Linden Scripting Language Guide

Aaron Brashears

Andrew Meadows

Cory Ondrejka

Doug Soo

Linden Scripting Language Guide

by Aaron Brashears

by Andrew Meadows

by Cory Ondrejka

by Doug Soo

Table of Contents

List of Tables

Chapter 1. Introduction

The Linden Scripting Language (LSL) is a simple, powerful language used to attach behaviors to the objects found in Second Life. It follows the familiar syntax of a c/Java style language, with an implicit state machine for every script.

Multiple scripts may also be attached to the same object, allowing a style of small, single-function scripts to evolve. This leads to scripts that perform specific functions ("hover", "follow", etc.) and allows them to be combined to form new behaviors.

The text of the script is compiled into an executable byte code, much like Java. This byte code is then run within a virtual machine on the simulator. Each script receives a time slice of the total simulator time allocated to scripts, so a simulator with many scripts would allow each individual script less time rather than degrading its own performance. In addition, each script executes within its own chunk of memory, preventing scripts from writing into protected simulator memory or into other scripts, making it much harder for scripts to crash the simulator.

This tutorial introduces the reader to the basic features of LSL, how to edit and apply your scripts, and a complete reference for standard linden constants, events, and library functions.

Chapter 2. Getting Started

You’re probably wondering what you can do with LSL, and how quickly you can do it. We’ll start with some simple examples, disect them, and introduce you the the script development process while we’re at it.

2.1. Hello Avatar

Continuing a long tradition of getting started by looking at a script that says "Hello", we’ll do just that. Though obviously not a particularly useful example on it’s own, this example will introduce us to:

- Creating a basic script
- Script states
- Calling functions
- Script events
- Applying a script to an object

2.1.1. Creating the Script

Start by opening your inventory and selecting ’Create|New Script’ from the inventory pull down menu. This will create an empty script called ’New Script’ in your ’Scripts’ folder. Double click on the text or icon of the script to open the script in the built in editor. When you open the script, the viewer will automatically insert a basic skeleton for lsl. It should look like:

```
default
{
    state_entry()
    {
        llSay(0, "Hello, Avatar!");
    }

    touch_start(integer total_number)
    {
        llSay(0, "Touched.");
    }
}
```

A casual inspection of this script reveals that this scipt probably says ’Hello, Avatar!’ when it enters some state, and it says ’Touched.’ when it is touched. But since this is also probably the first time you have seen a script we’ll disect this short listing, explaining each segment individually.

2.1.2. Default State

```
default
{
...
}
```

All LSL scripts have a simple implicit state machine with one or more states. All scripts must have a default state, so iff there is only one state, it will be the ’default’ state. When a script is first started or reset, it will start out in the default state.

The default state is declared by placing the default at the root level of the document, and marking the beginning with an open brace ’{’ and ending with a close brace ’}’. Because of it’s privileged status, you do not declare that it is fact a state like you normally would with other states.

Every time you enter a state, the script engine will automatically call the state_entry() event and execute the code found there. On state exit, the script engine will automatically call the state_exit() event before calling the next state’s state_entry handler. In our example, we call the llSay() function in state_entry() and do not bother to define a state_exit() handler. the state entry and exit handlers are a convenient place to initialize state data and clean up state specific data such as listen event callback.

You can read more about the default state, and how to create and utilize other states in the states chapter.

2.1.3. Functions

The language comes with well over 200 built in functions which allow scripts and objects to interact with their enviornment. All of the built in functions start with ’ll’.

The example calls the ’llSay()’ function twice. llSay() is used to emit text on the specified channel.

```
llSay( integer channel string text );
```

Say text on channel. Channel 0 is the public chat channel that all avatars see as chat text. Channels 1 to 2,147,483,648 are private channels that aren’t sent to avatars but other scripts can listen for.

You can define your own functions as long as the name does not conflict with a reserved word, built in constant, or built in function.

2.1.4. Touch Event

```
touch_start(integer total_number)
{
    llSay(0, "Touched.");
}
```

There are many events that can be detected in your scripts by declaring a handler. The touch_start() event is raised when a user touches the object through the user interface.

2.1.5. Try it Out

Now that we have seen the default script, and examined it in some detail, it is time to to see the script in action. Save the script by clicking on Save. During the save process, the editor will save the text of the script and compile the script into bytecode and then save that. When you see message 'Compile successful!' in the preview window, you know the compile and save is done.

To test the script you will have to apply it to an object in the world. Create a new object in the world by context clicking in the main world view and selecting Create. When the wand appears, you can create a simple primitive by clicking in the world. Once the object appears, you can drag your newly created script onto the object to start the script.

Soon after dragging the script onto the object, you will see the message `Object: Hello Avatar!`

Make sure the touch event is working by clicking on the object. You should see the message `Touched printed into the chat history.`

2.2. Using The Built-In Editor

The built in editor comes with most of the typical features you would expect from a basic text editor. Highlight text with the mouse, or by holding down the shift key while using the arrow keys. You can cut, copy, paste, and delete your selection using the 'Edit' pull down menu or by pressing the usual shortcut key.

2.3. Using Alternative Editors

Since the built-in editor supports pasting text from the clipboard, you can employ a different editor to edit your scripts, copying them into Second Life when you're ready to save them.

Chapter 3. Basics

Now that we have seen a very simple script in action, we need to look at the our toolchest for writing scripts. The next set of tools we will consider are the basic building blocks for programming a script, and will be used in every non-tribial scipt you write.

3.1. Comments

Commmenting your scripts is a good idea, and will help when you update and modify the script, or when you adapt parts of it into other scripts. Unless the meaning is obvious, you should add comments:

- at the start of the script to explain the purpose of the script
- before every global variable to describe what it holds
- before every global function to describe what it does
- sprinked through your script wherever the code solves a problem that took you more than a few minutes to figure out.

LSL uses Java/C++ style single line comments.

```
// This script toggles a the rotation of an object

// g_is_rotating stores the current state of the rotation. TRUE is
// rotating, FALSE otherwise.
integer g_is_rotating = FALSE;
default
{
    // toggle state during the touch handler
    touch(integer num)
    {
        if(g_is_rotating)
        {
            // turn off rotation
            llTargetOmega(<0,0,1%gt;, 0, 0);
            g_is_rotating = FALSE;
        }
        else
        {
            // rotate around the positive z axis - up.
            llTargetOmega(<0,0,1>, 4, 1);
            g_is_rotating = TRUE;
        }
    }
}
```

3.2. Arithmetic Operations

Most of the common arithmetic operations are supported in lsl, and follow the C/Java syntax.

3.2.1. Assignment

The most common arithmetic operation is assignment, denoted with the '=' sign. Loosely translated, it means, take what you find on the right hand side of the equal sign and assign it to the left hand side. Any expression that evaluates to a basic type can be used as the right hand side of an assignment, but the left hand side must be a normal variable.

All basic types support assignment '=', equality '==' and inequality '!=' operators.

```
// variables to hold a information about the target
key g_target;
vector g_target_postion;
float g_target_distance;

// function that demonstrates assignment
set_globals(key target, vector pos)
{
    g_target = target;
    g_target_position = pos;

    // assignment from the return value of a function
    vector my_pos = llGetPos();
    g_target_distance = llVecDist(g_target_position, my_pos);
}
```

3.2.2. Binary Arithmetic Operators

Binary arithmetic operators behave like a funtion call that accepts two parameters of the same type, and then return that type; however, the syntax is slightly different.

Table 3-1. Binary Arithmetic Operators

Operator	**Meaning**
+	Addition
-	Subtraction
*	Multiplication

Operator	Meaning
/	Division
%	Modulo (remainder)
^	Exclusive OR

Where noted, each type may have a special interpretation of a binary arithmetic operator. See the lsl types section for more details.

3.2.3. Boolean Operators

Table 3-2. Boolean Operators

Operator	Meaning
<	Operator returns TRUE if the left hand side is less than the right hand side.
>	Operator returns TRUE if the left hand side is greater than the right hand side.
<=	Operator returns TRUE if the left hand side is less than or equal to the right hand sid.
>=	Operator returns TRUE if the left hand side is greater than or equal to the right hand sid.
&&	Operator returns TRUE if the left hand side and right hand side are both true.
||	Operator returns TRUE if either the left hand or right hand side are true.
!	Unary operator returns the logical negation of the expression to the right.

3.2.4. Bitwise Operators

Table 3-3. Bitwise Operators

Operator	Meaning
&	Returns the bitwise and of the left and right hand side.
|	Returns the bitwise or of the left and right hand side.
~	Unary operator returns the bitwise complement of the expression to the right.

3.3. Types

Variables, return values, and parameters have type information. LSL provides a small set of basic types that are used througout the language.

LSL Types

integer

A signed, 32-bit integer value with valid range from -2147483648 to 2147483647.

float

An IEEE 32-bit floating point value with values ranging from 1.175494351E-38 to 3.402823466E+38.

key

A unique identifier that can be used to reference objects and agents in Second Life.

vector

3 floats that are used together as a single item. A vector can be used to represent a 3 dimensional position, direction, velocity, force, impulse, or a color. Each component can be accessed via '.x', '.y', and '.z'.

Table 3-4. Vector Arithmetic Operators

Operator	Meaning
+	Add two vectors together
-	Subract one vector from another
*	Vector dot product
%	Vector cross product

rotation

4 floats that are used together as a single item to represent a roation. This data is interpreted as a quaternion. Each compnent can be accessed via '.x', '.y', '.z', and '.w'.

Table 3-5. Rotation Arithmetic Operators

Operator	Meaning
+	Add two rotations together
-	Subract one rotation from another
*	Rotate the first rotation by the second
/	Rotate the first rotation by the inverse of the second

list

A heterogeneous list of the other data types. Lista are created via comma separated values of the other data types enclosed by '[' and ']'.

```
string StringVar = "Hello, Carbon Unit";
list MyList = [ 1234, ZERO_ROTATION, StringVar ];
```

Yields the list: `[ 1234, <0,0,0,1>, "Hello, Carbon Unit" ]`

Lists can be combined with other lists. For example:

```
MyList = 3.14159 + MyList;
```

Yields the list: `[ 3.14159, 1234, <0,0,0,1>, "Hello, Carbon Unit" ]` And similarly,

```
MyList = MyList + MyList;
```

Yields: `[ 3.14159, 1234, <0,0,0,1>, "Hello, Carbon Unit", 3.14159, 1234, <0,0,0,1>, "Hello, Carbon Unit" ]`

Library functions exist used to copy data from lists, sort lists, copy/remove sublists.

3.3.1. Type Conversion

Type conversion can either occur implicitly or explicitly. Explicit type casts are accomplished using C syntax:

```
float foo_float = 1.0;
integer  foo_int = (integer)foo_float;
```

3.3.1.1. Implicit Casting

LSL only supports two implicit type casts: integer to float and string to key. Thus, any place you see a float specified you can supply an integer, and any place you see a key specified, you can supply a string.

3.3.1.2. Explicit Casting

LSL supports the following explicit casts:

- Integer to String
- Float to Integer
- Float to String
- Vector to String
- Rotation to String

- Integer to List
- Float to List
- Key to List
- String to List
- Vector to List
- Rotation to List
- String to Integer
- String to Float
- String to Vector
- String to Rotation

3.4. Global Functions

Global functions are also declared much like Java/C, with the exception that no ’void’ return value exists. Instead, if no return value is needed, just don?t specify one:

```
make_physical_and_spin(vector torque)
{
    // double the torque
    vector double_torque = 2.0*torque;
    llSetState(STATUS_PHYSICS, TRUE);
    llApplyTorque(double_torque);
}
```

3.5. Global Variables

Global variables and functions are accessible from anywhere in the file. Global variables are declared much like Java or C, although only one declaration may be made per line:

```
vector gStartPosition;
```

Global variables may also be initialized if desired, although uninitialized global and local variables are initialized to legal zero values:

```
vector gStartPosition = <10.0,10.0,10.0>
```

3.6. Local Variables

Local variables are scoped below their declaration within the block of code they are declared in and may be declared within any block of code. Thus the following code is legal and will work like C:

```
integer test_function()
{
    // Test vector that we can use anywhere in the function
    vector test = <1,2,3>;
    integer j;
    for (j = 0; j < 10; j++)
    {
        // This vector is a different variable than the one declared above
        // This IS NOT good coding practice
        vector test = <j, j, j>;
    }
    // this test fails
    if (test == <9,9,9>)
    {
        // never reached
    }
}
```

Chapter 4. Flow Control

but the most basic scripts will require some kind of flow contol. LSL comes with a complete complement of constructs meant to deal with conditional processing, looping, as well as simply jumping to a different point in the script.

4.1. Conditional Statements

The ’if’ statement operates and has the same syntax as the Java/C version.

```
check_message(string message)
{
    if(message == "open")
    {
        open();
    }
    else if(message == "close")
    {
        close();
    }
    else
    {
        llSay(0, "Unknown command: " + message);
    }
}
```

The statements between the open and close curly brace are performed if the conditional inside the parentheses evaluates to a non-zero integer. Once a conditional is determined to be true (non-zero), no further processing of ’else’ conditionals will be considered. The NULL_KEY constant is counted as FALSE by conditional expressions.

There can be zero or more ’else if’ statements, and an optional final ’else’ to handle the case when none of the if statements evaluate to a non-zero integer.

The usual set of integer arithmetic and comparison operators are available.

```
// a function that accepts some information about its environment and
// determines the the ’best’ next step. This kind of code might be
// part of a simple box meant to move close to an agent and attach to
// them once near. This code sample relies on the standard linden
// library functions as well as two other methods not defined here.
assess_next_step(integer perm, integer attached, integer balance, float dist)
{
```

```
    string msg;
    if(!attached)
    {
        if((perm & PERMISSION_ATTACH) && (dist < 10.0))
        {
             attach();
        }
        else if((dist > 10.0) || ((dist > 20.0) && (balance > 1000)))
        {
            move_closer();
        }
        else
        {
            llRequestPermissions(llGetOwner(), PERMISSION_ATTACH);
        }
    }
}
```

4.2. Loop Constructs

Loops are a basic building block of most useful programming languages, and LSL offers the the same loop costructs as found in Java or C.

