# Th A gro 5 ibrar

# R f r nc Manua

© 2008 — 2011

# C ntents

Contents iii


1 Getting started guide 1
1.1 Introduction . . . 1
1.2 Structure of the library and its addons . . . 1
1.3 Initialisation . . . 2
1.4 Opening a window . . . 2
1.5 Display an image . . . 2
1.6 Changing the drawing target . . . 2
1.7 Event queues and input . . . 2
1.8 Displaying some text . . . 3
1.9 Drawing primitives . . . 3
1.10 Blending . . . 3
1.11 Sound . . . 3
1.12 Not the end . . . 3

2 Configuration files 5
2.1 ALLEGRO_CONFIG . . . 5
2.2 al_create_config . . . 5
2.3 al_destroy_config . . . 5
2.4 al_load_config_file . . . 5
2.5 al_load_config_file_f . . . 6
2.6 al_save_config_file . . . 6
2.7 al_save_config_file_f . . . 6
2.8 al_add_config_section . . . 6
2.9 al_add_config_comment . . . 6
2.10 al_get_config_value . . . 7
2.11 al_set_config_value . . . 7
2.12 al_get_first_config_section . . . 7
2.13 al_get_next_config_section . . . 7
2.14 al_get_first_config_entry . . . 8
2.15 al_get_next_config_entry . . . 8
2.16 al_merge_config . . . 8
2.17 al_merge_config_into . . . 8

3 Display routines 9
3.1 Display creation . . . 9
3.1.1 ALLEGRO_DISPLAY . . . 9
3.1.2 al_create_display . . . 9
3.1.3 al_destroy_display . . . 9
3.1.4 al_get_new_display_flags . . . 10
3.1.5 al_get_new_display_refresh_rate . . . 10
3.1.6 al_get_new_window_position . . . 10
3.1.7 al_set_new_display_option . . . 10

3.1.8 al_get_new_display_option . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
3.1.9 al_reset_new_display_options . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
3.1.10 al_set_new_display_flags . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
3.1.11 al_set_new_display_refresh_rate . . . . . . . . . . . . . . . . . . . . . . . . . . 13
3.1.12 al_set_new_window_position . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
3.2 Display operations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
3.2.1 al_acknowledge_resize . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
3.2.2 al_flip_display . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
3.2.3 al_get_backbuffer . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
3.2.4 al_get_display_flags . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
3.2.5 al_get_display_format . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
3.2.6 al_get_display_height . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
3.2.7 al_get_display_refresh_rate . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
3.2.8 al_get_display_width . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
3.2.9 al_get_window_position . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
3.2.10 al_inhibit_screensaver . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
3.2.11 al_resize_display . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
3.2.12 al_set_display_icon . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
3.2.13 al_get_display_option . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
3.2.14 al_set_window_position . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
3.2.15 al_set_window_title . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
3.2.16 al_toggle_display_flag . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
3.2.17 al_update_display_region . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
3.2.18 al_wait_for_vsync . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
3.2.19 al_get_display_event_source . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
3.3 Fullscreen display modes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
3.3.1 ALLEGRO_DISPLAY_MODE . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
3.3.2 al_get_display_mode . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
3.3.3 al_get_num_display_modes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
3.4 Mon-84es . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
343.1 ALLEGROMONITOR_INFOs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
3.3.2 al_get_new_displayadaptver . . . . . . . . . . . . . . . . . . . . . . . . . . . . 98
3.3.3 alsget_new_displayadaptver. . . . . . . . . . . . . . . . . . . . . . . . . . . . 98
343.4 al_getmMon-84_infon . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 98
3.353 al_get_numvideoyadaptvens . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 98

Eeven3system3and3_evense 218
ALLEGROEVENTh . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
ALLEGROEVENT_JOYSTICK_AXISn . . . . . . . . . . . . . . . . . . . . . . . . . . .
ALLEGROEVENT_JOYSTICK_BUT9TON_DOWNn . . . . . . . . . . . . . . . . . . . . .
ALLEGROEVENT_JOYSTICK_BUT9TON_UPe . . . . . . . . . . . . . . . . . . . . . . . .
ALLEGROEVENT_JOYSTICK_CONFIGURATIONn . . . . . . . . . . . . . . . . .
ALLEGROEVENT_KEY_DOWNn . . . . . . . . . . . . . . . . . . . . . . . .
ALLEGROEVENT_KEY_UPe. . . . . . . . . . . . . . . . . . . . . . . .
ALLEGROEVENT_KEY_CHARe . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
ALLEGROEVENT_MOUSE_AXESn . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
ALLEGROEVENT_MOUSE_BUT9TON_DOWNn . . . . . . . . . .
ALLEGROEVENT_MOUSE_BUT9TON_UPe . . . . . . . . . . . . . . . . . . . . . .
ALLEGROEVENT_MOUSE_WRPEDn . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
ALLEGROEVENT_MOUSE_ENTERO_DISPLAt . . . . . . . . . . . . . . . . . . . . . . . .
ALLEGROEVENT_MOUSE_LEAt . . . . . . . . . . . . . . . . . .
ALLEGROEVENT_TIMERn . . . . . . . . . . . . . . . . . . . . . . . . . . .
ALLEGROEVENT__DISPLAYEXPOSEs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
ALLEGROEVENT__DISPLAYRESIZEs . . . . . . . . . . . . . . . . . . . .
ALLEGROEVENT__DISPLAYCLOSEs. . . . . . . . . . . . . . . . . . . . . . . . . . . . .
ALLEGROEVENT__DISPLAYLOSTt . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . .

4.21 ALLEGRO_EVENT_DISPLAY_SWITCH_OUT . . . 27
4.22 ALLEGRO_EVENT_DISPLAY_SWITCH_IN . . . 27
4.23 ALLEGRO_EVENT_DISPLAY_ORIENTATION . . . 27
4.24 ALLEGRO_USER_EVENT . . . 28
4.25 ALLEGRO_EVENT_QUEUE . . . 28
4.26 ALLEGRO_EVENT_SOURCE . . . 28
4.27 ALLEGRO_EVENT_TYPE . . . 28
4.28 ALLEGRO_GET_EVENT_TYPE . . . 29
4.29 ALLEGRO_EVENT_TYPE_IS_USER . . . 29
4.30 al_create_event_queue . . . 29
4.31 al_init_user_event_source . . . 29
4.32 al_destroy_event_queue . . . 30
4.33 al_destroy_user_event_source . . . 30
4.34 al_drop_next_event . . . 30
4.35 al_emit_user_event . . . 30
4.36 al_is_event_queue_empty . . . 31
4.37 al_flush_event_queue . . . 31
4.38 al_get_event_source_data . . . 31
4.39 al_get_next_event . . . 31
4.40 al_peek_next_event . . . 32
4.41 al_register_event_source . . . 32
4.42 al_set_event_source_data . . . 32
4.43 al_unref_user_event . . . 32
4.44 al_unregister_event_source . . . 32
4.45 al_wait_for_event . . . 33
4.46 al_wait_for_event_timed . . . 33
4.47 al_wait_for_event_until . . . 33

**5 File I/O** **35**
5.1 ALLEGRO_FILE . . . 35
5.2 ALLEGRO_FILE_INTERFACE . . . 35
5.3 ALLEGRO_SEEK . . . 36
5.4 al_fopen . . . 36
5.5 al_fopen_interface . . . 36
5.6 al_fclose . . . 36
5.7 al_fread . . . 37
5.8 al_fwrite . . . 37
5.9 al_fflush . . . 37
5.10 al_ftell . . . 37
5.11 al_fseek . . . 37
5.12 al_feof . . . 38
5.13 al_ferror . . . 38
5.14 al_fclearerr . . . 38
5.15 al_fungetc . . . 39
5.16 al_fsize . . . 39
5.17 al_fgetc . . . 39
5.18 al_fputc . . . 39
5.19 al_fread16le . . . 39
5.20 al_fread16be . . . 40
5.21 al_fwrite16le . . . 40
5.22 al_fwrite16be . . . 40
5.23 al_fread32le . . . 40
5.24 al_fread32be . . . 40
5.25 al_fwrite32le . . . 41
5.26 al_fwrite32be . . . 41
5.27 al_fgets . . . 41
5.28 al_fget_ustr . . . 41

5.29 al_fputs . . . 42
5.30 Standard I/O specific routines . . . 42
5.30.1 al_fopen_fd . . . 42
5.30.2 al_make_temp_file . . . 42
5.31 Alternative file streams . . . 43
5.31.1 al_set_new_file_interface . . . 43
5.31.2 al_set_standard_file_interface . . . 43
5.31.3 al_get_new_file_interface . . . 43
5.31.4 al_create_file_handle . . . 43
5.31.5 al_get_file_userdata . . . 43

**6 File system routines** **45**
6.1 ALLEGRO_FS_ENTRY . . . 45
6.2 ALLEGRO_FILE_MODE . . . 45
6.3 al_create_fs_entry . . . 45
6.4 al_destroy_fs_entry . . . 46
6.5 al_get_fs_entry_name . . . 46
6.6 al_update_fs_entry . . . 46
6.7 al_get_fs_entry_mode . . . 46
6.8 al_get_fs_entry_atime . . . 46
6.9 al_get_fs_entry_ctime . . . 47
6.10 al_get_fs_entry_mtime . . . 47
6.11 al_get_fs_entry_size . . . 47
6.12 al_fs_entry_exists . . . 47
6.13 al_remove_fs_entry . . . 47
6.14 al_filename_exists . . . 47
6.15 al_remove_filename . . . 48
6.16 Directory functions . . . 48
6.16.1 al_open_directory . . . 48
6.16.2 al_read_directory . . . 48
6.16.3 al_close_directory . . . 48
6.16.4 al_get_current_directory . . . 48
6.16.5 al_change_directory . . . 49
6.16.6 al_make_directory . . . 49
6.16.7 al_open_fs_entry . . . 49
6.17 Alternative filesystem functions . . . 49
6.17.1 ALLEGRO_FS_INTERFACE . . . 49
6.17.2 al_set_fs_interface . . . 50
6.17.3 al_set_standard_fs_interface . . . 50
6.17.4 al_get_fs_interface . . . 50

**7 Fixed point math routines** **51**
7.1 al_fixed . . . 51
7.2 al_itofix . . . 51
7.3 al_fixtoi . . . 52
7.4 al_fixfloor . . . 52
7.5 al_fixceil . . . 52
7.6 al_ftofix . . . 53
7.7 al_fixtof . . . 53
7.8 al_fixmul . . . 54
7.9 al_fixdiv . . . 54
7.10 al_fixadd . . . 55
7.11 al_fixsub . . . 55
7.12 Fixed point trig . . . 56
7.12.1 al_fixtorad_r . . . 56
7.12.2 al_radtofix_r . . . 56
7.12.3 al_fixsin . . . 56
7.12.4 al_fixcos . . . 57

7.12.5 al_fixtan . . . 57
7.12.6 al_fixasin . . . 58
7.12.7 al_fixacos . . . 58
7.12.8 al_fixatan . . . 58
7.12.9 al_fixatan2 . . . 59
7.12.10 al_fixsqrt . . . 59
7.12.11 al_fixhypot . . . 59

**8 Graphics routines** **61**
8.1 Colors . . . 61
8.1.1 ALLEGRO_COLOR . . . 61
8.1.2 al_map_rgb . . . 61
8.1.3 al_map_rgb_f . . . 61
8.1.4 al_map_rgba . . . 61
8.1.5 al_map_rgba_f . . . 62
8.1.6 al_unmap_rgb . . . 62
8.1.7 al_unmap_rgb_f . . . 62
8.1.8 al_unmap_rgba . . . 62
8.1.9 al_unmap_rgba_f . . . 62
8.2 Locking and pixel formats . . . 62
8.2.1 ALLEGRO_LOCKED_REGION . . . 62
8.2.2 ALLEGRO_PIXEL_FORMAT . . . 63
8.2.3 al_get_pixel_size . . . 64
8.2.4 al_get_pixel_format_bits . . . 65
8.2.5 al_lock_bitmap . . . 65
8.2.6 al_lock_bitmap_region . . . 65
8.2.7 al_unlock_bitmap . . . 65
8.3 Bitmap creation . . . 66
8.3.1 ALLEGRO_BITMAP . . . 66
8.3.2 al_create_bitmap . . . 66
8.3.3 al_create_sub_bitmap . . . 66
8.3.4 al_clone_bitmap . . . 66
8.3.5 al_destroy_bitmap . . . 67
8.3.6 al_get_new_bitmap_flags . . . 67
8.3.7 al_get_new_bitmap_format . . . 67
8.3.8 al_set_new_bitmap_flags . . . 67
8.3.9 al_add_new_bitmap_flag . . . 69
8.3.10 al_set_new_bitmap_format . . . 69
8.4 Bitmap properties . . . 70
8.4.1 al_get_bitmap_flags . . . 70
8.4.2 al_get_bitmap_format . . . 70
8.4.3 al_get_bitmap_height . . . 70
8.4.4 al_get_bitmap_width . . . 70
8.4.5 al_get_pixel . . . 70
8.4.6 al_is_bitmap_locked . . . 70
8.4.7 al_is_compatible_bitmap . . . 71
8.4.8 al_is_sub_bitmap . . . 71
8.5 Drawing operations . . . 71
8.5.1 al_clear_to_color . . . 71
8.5.2 al_draw_bitmap . . . 71
8.5.3 al_draw_tinted_bitmap . . . 72
8.5.4 al_draw_bitmap_region . . . 72
8.5.5 al_draw_tinted_bitmap_region . . . 72
8.5.6 al_draw_pixel . . . 73
8.5.7 al_draw_rotated_bitmap . . . 73
8.5.8 al_draw_tinted_rotated_bitmap . . . 73
8.5.9 al_draw_scaled_rotated_bitmap . . . 74

8.5.10 al_draw_tinted_scaled_rotated_bitmap . . . . . . . . . . . . . . . . . . . . . . . . . 74
8.5.11 al_draw_scaled_bitmap . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
8.5.12 al_draw_tinted_scaled_bitmap . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
8.5.13 al_get_target_bitmap . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
8.5.14 al_put_pixel . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
8.5.15 al_put_blended_pixel . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
8.5.16 al_set_target_bitmap . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76
8.5.17 al_set_target_backbuffer . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76
8.5.18 al_get_current_display . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
8.6B_bleing.6modesay

