Event system and events

These functions are declared in the main Allegro header file:

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

ALLEGRO_EVENT

```
typedef union ALLEGRO_EVENT ALLEGRO_EVENT;
```

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

type (ALLEGRO_EVENT_TYPE)
: Indicates the type of event.

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

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

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

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

ALLEGRO_EVENT_JOYSTICK_AXIS

A joystick axis value changed.

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

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

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

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

ALLEGRO_EVENT_JOYSTICK_BUTTON_DOWN

A joystick button was pressed.

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

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

ALLEGRO_EVENT_JOYSTICK_BUTTON_UP

A joystick button was released.

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

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

ALLEGRO_EVENT_JOYSTICK_CONFIGURATION

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

ALLEGRO_EVENT_KEY_DOWN

A keyboard key was pressed.

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

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

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

ALLEGRO_EVENT_KEY_UP

A keyboard key was released.

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

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

ALLEGRO_EVENT_KEY_CHAR

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

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

keyboard.unichar (int)
: A Unicode code point (character). This *may* be zero or negative if the event was generated for a non-

visible "character", such as an arrow or Function key. In that case you can act upon the `keycode` field.

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

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

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

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

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

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

ALLEGRO_EVENT_MOUSE_AXES

One or more mouse axis values changed.

mouse.x (int)
: x-coordinate

mouse.y (int)
: y-coordinate

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

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

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

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

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

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

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

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

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

ALLEGRO_EVENT_MOUSE_BUTTON_DOWN

A mouse button was pressed.

mouse.x (int)
: x-coordinate

mouse.y (int)
: y-coordinate

mouse.z (int)
: z-coordinate

mouse.w (int)
: w-coordinate

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

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

ALLEGRO_EVENT_MOUSE_BUTTON_UP

A mouse button was released.

mouse.x (int)
: x-coordinate

mouse.y (int)
: y-coordinate

mouse.z (int)
z-coordinate

mouse.w (int)
w-coordinate

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

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

ALLEGRO_EVENT_MOUSE_WARPED

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

ALLEGRO_EVENT_MOUSE_ENTER_DISPLAY

The mouse cursor entered a window opened by the program.

mouse.x (int)
x-coordinate

mouse.y (int)
y-coordinate

mouse.z (int)
z-coordinate

mouse.w (int)
w-coordinate

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

ALLEGRO_EVENT_MOUSE_LEAVE_DISPLAY

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

mouse.x (int)
 x-coordinate

mouse.y (int)
 y-coordinate

mouse.z (int)
 z-coordinate

mouse.w (int)
 w-coordinate

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

ALLEGRO_EVENT_TIMER

A timer counter incremented.

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

timer.count (int64_t)
 The timer count value.

ALLEGRO_EVENT_DISPLAY_EXPOSE

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

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

display.x (int)

display.y (int)

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

display.width (int)

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

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

ALLEGRO_EVENT_DISPLAY_RESIZE

The window has been resized.

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

display.x (int)

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

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

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

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

ALLEGRO_EVENT_DISPLAY_CLOSE

The close button of the window has been pressed.

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

ALLEGRO_EVENT_DISPLAY_LOST

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

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

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

ALLEGRO_EVENT_DISPLAY_FOUND

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

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

ALLEGRO_EVENT_DISPLAY_SWITCH_OUT

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

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

ALLEGRO_EVENT_DISPLAY_SWITCH_IN

The window is the active one again.

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

ALLEGRO_EVENT_DISPLAY_ORIENTATION

Generated when the rotation or orientation of a display changes.

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

event.display.orientation

Contains one of the following values:

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

ALLEGRO_EVENT_DISPLAY_HALT_DRAWING

(since: 5.1.0)

When a display receives this event it should stop doing any drawing and then call al_acknowledge_drawing_halt.

This is currently only relevant for iOS. It will be sent when the application is switched to background mode, in addition to ALLEGRO_EVENT_DISPLAY_SWITCH_OUT. The latter may also be sent in situations where the application is not active but still should continue drawing, for example when a popup is display in front of it.

ALLEGRO_EVENT_DISPLAY_RESUME_DRAWING

(since: 5.1.0)

This is called under iOS when an application returns from background mode and is allowed to draw to the display again.

ALLEGRO_EVENT_TOUCH_BEGIN

(since: 5.1.0)

The touch input device registered a new touch.

touch.display (ALLEGRO_DISPLAY)
: The display which was touched.

touch.id (int)
: An identifier for this touch. If supported by the device it will stay the same for events from the same finger until the touch ends.

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

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

touch.dx (float)
: Movement speed in pixels in x direction.

touch.dy (float)
: Movement speed in pixels in y direction.

touch.primary (bool)
: Wether this is the only/first touch or an additional touch.

ALLEGRO_EVENT_TOUCH_END

(since: 5.1.0)

A touch ended.

Has the same fields as ALLEGRO_EVENT_TOUCH_BEGIN.

ALLEGRO_EVENT_TOUCH_MOVE

(since: 5.1.0)

The position of a touch changed.

Has the same fields as ALLEGRO_EVENT_TOUCH_BEGIN.

ALLEGRO_EVENT_TOUCH_CANCEL

(since: 5.1.0)

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

Has the same fields as ALLEGRO_EVENT_TOUCH_BEGIN.