4.2.1. for loop

A for loop is most useful for when you know how many times you need to iterate over an operation. Just like a Java or C for loop, the parentheses have three parts, the initializer, the continuation condition, and the increment. The loop continues while the middle term evaluates to true, and the increment step is performed at the end of every loop.

```
// move a non-physical block smootly upward (positive z) the the total
// distance specified divided into steps discrete moves.
move_up(float distance, integer steps)
{
    float step_distance = distance / (float)steps;
    vector offset = <0.0, 0.0, step_distance>;
    vector base_pos = llGetPos();
    integer i;
    for(i = 0; i <= steps; ++i)
    {
        llSetPos(base_pos + i * offset);
        llSleep(0.1);
    }
}
```

4.2.2. do-while loop

The do-while loop construct is most useful when you are sure that you want to perform an operation at least once, but you are not sure how many times you want to loop. The syntax is the same as you would find in a Java or C program. A simple english transation would be 'do the code inside the curly braces and continue doing it if the statement after the while is true.

```
// output the name of all inventory items attached to this object
talk_about_inventory(integer type)
{
    string name;
    integer i = 0;
    integer continue = TRUE;
    do
    {
        name = llGetInventoryName(type, i);
        if(llStringLength(name) > 0)
        {
            llSay(0, "Inventory " + (string)i + ": " + name);
        }
        else
        {
            llSay(0, "No more inventory items");
            continue = FALSE;
        }
    } while(continue);
}
```

4.2.3. while loop

The while loop behaves similarly to the do-while loop, excpet it allows you to exit the loop without doing a single iteration inside.

```
mention_inventory_type(integer type)
{
    integer i = llGetInventoryNumber(type);
    while(i--)
    {
        llSay(0, "item: " + llGetInventory(i));
    }
}
```

4.3. Jumps

A jump is used to move the running script to a new point inside of a function or event handler. You cannot jump into other functions or event handlers. Usually, you will want to use a jump for in situations where the if..else statements would become too cumbersome. For example, you may want to check several preconditions, and exit if any of them are not met.

```
attach_if_ready(vector target_pos)
{
    // make sure we have permission
    integer perm = llGetPerm();
    if(!(perm & PERMISSION_ATTACH))
    {
        jump early_exit;
    }

    // make sure we’re 10 or less meters away
    vector pos = llGetPos()
    float dist = llVecDist(pos, target_pos);
    if(dist > 10.0)
    {
        jump early_exit;
    }

    // make sure we’re roughly pointed toward the target.
    // the calculation of max_cos_theta could be precomputed
    // as a constant, but is manually computed here to
    // illustrate the math.
    float max_cos_theta = llCos(PI / 4.0);
    vector toward_target = llVecNorm(target_pos - pos);
    rotation rot = llGetRot();
    vector fwd = llRot2Fwd(rot);
    float cos_theta = toward_target * fwd;
    if(cos_theta > max_cos_theta)
    {
        jump early_exit;
    }

    // at this point, we’ve done all the checks.
    attach();

    @early_exit;
}
```

4.4. State Change

State change allow you to move through the lsl virtual machine’s flexible state machine by transitioning your script to and from user defined states and the default state. You can define your own script state by placing the

keyword ’state’ before its name and enclosing the event handlers with open and close curly braces (’{’ and ’}’.) You can invoke the transition to a new state by calling it with the syntax: ’state <statename>’.

```
default
{
    state_entry()
    {
        llSay(0, "I am in the default state");
        llSetTimer(1.0);
    }

    timer()
    {
        state SpinState;
    }
}

state SpinState
{
    state_entry()
    {
        llSay(0, "I am in SpinState!");
        llTargetOmega(<0,0,1>, 4, 1.0);
        llSetTimer(2.0);
    }

    timer()
    {
        state default;
    }

    state_exit()
    {
        llTargetOmega(<0,0,1>, 0, 0.0);
    }
}
```

Chapter 5. States

All scripts must have a ’default’ state, which is the first state entered when the script starts. States contain event handlers that are triggered by the LSL virtual machine. All states must supply at least one event handler - it’s not really a state without one.

When state changes, all callback settings are retained and all pending events are cleared. For example, if you have set a listen callback in the default state and do not remove it during state_exit(), the listen callback will be called in your new state if a new listen event passes the filter set in the default state.

5.1. state_entry()

The state_entry event occurs whenever a new state is entered, including program start, and is always the first event handled. No data is passed to this event handler.

You will usually want to set callbacks for things such as timers and seonsor in the state_entry() callback of the state to put your object into a useful condition for that state.

Warning

It is a common mistake to assume that the state_entry() callback is called when you rez an object out of your inventory. When you derez an object into your inventory the current state of the script is saved, so there will not be a call to state_entry() during the rez. If you need to provide startup code every time an object is created, you should create a global function and call it from both state_entry() and the on_rez() callbacks.

```
// global initialization function.
init()
{
    // Set up a listen callback for whoever owns this object.
    key owner = llGetOwner();
    llListen(0, "", owner, "");
}

default
{
    state_entry()
    {
        init();
    }

    on_rez(integer start_param)
    {
        init();
    }

    listen(integer channel, string name, key id, string message)
    {
        llSay(0, "Hi " + name + "! You own me.");
    }
}
```

5.2. state_exit()

The state_entry event occurs whenever the state command is used to transition to another state. It is handled before the new state's state_entry event.

You will want to provide a state_exit() if you need to clean up any events that you have requested in the current state, but do not expect in the next state.

```
default
{
    state_entry()
    {
        state TimerState;
    }
```

```
}

state TimerState
{
    state_entry()
    {
        // set a timer event for 5 seconds in the future.
        llSetTimerEvent(5.0);
    }

    timer()
    {
        llSay(0, "timer");
        state ListenState;
    }

    state_exit()
    {
        // turn off future timer events.
        llSetTimerEvent(0.0);
    }
}

integer g_listen_control;

state ListenState
{
    state_entry()
    {
        // listen for anything on the public channel
        g_listen_control = llListen(0, "", NULL_KEY, "");
    }

    listen(integer channel, string name, key id, string message)
    {
        llSay(0, "listen");
        state TimerState;
    }

    state_exit()
    {
        // turn off the listener
        llListenRemove(g_listen_control);
    }
}
```

The state_exit() handler is not called when an object is being deleted - all callbacks, handlers, sounds, etc, will be cleaned up automatically for you.

5.3. States vs. Global variables

A state and a set of global variables can serve the same purpose, and each can be expressed in terms of the other. In general, you should prefer the use of states over global variables since states allow you to immediately assume script state without making comparisons. The less comparisons a script makes, the more regular code statements it can run.

Chapter 6. Math

6.1. Tables of Functions

Table 6-1. Trigonometry Functions

Function
llAbs
llAcos
llAsin
llAtan2
llCeil
llCos
llFabs
llFloor
llFrand
llPow
llRound
llSin
llSqrt
llTan

Table 6-2. Vector Functions

Function
llVecDist
llVecMag
llVecNorm

Table 6-3. Rotation Functions

Function
llAngleBetween
llAxes2Rot
llAxisAngle2Rot
llEuler2Rot
llRot2Angle
llRot2Axis
llRot2Euler
llRot2Fwd
llRot2Left
llRot2Up
llRotBetween

Chapter 7. Strings

7.1. Tables of Functions

Table 7-1. String Functions

Function
llDeleteSubString
llGetSubString
llInsertString
llStringLength
llSubStringIndex
llToLower
llToUpper

Chapter 8. Lists

8.1. Tables of Functions

Table 8-1. List Functions

Function
llCSV2List
llDeleteSubList
llGetListEntryType
llGetListLength
llList2CSV
llList2Float
llList2Integer
llList2Key
llList2List
llList2ListStrided
llList2Rot
llList2String
llList2Vector
llListFindList
llListInsertList
llListRandomize
llListSort
llParseString2List

Chapter 9. Communication

9.1. Tables of Functions

Table 9-1. In World Functions

Function
llListen
llListenControl
llListenRemove
llSay
llShout
llWhisper

Table 9-2. Messaging Functions

Function
llEmail
llGetNextEmail
llInstantMessage

Chapter 10. Inventory

10.1. Tables of Functions

Table 10-1. Inventory Functions

Function
llAllowInventoryDrop
llGetInventoryKey
llGetInventoryName
llGetInventoryNumber
llGetNotecardLine
llGiveInventory
llGiveInventoryList
llRemoveInventory
llRequestInventoryData
llRezObject

Chapter 11. Vehicles

Vehicles are a new feature now available for use through LSL. This chapter will cover the basics of how vehicles work, the terms used when describing vehicles, and a more thorough examination of the api available.

There are several ways to make scripted objects move themselves around. One way is to turn the object into a "vehicle". This feature is versatile enough to make things that slide, hover, fly, and float. Some of the behaviors that can be enabled are:

- deflection of linear and angular velocity to preferred axis of motion
- asymmetric linear and angular friction
- hovering over terrain/water or at a global height
- banking on turns
- linear and angular motor for push and turning

11.1. Overview

Each scripted object can have one vehicle behavior that is configurable through the llSetVehicleType, llSetVehicleFloatParam, llSetVehicleVectorParam, llSetVehicleRotationParam, llSetVehicleFlags, and llRemoveVehicleFlags library calls.

These script calls are described in more detail below, but the important thing to notice here is that the vehicle behavior has several parameters that can be adjusted to change how the vehicle handles. Depending on the values chosen the vehicle can veer like a boat in water, or ride like a sled on rails.

Setting the vehicle flags allow you to make exceptions to some default behaviors. Some of these flags only have an effect when certain behaviors are enabled. For example, the VEHICLE_FLAG_HOVER_WATER_ONLY will make the vehicle ignore the height of the terrain, however it only makes a difference if the vehicle is hovering.

11.2. Warnings

Vehicles are new in Second Life 1.1 and some of the details of their behavior may be changed as necessary to ensure stability and user safety. In particular, many of the limits and defaults described in the appendices will probably change and should not be relied upon in the long term.

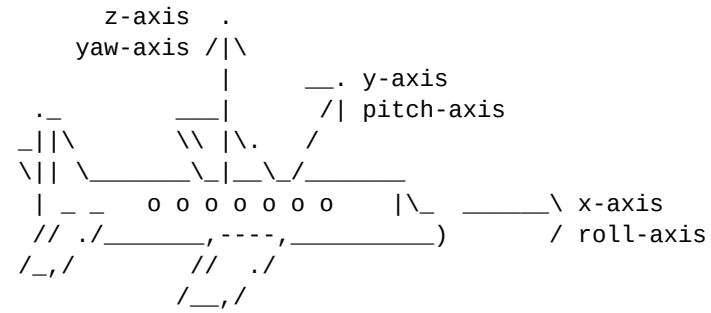
It is not recommended that you mix vehicle behavior with some of the other script calls that provide impulse and forces to the object, especially llSetBuoyancy, llSetForce, llSetTorque, and llSetHoverHeight.

While the following methods probably don't cause any instabilities, their behavior may conflict with vehicles and cause undesired and/or inconsistent results, so use llLookAt, llRotLookAt, llMoveToTarget, and llTargetOmega at your own risk.

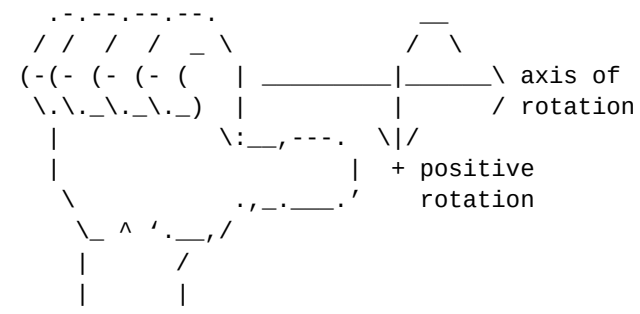
If you think you have found a bug relating to how vehicle's work, one way to submit the problem is to give a copy of the vehicle and script to Andrew Linden with comments or a notecard describing the problem. Please name all submissions "Bugged Vehicle XX" where XX are your Second Life initials. The vehicle and script will be examined at the earliest convenience.

11.3. Definitions

The terms "roll", "pitch", and "yaw" are often used to describe the modes of rotations that can happen to a airplane or boat. They correspond to rotations about the local x-, y-, and z-axis respectively.



The right-hand-rule, often introduced in beginning physics courses, is used to define the direction of positive rotation about any axis. As an example of how to use the right hand rule, consider a positive rotation about the roll axis. To help visualize how such a rotation would move the airplane, place your right thumb parallel to the plane's roll-axis such that the thumb points in the positive x-direction, then curl the four fingers into a fist. Your fingers will be pointing in the direction that the plane will spin.



Many of the parameters that control a vehicle's behavior are of the form: VEHICLE_BEHAVIOR_TIMESCALE. A behavior's "timescale" can usually be understood as the time for the behavior to push, twist, or otherwise affect the vehicle such that the difference between what it is doing, and what it is supposed to be doing, has been reduced to 1/e of what it was, where "e" is the natural exponent (approximately 2.718281828). In other words, it is the timescale for exponential decay toward full compliance to the desired behavior. When you want the vehicle to be very responsive use a short timescale of one second or less, and if you want to disable a behavior then set the timescale to a very large number like 300 (5 minutes) or more. Note, for stability reasons, there is usually a limit to how small a timescale is allowed to be, and is usually on the order of a tenth of a second. Setting a timescale to zero is safe and is always equivalent to setting it to its minimum. Any feature with a timescale can be effectively disabled by setting the timescale so large that it would take them all day to have any effect.

11.4. Setting the Vehicle Type

Before any vehicle parameters can be set the vehicle behavior must first be enabled. It is enabled by calling llSetVehicleType with any VEHICLE_TYPE_*, except VEHICLE_TYPE_NONE which will disable the vehicle. See the vehicle type constants section for currently available types. More types will be available soon.

Setting the vehicle type is necessary for enabling the vehicle behavior and sets all of the parameters to its default values. For each vehicle type listed we provide the corresponding equivalent code in long format. Is is *important* to realize that the defaults are *not* the optimal settings for any of these vehicle types and that they will definitely be changed in the future. Do not rely on these values to be constant until specified.

Should you want to make a unique or experimental vehicle you will still have to enable the vehicle behavior with one of the default types first, after which you will be able to change any of the parameters or flags within the allowed ranges.

Setting the vehicle type does not automatically take controls or otherwise move the object. However should you enable the vehicle behavior while the object is free to move and parked on a hill then it may start to slide away.

We're looking for new and better default vehicle types. If you think you've found a set of parameters that make a better car, boat, or any other default type of vehicle then you may submit your proposed list of settings to Andrew Linden via a script or notecard.

11.5. Linear and Angular Deflection

A common feature of real vehicles is their tendency to move along "preferred axes of motion". That is, due to their wheels, wings, shape, or method of propulsion they tend to push or redirect themselves along axes that are static in the vehicle's local frame. This general feature defines a class of vehicles and included in this category a common dart is a "vehicle": it has fins in the back such that if it were to tumble in the air it would eventually align itself to move point-forward -- we'll call this alignment effect *angular deflection*.

A wheeled craft exhibits a different effect: when a skateboard is pushed in some direction it will tend to redirect the resultant motion along that which it is free to roll -- we'll call this effect *linear deflection*.

So a typical Second Life vehicle is an object that exhibits linear and/or angular deflection along the "preferential axes of motion". The default preferential axes of motion are the local x- (at), y- (left), and z- (up) axes of the *local frame* of the vehicle's root primitive. The deflection behaviors relate to the x-axis (at): linear deflection will tend to rotate its velocity until it points along it's positive local x-axis while the angular deflection will tend to reorient the vehicle such that it's x-axis points in the direction that it is moving. The other axes are relevant to vehicle behaviors that are described later, such as the vertical attractor which tries to keep a vehicle's local z-axis pointed toward the world z-axis (up). The vehicle axes can be rotated relative to the object's actual local axes by using the VEHICLE_REFERENCE_FRAME parameter, however that is an advanced feature and is covered in detail in a later section of these documents.

Depending on the vehicle it might be desirable to have lots of linear and/or angular delfection or not. The speed of the deflections are controlled by setting the relevant parameters using the llSetVehicleFloatParam script call. Each variety of deflection has a "timescale" parameter that determines how quickly a full deflection happens. Basically the timescale it the time coefficient for exponential decay toward full deflection. So, a vehicle that deflects quickly should have a small timescale. For instance, a typical dart might have a angular deflection timescale of a couple of seconds but a linear deflection of several seconds; it will tend to reorient itself before it changes direction. To set the deflection timescales of a dart you might use the lines below:

```
llSetVehicleFloatParam(VEHICLE_ANGULAR_DEFLECTION_TIMESCALE, 2.0);
llSetVehicleFloatParam(VEHICLE_LINEAR_DEFLECTION_TIMESCALE, 6.0);
```

Each variety of deflection has an "efficiency" parameter that is a slider between 0.0 and 1.0. Unlike the other efficiency parameter of other vehicle behaviors, the deflection efficiencies do not slide between "bouncy" and "damped", but instead slide from "no deflection whatsoever" (0.0) to "maximum deflection" (1.0). That is, they behave much like the deflection timescales, however they are normalized to the range between 0.0 and 1.0.