10.3 Keyboard modifier flags . . . 90
10.4 al_install_keyboard . . . 91
10.5 al_is_keyboard_installed . . . 91
10.6 al_uninstall_keyboard . . . 91
10.7 al_get_keyboard_state . . . 91
10.8 al_key_down . . . 91
10.9 al_keycode_to_name . . . 91
10.10 al_set_keyboard_leds . . . 92
10.11 al_get_keyboard_event_source . . . 92

**11 Memory management routines 93**
11.1 al_malloc . . . 93
11.2 al_free . . . 93
11.3 al_realloc . . . 93
11.4 al_calloc . . . 94
11.5 al_malloc_with_context . . . 94
11.6 al_free_with_context . . . 94
11.7 al_realloc_with_context . . . 94
11.8 al_calloc_with_context . . . 94
11.9 ALLEGRO_MEMORY_INTERFACE . . . 95
11.10 al_set_memory_interface . . . 95

**12 Miscellaneous routines 97**
12.1 ALLEGRO_PI . . . 97
12.2 al_run_main . . . 97

**13 Mouse routines 99**
13.1 ALLEGRO_MOUSE_STATE . . . 99
13.2 al_install_mouse . . . 99
13.3 al_is_mouse_installed . . . 99
13.4 al_uninstall_mouse . . . 100
13.5 al_get_mouse_num_axes . . . 100
13.6 al_get_mouse_num_buttons . . . 100
13.7 al_get_mouse_state . . . 100
13.8 al_get_mouse_state_axis . . . 101
13.9 al_mouse_button_down . . . 101

14.6 al_rebase_path . . . 106
14.7 al_get_path_drive . . . 106
14.8 al_get_path_num_components . . . 106
14.9 al_get_path_component . . . 107
14.10 al_get_path_tail . . . 107
14.11 al_get_path_filename . . . 107
14.12 al_get_path_basename . . . 107
14.13 al_get_path_extension . . . 107
14.14 al_set_path_drive . . . 108
14.15 al_append_path_component . . . 108
14.16 al_insert_path_component . . . 108
14.17 al_replace_path_component . . . 108
14.18 al_remove_path_component . . . 108
14.19 al_drop_path_tail . . . 108
14.20 al_set_path_filename . . . 109
14.21 al_set_path_extension . . . 109
14.22 al_path_cstr . . . 109
14.23 al_make_path_canonical . . . 109

**15 State 111**
15.1 ALLEGRO_STATE . . . 111
15.2 ALLEGRO_STATE_FLAGS . . . 111
15.3 al_restore_state . . . 112
15.4 al_store_state . . . 112
15.5 al_get_errno . . . 112
15.6 al_set_errno . . . 112

**16 System routines 113**
16.1 al_install_system . . . 113
16.2 al_init . . . 113
16.3 al_uninstall_system . . . 113
16.4 al_is_system_installed . . . 114
16.5 al_get_allegro_version . . . 114
16.6 al_get_standg6e_path_canonical . . ... . . . .al_g

**16 System routines 113**
16.1 .al_get_allegro_version 42(.)-500(500(.)-500(.)-50500(.)-con0(.)m81a5l st.)-500(.)-500(.)-5113(.)-500(.)-500
16.1 .al_get_allegro_version . . . ..... . . . .
16.1 al_install_system . . . .. . . .
14.21 al_set_path_extension. . . .. . . .

17.18 al_wait_cond . . . . . 120
17.19 al_wait_cond_until . . . . . 121
17.20 al_broadcast_cond . . . . . 121
17.21 al_signal_cond . . . . . 121

**18 Time routines 123**
18.1 ALLEGRO_TIMEOUT . . . . . 123
18.2 al_get_time . . . . . 123
18.3 al_current_time . . . . . 123
18.4 al_init_timeout . . . . . 123
18.5 al_rest . . . . . 124

**19 Timer routines 125**
19.1 ALLEGRO_TIMER

21.3.8 al_ustr_dup . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 137
21.3.9 al_ustr_dup_substr . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 138

21.14.2 al_ustr_compare . . . . . 147
21.14.3 al_ustr_ncompare . . . . . 147
21.14.4 al_ustr_has_prefix . . . . . 147
21.14.5 al_ustr_has_prefix_cstr . . . . . 147
21.14.6 al_ustr_has_suffix . . . . . 148
21.14.7 al_ustr_has_suffix_cstr . . . . . 148
21.15 UTF-16 conversion . . . . . 148
21.15.1 al_ustr_new_from_utf16 . . . . . 148
21.15.2 al_ustr_size_utf16 . . . . . 148
21.15.3 al_ustr_encode_utf16 . . . . . 148
21.16 Low-level UTF-8 routines . . . . . 149
21.16.1 al_utf8_width . . . . . 149
21.16.2 al_utf8_encode . . . . . 149
21.17 Low-level UTF-16 routines . . . . . 149
21.17.1 al_utf16_width . . . . . 149
21.17.2 al_utf16_encode . . . . . 149

**22 Platform-specific functions** **151**
22.1 Windows . . . . . 151
22.1.1 al_get_win_window_handle . . . . . 151
22.2 iPhone . . . . . 151
22.2.1 al_iphone_program_has_halted . . . . . 151
22.2.2 al_iphone_override_screen_scale . . . . . 151

**23 Direct3D integration** **153**
23.1 al_get_d3d_device . . . . . 153
23.2 al_get_d3d_system_texture . . . . . 153
23.3 al_get_d3d_video_texture . . . . . 153
23.4 al_have_d3d_non_pow2_texture_support . . . . . 153
23.5 al_have_d3d_non_square_texture_support . . . . . 154
23.6 al_get_d3d_texture_position . . . . . 154
23.7 al_is_d3d_device_lost . . . . . 154

**24 OpenGL integration** **155**
24.1 al_get_opengl_extension_list . . . . . 155
24.2 al_get_opengl_proPension_list . . . . . . .. .

25.1.10 ALLEGRO_AUDIO_STREAM . . . 161
25.1.11 ALLEGRO_VOICE . . . 162
25.2 Setting up audio . . . 162
25.2.1 al_install_audio . . . 162
25.2.2 al_uninstall_audio . . . 162
25.2.3 al_is_audio_installed . . . 162
25.2.4 al_reserve_samples . . . 162
25.3 Misc audio functions . . . 163
25.3.1 al_get_allegro_audio_version . . . 163
25.3.2 al_get_audio_depth_size . . . 163
25.3.3 al_get_channel_count . . . 163
25.4 Voice functions . . . 163
25.4.1 al_create_voice . . . 163

25.6.21 al_get_sample_instance_playing . . . 170
25.6.22 al_set_sample_instance_playing . . . 170
25.6.23 al_get_sample_instance_attached . . . 171
25.6.24 al_detach_sample_instance . . . 171
25.6.25 al_get_sample . . . 171
25.6.26 al_set_sample . . . 171
25.7 Mixer functions . . . 171
25.7.1 al_create_mixer . . . 171
25.7.2 al_destroy_mixer . . . 172
25.7.3 al_get_default_mixer . . . 172
25.7.4 al_set_default_mixer . . . 172
25.7.5 al_restore_default_mixer . . . 172
25.7.6 al_attach_mixer_to_mixer . . . 172
25.7.7 al_attach_sample_instance_to_mixer . . . 173
25.7.8 al_attach_audio_stream_to_mixer . . . 173
25.7.9 al_get_mixer_frequency . . . 173
25.7.10 al_set_mixer_frequency . . . 173
25.7.11 al_get_mixer_channels . . . 173
25.7.12 al_get_mixer_depth . . . 173
25.7.13 al_get_mixer_quality . . . 174
25.7.14 al_set_mixer_quality . . . 174
25.7.15 al_get_mixer_playing . . . 174
25.7.16 al_set_mixer_playing . . . 174
25.7.17 al_get_mixer_attached . . . 174
25.7.18 al_detach_mixer . . . 174
25.7.19 al_set_mixer_postprocess_callback . . . 175
25.8 Stream functions . . . 175
25.8.1 al_create_audio_stream . . . 175
25.8.2 al_destroy_audio_stream . . . 176
25.8.3 al_get_audio_stream_event_source . . . 176
25.8.4 al_drain_audio_stream . . . 176
25.8.5 al_rewind_audio_stream . . . 176
25.8.6 al_get_audio_stream_frequency . . . 176
25.8.7 al_get_audio_stream_channels . . . 176
25.8.8 al_get_audio_stream_depth . . . 177
25.8.9 al_get_audio_stream_length . . . 177
25.8.10 al_get_audio_stream_speed . . . 177
25.8.11 al_set_audio_stream_speed . . . 177
25.8.12 al_get_audio_stream_gain . . . 177
25.8.13 al_set_audio_stream_gain . . . 177
25.8.14 al_get_audio_stream_pan . . . 178
25.8.15 al_set_audio_stream_pan . . . 178
25.8.16 al_get_audio_stream_playing . . . 178
25.8.17 al_set_audio_stream_playing . . . 178
25.8.18 al_get_audio_stream_playmode . . . 178
25.8.19 al_set_audio_stream_playmode . . . 178
25.8.20 al_get_audio_stream_attached . . . 179
25.8.21 al_detach_audio_stream . . . 179
25.8.22 al_get_audio_stream_fragment . . . 179
25.8.23 al_set_audio_stream_fragment . . . 179
25.8.24 al_get_audio_stream_fragments . . . 179
25.8.25 al_get_available_audio_stream_fragments . . . 180
25.8.26 al_seek_audio_stream_secs . . . 180
25.8.27 al_get_audio_stream_position_secs . . . 180
25.8.28 al_get_audio_stream_length_secs . . . 180
25.8.29 al_set_audio_stream_loop_secs . . . 180
25.9 Audio file I/O . . . 180

25.9.1 al_register_sample_loader . . . 180
25.9.2 al_register_sample_loader_f . . . 181
25.9.3 al_register_sample_saver . . . 181
25.9.4 al_register_sample_saver_f . . . 181
25.9.5 al_register_audio_stream_loader . . . 182
25.9.6 al_register_audio_stream_loader_f . . . 182
25.9.7 al_load_sample . . . 182
25.9.8 al_load_sample_f . . . 182
25.9.9 al_load_audio_stream . . . 183
25.9.10 al_load_audio_stream_f . . . 183
25.9.11 al_save_sample . . . 184
25.9.12 al_save_sample_f . . . 184

**26 Audio codecs addon** **185**
26.1 al_init_acodec_addon . . . 185
26.2 al_get_allegro_acodec_version . . . 185

**27 Color addon** **187**
27.1 al_color_cmyk . . . 187
27.2 al_color_cmyk_to_rgb . . . 187
27.3 al_color_hsl . . . 187
27.4 al_color_hsl_to_rgb . . . 187
27.5 al_color_hsv . . . 188
27.6 al_color_hsv_to_rgb . . . 188
27.7 al_color_html . . . 188
27.8 al_color_html_to_rgb . . . 188
27.9 al_color_rgb_to_html . . . 189
27.10 al_color_name . . . 189
27.11 al_color_name_to_rgb . . . 189
27.12 al_color_rgb_to_cmyk . . . 190
27.13 al_color_rgb_to_hsl . . . 190
27.14 al_color_rgb_to_hsv . . . 190
27.15 al_color_rgb_to_name . . . 191
27.16 al_color_rgb_to_yuv . . . 191
27.17 al_color_yuv . . . 191
27.18 al_color_yuv_to_rgb . . . 191
27.19 al_get_allegro_color_version . . . 191

**28 Font addons** **193**
28.1 General font routines . . . 193
28.1.1 ALLEGRO_FONT . . . 193
28.1.2 al_init_font_addon . . . 193
28.1.3 al_shutdown_font_addon . . . 193
28.1.4 al_load_font . . . 194
28.1.5 al_destroy_font . . . 194
28.1.6 al_register_font_loader . . . 194
28.1.7 al_get_font_line_height . . . 1942. .
. . . 194
. . . 193
. . . 194

28.1.19 al_get_ustr_dimensions . . . 197
28.1.20 al_get_allegro_font_version . . . 197
28.2 Bitmap fonts . . . 197
28.2.1 al_grab_font_from_bitmap . . . 197
28.2.2 al_load_bitmap_font . . . 198
28.3 TTF fonts . . . 198
28.3.1 al_init_ttf_addon . . . 199
28.3.2 al_shutdown_ttf_addon . . . 199
28.3.3 al_load_ttf_font . . . 199
28.3.4 al_load_ttf_font_f . . . 199
28.3.5 al_get_allegro_ttf_version . . . 199

**29 Image I/O addon** **201**
29.1 al_init_image_addon . . . 201
29.2 al_shutdown_image_addon . . . 201
29.3 al_get_allegro_image_version . . . 201

**30 Memfile interface** **203**
30.1 al_open_memfile . . . 203
30.2 al_get_allegro_memfile_version . . . 203

**31 Native dialogs support** **205**
31.1 ALLEGRO_FILECHOOSER . . . 205
31.2 ALLEGRO_TEXTLOG . . . 205
31.3 al_create_native_file_dialog . . . 205
31.4 al_show_native_file_dialog . . . 206
31.5 al_get_native_file_dialog_count . . . 206
31.6 al_get_native_file_dialog_path . . . 206
31.7 al_destroy_native_file_dialog . . . 207
31.8 al_show_native_message_box . . . 207
31.9 al_open_native_text_log . . . 208
31.10 al_close_native_text_log . . . 208
31.11 al_append_native_text_log . . . 208
31.12 al_get_native_text_log_event_source . . . 209
31.13 al_get_allegro_native_dialog_version . . . 209

**32 PhysicsFS integration** **211**
32.1 al_set_physfs_file_interface . . . 211
32.2 al_get_allegro_physfs_version . . . 211

**33 Primitives addon** **213**
33.1 General . . . 213
33.1.1 al_get_allegro_primitives_version . . . 213
33.1.2 al_init_primitives_addon . . . 213
33.1.3 al_shutdown_primitives_addon . . . 213
33.2 High level drawing routines . . . 213
33.2.1 Pixel-precise output . . . 214
33.2.2 al_draw_line . . . 215
33.2.3 al_draw_triangle . . . 216
33.2.4 al_draw_filled_triangle . . . 216
33.2.5 al_draw_rectangle . . . 216
33.2.6 al_draw_filled_rectangle . . . 216
33.2.7 al_draw_rounded_rectangle . . . 217
33.2.8 al_draw_filled_rounded_rectangle . . . 217
33.2.9 al_calculate_arc . . . 217
33.2.10 al_draw_ellipse . . . 218
33.2.11 al_draw_filled_ellipse . . . 218
33.2.12 al_draw_circle . . . 218