See also: ALLEGRO_EVENT_SOURCE, ALLEGRO_EVENT_TYPE, ALLEGRO_USER_EVENT, ALLEGRO_GET_EVENT_TYPE

ALLEGRO_EVENT_DISPLAY_CONNECTED

(since: 5.1.1)

Not yet documented.

ALLEGRO_EVENT_DISPLAY_DISCONNECTED

(since: 5.1.1)

Not yet documented.

ALLEGRO_USER_EVENT

```
typedef struct ALLEGRO_USER_EVENT ALLEGRO_USER_EVENT;
```

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

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

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

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

To create a new user event you would do this:

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

al_init_user_event_source(&my_event_source);

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

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

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

See also: al_emit_user_event, ALLEGRO_GET_EVENT_TYPE

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

ALLEGRO_EVENT_SOURCE

```
typedef struct ALLEGRO_EVENT_SOURCE ALLEGRO_EVENT_SOUR
```

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

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

ALLEGRO_GET_EVENT_TYPE

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

Make an event type identifier, which is a 32-bit integer. Usually, but not necessarily, this will be made from four 8-bit character codes, for example:

```
#define MY_EVENT_TYPE   ALLEGRO_GET_EVENT_TYPE('M','I'
```

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

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

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

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

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

See also: ALLEGRO_EVENT, ALLEGRO_USER_EVENT, ALLEGRO_EVENT_TYPE_IS_USER

ALLEGRO_EVENT_TYPE_IS_USER

```
#define ALLEGRO_EVENT_TYPE_IS_USER(t)       ((t) >= 5
```

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.

al_create_event_queue

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

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

See also: al_register_event_source, al_destroy_event_queue, ALLEGRO_EVENT_QUEUE

al_destroy_event_queue

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

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

See also: al_create_event_queue, ALLEGRO_EVENT_QUEUE

al_register_event_source

```
void al_register_event_source(ALLEGRO_EVENT_QUEUE *que
   ALLEGRO_EVENT_SOURCE *source)
```

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

See also: al_unregister_event_source, ALLEGRO_EVENT_SOURCE

al_unregister_event_source

```
void al_unregister_event_source(ALLEGRO_EVENT_QUEUE *q
   ALLEGRO_EVENT_SOURCE *source)
```

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

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

See also: al_register_event_source

al_pause_event_queue

```
void al_pause_event_queue(ALLEGRO_EVENT_QUEUE *queue,
```

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

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

See also: al_is_event_queue_paused

Since: 5.1.0

al_is_event_queue_paused

```
bool al_is_event_queue_paused(const ALLEGRO_EVENT_QUEU
```

Return true if the event queue is paused.

See also: al_pause_event_queue

Since: 5.1.0

al_is_event_queue_empty

```
bool al_is_event_queue_empty(ALLEGRO_EVENT_QUEUE *queu
```

Return true if the event queue specified is currently empty.

See also: al_get_next_event, al_peek_next_event

al_get_next_event

```
bool al_get_next_event(ALLEGRO_EVENT_QUEUE *queue, ALI
```

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

See also: ALLEGRO_EVENT, al_peek_next_event, al_wait_for_event

al_peek_next_event

```
bool al_peek_next_event(ALLEGRO_EVENT_QUEUE *queue, AL
```

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

See also: ALLEGRO_EVENT, al_get_next_event, al_drop_next_event

al_drop_next_event

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

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

See also: al_flush_event_queue, al_is_event_queue_empty

al_flush_event_queue

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

Drops all events, if any, from the queue.

See also: al_drop_next_event, al_is_event_queue_empty

al_wait_for_event

```
void al_wait_for_event(ALLEGRO_EVENT_QUEUE *queue, ALI
```

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

See also: ALLEGRO_EVENT, al_wait_for_event_timed, al_wait_for_event_until, al_get_next_event

al_wait_for_event_timed

```
bool al_wait_for_event_timed(ALLEGRO_EVENT_QUEUE *queu
   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

al_wait_for_event_until

```
bool al_wait_for_event_until(ALLEGRO_EVENT_QUEUE *queu
   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

al_init_user_event_source

```
void al_init_user_event_source(ALLEGRO_EVENT_SOURCE *s
```

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

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

```
typedef struct THING THING;

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

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

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

    return thing;
}
```

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

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

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

See also: ALLEGRO_EVENT_SOURCE, al_destroy_user_event_source, al_emit_user_event, ALLEGRO_USER_EVENT

al_destroy_user_event_source

```
void al_destroy_user_event_source(ALLEGRO_EVENT_SOURCE
```

Destroy an event source initialised with al_init_user_event_source.

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

See also: ALLEGRO_EVENT_SOURCE

al_emit_user_event

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

Emit a user event. The event source must have been initialised with al_init_user_event_source. Returns `false` if the event source isn't registered with any queues, hence the event wouldn't have been delivered into any queues.

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

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

Once the reference count drops to zero `dtor` will be called with a copy of the event as an argument. It should free the resources associated with the event, but *not* the event itself (since it is just a copy).

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

See also: ALLEGRO_USER_EVENT, al_unref_user_event

al_unref_user_event

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

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

See also: al_emit_user_event, ALLEGRO_USER_EVENT

al_get_event_source_data

```
intptr_t al_get_event_source_data(const ALLEGRO_EVENT_
```

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

al_set_event_source_data

```
void al_set_event_source_data(ALLEGRO_EVENT_SOURCE *sc
```

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

Allegro version 5.1.3 (GIT) - Last updated: 2012-07-22 00:48:04 UTC