11.6. Moving the Vehicle

Once enabled, a vehicle can be pushed and rotated by external forces and/or from script calls such as llApplyImpulse, however linear and angular motors have been built in to make motion easier and smoother. Their directions can be set using the llSetVehicleVectorParam call. For example, to make the vehicle try to move at 5 meters/second along its local x-axis (the default look-at direction) you would put the following line in your script:

```
llSetVehicleVectorParam(VEHICLE_LINEAR_MOTOR_DIRECTION, <5, 0, 0>);
```

To prevent vehicles from moving too fast the magnitude of the linear motor is clamped to be no larger than about 30 meters/second. Note that this is clamped mostly because of limitations of the physics engine, and may be raised later when possible.

Setting the motor speed is not enough to enable all interesting vehicles. For example, some will want a car that immediately gets up to the speed they want, while others will want a boat that slowly climbs up to its maximum velocity. To control this effect you can use the VEHICLE_LINEAR_MOTOR_TIMESCALE parameter. Basically the "timescale" of a motor is the time constant for the vehicle to exponentially accelerate toward its full speed.

What would happen if you were to accidentally set the vehicle's linear velocity to maximum possible speed and then let go? It would run away and never stop, right? Not necessarily: an automatic "motor decay" has been built in such that all motors will gradually decrease their effectiveness after being set.

Each time the linear motor's vector is set its "grip" immediately starts to decay exponentially with a timescale determined by the VEHICLE_LINEAR_MOTOR_DECAY_TIMESCALE, such that after enough time the motor ceases to have any effect. This decay timescale serves two purposes. First, since it *cannot* be set longer than 120 seconds, and is *always* enabled it gaurantees that a vehicle will not push itself about forever in the absence of active control (from keyboard commands or some logic loop in the script). Second, it can be used to push some vehicles around using a simple impulse model. That is, rather than setting the motor "on" or "off" depending on whether a particular key is pressed "down" or "up" the decay timescale can be set short and the motor can be set "on" whenever the key transitions from "up" to "down" and allowed to automatically decay.

Since the motor's effectiveness is reset whenever the motor's vector is set, then setting it to a vector of length zero is different from allowing it to decay completely. The first case will cause the vehicle to try to reach zero velocity, while the second will leave the motor impotent.

The two motor timescales have very similar names, but have different effects, so try not to get them confused. VEHICLE_LINEAR_MOTOR_TIMESCALE is the time for motor to "win", and VEHICLE_LINEAR_MOTOR_DECAY_TIMESCALE is the time for the motor's "effectiveness" to decay toward zero. If you set one when you think you are changing the other you will have frustrating results. Also, if the motor's decay timescale is shorter than the regular timescale, then the effective magnitude of the motor vector will be diminished.

11.7. Steering the Vehicle

Much like the linear motor, there is also an angular motor that is always on, and whose direction and magnitude can be set. For example, to make a vehicle turn at 5 degrees/sec around it's local z-axis (its up-axis) you might add the following lines to its script:

```
vector angular_velocity = <0, 0, 5 * PI / 180>;
llSetVehicleVectorParam(VEHICLE_ANGULAR_MOTOR_DIRECTION, angular_velocity);
```

The magnitude of the angular motor is capped to be no more than two rotations per second (4*PI radians/sec).

Also like the linear motor it has an efficiency parameter, VEHICLE_ANGULAR_MOTOR_TIMESCALE, and a motor decay parameter, VEHICLE_ANGULAR_MOTOR_DECAY_TIMESCALE, which is set to the

maximum possible value of 120 seconds by default.

When steering a vehicle you probably don’t want it to turn very far or for very long. One way to do it using the angular motor would be to leave the decay timescale long, enable a significant amount of angular friction (to quickly slow the vehicle down when the motor is turned off) then set the angular motor to a large vector on a key press, and set it to zero when the key is released. Another way to do it is to set the VEHICLE_ANGULAR_MOTOR_DECAY_TIMESCALE to a short value and push the vehicle about with a more impulsive method that sets the motor fast on a key press down (and optionally setting the motor to zero on a key up) relying on the automatic exponential decay of the motor’s effectiveness rather than a constant angular friction.

Setting the angular motor to zero magnitude is different from allowing it to decay. When the motor completely decays it no longer affects the motion of the vehicle, however setting it to zero will reset the "grip" of the vehicle and will make the vehicle try to achieve zero angular velocity.

For some vehicles it will be possible to use the "banking feature" to turn. "Banking" is what airplanes and motorcycles do when they turn. When a banking vehicle twists about its roll-axis there is a resultant spin around its yaw-axis. Banking is only available when using the "vertical attractor" which is described below.

11.8. The Vertical Attractor

Some vehicles, like boats, should always keep their up-side up. This can be done by enabling the "vertical attractor" behavior that springs the vehicle’s local z-axis to the world z-axis (a.k.a. "up"). To take advantage of this feature you would set the VEHICLE_VERTICAL_ATTRACTION_TIMESCALE to control the period of the spring frequency, and then set the VEHICLE_VERTICAL_ATTRACTION_EFFICIENCY to control the damping. An efficiency of 0.0 will cause the spring to wobble around its equilibrium, while an efficiency of 1.0 will cause the spring to reach it’s equilibrium with exponential decay.

```
llSetVehicleVectorParam(VEHICLE_VERTICAL_ATTRACTION_TIMESCALE, 4.0);
llSetVehicleVectorParam(VEHICLE_VERTICAL_ATTRACTION_EFFICIENCY, 0.5);
```

The vertical attractor is disabled by setting its timescale to anything larger than 300 seconds.

Note that by default the vertical attractor will prevent the vehicle from diving and climbing. So, if you wanted to make a airplane you would probably want to unlock the attractor around the pitch axis by setting the VEHICLE_FLAG_LIMIT_ROLL_ONLY bit:

```
llSetVehicleFlags(VEHICLE_FLAG_LIMIT_ROLL_ONLY);
```

11.9. Banking

The vertical attractor feature must be enabled in order for the banking behavior to function. The way banking works is this: a rotation around the vehicle’s roll-axis will produce a angular velocity around the yaw-axis, causing the vehicle to turn. The magnitude of the yaw effect will be proportional to the VEHICLE_BANKING_COEF, the angle of the roll rotation, and sometimes the vehicle’s velocity along it’s preferred axis of motion.

The VEHICLE_BANKING_COEF can vary between -1 and +1. When it’s positive then any positive rotation (by the right-hand rule) about the roll-axis will effect a (negative) torque around the yaw-axis, making it turn to the right -- that is the vehicle will lean into the turn, which is how real airplanes and motorcycle’s work. Negating the banking coefficient will make it so that the vehicle leans to the outside of the turn (not very "physical" but might allow interesting vehicles so why not?).

The VEHICLE_BANKING_MIX is a fake (i.e. non-physical) parameter that is useful for making banking vehicles do what you want rather than what the laws of physics allow. For example, consider a real motorcycle... it must be moving forward in order for it to turn while banking, however video-game motorcycles are often configured to turn in place when at a dead stop -- because they’re often easier to control that way using the limited interface of the keyboard or game controller. The VEHICLE_BANKING_MIX enables combinations of both realistic and non-realistic banking by fuctioning as a slider between a banking that is correspondingly totally static (0.0) and totally dynamic (1.0). By "static" we mean that the banking effect depends only on the vehicle’s rotation about it’s roll-axis compared to "dynamic" where the banking is also proportional to it’s velocity along it’s roll-axis. Finding the best value of the "mixture" will probably require trial and error.

The time it takes for the banking behavior to defeat a pre-existing angular velocity about the world z-axis is determined by the VEHICLE_BANKING_TIMESCALE. So if you want the vehicle to bank quickly then give it a banking timescale of about a second or less, otherwise you can make a sluggish vehicle by giving it a timescale of several seconds.

11.10. Friction Timescales

VEHICLE_LINEAR_FRICTION_TIMESCALE is a vector parameter that defines the timescales for the vehicle to come to a complete stop along the three local axes of the vehicle’s reference frame. The timescale along each axis is independent of the others. For example, a sliding ground car would probably have very little friction along its x- and z-axes (so it can easily slide forward and fall down) while there would usually significant friction along its y-axis:

```
llSetVehicleVectorParam(VEHICLE_LINEAR_FRICTION_TIMESCALE, <1000, 1000, 3>);
```

Remember that a longer timescale corresponds to a weaker friction, hence to effectively disable all linear friction you would set all of the timescales to large values.

Setting the linear friction as a scalar is allowed, and has the effect of setting all of the timescales to the same value. Both code snippets below are equivalent, and both make friction negligible:

```
// set all linear friction timescales to 1000
llSetVehicleVectorParam(VEHICLE_LINEAR_FRICTION_TIMESCALE, <1000, 1000, 1000>);
```

```
// same as above, but fewer characters
llSetVehicleFloatParam(VEHICLE_LINEAR_FRICTION_TIMESCALE, 1000);
```

VEHICLE_ANGULAR_FRICTION_TIMESCALE is also a vector parameter that defines the timescales for the vehicle to stop rotating about the x-, y-, and z-axes, and are set and disabled in the same way as the linear friction.

11.11. Buoyancy

The vehicle has a built-in buoyancy feature that is independent of the llSetBuoyancy call. It is recommended that the two buoyancies do not mix! To make a vehicle buoyant, set the VEHICLE_BUOYANCY parameter to something between 0.0 (no buoyancy whatsoever) to 1.0 (full anti-gravity).

The buoyancy behavior is independent of hover, however in order for hover to work without a large offset of the VEHICLE_HOVER_HEIGHT, the VEHICLE_BUOYANCY should be set to 1.0.

It is not recommended that you mix vehicle buoyancy with the llSetBuoyancy script call. It would probably cause the object to fly up into space.

11.12. Hover

The hover behavior is enabled by setting the VEHICLE_HOVER_TIMESCALE to a value less than 300 seconds; larger timescales totally disable it. Most vehicles will work best with short hover timescales of a few seconds or less. The shorter the timescale, the faster the vehicle will slave to is target height. Note, that if the values of VEHICLE_LINEAR_FRICTION_TIMESCALE may affect the speed of the hover.

Hover is independent of buoyancy, however the VEHICLE_BUOYANCY should be set to 1.0, otherwise the vehicle will not lift itself off of the ground until the VEHICLE_HOVER_HEIGHT is made large enough to counter the acceleration of gravity, and the vehicle will never float all the way to its target height.

The VEHICLE_HOVER_EFFICIENCY can be thought of as a slider between bouncy (0.0) and smoothed (1.0). When in the bouncy range the vehicle will tend to hover a little lower than its target height and the VEHICLE_HOVER_TIMESCALE will be approximately the oscillation period of the bounce (the real period will tend to be a little longer than the timescale).

For performance reasons, until improvements are made to the Second Life physics engine the vehicles can only hover over the terrain and water, so they will not be able to hover above objects made out of primitives, such as bridges and houses. By default the hover behavior will float over terrain and water, however this can be changed by setting some flags:

If you wanted to make a boat you should set the VEHICLE_HOVER_WATER_ONLY flag, or if you wanted to drive a hover tank under water you would use the VEHICLE_HOVER_TERRAIN_ONLY flag instead. Finally, if you wanted to make a submarine or a balloon you would use the VEHICLE_HOVER_GLOBAL_HEIGHT. Note that the flags are independent of each other and that setting two contradictory flags will have undefined behavor. The flags are set using the script call llSetVehicleFlags().

The VEHICLE_HOVER_HEIGHT determines how high the vehicle will hover over the terrain and/or water, or the global height, and has a maximum value of 100 meters. Note that for hovering purposes the "center" of the vehicle is its "center of mass" which is not always obvious to the untrained eye, and it changes when avatar’s sit on the vehicle.

11.13. Reference Frame

The vehicle relies on the x- (at), y- (left), and z- (up) axes in order to figure out which way it preferres to move and which end is up. By default these axes are identical to the local axes of the root primitive of the object, however this means that the vehicle’s root primitive must, by default, be oriented to agree with the designed at, left, and up axes of the vehicle. But, what if the vehicle object was already pre-built with the root primitive in some non-trivial orientation relative to where the vehicle as a whole should move? This is where the VEHICLE_REFERENCE_FRAME parameter becomes useful; the vehicle’s axes can be arbitrarily reoriented by setting this parameter.

As an example, suppose you had built a rocket out of a big cylinder, a cone for the nose, and some stretched cut boxes for the fins, then linked them all together with the cylinder as the root primitive. Ideally the rocket would move nose-first, however the cylinder’s axis of symmetry is its local z-axis while the default "at-axis" of the vehicle, the axis it will want to deflect to forward under angular deflection, is the local x-axis and points out from the curved surface of the cylinder. The script code below will rotate the vehicle’s axes such that the local z-axis becomes the "at-axis" and the local negative x-axis becomes the "up-axis":

```
// rotate the vehicle frame -PI/2 about the local y-axis (left-axis)
rotation rot = llEuler2Rot(0, PI/2, 0);
llSetVehicleRotationParam(VEHICLE_REFERENCE_FRAME, rot);
```

Another example of how the reference frame parameter could be used is to consider flying craft that uses the vertical attractor for stability during flying but wants to use VTOL (vertical takeoff and landing). During flight the craft’s dorsal axis should point up, but during landing its nose-axis should be up. To land the vehicle: while the vertical attractor is in effect, rotate the existing VEHICLE_REFERENCE_FRAME by +PI/2 about the left-axis, then the vehicle will pitch up such that it’s nose points toward the sky. The vehicle could be allowed to fall to the landing pad under friction, or a decreasing hover effect.

Appendix A. Linden Library Functions

Complete listing of the Linden Library function calls available in lsl.

A.1. llAbs

```
integer llAbs(integer val);
```

Returns the absolute value of *val*.

A.2. llAcos

```
float llAcos(float val);
```

Returns the arccosine in radians of *val*.

A.3. llAddToLandPassList

```
llAddToLandPassList(key avatar, float hours);
```

Add *avatar* to the land pass list for *hours*.

A.4. llAdjustSoundVolume

```
llAdjustSoundVolume(float volume);
```

Adjusts the volume of the currently playing attached sound started with llPlaySound or llLoopSound. This function Has no effect on sounds started with llTriggerSound.

A.5. llAllowInventoryDrop

```
llAllowInventoryDrop(integer add);
```

If *add* == TRUE, users that do no have object modify permissions can still drop inventory items onto object.

A.6. llAngleBetween

```
float llAngleBetween(rotation a, rotation b);
```

Returns the angle in radians between rotations *a* and *b*.

A.7. llApplyImpulse

```
llApplyImpulse(vector force, integer local);
```

Applies the *impulse* in local coordinates if *local* == TRUE. Otherwise the impulse is applied in global coordinates. This function only works on physical objects.

A.8. llApplyRotationalImpulse

```
llApplyRotationalImpulse(vector force, integer local);
```

Applies a rotational *impulse* force in local coordinates if *local* == TRUE. Otherwise the impulse is applied in global coordinates. This function only works on physical objects.

A.9. llAsin

```
float llAsin(float val);
```

Returns the arcsine in radians of *val*.

A.10. llAtan2

```
float llAtan2(float y, float x);
```

returns the arctangent2 of *y*, *x*

A.11. llAttachToAvatar

```
llAttachToAvatar(key avatar, integer attachment);
```

Attach to *avatar* at point *attachment*. Requires the PERMISSION_ATTACH runtime permission.

A.12. llAvatarOnSitTarget

```
key llAvatarOnSitTarget(void);
```

If an avatar is sitting on the sit target, return the avatar's key, NULL_KEY otherwise. This only will detect avatars sitting on sit targets defined with llSitTarget.

A.13. llAxes2Rot

```
rotation llAxes2Rot(vector fwd, vector left, vector up);
```

Returns the rotation represented by coordinate axes *fwd*, *left*, and *up*.

A.14. llAxisAngle2Rot

```
rotation llAxisAngle2Rot(vector axis, float angle);
```

Returns the rotation generated *angle* about *axis*.