33.2.13 al_draw_filled_circle . . . . . . . . . . 218
33.2.14 al_draw_arc . . . . . . . . . . 219
33.2.15 al_calculate_spline . . . . . . . . . . 219
33.2.16 al_draw_spline . . . . . . . . . . 219
33.2.17 al_calculate_ribbon . . . . . . . . . . 220
33.2.18 al_draw_ribbon . . . . . . . . . . 220
33.3 Low level drawing routines . . . . . . . . . . 220
33.3.1 al_draw_prim . . . . . . . . . . 221
33.3.2 al_draw_indexed_prim . . . . . . . . . . 221
33.3.3 al_create_vertex_decl . . . . . . . . . . 222
33.3.4 al_destroy_vertex_decl . . . . . . . . . . 222
33.3.5 al_draw_soft_triangle . . . . . . . . . . 222
33.3.6 al_draw_soft_line . . . . . . . . . . 223
33.4 Structures and types . . . . . . . . . . 223
33.4.1 ALLEGRO_VERTEX . . . . . . . . . . 223
33.4.2 ALLEGRO_VERTEX_DECL . . . . . . . . . . 224
33.4.3 ALLEGRO_VERTEX_ELEMENT . . . . . . . . . . 224
33.4.4 ALLEGRO_PRIM_TYPE . . . . . . . . . . 224
33.4.5 ALLEGRO_PRIM_ATTR . . . . . . . . . . 225
33.4.6 ALLEGRO_PRIM_STORAGE . . . . . . . . . . 225
33.4.7 ALLEGRO_VERTEX_CACHE_SIZE . . . . . . . . . . 225
33.4.8 ALLEGRO_PRIM_QUALITY . . . . . . . . . . 226

# 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.

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`

143agwindowl

al_create_displayYt



## 1.3 Initialisation

Before using Allegro you must call al_init

## 1.8 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.

## 1.9 Drawing primitives

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

## 1.10 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.

## 1.11 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

## 2.5 al_load_config_file_f

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

Read a configuration file from an already open file.

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

See also: al_load_config_file

## 2.6 al_save_config_file

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

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

See also: al_save_config_file_f, al_load_config_file

## 2.7 al_save_config_file_f

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

Write out a configuration file to an already open file.

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

See also: al_save_config_file

## 2.8 al_add_config_section

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

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

## 2.9 al_add_config_comment

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

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

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

See also: al_add_config_section

## 2.10 al_get_config_value

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

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

See also: al_set_config_value

## 2.11 al_set_config_value

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

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

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

See also: al_get_config_value

## 2.12 al_get_first_config_section

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

Returns the name of the first section in the given config file. Usually this will return an empty string for the global section. The `iterator` parameter will receive an opaque iterator which is used by
s278(values)-277.2 0.4T0.g.955 Tn wurn as, anyreceive anbuffer th only remain valid recetpasor long as the

## 2.14 al_get_first_config_entry

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

Returns the name of the first key in the given section in the given config. 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

## 2.15 al_get_next_config_entry

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

Returns the next key for the iterator obtained by al_get_first_config_entry.

## 2.16 al_merge_config

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

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

See also: al_merge_config_into

## 2.17 al_merge_config_into

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

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

See also: al_merge_config

# 3

# Display r utines

These functions are declared in the main Allegro header file:

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

## 3.1 Display creationn3(rgtions)-8(main)ope8(main)d000(Displin)oheader.ation

FIXME: document them all in detail

See also: al_set_new_display_flags

### 3.1.8 al_get_new_display_option

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

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

### 3.1.9 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.

### 3.1.10 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

ALLEGRNOFRAMEGL

ALLEGRO_OPEN_FORWARD_COMPATIBLE_GL ALLEGRDIRECT3DGL GENERATE_EXPOSE_EVENTSGL

ALLEGRO_OPEN_3_0GL Require the driver to dto provide

**ALLEGRO_OPENGL** Require the driver to provide an initialized Ope659 csucontextde Ope659 csurivsati59 csu3.0.GL

Require the driver to provide an initialized Ope659 csucontextde.GL

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

### 3.2.2 al_flip_display

```
void al_flip_display(void)
```

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

Several display options change how this function behaves:

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

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

If ALLEGRO_VSYNC is 1, this function will force waiting for vsync. If ALLEGRO_VSYNC is 2, this function will not wait for vsync. With many drivers the vsync behavior is controlled by the user and not the application, and ALLEGRO_VSYNC will not be set; in this case al_flip_display 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

### 3.2.3 al_get_backbuffer

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

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

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

### 3.2.4 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

### 3.2.5 al_get_display_format

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

Gets the pixel format of the display.

See also: ALLEGRO_PIXEL_FORMAT

### 3.2.6 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

### 3.2.7 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

### 3.2.8 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

### 3.2.9 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

### 3.2.10 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.

### 3.2.11 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

### 3.2.12 al_set_display_icon

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

Changes the icon associated with the display (window).

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

See also: al_set_window_title

### 3.2.13 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

### 3.2.14 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

### 3.2.15 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

### 3.2.16 al_toggle_display_flag

```
bool al_toggle_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_NOFRAME

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.

See also: al_set_new_display_flags, al_get_display_flags

### 3.2.17 al_update_display_region

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

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

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

See also: al_flip_display, al_get_display_option

### 3.2.18 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

### 3.2.19 al_get_display_event_source

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

Retrieve the associated event source.

## 3.3 Fullscreen display modes

### 3.3.1 ALLEGRO_DISPLAY_MODE

```
typedef struct ALLEGRO_DISPLAY_MODE
```

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

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

See also: al_get_display_mode

### 3.3.2 al_get_display_mode

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

Retrieves a display mode. Display parameters should not be changed between a call of al_get_num_display_modes 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

### 3.3.3 al_get_num_display_modes

```
int al_get_num_display_modes(void)
```

Get the number of available fullscreen display modes for the current set of display parameters. This will use the values set with al_set_new_display_refresh_rate, 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

## 3.4 Monitors

### 3.4.1 ALLEGRO_MONITOR_INFO

```
typedef struct ALLEGRO_MONITOR_INFO
```

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

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

See also: al_get_monitor_info

### 3.4.2 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.

See also: al_set_new_display_adapter

### 3.4.3 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`.

## 4.3 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.

## 4.4 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.

## 4.5 ALLEGRO_EVENT_JOYSTICK_CONFIGURATION

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

## 4.6 ALLEGRO_EVENT_KEY_DOWN

A keyboard key was pressed.

**keyboard.keycode (int)**

## 4.8 ALLEGRO_EVENT_KEY_CHAR

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

> 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.

## 4.10 ALLEGRO_EVENT_MOUSE_BUTTON_DOWN

A mouse button was pressed.

**mouse.x (int)**
x-coordinate

**mouse.y (int)**
y-coordinate

**mouse.z (int)**
z-coordinate

**mouse.w (int)**
w-coordinate

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

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

## 4.11 ALLEGRO_EVENT_MOUSE_BUTTON_UP

A mouse button was released.

**mouse.x (int)**
x-coordinate

**mouse.y (int)**
y-coordinate

**mouse.z (int)**
z-coordinate

**mouse.w (int)**
w-coordinate

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

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

## 4.12 ALLEGRO_EVENT_MOUSE_WARPED

al_set_mouse_xy was called to move the mouse. This event is identical to ALLEGRO_EVENT_MOUSE_AXES otherwise.

## 4.13 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.

## 4.14 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.

## 4.15 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.

## 4.16 ALLEGRO_EVENT_DISPLAY_EXPOSE

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

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

overhead when using this method. Second, create a mechanism in your game for easily reloading all of your bitmaps — for example, wrap them in a class or data structure and have a "bitmap manager" that can reload them back to the desired state. Then, when you receive an ALLEGRO_EVENT_DISPLAY_FOUND event, tell the bitmap manager (or whatever your mechanism is) to restore your bitmaps.

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

## 4.20 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.

## 4.21 ALLEGRO_EVENT_DISPLAY_SWITCH_OUT

The window is no longer active, that is the user might have click-278(no)n(to)-278ano(thre)-278(window)-2780or tabb
.

**display.source (ALLEGRO_DISPLAY *)**
: The display which wasnoouht of37(t.)]TJ/F37 11.9552 Tf -24.907 -35.609 Td [(4.21)-1000(ALLEGRO_EVENT_DI

**display.source (ALLEGRO_DISPLAY *)**
: The display which the even(.)]TJ/F37 9.9626 Tf -24.907 .200335 Td [(even((display)134(oriventaionw)]TJ

## 4.24 ALLEGRO_USER_EVENT

```
typedef struct ALLEGRO_USER_EVENT ALLEGRO_USER_EVENT;
```

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

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

See also: al_emit_user_event

## 4.25 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

## 4.26 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

## 4.27 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

## 4.28 ALLEGRO_GET_EVENT_TYPE

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

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

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

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

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

See also: ALLEGRO_EVENT, ALLEGRO_EVENT_TYPE_IS_USER

## 4.29 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.

## 4.30 al_create_event_queue

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

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

See also: ALLEGRO_EVENT_QUEUE

## 4.31 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.

```
type-278(the)-278(he-278(the)-278(head,)-2i)-278(event)-9.
```

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 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

## 4.36 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.

## 4.37 al_flush_event_queue

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

Drops all events, if any, from the queue.

## 4.38 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

## 4.39 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

## 4.40 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

## 4.41 al_register_event_source

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

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

See also: ALLEGRO_EVENT_QUEUE, ALLEGRO_EVENT_SOURCE

## 4.42 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

## 4.43 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

## 4.44 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.

## 4.45 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

## 4.46 al_wait_for_event_timed

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

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

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

See also: ALLEGRO_EVENT, al_wait_for_event, al_wait_for_event_until

## 4.47 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

# 5

# File I/O

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_interface5 Td [3t55 .(full)]TJ Tf-28.215 T278(the)_neweldctionsA structure pontersAin or

## 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_fclose

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

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

## 5.7 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

## 5.8 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

## 5.9 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_32le5.9

Returns theoc8(count)positi-278(on)in78(to)-278(the)-278(given)-278(ccurs,)-278(or)–1(true)-278(on)-278(error)1ise.

See also:, al_get_32le5.9

bool *f, size_t*f, *f,size)by

- 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

## 5.12 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

## 5.13 al_ferror

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

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

See also: al_feof, al_fclearerr

## 5.14 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

## 5.15 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.

al_fun00 l SQ [o,s_fun0.2 0.2 0.303rg 0.2 0.2 0.303RG [_errnol_fun00 l SQg 0 GBT/F37 1111(34.8431 763.8966Td [(5

## 5.25 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

## 5.26 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

## 5.27 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

## 5.28 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

## 5.29 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

## 5.30 Standard I/O specific routines

### 5.30.1 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

### 5.30.2 al_make_temp_file

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

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

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

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

Returns the opened ALLEGRO_FILE on success, NULL on failure.

# 6

# File system r utines

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).

## 6.1 ALLEGRO_FS_ENTRY

```
typedef struct ALLEGRO_FS_ENTRY ALLEGRO_FS_ENTRY;
```

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

## 6.4 al_destroy_fs_entry

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

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

Does nothing if passed NULL.

## 6.5 al_get_fs_entry_name

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

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

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

## 6.6 al_update_fs_entry

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

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

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

See also: al_get_errno, al_get_fs_entry_atime, al_get_fs_entry_ctime, al_get_fs_entry_mode

## 6.7 al_get_fs_entry_mode

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

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

See also: al_get_errno, ALLEGRO_FILE_MODE

## 6.8 al_get_fs_entry_atime

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

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

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

See also: al_get_fs_entry_ctime, al_get_fs_entry_mtime, al_update_fs_entry

## 6.15 al_remove_filename

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

Delete the given path from the filesystem, which may be a file or an empty directory. This is the same as al_remove_fs_entry, except it expects the  a398 w -278(the)8nc7h.ueturnxpectsetuerrno34(.)-345(Thise)-27lediven.D

```
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);
```

### 6.17.2 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:*e0.2dset_fs_interface*e face*e face

# Fi ed p int math r utines

These functions are declared in the main Allegro header file:

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

## 7.1 al_fixed

```
typedef int32_t al_fixed;
```

A fixed point number.

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

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

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

## 7.2 al_itofix

```
al_fixed al_oeatne o oi1fixed;
```

```
number = al_itofix(64   );

/* This check will fail in debug builds. */
assert(al_fixtoi(number) == 64   );
```

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

See also: al_fixtoi, al_ftofix, al_fixtof.

## 7.3 al_fixtoi

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

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

Example:

```
int result;

/* This will put 33 into 'result'. */
result = al_fixtoi(al_itofix(1  ) / 3);

/* But this will round up to 17. */
result = al_fixtoi(al_itofix(1  ) / 6);
```

See also: al_itofix, al_ftofix, al_fixtof, al_fixfloor, al_fixceil.

## 7.4 al_fixfloor

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

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

Example:

```
int result;

/* This will put 33 into 'result'. */
result = al_fixfloor(al_itofix(1  ) / 3);

/* And this will round down to 16. */
result = al_fixfloor(al_itofix(1  ) / 6);
```

See also: al_fixtoi, al_fixceil.

## 7.5 al_fixceil

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

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

Example:

```
int result;

/* This will put 34 into 'result'. */
result = al_fixceil(al_itofix(1  ) / 3);

/* This will round up to 17. */
result = al_fixceil(al_itofix(1  ) / 6);
```

See also: al_fixtoi, al_fixfloor.

## 7.6 al_ftofix

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

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

Example:

```
al_fixed number;

number = al_itofix(-4    );
assert(al_fixfloor(number) == -32768);

number = al_itofix(64   );
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

## 7.7 al_fixtof

```
double 51 51 51 a*33333x
```

## 7.8 al_fixmul

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

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

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

Example:

```
al_fixed result;

/* This will put 3     into 'result'. */
result = al_fixmul(al_itofix(1 ), al_itofix(3   ));

/* But this overflows, and sets errno. */
result = al_fixmul(al_itofix(1  ), al_itofix(3   ));
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.

## 7.9 al_fixdiv

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

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

Example:

```
al_fixed result;

/* This will put  . 6 6  'result'. */
result = al_fixdiv(al_itofix(2), al_itofix(33));

/* This will put   into 'result'. */
result = al_fixdiv( , al_itofix(-3 ));

/* Sets errno and puts -32768 into 'result'. */
result = al_fixdiv(al_itofix(-1  ), al_itofix( ));
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.

