



APPENDIX D: EVENT HANDLERS

This chapter lists all of the event handlers, their parameters, and the conditions required to trigger the events. Table D.1 shows a brief summary; longer descriptions follow. Events are triggered with a default interval of 0.1 second. The function `llMinEventDelay()` can set the minimum time between events being handled (but no less than 0.05 second).



DEFINITION

`llMinEventDelay(float delay)`

Set the minimum time, in seconds, between events being handled. Default value is 0.1 second, minimum is 0.05 second.

TABLE D.1: EVENT HANDLERS, FUNCTION CALLS REQUIRED TO CAUSE OR MONITOR FOR THE CONDITIONS, AND CONDITIONS THAT TRIGGER THE EVENT

EVENT HANDLER	FUNCTION TO SET UP THE EVENT	TRIGGER
<code>at_rot_target</code>	<code>llRotTarget</code>	Arrived at specified target rotation
<code>at_target</code>	<code>llTarget</code>	Not arrived at specified target rotation
<code>attach</code>	—	Object attached or detached from avatar
<code>changed</code>	—	Object changes in one of several ways (link, color, sitting, etc.)
<code>collision</code>	—	Physical object collision, but not for <code>llVolumeDetect()</code>
<code>collision_end</code>	—	Physical object stops collision
<code>collision_start</code>	—	Physical object starts collision
<code>control</code>	<code>llTakeControls</code>	"Taken" keyboard or mouse controls are pressed, held, or released
<code>dataserver</code>	<code>llGetNotecardLine</code> <code>llGetNumberOfNotecardLines</code> <code>llRequestAgentData</code> <code>llRequestInventoryData</code> <code>llRequestSimulatorData</code>	Asynchronous request for data is available
<code>email</code>	<code>llGetNextEmail</code>	Asynchronous request for email is available
<code>http_response</code>	<code>llHTTPRequest</code>	Asynchronous request for http data is available
<code>land_collision</code>	—	Physical object collides with land
<code>land_collision_end</code>	—	Physical object starts to collide with land
<code>land_collision_start</code>	—	Physical object stops colliding with land
<code>link_message</code>	<code>llMessageLinked</code>	Prim in linkset sends a message
<code>listen</code>	<code>llListen</code>	A chat message matching current <code>llListen()</code> filters is sent within "hearing" range

EVENT HANDLER	FUNCTION TO SET UP THE EVENT	TRIGGER
money	—	The object is paid
moving_end	—	Object stops moving or exits the current simulator
moving_start	—	Object starts moving or enters a sim
no_sensor	llSensor llSensorRepeat	Sensor request has found nothing
not_at_rot_target	llRotTarget	Object is not yet at specified target rotation
not_at_target	llTarget	Object is not yet at specified target position
object_rez	llRezObject	The script has successfully rezzed another object
on_rez	—	Object is rezzed from inventory or by another script
remote_data	llOpenRemoteDataChannel llSendRemoteData	XML-RPC communication is received by the script
run_time_permissions	llRequestPermissions	Run-time permissions have been granted as requested
sensor	llSensor llSensorRepeat	Sensor request is returning results
state_entry	—	Transition into the state
state_exit	—	Transition out of the state using a state command
timer	llSetTimerEvent	At least the specified time interval has passed
touch	—	A user is currently "touching" the object
touch_end	—	A user has finished "touching" the object
touch_start	—	A user has started "touching" the object

LSL is an *event-driven* language. This behavior is a major discriminating feature between LSL and many other languages. Essentially, the script's flow is determined by *events* such as receiving messages from other objects or avatars, or receiving sensor signals. Functionally, events are messages that are sent by a *Second Life* server to the scripts active in that sim. Event messages can be received by defining the matching *event handlers* in the state(s) of your scripts. *Nothing* happens in a script without event handlers: even the passing of time is marked by the delivery of timer events. Some events cannot happen without the matching event handler active somewhere in the object (in at least one of its prims). For instance, if an object doesn't have a script with a `money()` event handler, avatars cannot pay that object.

