGRO_EVENT_VIDEO_FRAME_SHOW`

Since: 5.1.0

### 40.2 al\_open\_video

```
ALLEGRO_VIDEO *al_open_video(char const *filename)
```

Reads a video file. This does not start streaming yet but reads the meta info so you can query e.g. the size or audio rate.

Since: 5.1.0

### 40.3 al\_close\_video

```
void al_close_video(ALLEGRO_VIDEO *video)
```

Closes the video and frees all allocated resources. The video pointer is invalid after the function returns.

Since: 5.1.0

### 40.4 al\_start\_video

```
void al_start_video(ALLEGRO_VIDEO *video, ALLEGRO_MIXER *mixer)
```

Starts streaming the video from the beginning.

Since: 5.1.0

### 40.5 `al_start_video_with_voice`

```
void al_start_video_with_voice(ALLEGRO_VIDEO *video, ALLEGRO_VOICE *voice)
```

Like `al_start_video` but audio is routed to the provided voice.

Since: 5.1.0

### 40.6 `al_get_video_event_source`

```
ALLEGRO_EVENT_SOURCE *al_get_video_event_source(ALLEGRO_VIDEO *video)
```

Get an event source for the video. The possible events are described under [ALLEGRO\\_VIDEO\\_EVENT\\_TYPE](#).

Since: 5.1.0

### 40.7 `al_pause_video`

```
void al_pause_video(ALLEGRO_VIDEO *video, bool paused)
```

Paused or resumes playback.

Since: 5.1.0

### 40.8 `al_is_video_paused`

```
bool al_is_video_paused(ALLEGRO_VIDEO *video)
```

Returns true if the video is currently paused.

Since: 5.1.0

### 40.9 `al_get_video_aspect_ratio`

```
double al_get_video_aspect_ratio(ALLEGRO_VIDEO *video)
```

Returns the aspect ratio of the video. Videos often do not use square pixels so you should always check the aspect ratio before displaying video frames.

Returns zero or negative value if the aspect ratio is unknown.

Since: 5.1.0

### 40.10 `al_get_video_audio_rate`

```
double al_get_video_audio_rate(ALLEGRO_VIDEO *video)
```

Returns the audio rate of the video, in Hz.

Since: 5.1.0

### 40.11 `al_get_video_fps`

```
double al_get_video_fps(ALLEGRO_VIDEO *video)
```

Returns the speed of the video in frames per second. Often this will not be an integer value.

Since: 5.1.0



## 40.12 al\_get\_video\_width

```
int al_get_video_width(ALLEGRO_VIDEO *video)
```

Returns the number of pixel raw pixel columns in the video stream. Multiply this with the aspect ratio to get the true width.

Since: 5.1.0

See also: [al\\_get\\_video\\_aspect\\_ratio](#)

## 40.13 al\_get\_video\_height

```
int al_get_video_height(ALLEGRO_VIDEO *video)
```

Returns the number of rows in the video. Typically this will be 720 or 1080.

Since: 5.1.0

## 40.14 al\_get\_video\_frame

```
ALLEGRO_BITMAP *al_get_video_frame(ALLEGRO_VIDEO *video)
```

Returns the current video frame. The bitmap is owned by the video so do not attempt to free it. The bitmap will stay valid until the next call to `al_get_video_frame`.

Since: 5.1.0

## 40.15 al\_get\_video\_position

```
double al_get_video_position(ALLEGRO_VIDEO *video, int which)
```

Returns the current position of the video stream in seconds since the beginning. The parameter has the following meaning:

- 0: Return the actual position.
- 1: Return the video decoding position. If this lags behind video decoding is taking too long and the video can't be displayed properly.
- 2: Return the audio decoding position. Audio gets easily out of sync for some reason - but this addon does not provide any means to do much about it.

Since: 5.1.0

## 40.16 al\_seek\_video

```
bool al_seek_video(ALLEGRO_VIDEO *video, double pos_in_seconds)
```

Seek to a different position in the video. Right now this does not work very well in the ffmpeg backend when seeking backwards and will often lose audio/video synchronization if doing so.

Since: 5.1.0