## 7.12 Fixed point trig

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

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

### 7.12.1 al_fixtorad_r

```
const al_fixed al_fixtorad_r = (al_fixed)16 8;
```

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:

```
a9.9626Tf27.o797e.632Td[(This)-277(constant)-277(gives)-276(nt)-27;]TJ/F432 (F32Td[/*(givesSerad_r)
```

```
al_fixed angle;
int result;

/* Set the binary angle to 9  degrees. */
angle = al_itofix(64);

/* The sine of 9  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.

### 7.12.4 al_fixcos

### 7.12.6 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  .24 5. */
angle = al_fixtof(al_fixmul(al_fixasin(al_ftofix( .238)), al_fixtorad_r));

/* This will trigger the assert. */
val = al_fixasin(al_ftofix(-1. 9));
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.

### 7.12.7 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.

### 7.12.8 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( .326));
```

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

### 7.12.9 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),  );

/* Sets 'result' to binary angle -1 9. */
result = al_fixatan2(al_itofix(-1), al_itofix(-2));

/* Fails the assert. */
result = al_fixatan2( ,  );
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.

### 7.12.10 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.

### 7.12.11 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.

### 8.1.5 al_map_rgba_f

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

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

See also: al_map_rgba, al_map_rgb, al_map_rgb_f

### 8.1.6 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

### 8.1.7 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

### 8.1.8 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

### 8.1.9 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

## 8.2 Locking and pixel formats

### 8.2.1 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.
- format

### 8.2.4 al_get_pixel_format_bits

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

### 8.3.5 al_destroy_bitmap

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

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

### 8.3.6 al_get_new_bitmap_flags

```
int al_get_new_bitmap_flags(void)
```

Returns the flags used for newly created bitmaps.

See also: al_set_new_bitmap_flags

### 8.3.7 al_get_new_bitmap_format

```
int al_get_new_bitmap_format(void)
```

Returns the format used for newly created bitmaps.

See also: ALLEGRO_PIXEL_FORMAT, al_set_new_bitmap_format

### 8.3.8 al_set_new_bitmap_flags

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

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

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

**ALLEGRO_MEMORY_BITMAP**
: Create a bitmap residing in system memory. Operations on, and with, memory bitmaps will not be hardware accelerated. However, direct pixel access can be relatively quick compared to video bitmaps, which depend on the display driver in use.

  Note: Allegro's software rendering routines are currently very unoptimised.

**ALLEGRO_KEEP_BITMAP_FORMAT**
: Only used when loading bitmaps from disk files, forces the resulting ALLEGRO_BITMAP to use the same format as the file.

  This is not yet honoured.

**ALLEGRO_FORCE_LOCKING**
: When drawing to a bitmap with this flag set, always use pixel locking and draw to it using Allegro's software drawing primitives. This should never be used if you plan to draw to the bitmap using Allegro's graphics primitives as it would cause severe performance penalties. However if you know that the bitmap will only ever be accessed by locking it, no unneeded FBOs will be created for it in the OpenGL drivers.

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

**ALLEGRO_ALPHA_TEST**
This is a driver hint only. It tells the graphics driver to do alpha testing instead of alpha blending on bitmaps created with this flag. Alpha testing is usually faster and preferred if your bitmaps have only one level of alpha (0). This flag is currently not widely implemented (i.e., only for memory bitmaps).

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

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

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

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

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

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

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

The reason for this can be seen in the Allegro example ex_premulalpha, ie, using pre-multiplied

8. G

### 8.4.7 al_is_compatible_bitmap

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

## 8.5. Drawing operations

> 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,

### 8.5.6 al_draw_pixel

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

Draws a single pixel at x, y. This function, unlike al_put_pixel, does blending and, unlike al_put_blended_pixel, respects the transformations. This function can be slow if called often; if you need to draw a lot of pixels considi97 -24.8lls considi97 -24.8lcprimitivblendingaddon. 0 G [(,)-278ddon. 0 ding

### 8.5.9 al_draw_scaled_rotated_bitmap

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

Like al_draw_rotated_bitmap, but can also scale the bitmap.

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

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

See also: al_draw_bitmap, al_draw_bitmap_region, al_draw_scaled_bitmap, al_draw_rotated_bitmap

### 8.5.10 al_draw_tinted_scaled_rotated_bitmap

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

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

See also: al_draw_tinted_bitmap

### 8.5.11 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 dw - destheighton width

### 8.5.16 al_set_target_bitmap

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

7.5.16 al_set_targetackbufferhp

8. G

### 8.6.4 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

## 8.7 Clipping

### 8.7.1 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

### 8.7.2 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

## 8.8 Graphics utility functions

### 8.8.1 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

## 8.9 Deferred drawing

### 8.9.1 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.

### 8.10.3 al_register_bitmap_loader_f

```
bool al_register_bitmap_loader_f(const char *extension,
```

b19.2281Tgech7.97-2795.445Td(8.1 11Tge)-2argument1 may1 b1 NULL1 to1 unrer_f) an1 4ntry.3.7 1

b19.2281Tg3.7 1-56.867T5 7.572ageeturns1 tru1 1 success,1 fals1 1 erro511112/0Returns1

### 8.10.7 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

### 8.10.8 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 Joystick routines

These functions are declared in the main Allegro header file:

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

## 9.1 ALLEGRO_JOYSTICK

```
typedef struct ALLEGRO_JOYSTICK ALLEGRO_JOYSTICK;
```

This is an abstract data type representing a physical joystick.

See also: al_get_joystick

```
typedef struct ALL-1TATEE struct ALL-1TATEEGRO_JOYSTICK;
```

ThisThisThis dtn Aold(This)-(This)“snapshot”(This)of(This)-(This)[-345(al′[(This)-xe[(This)-2d(This)butte)-278(funt(r

See also: al_tyate_get_joystick

## 9.4 al_install_joystick

```
bool al_install_joystick(void)
```

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

See also: al_uninstall_joystick

## 9.5 al_uninstall_joystick

```
void al_uninstall_joystick(void)
```

Uninstalls the active joystick driver. All outstanding ALLEGRO_JOYSTICK structures are invalidated. If no joystick driver was active, this function does nothing.

This function is automatically called when Allegro is shut down.

See also: al_install_joystick

## 9.6 al_is_joystick_installed

```
bool al_is_joystick_installed(void)
```

Returns true if al_install_joystick was called successfully.

## 9.7 al_reconfigure_joysticks

```
bool al_reconfigure_joysticks(void)
```

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

Afterwards, the number returned by al_get_num_joysticks may be difhonse, arH0void9g8(be)-27 is automaticahlc7 -11.

## 9.8 al_get_num_joysticks

```
int al_get_num_joysticks(void)
```

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

Returns 0 if there is no joystick driver installed.

See also: al_get_joystick, al_get_joystick_active

## 9.9 al_get_joystick

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

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

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

See also: al_get_num_joysticks, al_reconfigure_joysticks, al_get_joystick_active

## 9.10 al_release_joystick

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

This function currently does nothing.

See also: al_get_joystick

## 9.11 al_get_joystick_active

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

Return if the joystick handle is "active", i.e. in the current configuration, the handle represents some physical device plugged into the system. al_get_joystick returns a8(hand 0.2 0.4nked.)-345(A)190 G [- configuraected

# Keyboard routines

These functions are declared in the main Allegro header file:

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

## 10.1 ALLEGRO_KEYBOARD_STATE

```
typedef struct ALLEGRO_KEYBOARD_STATE ALLEGRO_KEYBOARD_STATE;
```

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

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

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

## 10.2 Key codes

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

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

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

```
ALLEGRO_KEY_A ... ALLEGRO_KEY_Z,
ALLEGRO_KEY_0 ... ALLEGRO_KEY_9,
ALLEGRO_KEY_PAD_0 ... ALLEGRO_KEY_PAD_9,
ALLEGRO_KEY_F1 ... ALLEGRO_KEY_F12,
ALLEGRO_KEY_ESCAPE,
ALLEGRO_KEY_TILDE,
ALLEGRO_KEY_MINUS,
ALLEGRO_KEY_EQUALS,
ALLEGRO_KEY_BACKSPACE,
ALLEGRO_KEY_TAB,
ALLEGRO_KEY_OPENBRACE, ALLEGRO_KEY_CLOSEBRACE,
```

```
ALLEGRO_KEY_ENTER,
ALLEGRO_KEY_SEMICOLON,
ALLEGRO_KEY_QUOTE,
ALLEGRO_KEY_BACKSLASH, ALLEGRO_KEY_BACKSLASH2,
ALLEGRO_KEY_COMMA,
ALLEGRO_KEY_FULLSTOP,
ALLEGRO_KEY_SLASH,
ALLEGRO_KEY_SPACE,
ALLEGRO_KEY_INSERT, ALLEGRO_KEY_DELETE,
ALLEGRO_KEY_HOME, ALLEGRO_KEY_END,
ALLEGRO_KEY_PGUP, ALLEGRO_KEY_PGDN,
ALLEGRO_KEY_LEFT, ALLEGRO_KEY_RIGHT,
ALLEGRO_KEY_UP, ALLEGRO_KEY_DOWN,
ALLEGRO_KEY_PAD_SLASH, ALLEGRO_KEY_PAD_ASTERISK,
ALLEGRO_KEY_PAD_MINUS, ALLEGRO_KEY_PAD_PLUS,
ALLEGRO_KEY_PAD_DELETE, ALLEGRO_KEY_PAD_ENTER,
ALLEGRO_KEY_PRINTSCREEN, ALLEGRO_KEY_PAUSE,
ALLEGRO_KEY_ABNT_C1, ALLEGRO_KEY_YEN, ALLEGRO_KEY_KANA,
ALLEGRO_KEY_CONVERT, ALLEGRO_KEY_NOCONVERT,
ALLEGRO_KEY_AT, ALLEGRO_KEY_CIRCUMFLEX,
ALLEGRO_KEY_COLON2, ALLEGRO_KEY_KANJI,
ALLEGRO_KEY_LSHIFT, ALLEGRO_KEY_RSHIFT,
ALLEGRO_KEY_LCTRL, ALLEGRO_KEY_RCTRL,
ALLEGRO_KEY_ALT, ALLEGRO_KEY_ALTGR,
ALLEGRO_KEY_LWIN, ALLEGRO_KEY_RWIN,
ALLEGRO_KEY_MENU,
ALLEGRO_KEY_SCROLLLOCK,
ALLEGRO_KEY_NUMLOCK,
ALLEGRO_KEY_CAPSLOCK
ALLEGRO_KEY_PAD_EQUALS,
ALLEGRO_KEY_BACKQUOTE,
ALLEGRO_KEY_SEMICOLON2,
ALLEGRO_KEY_COMMAND
```

## 10.3 Keyboard modifier flags

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

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

## 10.4 al_install_keyboard

```
bool al_install_keyboard(void)
```

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

See also: al_uninstall_keyboard, al_is_keyboard_installed

## 10.5 al_is_keyboard_installed

```
bool al_is_keyboard_installed(void)
```

Returns true if al_install_keyboard was called successfully.

## 10.6 al_uninstall_keyboard

```
void al_uninstall_keyboard(void)
```

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

This function is automatically called when Allegro is shut down.

See also: al_install_keyboard

## 10.7 al_get_keyboard_state

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

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

See also: al_key_down, ALLEGRO_KEYBOARD_STATE

## 10.8 al_key_down

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

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

See also: ALLEGRO_KEYBOARD_STATE

## 10.9 al_keycode_to_name

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

Converts the given keycode to a description of the key.

## 10.10 al_set_keyboard_leds

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

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

## 10.11 al_get_keyboard_event_source

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

Retrieve the keyboard event source.

Returns NULL if the keyboard subsystem was not installed.

# Memory management routines

These functions are declared in the main Allegro header file:

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

## 11.1 al_malloc

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

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

This is a macro.

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

## 11.2 al_free

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

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

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

This is a macro.

See also: al_malloc, al_free_with_context

## 11.3 al_realloc

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

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

This is a macro.

See also: al_malloc, al_realloc_with_context

## 11.9 ALLEGRO_MEMORY_INTERFACE

```
typedef struct ALLEGRO_MEMORY_INTERFACE ALLEGRO_MEMORY_INTERFACE;
```

This structure has the following fields.

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

See also: al_set_memory_interface

## 11.10 al_set_memory_interface

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

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

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

See also: ALLEGRO_MEMORY_INTERFACE

# 12 Miscellaneous routines

These functions are declared in the main Allegro header file:

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

## 12.1 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.

## 12.2 al_run_main

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

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

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

3

# M use r utines

These functions are declared in the main Allegro header file:

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

## 13.1 ALLEGRO_MOUSE_STATE

```
typedef struct ALLEGRO_MOUSE_STATE ALLEGRO_MOUSE_STATE;
```

Public fields (read only):

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

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

See also: al_get_mouse_state, al_get_mouse_state_axis, al_mouse_button_down

## 13.2 al_install_mouse

```
bool al_install_mouse(void)
```

Install a mouse driver.

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

## 13.3 al_is_mouse_installed

```
bool al_is_mouse_installed(void)
```

Returns true if al_install_mouse was called successfully.

## 13.4 al_uninstall_mouse

```
void al_uninstall_mouse(void)
```

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

This function is automatically called when Allegro is shut down.