A.15. llBreakAllLinks

```
llBreakAllLinks(void);
```

Delinks all objects in the link set. Requires the permission PERMISSION_CHANGE_LINKS be set.

A.16. llBreakLink

```
llBreakLink(integer linknum);
```

Delinks the object with the given *link* number. Requires permission PERMISSION_CHANGE_LINKS be set.

A.17. llCSV2List

```
list llCSV2List(string src);
```

Create a list from a string of comma separated values specified in *src*.

A.18. llCeil

```
integer llCeil(float val);
```

Returns largest integer value >= *val*.

A.19. llCloud

```
float llCloud(vector offset);
```

Returns the cloud density at the object position + *offset*.

A.20. llCollisionFilter

```
llCollisionFilter(string name, key id, iteger accept);
```

If *accept* == TRUE, only accept collisions with objects *name* and *id*, otherwise with objects not *name* or *id*. Specify an empty string or NULL_KEY to not filter on the corresponding parameter.

A.21. llCollisionSound

```
llCollisionSound(string impact_sound, float impact_volume);
```

Suppress default collision sounds, replace default impact sounds with *impact_sound* found in the object inventory. Supply an empty string to suppress collision sounds.

A.22. llCollisionSprite

```
llCollisionSprite(string impact_sprite);
```

Suppress default collision sprites, replace default impact sprite with *impact_sprite* found in the object inventory. Supply an empty string to just suppress.

A.23. llCos

```
float llCos(float theta);
```

Returns the cosine of *theta* radians.

A.24. llCreateLink

```
llCreateLink(key target, integer parent);
```

Attempt to link object script is attached to and *target*. Requires permission PERMISSION_CHANGE_LINKS be set. If *parent* == TRUE, object script is attached to is the root.

A.25. llDeleteSubList

```
list llDeleteSubList(list src, integer start, integer end);
```

Remove the slice from the list and return the remainder. The *start* and *end* are inclusive, so 0, length - 1 would delete the entire list and 0,0 would delete the first list entry. Using negative numbers for *start* and/or *end* causes the index to count backwards from the length of the list, so 0,-1 would delete the entire list. If *start* is larger than *end* the list deleted is the exclusion of the entries, so 6,4 would delete the entire list except for the 5th list entry.

A.26. llDeleteSubString

```
stiring llDeleteSubString(string src, integer start, integer end);
```

Removes the indicated substring and returns the result. The *start* and *end* are inclusive, so 0,length-1 would delete the entire string and 0,0 would delete the first character. Using negative numbers for *start* and/or *end* causes the index to count backwards from the length of the string, so 0,-1 would delete the entire string. If *start* is larger than end the sub string is the exclusion of the entries, so 6,4 would delete the entire string except for the 5th character.

A.27. llDetachFromAvatar

```
llDetachFromAvatar(key avatar);
```

Drop off of *avatar*.

A.28. llDetectedGrab

```
vector llDetectedGrab(integer number);
```

Returns the grab offset of detected object *number*. Returns <0,0,0> if number is not valid sensed object.

A.29. llDetectedKey

```
key llDetectedKey(integer number);
```

Returns the key of detected object *number*. Returns NULL_KEY if number is not valid sensed object.

A.30. llDetectedLinkNumber

```
integer llDetectedLinkNumber(integer number);
```

Returns the link position of the triggered event for touches. 0 for a non-linked object, 1 for the root of a linked object, 2 for the first child, etc.

A.31. llDetectedName

```
string llDetectedName(integer number);
```

Returns the name of detected object *number*. Returns empty string if *number* is not valid sensed object.

A.32. llDetectedOwner

```
key llDetectedOwner(integer number);
```

Returns the key of detected *number* object's owner. Returns invalid key if *number* is not valid sensed object.

A.33. llDetectedPos

```
vector llDetectedPos(integer number);
```

Returns the position of detected object *number*. Returns <0,0,0> if *number* is not valid sensed object.

A.34. llDetectedRot

```
rotation llDetectedRot(integer number);
```

Returns the rotation of detected object *number*. Returns <0,0,0,1> if *number* is not valid sensed object).

A.35. llDetectedType

```
integer llDetectedType(integer number);
```

Returns the type (AGENT, ACTIVE, PASSIVE, SCRIPTED) of detected object *number*. Returns 0 if *number* is not valid sensed object. Note that *number* is a bitfield, so comparisons need to be a bitwsie and check. eg:

```
integer type = llDetectedType(0);
if (type & AGENT)
{
  // ...do stuff with the agent
}
```

A.36. llDetectedVel

```
vector llDetectedVel(integer number);
```

Returns the velocity of detected object *number*. Returns <0,0,0> if *number* is not valid sensed object.

A.37. llDialog

```
llDialog(key avatar, string message, list buttons, integer channel);
```

Opens a "notify box" in the top-right corner of the given avatar's screen displaying the message. Up to four buttons can be specified in a list of strings. When the player clicks a button, the name of the button is chatted on the specified channel. Channels work just like llSay(), so channel 0 can be heard by everyone. The chat originates at the object's position, not the avatar's position. e.g.

```
LLDialog(who, "Are you a boy or a girl?", [ "Boy", "Girl" ], 4913);
LLDialog(who, "This shows only an OK button.", [], 192);
llDialog(who, "This chats so you can hear it.", ["Hooray"], 0);
```

A.38. llDie

```
llDie(void);
```

Delete the object which holds the script.

A.39. llDumpList2String

```
string llDumpList2String(list src, string separator);
```

Write the list out in a single string using separator between values.

A.40. llEdgeOfWorld

```
integer llEdgeOfWorld(vector pos, vector dir);
```

Returns TRUE if the line along *`dir`* from *`pos`* hits the edge of the world in the current simulator and returns FALSE if that edge crosses into another simulator.

A.41. llEjectFromLand

```
llEjectFromLand(key pest);
```

Ejects *`pest`* from land that you own.

A.42. llEmail

```
llEmail(string address, string subject, string message);
```

Sends email to *`address`* with *`subject`* and *`message`*.

A.43. llEuler2Rot

```
rotation llEuler2Rot(vector vec);
```

Returns the rotation represented by Euler Angle *vec*.

A.44. llFabs

```
float llFabs(float val);
```

Returns the absolute value of *val*.

A.45. llFloor

```
integer llFloor(float val);
```

Returns largest integer value <= *val*.

A.46. llFrand

```
float llFrand(float mag);
```

Returns a pseudo-random number between [0, *mag*).

A.47. llGetAccel

```
vector llGetAccel(void);
```

Gets the acceleration.

A.48. llGetAttached

```
integer llGetAttached(void);
```

Returns the object attachment point or 0 if not attached.

A.49. llGetAgentInfo

```
integer llGetAgentInfo(key id);
```

Returns information about the given agent *id*. Returns a bitfield of agent info constants.

A.50. llGetAgentSize

```
vector llGetAgentSize(key id);
```

If the agent *id* is in the same sim as the object, returns the size of the avatar.

A.51. llGetAlpha

```
float llGetAlpha(integer face);
```

Returns the alpha of the given *face*. If *face* is ALL_SIDES the value returned is the mean average of all faces.

A.52. llGetAndResetTime

```
float llGetAndResetTime(void);
```

Gets the time in seconds since creation and sets the time to zero.

A.53. llGetAnimation

```
string llGetAnimation(key id);
```

Returns the currently playing animation for avatar *id*.

A.54. llGetCenterOfMass

```
vector llGetCenterOfMass(void);
```

Returns the center of mass of the root object.

A.55. llGetColor

```
vector llGetColor(integer face);
```

Returns the color of *face* as a vector of red, green, and blue values between 0 and 1. If *face* is ALL_SIDES the color returned is the mean average of each channel.

A.56. llGetDate

```
string llGetDate(void);
```

Returns the current UTC date as YYYY-MM-DD.

A.57. llGetEnergy

```
float llGetEnergy(void);
```

Returns how much energy is in the object as a percentage of maximum.

A.58. llGetForce

```
vector llGetForce(void);
```

Returns the current force if the script is physical.

A.59. llGetFreeMemory

```
integer llGetFreeMemory(void);
```

Returns the available heap space for the current script.

A.60. llGetInventoryKey

```
key llGetInventoryKey(string name);
```

Returns the key of the inventory *name*.

A.61. llGetInventoryName

```
string llGetInventoryName(integer type, integer number);
```

Get the name of the inventory item *number* of *type*. Use the inventory constants to specify the *type*.

A.62. llGetInventoryNumber

```
integer llGetInventoryNumber(integer type);
```

Get the number of items of *type* in the object inventory. Use the inventory constants to specify the *type*.

A.63. llGetKey

```
key llGetKey(void);
```

Get the key for the object which has this script.

A.64. llGetLandOwnerAt

```
key llGetLandOwnerAt(vector pos);
```

Returns the key of the land owner at *pos* or NULL_KEY if public.

A.65. llGetLinkKey

```
key llGetLinkKey(integer linknum);
```

Returns the key of *linknum* in the link set.

A.66. llGetLinkName

```
string llGetLinkName(integer linknum);
```

Returns the name of *linknum* in the link set.

A.67. llGetLinkNumber

```
integer llGetLinkNumber(void);
```

Returns what link number in a link set the for the object which has this script. 0 means no link, 1 the root, 2 for first child, etc.

A.68. llGetListEntryType

```
integer llGetListEntryType(list src, integer index);
```

Returns the type of the variable at *index* in *src*.

A.69. llGetListLength

```
integer llGetListLength(list src);
```

Returns the number of elements in *src*.

A.70. llGetLocalPos

```
vector llGetLocalPos(void);
```

Returns the local position of a child object relative to the root.

A.71. llGetLocalRot

```
rotation llGetLocalRot(void);
```

Returns the local rotation of a child object relative to the root.

A.72. llGetNextEmail

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

Get the next waiting email with appropriate *address* and/or *subject*. If the parameters are blank, they are not used for filtering.

A.73. llGetNotecardLine

```
key llGetNotecardLine(string name, integer line);
```

This function fetches line number *line* of notecard *name* and returns the data through the dataserver event. The line count starts at zero. If the requested line is past the end of the notecard the dataserver event will return the constant EOF string. The key returned by this function is a unique identifier which will be supplied to the dataserver event in the *requested* parameter.

A.74. llGetNumberOfSides

```
key llGetNumberOfSides(void);
```

Returns the number of sides of the current which has the script.

A.75. llGetObjectName

```
string llGetObjectName(void);
```

Returns the name of the object which has the script.

A.76. llGetOmega

```
vector llGetOmega(void);
```

Returns the omega.

A.77. llGetOwner

```
key llGetOwner(void);
```

Returns the owner of the object.

A.78. llGetOwnerKey

```
key llGetOwnerKey(key id);
```

Returns the owner of object *id*.

A.79. llGetPermissions

```
integer llGetPermissions(void);
```

Returns what permissions have been enabled.eg:

```
integer perm = llGetPermissions();
if((perm & PERMISSION_DEBIT) == PERMISSION_DEBIT)
{
    // code goes here
}
```

A.80. llGetPermissionsKey

```
key llGetPermissionsKey(void);
```

Retuerns avatar that has enabled permissions. Returns NULL_KEY if not enabled.

A.81. llGetPos

```
vector llGetPos(void);
```

Returns the position.

A.82. llGetRegionFPS

```
llGetRegionFPS(void);
```

Returns the mean region frames per second.

A.83. llGetRegionName

```
string llGetRegionName(void);
```

Returns the current region name.

A.84. llGetRegionTimeDilation

```
float llGetRegionTimeDilation(void);
```

Returns the current time dilation as a float between 0 and 1.

A.85. llGetRot

```
rotation llGetRot(void);
```

Returns the rotation.

A.86. llGetScale

```
vector llGetScale(void);
```

Returns the scale.

A.87. llGetScriptName

```
string llGetScriptName(void);
```

Returns the name of this script.

A.88. llGetStartParameter

```
integer llGetStartParameter(void);
```

Returns the start parameter passed to llRezObject. If the object was created from agent inventory, this function returns 0.

A.89. llGetScriptState

```
integer llGetScriptState(string name);
```

Resets TRUE if script *name* is running

A.90. llGetStatus

```
integer llGetStatus(integer status);
```

Returns the value of *status*. The value will be one of the status constants.

A.91. llGetSubString

```
string llGetSubString(string src, integer start, integer end);
```

Returns the indicated substring from *src*. The *start* and *end* are inclusive, so 0,length-1 would capture the entire string and 0,0 would capture the first character. Using negative numbers for *start* and/or *end* causes the index to count backwards from the length of the string, so 0,-1 would capture the entire string. If start is larger than end the sub string is the exclusion of the entries, so 6,4 would give the entire string except for the 5th character.

A.92. llGetSunDirection

```
vector llGetSunDirection(void);
```

Returns the sun direction on the simulator.

A.93. llGetTexture

```
string llGetTexture(integer face);
```

Returns the texture of *face* if it is found in object inventory.

A.94. llGetTextureOffset

```
vector llGetTextureOffset(integer side);
```

Returns the texture offset of *side* in the x and y components of a vector.

A.95. llGetTextureRot

```
float llGetTextureRot(integer side);
```

Returns the texture rotation of *side*.

A.96. llGetTextureScale

```
vector llGetTextureScale(integer side);
```

Returns the texture scale of *side* in the x and y components of a vector.

A.97. llGetTime

```
float llGetTime(void);
```

Returns the time in seconds since creation of this script.

A.98. llGetTimeOfDay

```
float llGetTimeOfDay(void);
```

Gets the time in seconds since midnight in Second Life.

A.99. llGetTorque

```
vector llGetTorque(void);
```

Returns the torque if the script is physical.

A.100. llGetVel

```
vector llGetVel();
```

Returns the velocity.

A.101. llGetWallclock

```
float llGetWallclock(void);
```

Returns the time in seconds since simulator timezone midnight.

A.102. llGiveInventory

```
llGiveInventory(key destination, string inventory);
```

Give the named inventory item to the keyed avatar or object in the same simulator as the giver. If the recipient is an avatar, the avatar then follows the normal procedure of accepting or denying the offer. If the recipient is an object, the same permissions apply as if you were dragging inventory onto the object by hand, ie if llAllowInventoryDrop has been called with TRUE, any other object can pass objects to its inventory.

A.103. llGiveInventoryList

```
llGiveInventoryList(key destination, string category, list inventory);
```

Give the list of named inventory items to the keyed avatar or object in the same simulator as the giver. If the recipient is an avatar, the avatar then follows the normal procedure of accepting or denying the offer. The offered inventory is then placed in a folder named *`category`* in the recipients inventory. If the recipient is an object, the same permissions apply as if you were dragging inventory onto the object by hand, ie if llAllowInventoryDrop has been called with TRUE, any other object can pass objects to its inventory.If the recipient is an object, the *`category`* parameter is ignored.

A.104. llGiveMoney

```
llGiveMoney(key destination, integer amount);
```

Transfer *`amount`* from the script owner to *`destination`*. This call will fail if PERMISSION_DEBIT has not been set.

A.105. llGround

```
float llGround(vector offset);
```

Retuns the ground hieght at the object position + *`offset`*.

A.106. llGroundContour

```
vector llGroundContour(vector offset);
```

Retuns the ground contour at the object position + *`offset`*.

A.107. llGroundNormal

```
vector llGroundNormal(vector offset);
```

Retuns the ground contour at the object position + *offset*.

A.108. llGroundRepel

llGroundRepel(float *height*, integer *water*, float *tau*);

Critically damps to *height* if within *height* * 0.5 of *level*. The *height* is above ground level if *water* is FALSE or above the higher of land and water if *water* is TRUE.

A.109. llGroundSlope

vector **llGroundSlope**(vector *offset*);

Returns the ground slope at the object position + *offset*.