Many library functions have effects that take a relatively long time to accomplish. Rather than *blocking* (stopping execution), LSL uses *asynchronous* communications: your script fires off a request that something happen, and it is notified with an event when the request is satisfied. This pattern is used any time your script makes a call to a library function that cannot be handled locally by the host sim (for instance, the function `llHTTPRequest()` and the `http_response()` event handler) when physical simulation effects aren't going to be instantaneous (`llTarget()` and `at_target()`). Repeating events (`llSensorRepeat()` and `sensor()`) and even listeners (`llListen()` and `listen()`) follow the same model: make a request and get responses later. Asynchronous execution allows your script to keep functioning, handling other events and interactions without stopping to wait for long-term requests to happen.

As events occur, the script queues applicable ones (those that have handlers defined on the current state) for execution in the order they were raised. You should be aware of several important constraints on the delivery of event messages to handlers:





- By default, events are delivered to a script no more frequently than every 0.1 second. You can adjust this with `llMinEventDelay` (float *seconds*) but it can not be less than 0.05 second.
- A maximum of 64 events can be queued on a script—any additional events will be discarded silently!
- Queued events are discarded silently when a script transitions between states.

The combination of these factors means that if you have code that expects to get events rapidly and takes a relatively long time to execute (including artificial delays!), you may run the risk of missing events. For example, sending an IM (which delays the script by 2 seconds) in response to collide events (which can happen as often as every 0.1 second) is probably asking for trouble.

Similarly, the LSL function `llSleep` (float *seconds*) pauses script execution for the specified number of seconds without leaving the current event handler, similar to the way that many LSL functions introduce artificial delays. In both cases, there is a potential problem if your script is likely to handle lots of events.

Some events are delivered to a script only if the script has made the appropriate request. For instance, a `listen()` event handler is never called unless another handler in the same state has called `llListen()`. The list of these functions is shown in the "Function to Set Up Event" column of Table D.1.

LSL EVENT HANDLERS

This section describes the event handlers from Table D.1 in more detail, describing the event's parameters, the conditions under which it triggers, and any functions required before the event will trigger.

`at_rot_target(integer targetHandle, rotation targetRot, rotation currRot)`

- integer *targetHandle* is the handle returned by the call to `llRotTarget()`.
- rotation *targetRot* is the desired rotation, and is the same value that was passed into `llRotTarget()`.
- rotation *currRot* is the current rotation, equivalent to `llGetRot()`.

Triggered when the object arrives at the target. Enabled by a call to the function

```
integer llRotTarget(rotation targetRot, float tau);
```

Only one target can be specified. Also see `not_at_rot_target()`.

`at_target(integer targetHandle, vector targetPos, vector currPos)`

- integer *targetHandle* is the handle returned by the call to `llTarget()`.
- vector *targetPos* is the desired position, and is the same value that was passed into `llTarget()`.
- vector *currPos* is the current position, equivalent to `llGetPos()`.

Triggered when the object arrives at the target. Enabled by a call to the function

```
integer llTarget (vector position, float range);
```

Only one target can be specified. Also see `not_at_target ()`.

attach(key *id*)

- key *id* is the UUID identity of the avatar if the object is attached. If the object is not attached, then the value is `NULL_KEY`.

Triggered when an object attaches or detaches from an avatar. If you are editing an attached object, you must detach it and then reattach it before the event will trigger again. When you wear an object from inventory, both the `on_rez ()` and `attach ()` events trigger; if `on_rez ()` triggers first and calls `llResetScript ()`, then the key for `attach ()` will be forgotten. The bowling glove and HUD in the online chapter "Project 2: Bowling" show an example workaround.

changed(integer *change*)

- integer *change* is a bitfield of flags, shown in Table D.2.

Triggered by changes to the object or prim. Multiple changes can be represented in a single event, so use bitwise arithmetic.

TABLE D.2: BITFIELD FOR THE `CHANGED ()` EVENT.