## 13.5 al_get_mouse_num_axes

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

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

See also: al_get_mouse_num_buttons

## 13.6 al_get_mouse_num_buttons

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

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

See also: al_get_mouse_num_axes

## 13.7 al_get_mouse_state

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

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

Example:

```
ALLEGRO_MOUSE_STATE state;

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

See also: ALLEGRO_MOUSE_STATE, al_get_mouse_state_axis, al_mouse_button_down

## 13.8 al_get_mouse_state_axis

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

## 13.13 al_set_mouse_axis

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

Set the given mouse axis to the given value.

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

Returns true on success, false on failure.

See also: al_set_mouse_xy, al_set_mouse_z, al_set_mouse_w

## 13.14 al_get_mouse_event_source

```
ALLEGRO_EVENT_SOURCE *al_get_mouse_event_source(void)
```

Retrieve the mouse event source.

Returns NULL if the mouse subsystem was not installed.

## 13.15 Mouse cursors

### 13.15.1 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

### 13.15.2 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

### 13.15.3 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

# 14

# Path structures

These functions are declared in the main Allegro header file:

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

We define a path as an optional drive, followed by zero or more directory components, followed by an optional filename. The filename may be broken up into a basename and an extension, where the basename includes the start of the filename up to, but not including, the last dot (.) character. If no dot character exists the basename is the whole filename. The extension is everything from the last dot character to the end of the filename.

## 14.1 al_create_path

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

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

See also: al_create_path, al_destroy_path

## 14.2 al_create_path_for_directory

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

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

See also: al_create_path, al_destroy_path

## 14.3 al_destroy_path

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

Free a path structure. Does nothing if passed NULL.

See also: al_create_path, al_create_path_for_directory

## 14.4 al_clone_path

## 14.9 al_get_path_component

```
const char *al_get_path_component(const ALLEGRO_PATH *path, int i)
```

Return the i'th directory component of a path, counting from zero. If the index is negative then count from the right, i.e. -1 refers to the last path component. It is an error to pass an index which is out of bounds.

See also: al_get_path_num_components, al_get_path_tail

## 14.10 al_get_path_tail

```
const char *al_get_path_tail(const ALLEGRO_PATH *path)
```

Returns the last directory component, or NULL if there are no directory components.

## 14.11 al_get_path_filename

```
const char *al_get_path_filename(const ALLEGRO_PATH *path)
```

Return the filename part of the path, or the empty string if there is none.

The returned pointer is valid only until the filename part of the path is modified in any way, or until the path is destroyed.

See also: al_get_path_basename, al_get_path_extension, al_get_path_component

## 14.12 al_get_path_basename

```
const char *al_get_path_basename(const ALLEGRO_PATH *path)
```

Return the basename, i.e. filename with the extension removed. If the filename doesn't have an extension, the whole filename is the basename. If there is no filename part then the empty string is returned.

The returned pointer is valid only until the filename part of the path is modified in any way, or until the path is destroyed.

See also: al_get_path_filename, al_get_path_extension

## 14.13 al_get_path_extension

```
const char *al_get_path_extension(const ALLEGRO_PATH *path)
```

Return a pointer to the start of the extension of the filename, i.e. everything from the final dot ('.') character onwards. If no dot exists, returns an empty string.

The returned pointer is valid only until the filename part of the path is modified in any way, or until the path is destroyed.

See also: al_get_path_filename, al_get_path_basename

## 14.14 al_set_path_drive

```
void al_set_path_drive(ALLEGRO_PATH *path, const char *drive)
```

Set the drive string on a path. The drive may be NULL, which is equivalent to setting the drive string to the empty string.

See also: al_get_path_drive

## 14.15 al_append_path_component

```
void al_append_path_component(ALLEGRO_PATH *path, const char *s)
```

Append a directory component.

See also: al_insert_path_component

## 14.16 al_insert_path_component

```
void al_insert_path_component(ALLEGRO_PATH *path, int i, const char *s)
```

Insert a directory component at index i. If the index is negative then count from the right, i.e. -1 refers to the last path component.

It is an error to pass an index i which is not within these bounds: 0 <= i <= al_get_path_num_components(path).

See also: al_append_path_component,Td [(al_get_path_num_comg.)eplace5 RG [-345(al_append_path_component)]TJ

It is an error to pass index i

See also: al_insert_path_component,Td [(al_get_path_num_comg.)emove5 RG [-345(al_insert_path_component)]TJ0 g

It is an error to pass index i

See also: al_inserd_path_component,Td[(al_get_path_num_comg.)eplace5RG[-345(al_append_path_component)]TJ g

## 14.20 al_set_path_filename

```
void al_set_path_filename(ALLEGRO_PATH *path, const char *filename)
```

Set the optional filename part of the path. The filename may be NULL, which is equivalent to setting the filename to the empty string.

See also: al_set_path_extension, al_get_path_filename

## 14.21 al_set_path_extension

```
bool al_set_path_extension(ALLEGRO_PATH *path, char const *extension)
```

Replaces the extension of the path with the given one, i.e. replaces everything from the final dot ('.') character onwards, including the dot. If the filename of the path has no extension, the given one is appended. Usually the new extension you supply should include a leading dot.

Returns false if the path contains no filename part, i.e. the filename part is the empty string.

See also: al_set_path_filename, al_get_path_extension

## 14.22 al_path_cstr

```
const char *al_path_cstr(const ALLEGRO_PATH *path, char delim)
```

Convert a path to its string representation, i.e. optional drive, followed by directory components separated by 'delim', followed by an optional filename.

T8.93o 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.

## 14.23 al_make_path_canonical

```
bool al_make_path_canonical(ALLEGRO_PATH *path)
```

Removes any leading '..' directory components in absolute paths. Removes all '.' directory components.

Note that this does not collapse "x/../y" sections into "y". This is by design. If "/foo" on your system is a symlink to "/bar/baz", then "/foo/../quux" is actually "/bar/quux", not "/quux" as a naive removal of ".." components would give you.

5

# State

These functions are declared in the main Allegro header file:

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

## 15.1 ALLEGRO_STATE

```
typedef struct ALLEGRO_STATE ALLEGRO_STATE;
```

Opaque type which is passed to al_store_state/al_restore_state.

The various state kept internally by Allegro can be displayed like this:

```
global
    active system driver
        current config
per thread
    new bitmap params
    new display params
    active file interface
    errno
    current blending mode
    current display
        deferred drawing
    current target bitmap
        current transformation
        current clipping rectangle
        bitmap locking
```

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

## 15.2 ALLEGRO_STATE_FLAGS

```
typedef enum ALLEGRO_STATE_FLAGS
```

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

**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

## 16.7 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

## 16.8 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

## 16.9 al_get_app_name

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

Returns the global application name string.

See also: al_set_app_name

## 16.10 al_get_org_name

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

Returns the global organization name string.

See also: al_set_org_name

7

# 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>
```

## 17.1 ALLEGRO_THREAD

```
typedef struct ALLEGRO_THREAD ALLEGRO_THREAD;
```

An opaque structure representing a thread.

## 17.2 ALLEGRO_MUTEX

```
typedef struct ALLEGRO_MUTEX ALLEGRO_MUTEX;
```

An opaque structure representing a mutex.

## 17.3 ALLEGRO_COND

```
typedef struct ALLEGRO_COND ALLEGRO_COND;
```

An opaque structure representing a condition variable.

## 17.4 al_create_thread

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

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

Returns true if the thread was created, false if there was an error.

See also: al_start_thread, al_join_thread.

## 17.5 al_start_thread

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

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

Starting a thread which has already been started does nothing.

See also: al_create_thread.

## 17.6 al_join_thread

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

Wait for the thread to finish executing. This implicitly calls al_set_thread_should_stop first.

If `ret_value` is non-`NULL`, the value returned by the thread function will be stored at the location pointed to by `ret_value`.

See also: al_set_thread_should_stop, al_get_thread_should_stop, al_destroy_thread.

## 17.7 al_set_thread_should_stop

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

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

See also: al_join_thread, al_get_thread_should_stop.

## 17.8 al_get_thread_should_stop

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

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

Returns true if another thread has called al_join_thread or al_set_thread_should_stop on this thread.

See also: al_join_thread, al_set_thread_should_stop.

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

## 17.9 al_destroy_thread

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

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

Does nothing if `thread` is NULL.

See also: al_join_thread.

## 17.15 al_destroy_mutex

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

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

Does nothing if `mutex` is `NULL`.

## 17.16 al_create_cond

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

Create a condition variable.

Returns the condition value on success or `NULL` on error.

## 17.17 al_destroy_cond

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

Destroy a condition variable.

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

Does nothing if cond is `NULL`.

## 17.18 al_wait_cond

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

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

Example of proper use:

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

The mutex should be locked before checking the condition, and should be rechecked al_wait_cond returns. al_wait_cond can return for other reasons than the condition becoming true (e.g. the process was signalled). If multiple threads are blocked on the condition variable, the condition may no longer be true by the time the second and later threads are unblocked. Remember not to unlock the mutex prematurely.

See also: al_wait_cond_until, al_broadcast_cond, al_signal_cond.

## 17.19 al_wait_cond_until

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

Like al_wait_cond but the call can return if the absolute time passes `timeout` before the condition is signalled.

Returns zero on success, non-zero if the call timed out.

See also: al_wait_cond

## 17.20 al_broadcast_cond

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

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

See also: al_signal_cond.

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

## 17.21 al_signal_cond

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

Unblock at least one thread waiting on a condition variable.

Generally you should use al_broadcast_cond but al_signal_cond may be more efficient when it's applicable.

See also: al_broadcast_cond.

# Time r utines

These functions are declared in the main Allegro header file:

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

## 18.1 ALLEGRO_TIMEOUT

```
typedef struct ALLEGRO_TIMEOUT ALLEGRO_TIMEOUT;
```

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

See also: al_init_timeout

## 18.2 al_get_time

```
double al_get_time(void)
```

Return the number of seconds since the Allegro library was initialised. The return value is undefined if Allegro is uninitialised. The resolution depends on the used driver, but typically can be in the order of microseconds.

## 18.3 al_current_time

Alternate spelling of al_get_time.

## 18.4 al_init_timeout

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

Set timeout value of some number of seconds after the function call.

See also: ALLEGRO_TIMEOUT, al_wait_for_event_until

## 18.5 al_rest

```
void al_rest(double seconds)
```

Waits for the specified number seconds. This tells the system to pause the current thread for the given amount of time. With some operating systems, the accuracy can be in the order of 10ms. That is, even

```
al_rest( .      1)
```

might pause for something like 10ms. Also see the section on easier ways to time your program without using up all CPU.

## 19.12 al_set_timer_count

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

Set the timer's counter value. The timer can be started or stopped. The count value may be positive or negative, but will always be incremented by +1 at each tick.

See also: al_get_timer_count, al_add_timer_count

## 19.13 al_add_timer_count

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

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

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

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

See also: al_set_timer_count

## 19.14 al_get_timer_speed

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

Return the timer's speed, in seconds.

See also: al_set_timer_speed

## 19.15 al_set_timer_speed

```
void al_set_timer_speed(ALLEGRO_TIMER *timer, double new_speed_secs)
```

Set the timer's speed, i.e. the rate at which its counter will be incremented when it is started. This can be done when the timer is started or stopped. If the timer is currently running, it is made to look as though the speed change occured precisely at the last tick.

speed_secs has exactly the same meaning as with al_create_timer.

See also: al_get_timer_speed

## 19.16 al_get_timer_event_source

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

Retrieve the associated event source.

2

# Transformations

These functions are declared in the main Allegro header file:

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

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

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

```
al_identity_transform(&T);

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

The resultant matrix multiplication expression will look like this:

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

Since the point coordinate vector term will go on the right of that sequency 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, 1  ,   );
al_use_transform(T);
```

## 20.1 ALLEGRO_TRANSFORM

```
typedef struct ALLEGRO_TRANSFORM ALLEGRO_TRANSFORM;
```

Defines the generic transformation type, a 4x4 matrix. 2D transforms use only a small subsection of this matrix, namely the top left 2x2 matrix, and the right most 2x1 matrix, for a total of 6 values.

Fields:

- m - A 4x4 float matrix

## 20.2 al_copy_transform

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

Makes a copy of a transformation.

Parameters:

- dest - Source transformation
- src - Destination transformation

## 20.3 al_use_transform

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

Sets the transformation to be used for the the drawing operations on the target bitmap (each bitmap maintains its own transformation). Every drawing operation after this call will be transformed using this transformation. Call this function with an identity transformation to return to the default behaviour.

This function does nothing if there is no target bitmap.

Parameters:

- trans - Transformation to use

## 20.4 al_get_current_transform

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

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

Returns: A pointer to the current transformation.

## 20.5 al_invert_transform

```
void al_invert_transform(ALLEGRO_TRANSFORM *trans)
```

## 20.8 al_build_transform

```
void al_build_transform(ALLEGRO_TRANSFORM *trans, float x, float y,
```

2

# UTF–8 string r utines

These functions are declared in the main Allegro header file:

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

## 21.1 About UTF–8 string routines

Some parts of the Allegro API, such as the font rountines, expect Unicode strings encoded in UTF–8. These basic routines are provided to help you work with UTF–8 strings. You should use another library (e.g. ICU) if you require more functionality.

You should also see elsewhere for an introduction to Unicode. Extremely briefly, Unicode is a standard consisting of a large character set (of over 100,000 characters), and rules, such as how to sort strings. A code point is the integer value of a character, but not all code points are characters, as some code points have other uses. Clearly it is impossible represent each code point with a 8-bit byte or even a 16-bit integer, so there exist different Unicode Transformation Formats. UTF–8 has many nice properties, but the main advantages are that it is backwards compatible with C strings, and ASCII characters (code points <= 127) are encoded in UTF–8 exactly as they would be in ASCII.

Here is a diagram of the representation of the word "ål", with a NUL terminator.

```
                  ---------------- ---------------- --------------
          String         å                l               NUL
                  ---------------- ---------------- --------------
     Code points    U+  E5 (229)     U+  6C (1 8)     U+     ( )
                  ---------------- ---------------- --------------
  UTF-8 encoding      xC3,  xA5           x6C              x
                  ---------------- ---------------- --------------
UTF-16LE encoding     xE5,  x          x6C,  x          x  ,  x
                  ---------------- ---------------- --------------
```

U+00E5 is greater than 127 so requires two bytes to represent in UTF–8. U+006C and U+0000 both exist in the ASCII range, so take one byte each, exactly as in an ASCII string. UTF–16 is a different encoding, in which each code is represented by two or four bytes. In UTF–8 a zero byte is only present when it represents the NUL character, but this is not true for UTF–16.

In the Allegro API, be careful whether a function takes byte offsets or code-point indices. In general, all position parameters are in byte offsets, not code point indices. This may be surprising, but if you think about, is required for good performance. It also means many functions will work even if they do not contain UTF–8, so you may actually store arbitrary data in the strings.