A.110. llInsertString

string **llInsertString**(string *dst*, integer *position*, string *src*);

Inserts *src* into *dst* at *position* and returns the result.

A.111. llInstantMessage

llInstantMessage(key *user*, string *message*);

Send *message* to the *user* as an instant message.

A.112. llKey2Name

string **llKey2Name**(key *id*);

If object *id* is in the same simulator, return the name of the object.

A.113. llList2CSV

```
string llList2CSV(list src);
```

Create a string of comma separated values from *list*.

A.114. llList2Float

```
float llList2Float(list src, integer index);
```

Returns the float at *index* in the list *src*.

A.115. llList2Integer

```
integer llList2Integer(list src, integer index);
```

Returns the integer at *index* in the list *src*.

A.116. llList2Key

```
key llList2Key(list src, integer index);
```

Returns the key at *index* in the list *src*.

A.117. llList2List

```
list llList2List(list src, integer start, integer end);
```

Returns the slice of the list from *start* to *end* from the list *src* as a new list. The *start* and *end* parameters are inclusive, so 0,length-1 would copy the entire list and 0,0 would capture the first list entry. Using negative numbers for *start* and/or *end* causes the index to count backwards from the length of the list, so 0,-1 would capture the entire list. If *start* is larger than *end* the list returned is the exclusion of the entries, so 6,4 would give the entire list except for the 5th entry.

A.118. llList2ListStrided

```
list llList2ListStrided(list src, integer start, integer end, integer stride);
```

Copy the strided slice of *src* from *start* to *end*.

A.119. llList2Rot

```
rotation llList2Rot(list src, integer index);
```

Returns the rotation at *index* in *src*.

A.120. llList2String

```
string llList2String(list src, integer index);
```

Returns the string at *index* in *src*.

A.121. llList2Vector

```
llList2Vector(list src, integer index);
```

Returns the string at *index* in *src*.

A.122. llListFindList

```
integer llListFindList(list src, list test);
```

Returns the position of the first instance of *test* in *src*. Returns -1 if *test* is not in *src*.

A.123. llListInsertList

```
list llListInsertList(list dest, list src, integer pos);
```

Returns the list created by inserting *src* into *dest* at *pos*.

A.124. llListRandomize

```
list llListRandomize(list src, integer stride);
```

Returns *src* randomized into blocks of size *stride*. If the length of *src* dividided by *stride* is non-zero, this function does not randomize the list.

A.125. llListSort

```
list llListSort(list src, integer stride, integer ascending);
```

Returns *src* sorted into blocks of *stride* in ascending order if *ascending* is TRUE. Note that sort only works in the head of each sort block is the same type.

A.126. llListen

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

Sets a listen event callback for *msg* on *channel* from *name* and returns an identifier that can be used to deactivate or remove the listen. The *name*, *id* and/or *msg* parameters can be blank to indicate not to filter on that argument. Channel 0 is the public chat channel that all avatars see as chat text. Channels 1 to 2,147,483,648 are hidden channels that are not sent to avatars.

A.127. llListenControl

```
llListenControl(integer number, integer active);
```

Make a listen event callback active or inactive. Pass in the value returned from llListen to the *number* parameter to specify which event you are controlling. Use boolean values to spcify *active*

A.128. llListenRemove

```
llListenRemove(integer number);
```

Removes a listen event callback. Pass in the value returned from llListen to the *number* parameter to specify which event you are removing.

A.129. llLookAt

```
llLookAt(vector target, float strength, float damping);
```

Cause object to point the forward axis toward *target*. Good *strength* values are around half the mass of the object and good *damping* values are less than 1/10th of the *strength*. Asymmetrical shapes require smaller *damping*. A *strength* of 0.0 cancels the look at.

A.130. llLoopSound

```
llLoopSound(string sound, flaot volume);
```

Similar to llPlaySound, this function plays a sound attached to an object, but will continuously loop that sound until llStopSound or llPlaySound is called. Only one sound may be attached to an object at a time. A second call to llLoopSound with the same key will not restart the sound, but the new volume will be used. This allows control over the volume of already playing sounds. Setting the *volume* to 0 is not the same as calling llStopSound; a sound with 0 volume will continue to loop. To restart the sound from the beginning, call llStopSound before calling llLoopSound again.

A.131. llLoopSoundMaster

llLoopSoundMaster(string *sound*, flaot *volume*);

Behaviour is identical to llLoopSound, with the addition of marking the source as a "Sync Master", causing "Slave" sounds to sync to it. If there are multiple masters within a viewer’s interest area, the most audible one (a function of both distance and volume) will win out as the master. The use of multiple masters within a small area is unlikely to produce the desired effect.

A.132. llLoopSoundSlave

llLoopSoundSlave(string *sound*, flaot *volume*);

Behaviour is identical to llLoopSound, unless there is a "Sync Master" present. If a Sync Master is already playing the Slave sound will begin playing from the same point the master is in its loop syncronizing the loop points of both sounds. If a Sync Master is started when the Slave is already playing, the Slave will skip to the correct position to sync with the Master.

A.133. llMakeExplosion

llMakeExplosion(integer *particles*, float *scale*, float *velocity*, float *lifetime*, float *arc*, string *texture*, vector *offset*);

Make a round explosion of particles using *texture* from the object’s inventory.

A.134. llMakeFire

llMakeFire(integer *particles*, float *scale*, float *velocity*, float *lifetime*, float *arc*, string *texture*, vector *offset*);

Make fire particles using *texture* from the object’s inventory.

A.135. llMakeFountain

llMakeFountain(integer *particles*, float *scale*, float *velocity*, float *lifetime*, float *arc*, string *texture*, vector *offset*);

Make a fountain of particles using *texture* from the object's inventory.

A.136. llMakeSmoke

llMakeSmoke(integer *particles*, float *scale*, float *velocity*, float *lifetime*, float *arc*, string *texture*, vector *offset*);

Make smoky particles using *texture* from the object's inventory.

A.137. llMessageLinked

llMessageLinked(integer *linknum*, integer *num*, string *str*, key *id*);

Sends *num*, *str*, and *id* to members of the link set. The *linknum* parameter is either the linked number available through llGetLinkNumber or a link constant.

A.138. llMinEventDelay

llMinEventDelay(float *delay*);

Set the minimum time between events being handled.

A.139. llModifyLand

llModifyLand(integer *action*, integer *size*);

Modify land with *action* on *size* area. The parameters can be chosen from the land constants.

A.140. llMoveToTarget

```
llMoveToTarget(vector target, float float tau);
```

Critically damp to position *target* in *tau* seconds if the script is physical. Good *tau* values are greater than 0.2. A *tau* of 0.0 stops the critical damping.

A.141. llOffsetTexture

```
llOffsetTexture(float offset_s, float offset_t, integer face);
```

Sets the texture s and t offsets of *face*. If *face* is ALL_SIDES this function sets the texture offsets for all faces.

A.142. llOverMyLand

```
integer llOverMyLand(key id);
```

Returns TRUE if *id* is over land owned by the object owner, FALSE otherwise.

A.143. llParseString2List

```
list llParseString2List(string src, list separators, list spacers);
```

Breaks *src* into a list, discarding anything in *separators*, keeping any entry in *spacers*. The *separators* and *spacers* must be lists of strings with a maximum of 8 entries each. So, if you had made the call:

```
llParseString2List("Parsethisnow!  I dare:you to.", ["this", "!", " "], [":"]);
```

You would get the list:

```
["Parse", "now", "I", "dare", ":", "you", "to"]
```

A.144. llParticleSystem

llParticleSystem(list *parameters*);

Makes a particle system based on the parameter list. The *parameters* are specified as an ordered list of parameter and value. Valid parameters and their expected values can be found in the particle system constants. Here is a simple example:

```
llParticleSystem([PSYS_PART_FLAGS, PSYS_PART_WIND_MASK,
                  PSYS_PART_START_COLOR, <1,0,0>,
                  PSYS_SRC_PATTERN, PSYS_SRC_PATTERN_EXPLODE]);
```

A.145. llPassCollisions

llPassCollisions(integer *pass*);

If *pass* is TRUE, land and object collisions are passed from children on to parents.

A.146. llPassTouches

llPassTouches(integer *pass*);

If *pass* is TRUE, touches are passed from children on to parents.

A.147. llPlaySound

llPlaySound(string *sound*, float *volume*);

Plays a sound once. The sound will be attached to an object and follow object movement. Only one sound may be attached to an object at a time, and attaching a new sound or calling llStopSound will stop the previously attached sound. A second call to llPlaySound with the same *sound* will not restart the sound, but the new volume will be used, which allows control over the volume of already playing sounds. To restart the sound from the beginning, call llStopSound before calling llPlaySound again.

A.148. llPlaySoundSlave

llPlaySoundSlave(string *sound*, float *volume*);

Behaviour is identical to llPlaySound, unless there is a "Sync Master" present. If a Sync Master is already playing the Slave sound will not be played until the Master hits its loop point and returns to the beginning. llPlaySoundSlave will play the sound exactly once; if it is desired to have the sound play every time the Master loops, either use llLoopSoundSlave with extra silence padded on the end of the sound or ensure that llPlaySoundSlave is called at least once per loop of the Master.

A.149. llPointAt

llPointAt(vector *pos*);

Make avatar that owns object point at *pos*.

A.150. llPow

llPow(float *base*, float *exp*);

Returns *base* raised to the *exp*.

A.151. llPreloadSound

llPreloadSound(string *sound*);

Preloads *sound* from object inventory on nearby viewers.

A.152. llPushObject

llPushObject(key *id*, vector *impulse*, vector *angular_impulse*, integer *local*);

Applies *impulse* and *angular_impulse* to object *id*.

A.153. llReleaseControls

```
llReleaseControls(key avatar);
```

Stop taking inputs from *avatar*.

A.154. llRemoteLoadScript

```
llRemoteLoadScript(key target, string name, integer running, integer param);
```

If the owner of the object this script is attached can modify *target* and the objects are in the same region, copy script *name* onto *target*, if *running* == TRUE, start the script with *param*. If *name* already exists on *target*, it is replaced.

A.155. llRemoveInventory

```
llRemoveInventory(string inventory);
```

Remve the name *inventory* item from the object inventory.

A.156. llRemoveVehicleFlags

```
llRemoveVehicleFlags(integer flags);
```

Sets the vehicle *flags* to FALSE. Valid parameters can be found in the vehicle flags constants section.

A.157. llRequestAgentData

```
key llRequestAgentData(key id, integer data);
```

This function requests data about agent *id*. If and when the information is collected, the dataserver event is called with the returned key returned from this function passed in the *requested* parameter. See the agent data constants for details about valid values of *data* and what each will return in the dataserver event.

A.158. llRequestInventoryData

```
key llRequestInventoryData(string name);
```

Requests data from object inventory item *name*. When data is available the dataserver event will be raised with the key returned from this function in the *requested* parameter. The only request currently implemented is to request data from landmarks, where the data returned is in the form "<float, float, float>" which can be cast to a vector. This position is in region local coordinates.

A.159. llRequestPermissions

```
integer llRequestPermissions(key avatar, iteger perm);
```

Ask *avatar* to allow the script to do *perm*. The *perm* parameter should be a permission constant. Multiple permissions can be requested simultaneously by or'ing the constants together. Many of the permissions requests can only go to object owner. This call will not stop script execution - if the specified avatar grants the requested permissions, the run_time_permissions event will be called.

A.160. llResetScript

```
llResetScript(void);
```

Resets this script.

A.161. llResetOtherScript

```
llResetOtherScript(string name);
```

Resets the script *name*.

A.162. llResetTime

```
llResetTime(void);
```

Sets the internal timer to zero.

A.163. llRezObject

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

Creates object's *inventory* object at position *pos* with velocity *vel* and rotation *rot*. The *param* value will be available to the newly created object in the on_rez event or through the llGetStartParameter library function. The *vel* parameter is ignored if the rezzed object is not physical.

A.164. llRot2Angle

```
float llRot2Angle(rotation rot);
```

Returns the rotation angle represented by *rot*.

A.165. llRot2Axis

```
vector llRot2Axis(rotation rot);
```

Returns the rotation axis represented by *rot*.

A.166. llRot2Euler

```
vector llRot2Euler(rotation rot);
```

Returns the Euler Angle representation of *rot*.

A.167. llRot2Fwd

```
vector llRot2Fwd(rotation rot);
```

Returns the forward axis represented by *rot*.

A.168. llRot2Left

llRot2Left(rotation *rot*);

Returns the left axis represented by *rot*.

A.169. llRot2Up

llRot2Up(rotation *rot*);

Returns the up axis represented by *rot*.

A.170. llRotBetween

rotation **llRotBetween**(vector *a*, vector *b*);

Returns the rotation needed to rotate *a* to *b*.

A.171. llRotLookAt

llRotLookAt(rotation *rot*, float *strength*, float *damping*);

Cause object to rotate to *rot*. Good *strength* values are around half the mass of the object and good *damping* values are less than 1/10th of the *strength*. Asymmetrical shapes require smaller *damping*. A *strength* of 0.0 cancels the look at.

A.172. llRotTarget

integer **llRotTarget**(rotation *rot*, float *error*);

Set object rotation within *error* of *rotation* as a rotational target and return an integer number for the target. The number can be used in llRotTargetRemove.

A.173. llRotTargetRemove

llRotTargetRemove(integer *number*);

Remove rotational target *number*.

A.174. llRotateTexture

llRotateTexture(float *radians*, integer *face*);

Sets the texture rotation of *face* to *radians*. If *face* ALL_SIDES, rotate the texture of all faces.

A.175. llRound

integer **llRound**(float *val*);

returns *val* rounded to the nearest integer.

A.176. llSameGroup

integer **llSameGroup**(key *id*);

Returns TRUE if the object or agen *id* is in the same simulator and has the same active group as this object. Otherwise, returns FALSE.

A.177. llSay

llSay(integer *channel*, string *text*);

Say *text* on *channel*. Channel 0 is the public chat channel that all avatars see as chat text. Channels 1 to 2,147,483,648 are private channels that are not sent to avatars but other scripts can listen for through the llListen api.

A.178. llScaleTexture

```
llScaleTexture(integer scale_s, integer scale_t, integer face);
```

Sets the texture s and t scales of *face* to *scale_s* and *scale_t* respectively. If face is ALL_SIDES, scale the texture to all faces.

A.179. llScriptDanger

```
integer llScriptDanger(vector pos);
```

Returns true if pos is over public land, land that doesn't allow everyone to edit and build, or land that doesn't allow outside scripts.

A.180. llSensor

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

Performs a single scan for *name* and *id* with *type* within *range* meters and *arc* radians of forward vector. Specifying a blank name or NULL_KEY id will not filter results for any particular name or id A range of 0.0 does not perform a scan. The *type* parameter should be an object type constant vlaue.

A.181. llSensorRemove

```
llSensorRemove(void);
```

Remves the sensor.

A.182. llSensorRepeat

llSensorRepeat(string *name*, key *id*, integer *type*, float *range*, float *arc*, float *rate*);

Performs a single scan for *name* and *id* with *type* within *range* meters and *arc* radians of forward vector and repeats every *rate* seconds. Specifying a blank name or NULL_KEY id will not filter results for any particular name or id A range of 0.0 cancels the scan. The *type* parameter should be an object type constant vlaue.

A.183. llSetAlpha

llSetAlpha(float *alpha*, integer *face*);

Sets the alpha value for *face*. If face is ALL_SIDES, set the alpha to all faces. The *alpha* value is interpreted as an opacity percentage - 1.0 is fully opaque, and 0.2 is mostly transparent. This api will clamp *alpha* values less 0.1 to .1 and greater than 1.0 to 1.

A.184. llSetBuoyancy

llSetBuoyancy(float *buoyancy*);

Set the object buoyancy. A value of 0 is none, less than 1.0 sinks, 1.0 floats, and greater than 1.0 rises.