FLAG	VALUE	DESCRIPTION
<code>CHANGED_INVENTORY</code>	<code>0x001</code>	Prim inventory has changed by the owner adding or removing inventory.
<code>CHANGED_COLOR</code>	<code>0x002</code>	Prim color or alpha parameters have changed.
<code>CHANGED_SHAPE</code>	<code>0x004</code>	Prim shape has changed.
<code>CHANGED_SCALE</code>	<code>0x008</code>	Prim scale has changed.
<code>CHANGED_TEXTURE</code>	<code>0x010</code>	Prim texture parameters have changed.
<code>CHANGED_LINK</code>	<code>0x020</code>	Number of prims making up the object has changed, or an avatar has sat on or unsat from the object.
<code>CHANGED_ALLOWED_DROP</code>	<code>0x040</code>	A user other than the owner (or the owner if the object is no-mod) has added inventory to the prim.
<code>CHANGED_OWNER</code>	<code>0x080</code>	The object has changed owners.
<code>CHANGED_REGION</code>	<code>0x100</code>	The object has changed region.
<code>CHANGED_TELEPORT</code>	<code>0x200</code>	The avatar to whom this object is attached has teleported.

collision(integer *numDetected*)

- integer *numDetected* is the number of `collision ()` events in the queue.

Triggered while physical object is colliding with avatars and other physical objects. Attachments, phantom objects, and nonphysical objects will not trigger the event, nor will objects using `llVolumeDetect ()`. Also see `collision_end ()` and `collision_start ()`. Information on the avatars may be gathered via the `llDetected* ()` library functions.





`collision_end(integer numDetected)`

- integer `numDetected` is the number of `collision_end()` events in the queue.

Triggered when a physical object stops colliding with avatars or other physical objects. It will trigger for objects using `llVolumeDetect()`. It does not always trigger reliably. Also see `collision()` and `collision_start()`. Information on the avatars may be gathered via the `llDetected*()` library functions.

`collision_start(integer numDetected)`

- integer `numDetected` is the number of `collision_end()` events in the queue.

Triggered when a physical object starts colliding with avatars or other physical objects. It will trigger for objects using `llVolumeDetect()`. It does not always trigger reliably. Also see `collision()` and `collision_end()`. Information on the avatars may be gathered via the `llDetected*()` library functions.

`control(key id, integer held, integer recentChange)`

- key `id` is the UUID of the avatar using the controls.
- integer `held` is the bitfield of the keys being held down. Values are shown in Table D.3.
- integer `recentChange` is the bitfield of the keys changed since the last control event. Values are shown in Table D.3.

Triggered when one or more of the "taken" keyboard or mouse controls are pressed, held, or released. After `PERMISSION_TAKE_CONTROLS` has been granted with `llRequestPermissions()` as follows

```
llRequestPermissions(key agent, integer perm);
```

Controls are taken using

```
llTakeControls(integer controls, integer accept, integer passToAvatar)
```

You can work out whether a button has been held or touched by combining the `held` and `recentChange` parameters, as in the following example:

```
if (held & recentChange & CONTROL_FWD)
    llOwnerSay("forward just pressed");
if (~held & recentChange & CONTROL_FWD)
    llOwnerSay("forward just released");
if (held & ~recentChange & CONTROL_FWD)
    llOwnerSay("forward held down; saw on previous call to control");
if (~held & ~recentChange & CONTROL_FWD)
    llOwnerSay("forward untouched");
```

TABLE D.3: BITFIELD VALUES FOR THE CONTROL () EVENT

CONSTANT	VALUE	DESCRIPTION
CONTROL_FWD	0x00000001	Move-forward control
CONTROL_BACK	0x00000002	Move-back control
CONTROL_LEFT	0x00000004	Move-left control
CONTROL_RIGHT	0x00000008	Move-right control
CONTROL_ROT_LEFT	0x00000100	Rotate-left control
CONTROL_ROT_RIGHT	0x00000200	Rotate-right control
CONTROL_UP	0x00000010	Move-up control
CONTROL_DOWN	0x00000020	Move-down control
CONTROL_LBUTTON	0x10000000	Left-mouse-button control
CONTROL_ML_LBUTTON	0x40000000	Left-mouse-button control while in mousetlook

dataserver(key *queryHandle*, string *data*)

- key *queryHandle* matches the key value returned by the requesting function.
- string *data* is the requested data, cast as a string if necessary.