21. UTF-8

### 21.3.4 al_ustr_free

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

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

### 21.3.5 al_cstr

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

Get a `char *` pointer to the data in a string. This pointer will only be valid while the underlying string is not modified and not destroyed. The pointer may be passed to functions expecting C-style strings, with the following caveats:

- ALLEGRO_USTRs are allowed to contain embedded NUL ('\0') bytes. That means `al_ustr_size(u)` and `strlen(al_cstr(u))` may not agree.
- An ALLEGRO_USTR may be created in such a way that it is not NUL terminated. A string which is dynamically allocated will always be NUL terminated, but a string which references the middle of another string or region of memory will not be NUL terminated.
- If the ALLEGRO_USTR references another string, the returned c-string will point into the referenced string, the length of the string will be ignored.

See also: al_ustr_to_buffer, al_cstr_dup

### 21.3.6 al_ustr_to_buffer

```
void al_ustr_to_buffer(const ALLEGRO_USTR *us, char *buffer, int size)
```

Write the contents of the string into a pre-allocated buffer of the given size in bytes. The result will always be 0-terminated.

See also: al_cstr, al_cstr_dup

### 21.3.7 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

### 21.3.8 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

### 21.5.3 al_ref_ustr

```
ALLEGRO_USTR *al_ref_ustr(ALLEGRO_USTR_INFO *info, const ALLEGRO_USTR *us,
   int start_pos, int end_pos)
```

Create a read-only string that references the storage of another string. The information about the string (e.g. its size) is stored in the structure pointed to by the `info` parameter. The string will not have any other storage allocated of its own, so if you allocate the `info` structure on the stack then no explicit "free" operation is required.

The referenced interval is [start_pos, end_pos).

The string is valid until the underlying string is modified or destroyed.

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

See also: al_ref_cstr, al_ref_buffer

## 21.6 Sizes and offsets

### 21.6.1 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

### 21.6.2 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

### 21.6.3 al_ustr_offset

```
int al_ustr_offset(const ALLEGRO_USTR *us, int index)
```

Return the offset (in bytes from the start of the string) of the code point at the specified index in the string. A zero index parameter will return the first character of the string. If index is negative, it counts backward from the end of the string, so an index of -1 will return an offset to the last code point.

If the index is past the end of the string, returns the offset of the end of the string.

See also: al_ustr_length

### 21.6.4 al_ustr_next

```
bool al_ustr_next(const ALLEGRO_USTR *us, int *pos)
```

21. UTF-8

### 21.7.3 al_ustr_prev_get

```
int32_t al_ustr_prev_get(const ALLEGRO_USTR *us, int *pos)
```

Find the beginning of a code point before *pos, then return it. Note this performs a pre-increment.

On success returns the code point value. If pos was out of bounds (e.g. past the end of the string), return -1. On an error, such as an invalid byte sequence, return -2. As with al_ustr_prev, invalid byte sequences may be skipped while advancing.

See also: al_ustr_get_next

## 21.8 Inserting into strings

### 21.8.1 al_ustr_insert

```
bool al_ustr_insert(ALLEGRO_USTR *us1, int pos, const ALLEGRO_USTR *us2)
```

Insert us2 into us1 beginning at pos. pos cannot be less thote

Inspepos al_offset_prevFind the bytep78(of)-278(a)-278--278(s(byte)offset_ [(.)]T

### 21.9.2 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

### 21.9.3 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

### 21.9.4 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

### 21.9.5 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

## 21.10 Removing parts of strings

### 21.10.1 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

### 21.10.2 al_ustr_remove_range

```
bool al_ustr_remove_range(ALLEGRO_USTR *us, int start_pos, int end_pos)
```

Remove the interval [start_pos, end_pos) (in bytes) from a string. `start_pos` and `end_pos` may both be past the end of the string but cannot be less than 0 (the start of the string).

Returns true on success, false on error.

See also: al_ustr_remove_chr, al_ustr_truncate

### 21.10.3 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

### 21.10.4 al_ustr_ltrim_ws

```
bool al_ustr_ltrim_ws(ALLEGRO_USTR *us)
```

Remove leading whitespace characters from a string, as defined by the C function `isspace()`.

Returns true on success, or false if the function was passed an empty string.

See also: al_ustr_rtrim_ws, al_ustr_trim_ws

### 21.10.5 al_ustr_rtrim_ws

```
bool al_ustr_rtrim_ws(ALLEGRO_USTR *us)
```

Remove trailing ("right") whitespace characters from a string, as defined by the C function `isspace()`.

Returns true on success, or false if the function was passed an empty string.

See also: al_ustr_ltrim_ws, al_ustr_trim_ws

### 21.10.6 al_ustr_trim_ws

```
bool al_ustr_trim_ws(ALLEGRO_USTR *us)
```

Remove both leading and trailing whitespace characters from a string.

Returns true on success, or false if the function was passed an empty string.

See also: al_ustr_ltrim_ws, al_ustr_rtrim_ws

## 21.11 Assigning one string to another

### 21.11.1 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

### 21.11.2 al_ustr_assign_substr

```
bool al_ustr_assign_substr(ALLEGRO_USTR *us1, const ALLEGRO_USTR *us2,
   int start_pos, int end_pos)
```

Overwrite the string us1 with the contents of us2 in the byte interval [start_pos, end_pos). The end points will be clamed to the bounds of us2.

Usually you will first have to use al_ustr_offset to find the byte offsets.

Returns true on success, false on error.

See also: al_ustr_assign, al_ustr_assign_cstr

### 21.11.3 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

## 21.12 Replacing parts of string

### 21.12.1 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

### 21.12.2 al_ustr_replace_range

```
bool al_ustr_replace_range(ALLEGRO_USTR *us1, int start_pos1, int end_pos1,
   const ALLEGRO_USTR *us2)
```

### 21.13.6 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`.

### 21.13.7 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`

### 21.13.12 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`.

## 21.14 Comparing

### 21.14.1 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

### 21.14.2 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

### 21.14.3 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.

### 21.14.4 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

### 21.14.5 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

## 21.16 Low-level UTF–8 routines

### 21.16.1 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 [-3qt.2 0.2 0.45 RG [-3497 -28.29(ow-levr-24.367 Td [(22.16.1)-1000(al_utf8_encodh)]TJ

### al [-3qt.2 0.h

```
size_t al[-3qt.width(int c)
```

Returns the number of bytes that would be occupied by the specified code point when encoded in UTF21.68. This is78(is)278(be)-r8(and)-278(4)-278(bytes)-278(for)-278(legal)-278(code)-278(point)-278(values.)-34

22

# Platform-specific functions

## 22.1 Windows

These functions are declared in the following header file:

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

### 22.1.1 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.

## 22.2 iPhone

These functions are declared in the following header file:

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

### 22.2.1 al_iphone_program_has_halted

```
void al_iphone_program_has_halted(void)
```

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

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.

# 24

# OpenGL integration

These functions are declared in the following header file:

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

## 24.1 al_get_opengl_extension_list

```
ALLEGRO_OGL_EXT_LIST *al_get_opengl_extension_list(void)
```

Returns the list of OpenGL extensions supported by Allegro, for the given display.

Allegro will keep information about all extensions it knows about in a structure returned by `al_get_opengl_extension_list`.

For example:

```
if (al_get_opengl_extension_list()->ALLEGRO_GL_ARB_multitexture) {
    use it
}
```

The extension will be set to true if available for the given display and false otherwise. This means to use the definitions and functions from an OpenGL extension, all you need to do is to check for it as above at run time, after acquiring the OpenGL display from Allegro.

Under Windows, this will also work with WGL extensions, and under Unix with GLX extensions.

In case you want to manually check for extensions and load function pointers yourself (say, in case the Allegro developers did not include it yet), you can use the al_have_opengl_extension and al_get_opengl_proc_address functions instead.

## 24.2 al_get_opengl_proc_address

```
void *al_get_opengl_proc_address(const char *name)
```

Helper to get the address of an OpenGL symbol

Example:

How to get the function glMultiTexCoord3fARB that comes with ARB's Multitexture extension:

```
// define the type of the function
   ALLEGRO_DEFINE_PROC_TYPE(void, MULTI_TEX_FUNC,
      (GLenum, GLfloat, GLfloat, GLfloat));
// declare the function pointer
   MULTI_TEX_FUNC glMultiTexCoord3fARB;
// get the address of the function
   glMultiTexCoord3fARB = (MULTI_TEX_FUNC) al_get_opengl_proc_address(
      "glMultiTexCoord3fARB");
```

If glMultiTexCoord3fARB is not NULL then it can be used as if it has been defined in the OpenGL core library.

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

Parameters:

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

Return value:

A pointer to the symbol if available or NULL otherwise.

## 24.3 al_get_opengl_texture

```
GLuint al_get_opengl_texture(ALLEGRO_BITMAP *bitmap)
```

Returns the OpenGL texture id internally used by the given bitmap if it uses one, else 0.

Example:

```
bitmap = al_load_bitmap("my_texture.png");
texture = al_get_opengl_texture(bitmap);
if (texture != )
    glBindTexture(GL_TEXTURE_2D, texture);
```

## 24.4 al_get_opengl_texture_size

```
void al_get_opengl_texture_size(ALLEGRO_BITMAP *bitmap, int *w, int *h)
```

Retrieves the size of the texture used for the bitmap. This can be different from the bitmap size if OpenGL only supports power-of-two sizes or if it is a sub-bitmap.

## 24.5 al_get_opengl_texture_position

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

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

## 24.6 al_get_opengl_fbo

```
GLuint al_get_opengl_fbo(ALLEGRO_BITMAP *bitmap)
```

Returns the OpenGL FBO id internally used by the given bitmap if it uses one, else 0. An FBO is created for a bitmap when you call al_set_target_bitmap for it.

## 24.7 al_remove_opengl_fbo

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

If the bitmap has an OpenGL FBO created for it (see al_set_target_bitmap), it is freed. It also is freed automatically when the bitmap is destroyed.

## 24.8 al_have_opengl_extension

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

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

Example:

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

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

Returns true if the extension is available; false otherwise.

## 24.9 al_get_opengl_version

```
uint32_t al_get_opengl_version(void)
```

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

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

See also: al_get_opengl_variant

# 25

# Audi add n

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

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

## 25.1 Audio types

### 25.1.1 ALLEGRO_AUDIO_DEPTH

```
enum ALLEGRO_AUDIO_DEPTH
```

Sample depth and type, and signedness. Mixers only use 32-bit signed float (–1..+1), or 16-bit signed integers. The unsigned value is a bit-flag applied to the depth value.

- ALLEGRO_AUDIO_DEPTH_INT8
- ALLEGRO_AUDIO_DEPTH_INT16
- ALLEGRO_AUDIO_DEPTH_INT24
- ALLEGRO_AUDIO_DEPTH_FLOAT32
- ALLEGRO_AUDIO_DEPTH_UNSIGNED

For convenience:

- ALLEGRO_AUDIO_DEPTH_UINT8
- ALLEGRO_AUDIO_DEPTH_UINT16
- ALLEGRO_AUDIO_DEPTH_UINT24

### 25.1.2 ALLEGRO_AUDIO_PAN_NONE

```
#define ALLEGRO_AUDIO_PAN_NONE      (-1   . f)
```

Special value for the ALLEGRO_AUDIOPROP_PAN property. Use this value to play samples at their original volume with panning disabled.

### 25.1.3 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

### 25.1.4 ALLEGRO_MIXER

```
typedef struct ALLEGRO_MIXER ALLEGRO_MIXER;
```

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

### 25.1.5 ALLEGRO_MIXER_QUALITY

```
enum ALLEGRO_MIXER_QUALITY
```

- ALLEGRO_MIXER_QUALITY_POINT - point sampling
- ALLEGRO_MIXER_QUALITY_LINEAR - linear interpolation

### 25.1.6 ALLEGRO_PLAYMODE

```
enum ALLEGRO_PLAYMODE
```

Sample and stream playback mode.

- ALLEGRO_PLAYMODE_ONCE
- ALLEGRO_PLAYMODE_LOOP
- ALLEGRO_PLAYMODE_BIDIR

### 25.1.7 ALLEGRO_SAMPLE_ID

```
typedef struct ALLEGRO_SAMPLE_ID ALLEGRO_SAMPLE_ID;
```

#### 25.1.11 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.

### 25.2 Setting up audio

#### 25.2.1 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

#### 25.2.2 al_uninstall_audio

```
void al_uninstall_audio(void)
```

Uninstalls the audio subsystem.

See also: al_install_audio

#### 25.2.3 al_is_audio_installed

```
bool al_is_audio_installed(void)
```

Returns true if al_install_audio was called previously and returned successfully.

#### 25.2.4 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

25.3. Misc audio

# 25. AUDIO ADDON

### 25.4.10 al_get_voice_playing

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

Return true if the voice is currently playing.

### 25.4.11 al_set_voice_playing

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

Change whether a voice is playing or not. The voice must have a sample or mixer attached to it.

Returns true on success, false on failure.

### 25.4.12 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.

### 25.4.13 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.

Returns true on success, false on failure.

See also: al_get_voice_position.

## 25.5 Sample functions

### 25.5.1 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:

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

See also: al_destroy_sample, ALLEGRO_AUDIO_DEPTH, ALLEGRO_CHANNEL_CONF

### 25.5.2 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 withbufferbuencefreed theFree

### 25.5.6 al_get_sample_channels

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

Return the channel configuration.

See also: ALLEGRO_CHANNEL_CONF.

### 25.5.7 al_get_sample_depth

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

Return the audio depth.

See also: ALLEGRO_AUDIO_DEPTH.

### 25.5.8 al_get_sample_frequency

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

Return the frequency of the sample.

### 25.5.9 al_get_sample_length

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

Return the length of the sample in sample values.

### 25.5.10 al_get_sample_data

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

Return a pointer to the raw sample data.

## 25.6 Sample instance functions

### 25.6.1 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.

### 25.6.2 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!).

### 25.6.3 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.

### 25.6.4 al_stop_sample_instance

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

Stop an sample instance playing.

### 25.6.5 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.

### 25.6.6 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.

### 25.6.7 al_get_sample_instance_frequency

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

Return the frequency of the sample instance.

### 25.6.8 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.

### 25.6.9 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.

### 25.6.10 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.

### 25.6.11 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.

### 25.6.12 al_get_sample_instance_speed

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

Return the playback speed.