A.185. llSetCameraAtOffset

llSetCameraAtOffset(vector *offset*);

Sets the camera at offset used in this object if an avatar sits on it.

A.186. llSetCameraEyeOffset

llSetCameraEyeOffset(vector *offset*);

Sets the camera eye offset used in this object if an avatar sits on it.

A.187. llSetColor

llSetColor(vector *color*, integer *face*);

Sets the *color* of *face*. If face is ALL_SIDES, set the alpha to all faces.

A.188. llSetDamage

llSetDamage(float *damage*);

Sets the amount of damage that will be done to an object that this object hits. This object will be destoyed on damaging another object.

A.189. llSetForce

llSetForce(vector *force*, integer *local*);

If the object is physical, this function sets the *force*. The vector is in loal coordinates if local is TRUE, global if FALSE.

A.190. llSetForceAndTorque

llSetForceAndTorque(vector *force*, vector *torque*, integer *local*);

If the object is physical, this function sets the *force* and *torque*. The vectors are in loal coordinates if local is TRUE, global if FALSE.

A.191. llSetHoverHeight

llSetHoverHeight(float *height*, float *water*, float *tau*);

Critically damps to a height. The height is above ground and water if *water* is TRUE.

A.192. llSetLinkColor

llSetLinkColor(integer *linknumber*, vector *color*, integer *face*);

Sets the *color* of the linked child specified by *linknumber*. A value of 0 means no link, 1 the root, 2 for first child, etc. If *linknumber* is ALL_SIDES, set the color of all objects in the linked set. If *face* is ALL_SIDES, set the color of all faces.

A.193. llSetObjectName

llSetObjectName(string *name*);

Sets the object name to *name*.

A.194. llSetPos

llSetPos(vector *pos*);

If the object is not physical, this function sets the position in region coordinates. If the object is a child, the position is treated as root relative and the linked set is adjusted.

A.195. llSetRot

llSetRot(rotation *rot*);

If the object is not physical, this function sets the rotation. If the object is a child, the position is treated as root relative and the linked set is adjusted.

A.196. llSetScale

llSetScale(vector *scale*);

Sets the object scale.

A.197. llSetScriptState

```
llSetScriptState(string name, integer run);
```

Control the state of a script on the object.

A.198. llSetSitText

```
funcdef(string text);
```

Displays *text* rather than sit in viewer pie menu.

A.199. llSetSoundQueueing

```
llSetSoundQueueing(integer queue);
```

Sets whether successive calls to llPlaySound, llLoopSound, etc., (attached sounds) interrupt the playing sound. The default for objects is FALSE. Setting this value to TRUE will make the sound wait until the current playing sound reaches its end. The queue is one level deep.

A.200. llSetStatus

```
llSetStatus(integer status, integer value);
```

Sets the *status* to *value*. Use status constants for the values of *status*.

A.201. llSetText

```
llSetText(string text, vector color, float alpha);
```

Sets text that floats above object to *text*, using the specified *color* and *alpha*.

A.202. llSetTexture

llSetTexture(string *texture*, integer *face*);

Sets the *texture* from object inventory of *face*. If face is ALL_SIDES, set the texture to all faces.

A.203. llSetTextureAnim

llSetTextureAnim(integer *mode*, integer *face*, integer *sizex*, integer *sizey*, float *start*, float *length*, float *rate*);

Animates a texture by setting the texture scale and offset. The mode is a mask of texture animation constants. You can only have one texture animation on an object, calling llSetTextureAnim more than once on an object will reset it.

You can only do one traditional animation, ROTATE or SCALE at a time, you cannot combine masks. In the case of ROTATE or SCALE, *sizex* and *sizey* are ignored, and *start* and *length* are used as the start and length values of the animation. For rotation, *start* and *length* are in radians.

The *face* specified which face to animate. If *face* is ALL_SIDES, all textures on the object are animated.

The *sizex* and *sizey* describe the layout of the frames within the texture. *sizex* specifies how many horizontal frames and *sizey* is how many vertical frames.

start is the frame number to begin the animation on. Frames are numbered from left to right, top to bottom, starting at 0.

length is the number of frames to animate. 0 means to animate all frames after the start frame.

rate is the frame rate to animate at. 1.0 means 1 frame per second, 10.0 means 10 frames per second, etc.

A.204. llSetTimerEvent

llSetTimerEvent(float *sec*);

Sets the timer event to be triggered every *sec* seconds. Passing in 0.0 stops further timer events.

A.205. llSetTorque

llSetTorque(vector *torque*, integer *local*);

If the object is physical, this function sets the *torque*. The vector is in loal coordinates if local is TRUE, global if FALSE.

A.206. llSetTouchText

llSetTouchText(string *text*);

Displays *text* in viewer pie menu that acts as a touch.

A.207. llSetVehicleFlags

llSetVehicleFlags(integer *flags*);

Sets the vehicle *flags* to TRUE. Valid parameters can be found in the vehicle flags constants section.

A.208. llSetVehicleFloatParam

llSetVehicleFloatParam(integer *param_name*, float *param_value*);

Sets the vehicle floating point parameter *param_name* to *param_value*. Valid parameters and their expected values can be found in the vehicle parameter constants section.

A.209. llSetVehicleType

llSetVehicleType(integer *type*);

Activates the vehicle action and choose vehicle *type*. Valid types and an explination of their characteristeics can be found in the vehicle type constants section.

A.210. llSetVehicleRotationParam

```
llSetVehicleRotationParam(integer param_name, rotation param_value);
```

Sets the vehicle rotation parameter *param_name* to *param_value*. Valid parameters can be found in the vehicle parameter constants section.

A.211. llSetVehicleVectorParam

```
llSetVehicleVectorParam(integer param_name, vector param_value);
```

Sets the vehicle vector parameter *param_name* to *param_value*. Valid parameters can be found in the vehicle paramter constants section.

A.212. llShout

```
llShout(integer channel, string text);
```

Shout *text* on *channel*. Channel 0 is the public chat channel that all avatars see as chat text. Channels 1 to 2,147,483,648 are private channels that are not sent to avatars but other scripts can listen for through the llListen api.

A.213. llSin

```
float llSin(float theta);
```

Returns the sine of *theta* in radians.

A.214. llSitTarget

```
llSitTarget(vector offset, rotation rot);
```

Set the sit location for this object. If *offset* == ZERO_VECTOR clear the sit target.

A.215. llSleep

```
llSleep(float sec);
```

Puts the script to sleep for *sec* seconds.

A.216. llSqrt

```
float llSqrt(float val);
```

Returns the square root of *val*. If *val* is less than 0.0, this function returns 0.0 and raises a math runtime error.

A.217. llStartAnimation

```
llStartAnimation(string anim);
```

This function starts animation *anim* for the avatar that owns the object.

Valid strings for *anim*

hold_R_bazooka
hold_R_handgun
hold_R_rifle

> Holds the appropriately shaped weapon in the right hand. Automatically switches to the aims (below) when user enters mouse look

aim_R_bazooka
aim_R_handgun
aim_R_rifle

> Aims the appropriately shaped weapon along the direction the avatar is looking.

away

Flops over in "away from keyboard" state.

backflip

Performs a backflip.

bow

Bows at waist.

brush

Brushes dirt from shirt.

clap

Applauds.

courtbow

Bows with a courtly flourish.

crouch

Crouches in place.

crouchwalk

Walks in place while crouching.

dance1
dance2
dance3
dance4
dance5
dance6
dance7
dance8

Various dance maneuvers.

falldown

Freefall falling animation.

female_walk

Walks with hip sway.

fly

Flies forward.

flyslow

Flies forward at a less agressive angle.

hello

Waves.

hold_throw_R

Hold object in right hand, prepared to throw it.

hover

Hovers in place.

hover_down

Pretends to hover straight down.

hover_up

Pretends to hover straight up.

jump

Midair jump position.

kick_roundhouse_R

Roundhouse kick with right leg.

land

Lands after flying.

prejump

Prepares to jump.

punch_L

Punch with left hand.

punch_R

Punch with right hand.

punch_onetwo

Punch with one hand then the other.

run

Runs in place.

salute

Salutes with right hand.

sit

Sits on object at knee height.

sit_ground

Sits down on ground.

slowwalk

Walks in place slowly.

smoke_idle

Leans on imaginary prop while holding cigarette.

smoke_inhale

Leans on imaginary prop and smokes a cigarette.

smoke_throw_down

Leans on imaginary prop, throws down a cigarette, and stamps it out.

snapshot

Pantomimes taking a picture.

soft_land

Stumbles a bit as if landing.

stand

Stands in place.

standup

Falls on face and stands up.

stride

Legs extended as if stepping off of a ledge.

sword_strike_R

Strike with sword in right hand.

talk

Head moves as if talking.

target

Right arm points where avatar is looking (used automatically with aim anims).

throw_R

Throws object in right hand.

tryon_shirt

Turns around and models a new shirt.

turnleft

Pretends to turn left.

turnright

Pretends to turn right.

type

Makes typing motion.

uphillwalk

Walks uphill in place.

walk

Walks in place.

whisper

Whispers behind hand.

whistle

Whistles with hands in mouth.

yell

Shouts between cupped hands.

A.218. llStopAnimation

```
llStopAnimation(string anim);
```

Stop animation *anim* for avatar that owns object.

A.219. llStopHover

```
llStopHover(void);
```

Stop hover to a height.

A.220. llStopLookAt

```
llStopLookAt(void);
```

Stop causing object to look at target.

A.221. llStopMoveToTarget

```
llStopMoveToTarget(void);
```

Stops critically damped motion.

A.222. llStopPointAt

```
llStopPointAt(void);
```

Stop avatar that owns object pointing.

A.223. llStopSound

```
llStopSound(void);
```

Stops a currently playing attached sound started with llPlaySound or llLoopSound. Has no effect on sounds started with llTriggerSound.

A.224. llStringLength

```
integer llStringLength(string src);
```

Returns the number of characters in *src*.

A.225. llSubStringIndex

```
integer llSubStringIndex(string source, string pattern);
```

Finds index in source where pattern first appears. Returns -1 if no match is found found.

A.226. llTakeControls

```
llTakeControls(integer controls, integer accept, integer pass_on);
```

If (*accept* == (*controls* & input)), send input to object. If the boolean *pass_on* is TRUE, also send input to avatar.

A.227. llTan

```
float llTan(float theta);
```

Returns the tangent of *theta* radians.

A.228. llTarget

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

Set object position within *range* of *position* as a target and returns an integer ID for the target.

A.229. llTargetOmega

```
llTargetOmega(vector axis, float spinrate, float gain);
```

Attempt to spin at *spinrate* with strength *gain* on *axis*. A *spinrate* of 0.0 cancels the spin. This function always works in object local coordinates.

A.230. llTargetRemove

```
llTargetRemove(integer tnumber);
```

Remove target number *tnumber*.

A.231. llTeleportAgentHome

```
llTeleportAgentHome(key id);
```

Teleport agent on the owner's land to agent's home location.

A.232. llToLower

```
llToLower();
```

A.233. llToUpper

```
string llToUpper(string src);
```

Returns *src* in all lower case.

A.234. llTriggerSound

```
llTriggerSound(string sound, float volume);
```

Plays a transient sound NOT attached to an object. The sound plays from a stationary position located at the center of the object at the time of the trigger. There is no limit to the number of triggered sounds which can be generated by an object, and calling llTriggerSound does not affect the attached sounds created by llPlaySound and llLoopSound. This is very useful for things like collision noises, explosions, etc. There is no way to stop or alter the volume of a sound triggered by this function.

A.235. llTriggerSoundLimited

```
llTriggerSoundLimited(string sound, float volume, vector tne, vector bsw);
```

Plays a transient sound NOT attached to an object with its audible range limited by the axis aligned bounding box define by *tne* (top-north-eash) and *bsw* (bottom-south-west). The sound plays from a stationary position located at the center of the object at the time of the trigger. There is no limit to the number of triggered sounds which can be generated by an object, and calling llTriggerSound does not affect the attached sounds created by llPlaySound and llLoopSound. This is very useful for things like collision noises, explosions, etc. There is no way to stop or alter the volume of a sound triggered by this function.

A.236. llUnSit

```
llUnSit(key id);
```

If agent identified by *id* is sitting on the object the script is attached to or is over land owned by the objects owner, the agent is forced to stand up.

A.237. llVecDist

```
float llVecDist(vector a, vector b);
```

Returns the distance from *a* to *b*

A.238. llVecMag

```
float llVecMag(vector vec);
```

Returns the magnitude of *vec*.

A.239. llVecNorm

```
vector llVecNorm(vector vec);
```

Returns normalized *vec*.

A.240. llVolumeDetect

```
llVolumeDetect(integer detect);
```

When detect = TRUE, this makes the entire link set the script is attached to phantom but if another object interpenetrates it, it will get a collision_start event. When an object stops interpenetrating, a collision_end event is generated. While the other is interpenetrating, collision events are NOT generated. The script must be applied to the root object of the link set to get the collision events. Collision filters work normally.

A.241. llWater

```
float llWater(vector offset);
```

Returns the water height at the object position + *offset*.

A.242. llWhisper

```
llWhisper(integer channel, string text);
```

Whisper *text* on *channel*. Channel 0 is the public chat channel that all avatars see as chat text. Channels 1 to 2,147,483,648 are private channels that are not sent to avatars but other scripts can listen for through the llListen api.

A.243. llWind

```
vector llWind(vector offset);
```

Returns the wind velocity below the object position + *offset*.

Appendix B. Events

Every state must have at least one handler. You can choose to handle an event by defining one of the the reserved event handlers named here.

B.1. at_rot_target

```
at_rot_target(integer number, rotation target_rotation, rotation our_rotation);
```

This event is triggered when a script comes within a defined angle of a target rotation. The range is set by a call to llRotTarget.

B.2. attach

```
attach(key attached);
```

This event is triggered whenever a object with this script is attached or detached from an avatar. If it is attached, attached is the key of the avatar it is attached to, otherwise attached is NULL_KEY.

B.3. changed

```
changed(integer changed);
```

Triggered when various events change the object. The *changed* will be a bitfield of change constants.

B.4. collision

```
collision(integer total_number);
```

This event is raised while another object is colliding with the object the script is attached to. The number of detected objects is passed to the script. Information on those objects may be gathered via the llDetected* library functions. (Collisions are also generated if a user walks into an object.)

B.5. collision_end

```
collision_end(integer total_number);
```

This event is raised when another object stops colliding with the object the script is attached to. The number of detected objects is passed to the script. Information on those objects may be gathered via the llDetected* library functions. (Collisions are also generated if a user walks into an object.)

B.6. collision_start

```
collision_start(integer total_number);
```

This event is raised when another object begins to collide with the object the script is attached to. The number of detected objects is passed to the script. Information on those objects may be gathered via the llDetected* library functions. (Collisions are also generated if a user walks into an object.)

B.7. control

```
control(key name, integer levels, integer edges);
```

Once a script has the ability to grab control inputs from the avatar, this event will be used to pass the commands into the script. The `levels` and `edges` are bitfields of control constsants.

B.8. dataserver

```
dataserver(key requested, string data);
```

This event is triggered when the requested data is returned to the script. Data may be requested by the llRequestAgentData, the llRequestInventoryData, and the llGetNotecardLine function calls.

B.9. email

```
email(string time, string address, string subject, string body, integer remaining);
```

This event is triggered when an email sent to this script arrives. The *remaining* tells how many more emails are known as still pending.

B.10. land_collision

```
land_collision(vector position);
```

This event is raised when the object the script is attached to is colliding with the ground.

B.11. land_collision_end

```
land_collision_end(vector position);
```

This event is raised when the object the script is attached to stops colliding with the ground.

B.12. land_collision_start

```
land_collision_start(vector position);
```

This event is raised when the object the script is attached to begins to collide with the ground.