Triggered when the task receives asynchronous data. Multiple requests can be pending; `dataserver` answers do not necessarily come in the order they were requested, so always use the *queryHandle* to check which answer is being received. `dataserver` requests will trigger `dataserver()` events in all scripts in the same prim where the request was made. Library functions that set up this trigger include the following:

```
key llGetNotecardLine(string name, integer line);
key llGetNumberOfNotecardLines(string name);
key llRequestAgentData(key id, integer data);
key llRequestInventoryData(string name);
key llRequestSimulatorData(string simulator, integer data);
```

email(string *time*, string *address*, string *subject*, string *message*, integer *numRemaining*)

- string *time* is a UNIX integer timestamp, cast as string; same as what is returned by `llGetUnixTime()`.
- string *address* is an email address, limited to approximately 78 characters.
- string *subject* is the subject line, limited to approximately 78 characters.
- string *message* is the body of the message, limited to approximately 1,000 characters.
- integer *numRemaining* is the number of emails left to process. The queue is limited to 100 emails; any additional email is bounced.

Triggered when a request by `llGetNextEmail()` is answered:

```
llGetNextEmail( string address, string subject );
```





To send an email, use `llEmail()`. The body has several headers (see the wiki for details); to remove these headers use

```
message = llDeleteSubString(message, 0, llSubStringIndex(message, "\n\n") + 1);
```

`http_response(key requestHandle, integer status, list metadata, string body)`

- key `requestHandle` is the value returned by `llHTTPRequest()`.
- integer `status` is the HTTP return code. See www.w3.org/Protocols/rfc2616/rfc2616-sec10.html for a complete list. Two common ones are 404 for **Not Found** or 200 for **OK**.
- list `metadata` is a strided list filled with `<key, value>` pairs describing the response. Currently, the only key returned is `HTTP_BODY_TRUNCATED`, with a value equal to the point at which the response was truncated in bytes.
- string `body` is the value of the HTTP response. If the response includes a "Content-Type" header, then the actual content depends on what type the body contains. If it doesn't include the header, then the body is set to "Unsupported or unknown Content-Type". Limited to 2,048 characters.

Triggered when task receives a response to `llHTTPRequest()` or if a pending request fails or times out:

```
key llHTTPRequest(string url, list parameters, string body);
```

Multiple requests can be pending.

`land_collision(vector pos)`

- vector `pos` is the position of collision with the ground, in region coordinates.

Triggered when the object collides with land. Triggered even for phantom objects. In a linkset, the root prim will receive the event trigger for any children with no script and for scripted children who have invoked `llPassCollisions(TRUE)`. Also see `land_collision_start()` and `land_collision_end()`.

`land_collision_end(vector pos)`

- vector `pos` is the position of the last collision with the ground, in region coordinates.

Triggered when the object stops colliding with land. Triggered even for phantom objects. In a linkset, the root prim will receive the event trigger for any children with no script, and for scripted children who have invoked `llPassCollisions(TRUE)`. Also see `land_collision()` and `land_collision_start()`.

land_collision_start(vector *pos*)

- vector *pos* is the position of the last collision with the ground, in region coordinates.

Triggered when the object stops colliding with land. Triggered even for phantom objects. In a linkset, the root prim will receive the event trigger for any children with no script and for scripted children who have invoked `llPassCollisions(TRUE)`. Also see `land_collision()` and `land_collision_end()`.

link_message(integer *senderLinkNum*, integer *num*, string *message*, key *anyKey*)

- integer *senderLinkNum* is the identity of the linked prim sending the message.
- integer *num* is the value used in `llMessageLinked()`.
- string *message* is the value used in `llMessageLinked()`.
- key *anyKey* is the value used in `llMessageLinked()`.