### 25.6.13 al_set_sample_instance_speed

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

Set the playback speed.

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

### 25.6.14 al_get_sample_instance_gain

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

Return the playback gain.

### 25.6.15 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.

### 25.6.16 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.

### 25.6.17 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 constant sound power level is maintained as the sample is panned from left to right. As a consequence, a pan value of 0.0 will play the sample 3 dB softer than the original level. To disable panning and play a sample at its original level, set the pan value to ALLEGRO_AUDIO_PAN_NONE.

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

(A sound guy should explain that better; I only implemented it. Also this might be more properly called a balance control than pan. Also we don't attempt anything with more than two channels yet.)

See also: al_get_sample_instance_pan.

### 25.6.18 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.

### 25.6.19 al_get_sample_instance_playmode

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

Return the playback mode.

### 25.6.20 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.

### 25.6.21 al_get_sample_instance_playing

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

Return true if the sample instance is playing.

### 25.6.22 al_set_sample_instance_playing

```
bool al_set_sample_instance_playing(ALLEGRO_SAMPLE_INSTANCE *spl, bool val)
```

Change whether the sample instance is playing.

Returns true on success, false on failure.

### 25.6.23 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.

### 25.6.24 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.

### 25.6.25 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.

### 25.6.26 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.

## 25.7 Mixer functions

### 25.7.1 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

UD10 ADDON attach audio stream to mixer

### 25.7.19 al_set_mixer_postprocess_callback

```
bool al_set_mixer_postprocess_callback(ALLEGRO_MIXER *mixer,
   void (*pp_callback)(void *buf, unsigned int samples, void *data),
   void *pp_callback_userdata)
```

Sets a post-processing filter function that's called after the attached streams have been mixed. The buffer's format will be whatever the mixer was created with. The sample count and user-data pointer is also passed.

## 25.8 Stream functions

### 25.8.1 al_create_audio_stream

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

Creates an ALLEGRO_AUDIO_STREAM. The stream will be set to play by default. It will feed audio

## 25. AUDIO ADDON

### 25.8.8 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.

### 25.8.9 al_get_audio_stream_length

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

Return the stream length in samples.

### 25.8.10 al_get_audio_stream_speed

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

Return the playback speed.

See also: al_set_audio_stream_speed.

### 25.8.11 al_set_audio_stream_speed

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

Set the playback speed.

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

See also: al_get_audio_stream_speed.

### 25.8.12 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.

### 25.8.13 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.

### 25.8.14 al_get_audio_stream_pan

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

Get the pan value.

See also:

### 25.8.20 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.

### 25.8.21 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.

### 25.8.22 al_get_audio_stream_fragment

```
void *al_get_audio_stream_fragment(const ALLEGRO_AUDIO_STREAM *stream)
```

When using Allegro's audio streaming, you will use this function to continuously provide new sample data to a stream.

If the stream is ready for new data, the function will return the address of an internal buffer to be filled with audio data. The length and format of the buffer are specified with al_create_audio_stream 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

### 25.8.23 al_set_audio_stream_fragment

```
bool al_set_audio_stream_fragment(ALLEGRO_AUDIO_STREAM *stream, void *val)
```

This function needs to be called for every successful call of al_get_audio_stream_fragment to indicate that the buffer is filled with new data.

### 25.8.24 al_get_audio_stream_fragments

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

Returns the number of fragments this stream uses. This is the same value as passed to al_create_audio_stream when a new stream is created.

### 25.8.25 al_get_available_audio_stream_fragments

```
unsigned int al_get_available_audio_stream_fragments(
```

### 25.9.5 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

### 25.9.6 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

### 25.9.7 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

### 25.9.8 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

### 25.9.9 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

### 25.9.10 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

# 27

# C l r add n

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

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

## 27.1 al_color_cmyk

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

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

See also: al_color_cmyk_to_rgb, al_color_rgb_to_cmyk

## 27.2 al_color_cmyk_to_rgb

```
void al_color_cmyk_to_rgb(float cyan, float magenta, float yellow,
    float key, float *red, float *green, float *blue)
```

Convert CMYK values to RGB values.

See also: al_color_cmyk, al_color_rgb_to_cmyk

## 27.3 al_color_hsl

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

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

See also: al_color_hsl_to_rgb, al_color_hsv

## 27.4 al_color_hsl_to_rgb

```
void al_color_hsl_to_rgb(float hue, float saturation, float lightness,
   float *red, float *green, float *blue)
```

Convert values in HSL color model to RGB color model.

Parameters:

- hue - Color hue angle in the range 0..360.

- saturation - Color saturation in the range 0..1.
- lightness - Color lightness in the range 0..1.
- red, green, blue - returned RGB values in the range 0..1.

See also: al_color_rgb_to_hsl, al_color_hsl, al_color_hsv_to_rgb

## 27.5 al_color_hsv

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

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

See also: al_color_hsv_to_rgb, al_color_hsl

## 27.6 al_color_hsv_to_rgb

```
void al_color_hsv_to_rgb(float hue, float saturation, float value,
   float *red, float *green, float *blue)
```

Convert values in HSV color model to RGB color model.

Parameters:

- hue - Color hue angle in the range 0..360.
- saturation - Color saturation in the range 0..1.
- value - Color value in the range 0..1.
- red, green, blue - returned RGB values in the range 0..1.

See also: al_color_rgb_to_hsv, al_color_hsv, al_color_hsl_to_rgb

## 27.7 al_color_html

```
ALLEGRO_COLOR al_color_html(char const *string)
```

Interprets an HTML styled hex number (e.g. #00faff) as a color. Components that are malformed are set to 0.

See also: al_color_html_to_rgb, al_color_rgb_to_html

## 27.8 al_color_html_to_rgb

```
void al_color_html_to_rgb(char const *string,
   float *red, float *green, float *blue)
```

Interprets an HTML styled hex number (e.g. #00faff) as a color. Components that are malformed are set to 0.

See also: al_color_html, al_color_rgb_to_html

mediumseagreen, mediumslateblue, mediumspringgreen, mediumturquoise,
mediumvioletred, midnightblue, mintcream, mistyrose, moccasin, avajowhite, navy,
oldlace, olive, olivedrab, orange, orangered, orchid, palegoldenrod, palegreen,
paleturquoise, palevioletred, papayawhip, peachpuff, peru, pink, plum, posin, palevioletred,palevioletred, medium

## 27.15 al_color_rgb_to_name

```
char const *al_color_rgb_to_name(float r, float g, float b)
```

Given an RGB triplet with components in the range 0..1, find a color name describing it approximately.

See also: al_color_name_to_rgb, al_color_name

## 27.16 al_color_rgb_to_yuv

```
void al_color_rgb_to_yuv(float red, float green, float blue,
   float *y, float *u, float *v)
```

Convert RGB values to YUV color space.

See also: al_color_yuv, al_color_yuv_to_rgb

## 27.17 al_color_yuv

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

Return an ALLEGRO_COLOR structure from YUV values.

See also: al_color_yuv_to_rgb, al_color_rgb_to_yuv

## 27.18 al_color_yuv_to_rgb

```
void al_color_yuv_to_rgb(float y, float u, float v,
    float *red, float *green, float *blue)
```

Convert YUV color values to RGB color space.

See also: al_color_yuv, al_color_rgb_to_yuv

## 27.19 al_get_allegro_color_version

```
uint32_t al_get_allegro_color_version(void)
```

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

2

# F nt add ns

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

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

## 28.1 General font routines

### 28.1.1 ALLEGRO_FONT

```
typedef struct ALLEGRO_FONT ALLEGRO_FONT;
```

A handle identifying any kind of font. Usually you will create it with al_load_font 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.

### 28.1.2 al_init_font_addon

```
void al_init_font_addon(void)
```

Initialise the font addon.

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

See also: al_init_image_addon, al_init_ttf_addon, al_shutdown_font_addon

### 28.1.3 al_shutdown_font_addon

```
void al_shutdown_font_addon(void)
```

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

See also: al_init_font_addon

28. F

### 28.1.8 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

### 28.1.9 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

### 28.1.10 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

### 28.1.11 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

### 28.1.12 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 0-terminated string `text` onto `bmp` at position `x`, `y`, using the specified `font`.

The `flags` parameter can be 0 or one of the following flags:

- ALLEGRO_ALIGN_LEFT - Draw the text left-aligned (same as 0).
- ALLEGRO_ALIGN_CENTRE - Draw the text centered around the given position.
- ALLEGRO_ALIGN_RIGHT - Draw the text right-aligned to the given position.

See also: al_draw_ustr, al_draw_textf, al_draw_justified_text

### 28.1.13 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 0-terminated char array.

See also: al_draw_text, al_draw_justified_ustr

### 28.1.14 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 specified area.

See also: al_draw_justified_textf, al_draw_justified_ustr

### 28.1.15 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_ustr, but justifies the string to the specified area.

See also: al_draw_justified_text, al_draw_justified_textf.

### 28.1.16 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

### 28.1.17 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, ...)
```

Like al_draw_justified_text and al_draw_textf.

See also: al_draw_justified_text, al_draw_justified_ustr.

### 28.1.18 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

### 28.1.19 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

### 28.1.20 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.

## 28.2 Bitmap fonts

### 28.2.1 al_grab_font_from_bitmap

```
ALLEGRO_FONT *al_grab_font_from_bitmap(ALLEGRO_BITMAP *bmp,
   int ranges_n, int ranges[])
```

Creates a new font from an Allegro bitmap. You can delete the bitmap after the function returns as the font will contain a copy for itself.

Parameters:

- bmp: The bitmap with the glyphs drawn onto it
- n: Number of unicode ranges in the bitmap.
- ranges: 'n' pairs of first and last unicode point to map glyphs to for each range.

The bitmap format is as in the following example, which contains three glyphs for 1, 2 and 3.

```
.............
. 1 .222.333.
. 1 .  2.  3.
. 1 .222.333.
. 1 .2  .  3.
. 1 .222.333.
.............
```

In the above illustration, the dot is for pixels having the background color. It is determined by the color of the top left pixel in the bitmap. There should be a border of at least 1 pixel with this color to the bitmap edge and between all glyphs.

Each glyph is inside a rectangle of pixels not containing the background color. The height of all glyph rectangles should be the same, but the width can vary.

The placement of the rectangles does not matter, except that glyphs are scanned from left to right and top to bottom to match them to the specified unicode codepoints.

The glyphs will simply be drawn using al_draw_bitmap, 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(/nt int(n32,)-,[(hn/F255pass8(the)1th)e)odeT2Tf18. 23Td[(int)-5 (ranges[])-5 (=)-12
```

### 28.3.1 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.

### 28.3.2 al_shutdown_ttf_addon

```
void al_shutdown_ttf_addon(void)
```

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

### 28.3.3 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 pixel. The standard font size is measured in units per EM, if you instead want to specify the size as the total height of glyphs in pixel, pass it as a negative value.

> Note: If you want to display text at multiple sizes, load the font multiple times with different size parameters.

The 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)

See also: al_init_ttf_addon, al_load_ttf_font_f

### 28.3.4 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.

### 28.3.5 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.

# 29

# Image I/O addon

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

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

## 29.1 al_init_image_addon

```
bool al_init_image_addon(void)
```

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

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

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

## 29.2 al_shutdown_image_addon

```
void al_shutdown_image_addon(void)
```

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

## 29.3 al_get_allegro_image_version

```
uint32_t al_get_allegro_image_version(void)
```

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

3

# 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>
```

## 30.1 al_open_memfile

```
ALLEGRO_FILE *al_open_memfile(void *mem, int64_t size, const char *mode)
```

Returns a file handle to the block of memory. All read and write operations act upon the memory directly, so it must not be freed while the file remains open.

The mode can be any combination of "r" (readable) and "w" (writable). Regardless of the mode, the file always opens at position 0. The file size is fixed and cannot be expanded.

It should be closed with al_fclose. After the file is closed, you are responsible for freeing the memory (if needed).

## 30.2 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.

# 3

# Native dialogs support

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

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

## 31.1 ALLEGRO_FILECHOOSER

```
typedef struct ALLEGRO_FILECHOOSER ALLEGRO_FILECHOOSER;
```

Opaque handle to a native file dialog.

## 31.2 ALLEGRO_TEXTLOG

```
typedef struct ALLEGRO_TEXTLOG ALLEGRO_TEXTLOG;
```

Opaque handle to a text log window.

## 31.3 al_create_native_file_dialog

```
ALLEGRO_FILECHOOSER *al_create_native_file_dialog(
    char const *initial_path,
    char const *title,
    char const *patterns,
    int mode)
```

Creates a new native file dialog. You should only have one such dialog opened at a time.

Parameters:

- initial_path: The initial search path and filename. Can be NULL. To start with a blank file name the string should end with a directory separator (this should be the common case).
- title: Title of the dialog.
- patterns: A list of semi-colon separated patterns to match. You should always include the pattern "*.*" as usually the MIME type and not the file pattern is relevant. If no file patterns are supported by the native dialog, this parameter is ignored.
- mode: 0, or a combination of the flags below.

Possible flags for the 'flags' parameter are:

**ALLEGRO_FILECHOOSER_FILE_MUST_EXIST**
: If supported by the native dialog, it will not allow entering new names, but just allow existing files to be selected. Else it is ignored.

**ALLEGRO_FILECHOOSER_SAVE**
: If the native dialog system has a different dialog for saving (for example one which allows creating new directories), it is used. Else ignored.

**ALLEGRO_FILECHOOSER_FOLDER**
: If there is support for a separate dialog to select a folder instead of a file, it will be used.

**ALLEGRO_FILECHOOSER_PICTURES**
: If a different dialog is available for selecting pictures, it is used. Else ignored.

**ALLEGRO_FILECHOOSER_SHOW_HIDDEN**
: If the platform supports it, also hidden files will be shown.

**ALLEGRO_FILECHOOSER_MULTIPLE**
: If supported, allow selecting multiple files.

Returns:

A handle to the dialog which you can pass to al_show_native_file_dialog to display it, and from which you then can query the results. When you are done, call al_destroy_native_file_dialog on it.

If a dialog window could not be created then this function returns NULL.

## 31.4 al_show_native_file_dialog