B.13. link_message

```
link_message(integer sender_number, integer number, string message, key id);
```

Triggered when object receives a link message via llMessageLinked library function call.

B.14. listen

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

This event is raised whenever a chat message matching the constraints passed in the llListen command is heard. The *name* and *id* of the speaker as well as the *message* are passed in as parameters. Channel 0 is the public chat channel that all avatars see as chat text. Channels 1 through 2,147,483,648 are private channels that are not sent to avatars but other scripts can listen on those channels.

B.15. money

```
money(key giver, integer amount);
```

This event is triggered when user *giver* has given an *amount* of Linden dollars to the object.

B.16. moving_end

```
moving_end(void);
```

Triggered whenever a object with this script stops moving.

B.17. moving_start

```
moving_start(void);
```

Triggered whenever a object with this script starts moving.

B.18. no_sensor

```
no_sensor(void);
```

This event is raised when sensors are active (via the llSensor library call) but are not sensing anything.

B.19. not_at_rot_target

```
not_at_rot_target(void);
```

When a target is set via the llRotTarget library call, but the script is outside the specified angle this event is raised.

B.20. not_at_target

```
not_at_target(void);
```

When a target is set via the llTarget library call, but the script is outside the specified range this event is raised.

B.21. object_rez

```
object_rez(key id);
```

Triggered when object rez-es in another object from its inventory via the llRezObject api. The *id* is the globally unique key for the object.

B.22. on_rez

```
on_rez(integer start_param);
```

Triggered whenever a object is rez-ed from inventory or by another object. The *start_param* is the parameter passed in from the call to llRezObject.

B.23. run_time_permissions

```
run_time_permissions(integer permissions);
```

Scripts need permission from either the owner or the avatar they wish to act on before they perform certain functions, such as debiting money from their owner's account, triggering an animation on an avatar, or capturing control inputs. The llRequestPermissions library function is used to request these permissions and the various permissions integer constants can be supplied. The integer returned to this event handler contains the current set of permissions flags, so if *permissions* eqal 0 then no permissions are set.

B.24. sensor

```
sensor(integer total_number);
```

This event is raised whenever objects matching the constraints of the llSensor command are detected. The number of detected objects is passed to the script in the *`total_number`* parameter. Information on those objects may be gathered via the llDetected* library functions.

B.25. state_entry

```
state_entry(void);
```

The state_entry event occurs whenever a new state is entered, including program start, and is always the first event handled.

B.26. state_exit

```
state_exit(void);
```

The state_exit event occurs whenever the state command is used to transition to another state. It is handled before the new state's state_entry event.

B.27. timer

```
timer(void);
```

This event is raised at regular intervals set by the llSetTimerEvent library function.

B.28. touch

```
touch(integer total_number);
```

This event is raised while a user is touching the object the script is attached to. The number of touching objects is passed to the script in the *`total_number`* parameter. Information on those objects may be gathered via the llDetected* library functions.

B.29. touch_end

```
touch_end(integer total_number);
```

This event is raised when a user stops touching the object the script is attached to. The number of touching objects is passed to the script in the *`total_number`* parameter. Information on those objects may be gathered via the llDetected* library functions.

B.30. touch_start

```
touch_start(integer total_number);
```

This event is raised when a user first touches the object the script is attached to. The number of touching objects is passed to the script in the *`total_number`* parameter. Information on those objects may be gathered via the llDetected* library functions.

Appendix C. Constants

To ease scripting, many useful constants are defined by LSL.

C.1. Boolean Constants

The boolean constants represent the values for TRUE and FALSE. LSL represents booleans as integer values 1 and 0 respectively. Since there is no boolean type these constants act as a scripting aid usually employed for testing variables which conceptually represent boolean values.

- TRUE
- FALSE

C.2. Status Constants

The status constants are used in the llSetStatus and llGetStatus library calls. These constants can be bitwise or'ed together when calling the library functions to set the same value to more than one status flag

Status Constants

STATUS_PHYSICS

> Controls whether the object moves physically. This controls the same flag that the ui checkbox for 'Physical' controls. The default is FALSE.

STATUS_PHANTOM

> Controls whether the object collides or not. Setting the value to TRUE makes the object non-colliding with all objects. It is a good idea to use this for most objects that move or rotate, but are non-physical. It is also useful for simulating volumetric lighting. The default is FALSE.

STATUS_ROTATE_X
STATUS_ROTATE_Y
STATUS_ROTATE_Z

> Controls whether the object can physically rotate around the specific axis or not. This flag has no meaning for non-physical objects. Set the value to FALSE if you want to disable rotation around that axis. The default is TRUE for a physical object.
>
> A useful example to think about when visualizing the effect is a 'sit-and-spin' device. They spin around the Z axis (up) but not around the X or Y axis.

STATUS_BLOCK_GRAB

Controls whether the object can be grabbed. A grab is the default action when in third person, and is available as the ’hand’ tool in build mode. This is useful for physical objects that you don’t want other people to be able to trivially disturb. The default if FALSE

STATUS_SANDBOX

Controls whether the object can cross region boundaries and move more than 20 meters from its creation point. The default if FALSE.

STATUS_DIE_AT_EDGE

Controls whether the object is returned to the owner’s inventory if it wanders off the edge of the world. It is useful to set this status TRUE for things like bullets or rockets. The default is TRUE

C.3. Object Type Constants

These constants can be combined using the binary ’|’ operator and are used in the llSensor and related calls.

Object Type Constants

AGENT

Objects in world that are agents.

ACTIVE

Objects in world that are running a script or currently pysically moving.

PASSIVE

Static in-world objects.

SCRIPTED

Scripted in-world objects.

C.4. Permission Constants

The permission constants are used for passing values to llRequestPermissions, determing the value of llGetPermissions, and explicitly passed to the run_time_permissions event. For many of the basic library functions to work, a specific permission must be enabled. The permission constants can be or’ed together to be used in conjunction.

Permission Constants

PERMISSION_DEBIT

If this permission is enabled, the object can successfully call llGiveMoney to debit the owner’s account.

PERMISSION_TAKE_CONTROLS

If this permission enabled, the object can successfully call the llTakeControls libray call.

PERMISSION_REMAP_CONTROLS

(not yet implemented)

PERMISSION_TRIGGER_ANIMATION

If this permission is enabled, the object can successfully call llStartAnimation for the avatar that owns this object.

PERMISSION_ATTACH

If this permission is enabled, the object can successfully call llAttachToAvatar to attach to the given avatar.

PERMISSION_RELEASE_OWNERSHIP

(not yet implemented)

PERMISSION_CHANGE_LINKS

If this permission is enabled, the object can successfully call llCreateLink, llBreakLink, and llBreakAllLinks to change links to other objects.

PERMISSION_CHANGE_JOINTS

(not yet implemented)

PERMISSION_CHANGE_PERMISSIONS

(not yet implemented)

C.5. Inventory Constants

These constants can be used to refer to a specific inventory type in calls to llGetInventoryNumber andllGetInventoryName.

Inventory Constants

INVENTORY_TEXTURE
INVENTORY_SOUND
INVENTORY_OBJECT
INVENTORY_SCRIPT
INVENTORY_LANDMARK
INVENTORY_CLOTHING
INVENTORY_NOTECARD
INVENTORY_BODYPART

Each constant refers to the named type of inventory.

C.6. Attachment Constants

These constants are used to refer to attachment points in calls to llAttachToAvatar.

Attachment Constants

ATTACH_CHEST

Attach to the avatar chest.

ATTACH_HEAD

Attach to the avatar head.

ATTACH_LSHOULDER

Attach to the avatar left shoulder.

ATTACH_RSHOULDER

Attach to the avatar right shoulder.

ATTACH_LHAND

Attach to the avatar left hand.

ATTACH_RHAND

Attach to the avatar right hand.

ATTACH_LFOOT

Attach to the avatar left foot.

ATTACH_RFOOT

Attach to the avatar right foot.

ATTACH_BACK

Attach to the avatar back.

ATTACH_PELVIS

Attach to the avatar pelvis.

ATTACH_MOUTH

Attach to the avatar mouth.

ATTACH_CHIN

Attach to the avatar chin.

ATTACH_LEAR

Attach to the avatar left ear.

ATTACH_REAR

Attach to the avatar right ear.

ATTACH_LEYE

Attach to the avatar left eye.

ATTACH_REYE

Attach to the avatar right eye.

ATTACH_NOSE

Attach to the avatar nose.

ATTACH_RUARM

Attach to the avatar right upper arm.

ATTACH_RLARM

Attach to the avatar right lower arm.

ATTACH_LUARM

Attach to the avatar left upper arm.

ATTACH_LLARM

Attach to the avatar left lower arm.

ATTACH_RHIP

Attach to the avatar right hip.

ATTACH_RULEG

Attach to the avatar right upper leg.

ATTACH_RLLEG

Attach to the avatar right lower leg.

ATTACH_LHIP

Attach to the avatar left hip.

ATTACH_LULEG

Attach to the avatar lower upper leg.

ATTACH_LLLEG

Attach to the avatar lower left leg.

ATTACH_BELLY

Attach to the avatar belly.

ATTACH_RPEC

Attach to the avatar right pectoral.

ATTACH_LPEC

Attach to the avatar left pectoral.

C.7. Land Constants

These constants are only used in calls to llModifyLand. The constants are equivalent to the similarly labelled user interface elements for editing land in the viewer.

Land Constants

LAND_LEVEL

Action to level the land.

LAND_RAISE

Action to raise the land.

LAND_LOWER

Action to lower the land.

LAND_SMALL_BRUSH

Use a small brush size.

LAND_MEDIUM_BRUSH

Use a medium brush size.

LAND_LARGE_BRUSH

Use a large brush size.

C.8. Link Constants

These constants are used in calls to llSetLinkColor and llMessageLinked.

Link Constants

LINK_SET

This targets every object in the linked set.

LINK_ROOT

This targets the root of the linked set.

LINK_ALL_OTHERS

This targets every object in the linked set except the object with the script.

LINK_ALL_CHILDREN

This targets every object except the root in the linked set.

C.9. Control Constants

These constants are used in llTakeControls as well as the control event handler.

Control Constants

CONTROL_FWD

Test for the avatar move forward control.

CONTROL_BACK

Test for the avatar move back control.

CONTROL_LEFT

Test for the avatar move left control.

CONTROL_RIGHT

Test for the avatar move right control.

CONTROL_ROT_LEFT

Test for the avatar rotate left control.

CONTROL_ROT_RIGHT

Test for the avatar rotate right control.

CONTROL_UP

Test for the avatar move up control.

CONTROL_DOWN

Test for the avatar move down control.

CONTROL_LBUTTON

Test for the avatar left button control.

CONTROL_ML_BUTTON

Test for the avatar left button control while in mouse look.

C.10. Change Constants

These constants are used in the changed event handler.

Change Constants

CHANGED_INVENTORY

The object inventory has changed.

CHANGED_ALLOWED_DROP

The object inventory has changed because an item was added through the llAllowInventoryDrop interface.

CHANGED_COLOR

The object color has changed.

CHANGED_SHAPE

The object shape has changed, eg, a box to a cylinder

CHANGED_SCALE

The object scale has changed.

CHANGED_TEXTURE

The texture offset, scale rotation, or simply the object texture has changed.

CHANGED_LINK

The object has linked or its links were broken.

C.11. Type Constants

These constants are used to determine the variable type stored in a heterogenous list. The value returned from llGetListEntryType can be used for comparison against these constants.

Type Constants

TYPE_INTEGER

The list entry is an integer.

TYPE_FLOAT

The list entry is a float.

TYPE_STRING

The list entry is a string.

TYPE_KEY

The list entry is a key.

TYPE_VECTOR

The list entry is a vector.

TYPE_ROTATION

The list entry is a rotation.

TYPE_INVALID

The list entry is invalid.

C.12. Agent Info Constants

Each of these constants represents a bit in the integer returned from the llGetAgentInfo function and can be used in an expression to determine the specified information about an agent.

Agent Info Constants

AGENT_FLYING

The agent is flying.

AGENT_ATTACHMENTS

The agent has attachments.

AGENT_SCRIPTED

The agent has scripted attachments.

C.13. Texture Animation Constants

These constants are used in the llSetTextureAnim api to control the animation mode.

Texture Animation Constants

ANIM_ON

Texture animation is on.

LOOP

Loop the texture animation.

REVERSE

Play animation in reverse direction.

PING_PONG

play animation going forwards, then backwards.

SMOOTH

slide in the X direction, instead of playing separate frames.

ROTATE

Animate texture rotation.

SCALE

Animate the texture scale.

C.14. Particle System Constants

These constants are used in calls to the llParticleSystem api to specify parameters.

Particle System Parameters

PSYS_PART_FLAGS

Each particle that is emitted by the particle system is simulated based on the following flags. To use multiple flags, bitwise or (|) them together.

PSYS_PART_FLAGS Values

PSYS_PART_INTERP_COLOR_MASK

Interpolate both the color and alpha from the start value to the end value.

PSYS_PART_INTERP_SCALE_MASK

Interpolate the particle scale from the start value to the end value.

PSYS_PART_WIND_MASK

Particles have their velocity damped towards the wind velocity.

PSYS_PART_BOUNCE_MASK

Particles bounce off of a plane at the object's Z height.

PSYS_PART_FOLLOW_SRC_MASK

The particle position is relative to the source object's position.

PSYS_PART_FOLLOW_VELOCITY_MASK

The particle orientation is rotated so the vertical axis faces towards the particle velocity.

PSYS_PART_TARGET_POS_MASK

The particle heads towards the location of the target object as defined by PSYS_SRC_TARGET_KEY.

PSYS_PART_EMISSIVE_MASK

The particle glows.

PSYS_PART_RANDOM_ACCEL_MASK

(not implemented)

PSYS_PART_RANDOM_VEL_MASK

(not implemented)

PSYS_PART_TRAIL_MASK

(not implemented)

PSYS_SRC_PATTERN

The pattern which is used to generate particles. Use one of the following values:

PSYS_SRC_PATTERN Values

PSYS_PART_SRC_PATTERN_DROP

Drop particles at the source position.

PSYS_PART_SRC_PATTERN_EXPLODE

Shoot particles out in all directions, using the burst parameters.

PSYS_PART_SRC_PATTERN_ANGLE

Shoot particles across a 2 dimensional area defined by the arc created from PSYS_SRC_OUTERANGLE. There will be an open area defined by PSYS_SRC_INNERANGLE within the larger arc.

PSYS_PART_SRC_PATTERN_ANGLE_CONE

Shoot particles out in a 3 dimensional cone with an outer arc of PSYS_SRC_OUTERANGLE and an inner open area defined by PSYS_SRC_INNERANGLE.

PSYS_PART_START_COLOR

a vector <r,g,b> which determines the starting color of the object.

PSYS_PART_START_ALPHA

a float which determines the starting alpha of the object.

PSYS_PART_END_COLOR

a vector <r, g, b> which determines the ending color of the object.

PSYS_PART_END_ALPHA

a float which determines the ending alpha of the object.

PSYS_PART_START_SCALE

a vector <sx, sy, z>, which is the starting size of the particle billboard in meters (z is ignored).

PSYS_PART_END_SCALE

a vector <sx, sy, z>, which is the ending size of the particle billboard in meters (z is ignored).

PSYS_PART_MAX_AGE

age in seconds of a particle at which it dies.

PSYS_SRC_ACCEL

a vector <x, y, z> which is the acceleration to apply on particles.

PSYS_SRC_TEXTURE

an asset name for the texture to use for the particles.

PSYS_SRC_BURST_RATE

how often to release a particle burst (float seconds).

PSYS_SRC_INNERANGLE

specifies the inner angle of the arc created by the PSYS_PART_SRC_PATTERN_ANGLE or PSYS_PART_SRC_PATTERN_ANGLE_CONE source pattern. The area specified will not have particles in it..