Triggered when a script receives a link message from a prim in the same linkset (including itself) via the function call

```
llMessageLinked(integer receiverLinkNum, integer num,  
                string message, key anyKey);
```

num, *message* and *anyKey* are determined by the caller. Only 64 messages are queued; extras are discarded silently.

listen(integer *channel*, string *name*, key *id*, string *message*)

- integer *channel* is the communications channel that the message was heard on. Channel 0, `PUBLIC_CHANNEL`, is the public chat channel that all users see as chat text. Channels 1 to 2,147,483,647 and -1 to -2,147,483,647 are private channels that scripts can listen to. Avatars can chat only on positive channels. Channel 2,147,483,648 is the debug channel, `DEBUG_CHANNEL`.
- string *name* is the name of the speaking object or avatar.
- key *id* is the UUID of the speaker.
- string *message* is the message.

Triggered by avatar chat and scripted chat using `llSay()`, `llShout()`, `llWhisper()`, `llRegionSay()`, and `llDialog()`. Use `llListen()` to enable and create appropriate filters:

```
integer llListen(integer channel, string name, key id, string message);
```

money(key *avatarID*, integer *amount*)

- key *avatarID* is the UUID identifier of the avatar who paid.
- integer *amount* is the amount paid.

Triggered when money is paid to the owner of prim in the *amount* by *avatarID*.





moving_end()

Triggered under the following conditions:

- When any object is rezzed.
- When the user changes the object's position via the edit menu.
- When a physical object stops moving.
- For an attached object, if the avatar moves, then it is triggered continuously. If editing an attached object, you must detach it and then reattach it before the event will trigger again.
- `llSetPos()` does not trigger the event.

moving_start()

Triggered under the following conditions:

- When any object is rezzed.
- When the user changes the object's position with the edit menu; note this event is triggered when the user **stops** moving the object.
- When a physical object starts moving.
- For an attached object, if the avatar moves, then it is triggered continuously.
- `llSetPos()` does not trigger the event.

no_sensor()

Triggered when sensors are active but are not sensing anything. Called instead of `sensor()`. Sensors are created with `llSensor()` and `llSensorRepeat()` functions:

```
llSensor(string name, key id, integer type, float range, float arc);  
llSensorRepeat(string name, key id, integer type,  
               float range, float arc, float timerInterval);
```

not_at_rot_target()

Triggered when the object arrives at the target. Enabled by a call to the function

```
integer llRotTarget(rotation targetRot, float tau);
```

Called instead of `at_rot_target()`. It may be triggered multiple times if the target position has not been reached.

not_at_target()

Triggered when the object arrives at the target. Enabled by a call to the function

```
integer llTarget(vector position, float range);
```

Called instead of `at_target()`. It may be triggered multiple times if the target position has not been reached.

object_rez(key *id*)

- key *id* is the UUID of the object rezzed.

Triggered when the script rezzes another object using `llRezObject()`:

```
llRezObject(string inventory,  
            vector pos, vector vel, rotation rot,  
            integer param);
```

on_rez(integer *start_param*)

- integer *start_param* is the parameter supplied to `llRezObject()` or `llRezAtRoot()`; it is zero when rezzed by an avatar.

Triggered when an object is rezzed (by script or by user). Also triggered in attachments when a user logs in or when the object is attached from inventory. The start parameter can also be retrieved using `llGetStartParameter()`. Use `llResetScript()` to cause a script reset and a trigger of `state_entry()`.

remote_data(integer *event_type*, key *channel*, key *message_id*, string *sender*, integer *integer_data*, string *string_data*)

If *event_type* is `REMOTE_DATA_CHANNEL`, then

- *channel* is the channel opened by `llOpenRemoteDataChannel()`.
- *message_id* is `NULL_KEY`.
- *sender* is "".
- *integer_data* is 0.
- *string_data* is "".

If *event_type* is `REMOTE_DATA_REQUEST`, then

- *channel* is the channel that the message was received on.
- *message_id* is `NULL_KEY`.
- *sender* is "".
- *integer_data* is the integer data sent on the channel.
- *string_data* is a string up to 255 characters.

Triggered by `llOpenRemoteDataChannel()` to notify the script that the channel is open, and when any script or external XML-RPC client sends data, either as a new message or as a returned value from a call originated from this script. See Listing 12.8 (XML-RPC Flag Changer) in Chapter 12, "Reaching outside *Second Life*," for a complete example.

run_time_permissions(integer *perm*)

- integer *perm* is a bitfield of granted permissions, as shown in Table D.4.