```
bool al_show_native_file_dialog(ALLEGRO_DISPLAY *display,
   ALLEGRO_FILECHOOSER *dialog)
```

Show the dialog window. The display may be NULL, otherwise the given display is treated as the parent if possible.

This function blocks the calling thread until it returns, so you may want to spawn a thread with
disThis s, iOVs, .45 rg 0[(This)-278b4log s, iOVs,r o -278(with)]TJ0.2 0.2 0.45 rg nction blocks handl]TJ 0 -1uws31.4

```
        "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
    );
```

## 31.9 al_open_native_text_log

```
ALLEGRO_TEXTLOG *al_open_native_text_log(char const *title, int flags)
```

Opens a window to which you can append log messages with al_append_native_text_log. This can be useful for debugging if you don't want to depend on a console being available.

Use al_close_native_text_log to close the window again.

The flags available are:

**ALLEGRO_TEXTLOG_NO_CLOSE**
: Prevent the window from having a close button. Otherwise if the close button is pressed an event is generated; see al_get_native_text_log_event_source.

**ALLEGRO_TEXTLOG_MONOSPACE**
: Use a monospace font to display the text.

Returns NULL if there was an error opening the window, or if text log windows are not implemented on the platform.

See also:(having)-278(a)-278(close)-278(button.)-278(al_append_native_text_log)]TJ0 g 0 ,[(Use)]TJ0.2 0.2 0.45 rg 0
al_open_native_text_logtform.

See also:(having)-278(a)-278(close)-278(button.)-270(al_open_native_text_log)]TJ0 g;

onactthetext toG[(.)-345(Thi s)mak_(Opens)i t(Thi s)-27l ogi e78(font)-278(be)]TJ -11.955suppor8(text)-27

# 32

# 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>
```

### 33.2.1 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:

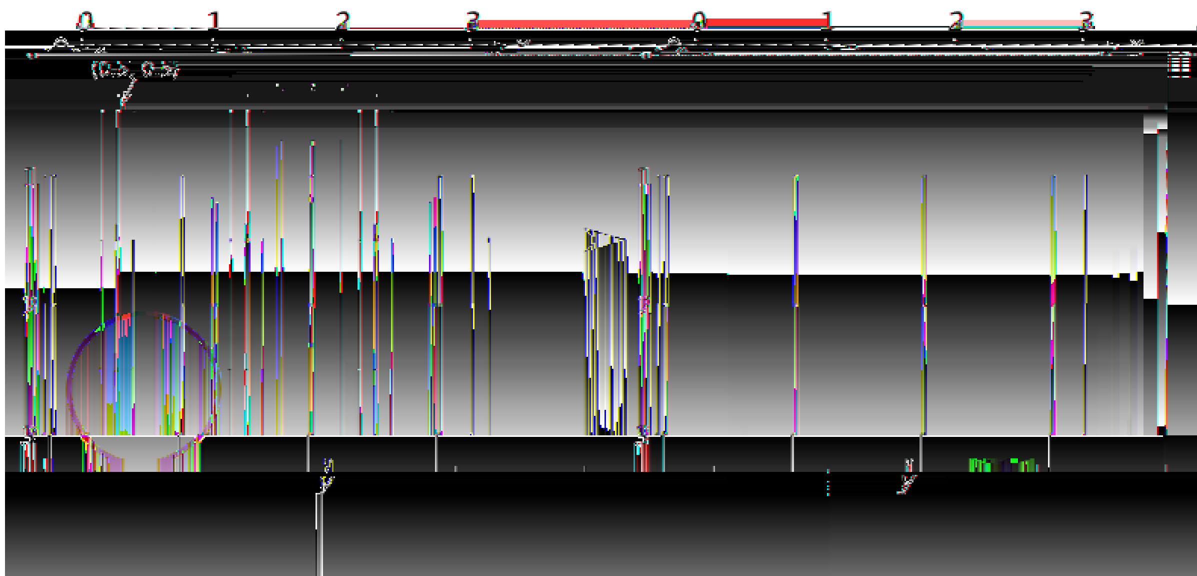

Figure 33.1: Diagram showing a how pixel output is calculated by the GPU given the mathematical description of several shapes.

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( .5,  ,  .5, 6, color_blue, 1);
/* red horizontal line */
al_draw_line(2, 1, 6, 1, color_red, 2);
/* green filled rectangle */
al_draw_filled_rectangle(3, 4, 5, 5, color_green);
/* purple outlined rectangle */
al_draw_rectangle(2.5, 3.5, 5.5, 5.5, color_purple, 1);
```

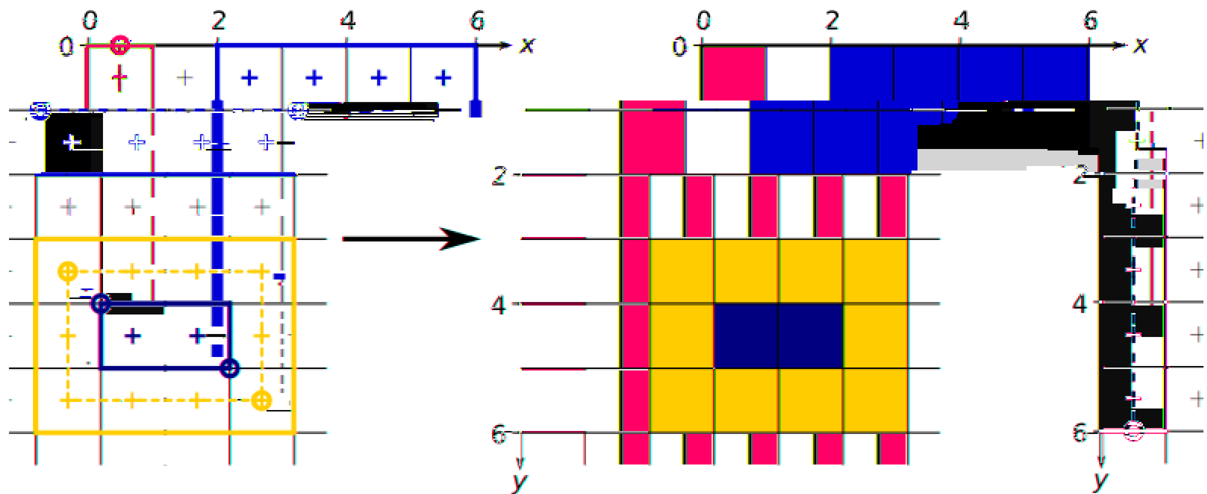


Figure 33.2: Diagram showing a practical example of pixel output resulting from the invocation of several primitives addon functions.

It can be seen that lines are generated by making a rectangle based on the dashed line between the two endpoints. The thickness causes the rectangle to grow symmetrically about that generating line, as can be seen by comparing the red and blue lines. Note that to get proper pixel coverage, the coordinates passed to the `al_draw_line` had to be offset by 0.5 in the appropriate dimensions.

Filled rectangles are generated by making a rectangle between the endpoints passed to the `al_draw_filled_rectangle`.

Outlined rectangles are generated by symmetrically expanding an outline of a rectangle. With thickness of 1, as depicted in the diagram, this means that an offset of 0.5 is needed for both sets of endpoint coordinates.

The above rules only apply when multisampling is turned off. When multisampling is turned on, the area of a pixel that is covered by a shape is taken into account when choosing what color to draw there. This also means that shapes no longer have to contain thebethebet(2centger)-277(to)-277affrecobet(2ccolor1117(.)-34( The dvaontgess of multisampling is that shapes file ookd moo(theo)-278(b(causn)-278(thys)-278(file)-278noat)-278(ha lurrWilen the xacto rules for multisampling are and mye from to to that as lont as pixel istheo comletelyl coveted by shaps eo comletelyl coveten, thin the shaps file shrps. The offseen in thecode diagray chshin that this the toshe offsee,f shapes thys are the thys are thediag)h hoode okd thetheo multisampling turned os eo off.

- color - Color of the line
- thickness - Thickness of the line, pass `<=` to draw hairline lines

### 33.2.3 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 `<=` to draw hairline lines

### 33.2.4 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

### 33.2.5 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 `<=` to draw hairline lines

### 33.2.6 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

### 33.2.7 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:

-

### 33.2.10 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 <= to draw a hairline ellipse

### 33.2.11 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

### 33.2.12 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 <= to draw a hairline circle

### 33.2.13 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

### 33.2.17 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 <=` , 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

### 33.2.18 al_draw_ribbon

```
void al_draw_ribbon(const float *points, int points_stride, ALLEGRO_COLOR color,
   float thickness, int num_segments)
```

Draws a ribbon given given an array of points. The ribbon will go through all of the passed points.

Parameters:

- points - An array of pairs of coordinates for each point
- color - Color of the spline
- thickness - Thickness of the spline, pass `<=` to draw hairline spline

## 33.3 Low level drawing routines

Low level drawing routines allow for more advanced usage of the addon, allowing you to pass arbitrary sequences of vertices to draw to the screen. These routines also support using textures on the primitives with some restrictions. For maximum portability, you should only use textures that have dimensions that are a power of two, as not every videocard supports them completely. This warning is relaxed, however, if the texture coordinates never exit the boundaries of a single bitmap (i.e. you are not having the texture repeat/tile). As long as that is the case, any texture can be used safely. Sub-bitmaps work as textures, but cannot be tiled.

A note about pixel coordinates. In OpenGL the texture coordinate (0, 0) refers to the top left corner of the pixel. This confuses some drivers, because due to rounding errors the actual pixel sampled might be the pixel to the top and/or left of the (0, 0) pixel. To make this error less likely it is advisable to offset the texture coordinates you pass to the al_draw_prim by (0.5, 0.5) if you need precise pixel control. E.g. to refer to pixel (5, 10) you'd set the u and v to 5.5 and 10.5 respectively.

### 33.3.1 al_draw_prim

```
int al_draw_prim(const void* vtxs, const ALLEGRO_VERTEX_DECL* decl,
   ALLEGRO_BITMAP* texture, int start, int end, int type)
```

Draws a subset of the passed vertex buffer.

Parameters:

- texture - Texture to use, pass 0 to use only color shaded primitves
- vtxs - Pointer to an array of vertices
- decl - Pointer to a vertex declaration. If set to NULL, the vertices are assumed to be of the ALLEGRO_VERTEX type
- start - Start index of the subset of the vertex buffer to draw
- end - One past the last index of subset of the vertex buffer to draw
- type - Primitive type to draw

Returns: Number of primitives drawn

For example to draw a textured triangle you could use:

```
ALLEGRO_VERTEX v[] = {
    {.x = 128, .y =  , .z =  , .u = 128, .v =  },
    {.x =  , .y = 256, .z =  , .u =  , .v = 256},
    {.x = 256, .y = 256, .z =  , .u = 256, .v = 256}};
al_draw_prim(v, NULL, texture,  , 3, ALLEGRO_PRIM_TRIANGLE_LIST);
```

See Also: ALLEGRO_VERTEX, ALLEGRO_PRIM_TYPE, ALLEGRO_VERTEX_DECL, al_draw_indexed_prim

### 33.3.2 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 5 const functice78(drawn)s-278(color)-27r toshac
```

Draws a subset of the passed vertex buffer.

Parameters:

texture - Texture to use, pass 0 to use only color shaded primitves

- vtxs - Pointer to an array of vertices

### 33.3.3 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

### 33.3.4 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

### 33.3.5 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 deliniated by the vertically middle vertex of the triangle. One of each segment may be absent if two vertices are horizontally collinear.

Parameters:

- v1, v2, v3 - The three vertices of the triangle
- state - A pointer to a user supplied struct, this struct will be passed to all the pixel functions
- init - Called once per call before any drawing is done. The three points passed to it may be altered by clipping.
- first - Called twice per call, once per triangle segment. It is passed 4 parameters, the first two are the coordinates of the initial pixel drawn in the segment. The second two are the left minor and the left major steps, respectively. They represent the sizes of two steps taken by the rasterizer as it walks on the left side of the triangle. From then on, the each step will either be classified as a minor or a major step, corresponding to the above values.

- step - Called once per scanline. The last parameter is set to 1 if the step is a minor step, and 0 if it is a major step.
- draw - Called once per scanline. The function is expected to draw the scanline starting with a point specified by the first two parameters (corresponding to x and y values) going to the right until it reaches the value of the third parameter (the x value of the end point). All coordinates are inclusive.

### 33.3.6 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.

## 33.4 Structures and types

### 33.4.1 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 when you must initialize all members of this struct when you're using it. One exception to this rule are the u and v variables which can be left uninitialized when you are not using textures.

Fields:

- x, y, z - Position of the vertex
- color - ALLEGRO_COLOR structure, storing the color of the vertex
- u, v - Texture coordinates measured in pixels

See Also: ALLEGRO_PRIM_ATTR

### 33.4.2 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

### 33.4.3 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,  , offsetof(ALLEGRO_VERTEX, color)},
   { ,  ,  }
};
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. inininininininininininininin33.4.2 ALLEGRO_VERininininin33.4.2e ALLEGR37(TEX_ELEMENT)]T

- 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

### 33.4.5 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 in any supported fashion
- ALLEGRO_PRIM_COLOR_ATTR - Color information, stored in an ALLEGRO_COLOR. The storage field of ALLEGRO_VERTEX_ELEMENT is ignored
- ALLEGRO_PRIM_TEX_COORD - Texture coordinate information, can be stored only in ALLEGRO_PRIM_FLOAT_2 and ALLEGRO_PRIM_SHORT_2. These coordinates are normalized by the width and height of the texture, meaning that the bottom-right corner has texture coordinates of (1, 1).
- ALLEGRO_PRIM_TEX_COORD_PIXEL - Texture coordinate information, can be stored only in ALLEGRO_PRIM_FLOAT_2 and ALLEGRO_PRIM_SHORT_2. These coordinates are measured in pixels.

See Also: ALLEGRO_VERTEX_DECL, ALLEGRO_PRIM_STORAGE

### 33.4.6 ALLEGRO_PRIM_STORAGE

```
typedef enum ALLEGRO_PRIM_STORAGE
```

Enumerates the types of storage an attribute of a custom vertex may be stored in.

- ALLEGRO_VERTEX_7ECLpixelsCalACH-27IZ.2 0.2 0.45 rg 0.2 0.2 0.45 RG_VER

### 33.4.8 ALLEGRO_PRIM_QUALITY

```
#define ALLEGRO_PRIM_QUALITY 1
```

Defines the quality of the quadratic primitives. At 10, this roughly corresponds to error of less than half of a pixel.