Triggered when a user grants run-time permissions to a script. Request permissions with a call to `llRequestPermissions(key agent, integer perm)`.





TABLE D.4: PERMISSIONS THAT NEED TO BE REQUESTED OF A USER

CONSTANTS	VALUE	ACTION	GRANTER
PERMISSION_DEBIT	0x002	Take money from user's account.	Owner
PERMISSION_TAKE_CONTROLS	0x004	Take user's controls.	Anyone
PERMISSION_TRIGGER_ANIMATION	0x010	Trigger animation on user.	Anyone
PERMISSION_ATTACH	0x020	Attach or detach from user.	Owner
PERMISSION_CHANGE_LINKS	0x080	Change links in a linkset.	Owner
PERMISSION_TRACK_CAMERA	0x400	Track user's camera position and rotation.	Anyone
PERMISSION_CONTROL_CAMERA	0x800	Control user's camera.	Anyone

Returns an integer bit pattern with the permissions the script has been granted. Permissions can be combined with the bitwise "or" operator (`|`), and tested with a bitwise "and" operator (`&`), as shown in the following example:

```
default
{
    state_entry() {
        integer reqPerms = PERMISSION_TRIGGER_ANIMATION | PERMISSION_ATTACH;
        llRequestPermissions(llGetOwner(), reqPerms);
    }
}

run_time_permissions(integer perms) {
    if (perms & PERMISSION_ATTACH) {
        llAttachToAvatar(ATTACH_CHIN);
    }
    if (perms & PERMISSION_TRIGGER_ANIMATION) {
        llStartAnimation("Walking");
    }
}
}
```

The function `llGetPermissions()` also returns the list of permissions granted to a script.

sensor(integer numDetected)

- integer *numDetected* is the number of objects sensed, to a maximum of 16. The objects are sorted from closest to farthest.

Result of the `llSensor()` and `llSensorRepeat()` functions:

```
llSensor(string name, key id, integer type, float range, float arc)
llSensorRepeat(string name, key id, integer type,
               float range, float arc, float timerInterval)
```

Information on those objects may then be gathered via the `llDetected*` library functions. If a sensor is active but no matches are found, `no_sensor()` will be raised instead.

state_entry()

Triggered on script startup, on any state transition, and after calls to `llResetScript()`. This event is *not* triggered every time the object is rezzed. Do not place calls to `llResetScript()` inside this event handler; you'll get infinite recursion.

state_exit()

Triggered whenever the `state` command is used to transition to another state, before the new state is entered.

timer()

Triggered at periodic intervals specified by the parameter to `llSetTimerEvent(float interval)`. Only one timer can be active in the state at one time. Time dilation (when simulation time appears to slow down relative to real time) in an overworked sim can cause the periodicity to be unpredictable. The timer survives state changes, meaning, for example, that a call to `llSetTimerEvent()` in the `default` state will trigger `timer()` handlers in every other state.

touch(integer numDetected)

- `integer numDetected` is the number of avatars who touched the object since the last time the event was triggered.

Triggered when a user is clicking the object. It will continue to be triggered until the mouse button has been released (it triggers multiple times). Information on the avatars may be gathered via the `llDetected*` () library functions shown in Table 2.1 of Chapter 2, "Making Your Avatar Stand Up and Stand Out."

touch_end(integer numDetected)

- `integer numDetected` is the number of avatars who touched the object since the last time the event was triggered.

Triggered when the user stops clicking on an object. Information on the avatars may be gathered via the `llDetected*` () library functions shown in Table 2.1.

touch_start(integer numDetected)

- `integer numDetected` is the number of avatars who touched the object since the last time the event was triggered.

Triggered by a user starting to click on an object. Information on the avatars may be gathered via the `llDetected*` () library functions shown in Table 2.1.

SAMPLE EVENT DELIVERY

A given in-world action can generate many different events. Depending on what you are trying to do, you may want to catch one, some, or all of the events. Table D.5 shows some event sequences as delivered to the script in Listing D.1. Note that each run of a script like this one may yield slightly different results. Also note that this script lacks function calls that would trigger certain events, such as `llListen()` for the `listen()` event handler.





TABLE D.5: SAMPLE EVENT SEQUENCES

ACTION	EVENTS GENERATED
An object is rezzed to the world from an agent's inventory.	on_rez (possibly but not necessarily state_entry!)
A new script is added to an object, manually resetting a script or recompiling a running script on a rezzed object.	state_entry (but not on_rez!)
A scripted object from inventory is worn/attached.	on_rez, attach, collision_start, collision, moving_start, collision, moving_end, moving_start, collision, collision_end, moving_end*
An attached/worn object is detached.	attach
An attached/worn object is dropped.	attach, moving_start, moving_end
A rezzed object is moved using the editor.	moving_start, moving_end

* Note that for one attach action, one event may be triggered multiple times. Also, triggered events can vary slightly each time. Notice that collision and moving sets don't need to nest! The collision events happen because the object intersects with the avatar; the moving events happen because the object is rezzed outside the avatar and moved to the correct place.

Listing D.1: An Event Probe

```
default {
  at_rot_target(integer tnum, rotation trot, rotation crot) {
    llOwnerSay("at_rot_target");
  }
  at_target(integer tnum, vector tpos, vector cpos) {
    llOwnerSay("at_target");
  }
  attach(key at) {
    llOwnerSay("attach");
  }
  changed(integer ev) {
    llOwnerSay("changed");
  }
  collision(integer ccount) {
    llOwnerSay("collision");
  }
  collision_end(integer ccount) {
    llOwnerSay("collision_end");
  }
  collision_start(integer ccount) {
    llOwnerSay("collision_start");
  }
  control(key id, integer held, integer change) {
    llOwnerSay("control");
  }
  dataserver(key qid, string data) {
    llOwnerSay("dataserver");
  }
  email(string time, string addr, string sub, string message, integer remain) {
    llOwnerSay("email");
  }
  http_response(key rid, integer status, list metadata, string body) {
    llOwnerSay("http_response");
  }
  land_collision(vector pos) {
    llOwnerSay("land_collision");
  }
  land_collision_end(vector pos) {
    llOwnerSay("land_collision_end");
  }
}
```

```

land_collision_start(vector pos) {
    llOwnerSay("land_collision_start");
}
link_message(integer sendernum, integer num, string str, key id) {
    llOwnerSay("link_message");
}
listen(integer chan, string name, key id, string message) {
    llOwnerSay("listen");
}
money(key id, integer amount) {
    llOwnerSay("money");
}
moving_end() {
    llOwnerSay("moving_end");
}
moving_start() {
    llOwnerSay("moving_start");
}
no_sensor() {
    llOwnerSay("no_sensor");
}
not_at_rot_target() {
    llOwnerSay("not_at_rot_target");
}
not_at_target() {
    llOwnerSay("not_at_target");
}
object_rez(key id) {
    llOwnerSay("object_rez");
}
on_rez(integer param) {
    llOwnerSay("on_rez");
}
remote_data(integer evtype, key channel, key message_id,
             string sender, integer idata, string sdata) {
    llOwnerSay("remote_data");
}
run_time_permissions(integer perm) {
    llOwnerSay("run_time_permissions");
}
sensor(integer num) {
    llOwnerSay("sensor");
}
state_entry() {
    llOwnerSay("state_entry");
}
state_exit() {
    llOwnerSay("state_exit");
}
timer() {
    llOwnerSay("timer");
}
touch(integer total_num) {
    llOwnerSay("touch");
}
touch_start(integer total_num) {
    llOwnerSay("touch_start");
}
touch_end(integer total_num) {
    llOwnerSay("touch_end");
}
}

```

The event sequences in Table D.5 are not close to a complete list, because the events actually generated will depend greatly on what else is happening. However, the table should give you some insight into what is happening when you ask **SL** to do some pretty simple things.

CH02 ADD.

CH06 ADD.

CH08 ADD.

CH12 ADD.

CHAPTER 16

PROJECT 1

PROJECT 2

PROJECT 3

PROJECT 4

PROJECT 5