PSYS_SRC_OUTERANGLE

specifies the outer angle of the arc created by the PSYS_PART_SRC_PATTERN_ANGLE or PSYS_PART_SRC_PATTERN_ANGLE_CONE source pattern. The area between the outer and inner angle will be filled with particles..

PSYS_SRC_BURST_PART_COUNT

how many particles to release in a burst.

PSYS_SRC_BURST_RADIUS

what distance from the center of the object to create the particles.

PSYS_SRC_BURST_SPEED_MIN

minimum speed that a particle should be moving.

PSYS_SRC_BURST_SPEED_MAX

maximum speed that a particle should be moving.

PSYS_SRC_MAX_AGE

how long this particle system should last, 0.0 menas forever.

PSYS_SRC_TARGET_KEY

the key of a target object to move towards if PSYS_PART_TARGET_POS_MASK is enabled.

PSYS_SRC_OMEGA

Sets the angular velocity to rotate the axis that SRC_PATTERN_ANGLE and SRC_PATTERN_ANGLE_CONE use..

C.15. Agent Data Constants

These constants are used in calls to the llRequestAgentData api to collect information about an agent which will be provided in the dataserver event.

Texture Animation Constants

DATA_ONLINE

"1" for online "0" for offline.

DATA_NAME

The name of the agent.

DATA_BORN

The date the agent was born returned in ISO 8601 format of YYYY-MM-DD.

DATA_RATING

Returns the agent ratings as a comma separated string of six integers. They are:

1. Positive rated behavior
2. Negative rated behavior
3. Positive rated appearance
4. Negative rated appearance
5. Positive rated building
6. Negative rated building

C.16. Float Constants

LSL provides a small collection of floating point constants for use in float arithmetic. These constants are usually employed while performing trigonometric calculations, but are sometimes useful for other applications such as specifying arc radians to sensor or particle system functions.

Float Constants

PI

3.14159265 - The radians of a hemicircle.

TWO_PI

6.28318530 - The radians of a circle.

PI_BY_TWO

1.57079633 - The radians of a quarter circle.

DEG_TO_RAD

0.01745329 - Number of radians per degree. You can use this to convert degrees to radians by multiplying the degrees by this number.

RAD_TO_DEG

57.2957795 - Number of degrees per radian. You can use this number to convert radians to degrees by multiplying the radians by this number.

SQRT2

1.41421356 - The square root of 2.

C.17. Key Constant

There is only one key constant which acts as an invalid key: NULL_KEY.

C.18. Miscellaneous Integer Constants

There is one uncategorized integer constant which is used in some of the texturing and coloring api: ALL_SIDES

C.19. Miscellaneous String Constants

There is one uncategorized string constant which is used in the dataserver event: EOF

C.20. Vector Constant

There is only one vector constant which acts as a zero vector: ZERO_VECTOR = <0,0,0>.

C.21. Rotation Constant

There is only one rotation constant which acts as a zero rotation: ZERO_ROTATION = <0,0,0,1>.

C.22. Vehicle Parameters

Parameters

VEHICLE_LINEAR_FRICTION_TIMESCALE

A vector of timescales for exponential decay of the vehicle's linear velocity along its preferred axes of motion (at, left, up). Range = [0.07, inf) seconds for each element of the vector.

VEHICLE_ANGULAR_FRICTION_TIMESCALE

A vector of timescales for exponential decay of the vehicle's angular velocity about its preferred axes of motion (at, left, up). Range = [0.07, inf) seconds for each element of the vector.

VEHICLE_LINEAR_MOTOR_DIRECTION

The direction and magnitude (in preferred frame) of the vehicle's linear motor. The vehicle will accelerate (or decelerate if necessary) to match its velocity to its motor. Range of magnitude = [0, 30] meters/second.

VEHICLE_LINEAR_MOTOR_TIMESCALE

The timescale for exponential approach to full linear motor velocity.

VEHICLE_LINEAR_MOTOR_DECAY_TIMESCALE

The timescale for exponential decay of the linear motor's magnitude.

VEHICLE_ANGULAR_MOTOR_DIRECTION

The direction and magnitude (in preferred frame) of the vehicle's angular motor.The vehicle will accelerate (or decelerate if necessary) to match its velocity to its motor.

VEHICLE_ANGULAR_MOTOR_TIMESCALE

The timescale for exponential approach to full angular motor velocity.

VEHICLE_ANGULAR_MOTOR_DECAY_TIMESCALE

The timescale for exponential decay of the angular motor's magnitude.

VEHICLE_HOVER_HEIGHT

The height (above the terrain or water, or global) at which the vehicle will try to hover.

VEHICLE_HOVER_HEIGHT_EFFICIENCY

A slider between minimum (0.0 = bouncy) and maximum (1.0 = fast as possible) damped motion of the hover behavior.

VEHICLE_HOVER_HEIGHT_TIMESCALE

The period of bounce (or timescale of exponential approach, depending on the hover efficiency) for the vehicle to hover to the proper height.

VEHICLE_BUOYANCY

A slider between minimum (0.0) and maximum anti-gravity (1.0).

VEHICLE_LINEAR_DEFLECTION_EFFICIENCY

A slider between minimum (0.0) and maximum (1.0) deflection of linear velocity. That is, it's a simple scalar for modulating the strength of linear deflection.

VEHICLE_LINEAR_DEFLECTION_TIMESCALE

The timescale for exponential success of linear deflection deflection. It is another way to specify how much time it takes for the vehicle's linear velocity to be redirected to it's preferred axis of motion.

VEHICLE_ANGULAR_DEFLECTION_EFFICIENCY

A slider between minimum (0.0) and maximum (1.0) deflection of angular orientation. That is, it's a simple scalar for modulating the strength of angular deflection such that the vehicle's preferred axis of motion points toward it's real velocity.

VEHICLE_ANGULAR_DEFLECTION_TIMESCALE

The timescale for exponential success of linear deflection deflection. It's another way to specify the strength of the vehicle's tendency to reorient itself so that it's preferred axis of motion agrees with it's true velocity.

VEHICLE_VERTICAL_ATTRACTION_EFFICIENCY

A slider between minimum (0.0 = wobbly) and maximum (1.0 = firm as possible) stability of the vehicle to keep itself upright.

VEHICLE_VERTICAL_ATTRACTION_TIMESCALE

The period of wobble, or timescale for exponential approach, of the vehicle to rotate such that it's preferred "up" axis is oriented along the world's "up" axis.

VEHICLE_BANKING_EFFICIENCY

A slider between anti (-1.0), none (0.0), and maxmum (1.0) banking strength.

VEHICLE_BANKING_MIX

A slider between static (0.0) and dynamic (1.0) banking. "Static" means the banking scales only with the angle of roll, whereas "dynamic" is a term that also scales with the vehicle's linear speed.

VEHICLE_BANKING_TIMESCALE

The timescale for banking to exponentially approach it's maximum effect. This is another way to scale the strength of the banking effect, however it affects the term that is proportional to the difference between what the banking behavior is trying to do, and what the vehicle is actually doing.

VEHICLE_REFERENCE_FRAME

A rotation of the vehicle's preferred axes of motion and orientation (at, left, up) with respect to the vehicle's local frame (x, y, z).

C.23. Vehicle Flags

Flags

VEHICLE_FLAG_NO_DEFLECTION_UP

This flag prevents linear deflection parallel to world z-axis. This is useful for preventing ground vehicles with large linear deflection, like bumper cars, from climbing their linear deflection into the sky.

VEHICLE_FLAG_LIMIT_ROLL_ONLY

For vehicles with vertical attractor that want to be able to climb/dive, for instance, airplanes that want to use the banking feature.

VEHICLE_FLAG_HOVER_WATER_ONLY

Ignore terrain height when hovering.

VEHICLE_FLAG_HOVER_TERRAIN_ONLY

Ignore water height when hovering.

VEHICLE_FLAG_HOVER_GLOBAL_HEIGHT

Hover at global height.

VEHICLE_FLAG_HOVER_UP_ONLY

Hover doesn't push down. Use this flag for hovering vehicles that should be able to jump above their hover height.

VEHICLE_FLAG_LIMIT_MOTOR_UP

Prevents ground vehicles from motoring into the sky.

C.24. Vehicle Types

Types

VEHICLE_TYPE_SLED

Simple vehicle that bumps along the ground, and likes to move along it’s local x-axis.

```
// most friction for left-right, least for up-down
llSetVehicleVectorParam( VEHICLE_LINEAR_FRICTION_TIMESCALE, <30, 1, 1000> );

// no angular friction
llSetVehicleVectorParam( VEHICLE_ANGULAR_FRICTION_TIMESCALE, <1000, 1000, 1000> );

// no linear motor
llSetVehicleVectorParam( VEHICLE_LINEAR_MOTOR_DIRECTION, <0, 0, 0> );
llSetVehicleFloatParam( VEHICLE_LINEAR_MOTOR_TIMESCALE, 1000 );
llSetVehicleFloatParam( VEHICLE_LINEAR_MOTOR_DECAY_TIMESCALE, 120 );

// no agular motor
llSetVehicleVectorParam( VEHICLE_ANGULAR_MOTOR_DIRECTION, <0, 0, 0> );
llSetVehicleFloatParam( VEHICLE_ANGULAR_MOTOR_TIMESCALE, 1000 );
llSetVehicleFloatParam( VEHICLE_ANGULAR_MOTOR_DECAY_TIMESCALE, 120 );

// no hover (but with timescale of 10 sec if enabled)
llSetVehicleFloatParam( VEHICLE_HOVER_HEIGHT, 0 );
llSetVehicleFloatParam( VEHICLE_HOVER_EFFICIENCY, 10 );
llSetVehicleFloatParam( VEHICLE_HOVER_TIMESCALE, 10 );
llSetVehicleFloatParam( VEHICLE_BUOYANCY, 0 );

// maximum linear deflection with timescale of 1 second
llSetVehicleFloatParam( VEHICLE_LINEAR_DEFLECTION_EFFICIENCY, 1 );
llSetVehicleFloatParam( VEHICLE_LINEAR_DEFLECTION_TIMESCALE, 1 );

// no angular deflection
llSetVehicleFloatParam( VEHICLE_ANGULAR_DEFLECTION_EFFICIENCY, 0 );
llSetVehicleFloatParam( VEHICLE_ANGULAR_DEFLECTION_TIMESCALE, 10 );

// no vertical attractor (doesn’t mind flipping over)
llSetVehicleFloatParam( VEHICLE_VERTICAL_ATTRACTION_EFFICIENCY, 1 );
llSetVehicleFloatParam( VEHICLE_VERTICAL_ATTRACTION_TIMESCALE, 1000 );

// no banking
llSetVehicleFloatParam( VEHICLE_BANKING_EFFICIENCY, 0 );
llSetVehicleFloatParam( VEHICLE_BANKING_MIX, 1 );
llSetVehicleFloatParam( VEHICLE_BANKING_TIMESCALE, 10 );

// default rotation of local frame
llSetVehicleRotationParam( VEHICLE_REFERENCE_FRAME, <0, 0, 0, 1> );

// remove these flags
llRemoveVehicleFlags( VEHICLE_FLAG_HOVER_WATER_ONLY
                      | VEHICLE_FLAG_HOVER_TERRAIN_ONLY
                      | VEHICLE_FLAG_HOVER_GLOBAL_HEIGHT
                      | VEHICLE_FLAG_HOVER_UP_ONLY );

// set these flags (the limit_roll flag will have no effect
```

```
// until banking is enabled, if ever)
llSetVehicleFlags( VEHICLE_FLAG_NO_DEFLECTION_UP
                   | VEHICLE_FLAG_LIMIT_ROLL_ONLY
                   | VEHICLE_FLAG_LIMIT_MOTOR_UP );
```

VEHICLE_TYPE_CAR

Another vehicle that bounces along the ground but needs the motors to be driven from external controls or timer events.

```
// most friction for left-right, least for up-down
llSetVehicleVectorParam( VEHICLE_LINEAR_FRICTION_TIMESCALE, <100, 2, 1000> );

// no angular friction
llSetVehicleVectorParam( VEHICLE_ANGULAR_FRICTION_TIMESCALE, <1000, 1000, 1000> );

// linear motor wins after about a second, decays after about a minute
llSetVehicleVectorParam( VEHICLE_LINEAR_MOTOR_DIRECTION, <0, 0, 0> );
llSetVehicleFloatParam( VEHICLE_LINEAR_MOTOR_TIMESCALE, 1 );
llSetVehicleFloatParam( VEHICLE_LINEAR_MOTOR_DECAY_TIMESCALE, 60 );

// agular motor wins after a second, decays in less time than that
llSetVehicleVectorParam( VEHICLE_ANGULAR_MOTOR_DIRECTION, <0, 0, 0> );
llSetVehicleFloatParam( VEHICLE_ANGULAR_MOTOR_TIMESCALE, 1 );
llSetVehicleFloatParam( VEHICLE_ANGULAR_MOTOR_DECAY_TIMESCALE, 0.8 );

// no hover
llSetVehicleFloatParam( VEHICLE_HOVER_HEIGHT, 0 );
llSetVehicleFloatParam( VEHICLE_HOVER_EFFICIENCY, 0 );
llSetVehicleFloatParam( VEHICLE_HOVER_TIMESCALE, 1000 );
llSetVehicleFloatParam( VEHICLE_BUOYANCY, 0 );

// maximum linear deflection with timescale of 2 seconds
llSetVehicleFloatParam( VEHICLE_LINEAR_DEFLECTION_EFFICIENCY, 1 );
llSetVehicleFloatParam( VEHICLE_LINEAR_DEFLECTION_TIMESCALE, 2 );

// no angular deflection
llSetVehicleFloatParam( VEHICLE_ANGULAR_DEFLECTION_EFFICIENCY, 0 );
llSetVehicleFloatParam( VEHICLE_ANGULAR_DEFLECTION_TIMESCALE, 10 );

// critically damped vertical attractor
llSetVehicleFloatParam( VEHICLE_VERTICAL_ATTRACTION_EFFICIENCY, 1 );
llSetVehicleFloatParam( VEHICLE_VERTICAL_ATTRACTION_TIMESCALE, 10 );

// weak negative critically damped banking
llSetVehicleFloatParam( VEHICLE_BANKING_EFFICIENCY, -0.2 );
llSetVehicleFloatParam( VEHICLE_BANKING_MIX, 1 );
llSetVehicleFloatParam( VEHICLE_BANKING_TIMESCALE, 1 );

// default rotation of local frame
llSetVehicleRotationParam( VEHICLE_REFERENCE_FRAME, <0, 0, 0, 1> );

// remove these flags
llRemoveVehicleFlags( VEHICLE_FLAG_HOVER_WATER_ONLY
                      | VEHICLE_FLAG_HOVER_TERRAIN_ONLY
```

```
                        | VEHICLE_FLAG_HOVER_GLOBAL_HEIGHT);

    // set these flags
    llSetVehicleFlags( VEHICLE_FLAG_NO_DEFLECTION_UP
                     | VEHICLE_FLAG_LIMIT_ROLL_ONLY
                     | VEHICLE_FLAG_HOVER_UP_ONLY
                     | VEHICLE_FLAG_LIMIT_MOTOR_UP );
```

VEHICLE_TYPE_BOAT

Hovers over water with lots of friction and some anglar deflection.

```
    // least for forward-back, most friction for up-down
    llSetVehicleVectorParam( VEHICLE_LINEAR_FRICTION_TIMESCALE, <10, 3, 2> );

    // uniform angular friction (setting it as a scalar rather than a vector)
    llSetVehicleFloatParam( VEHICLE_ANGULAR_FRICTION_TIMESCALE, 10 );

    // linear motor wins after about five seconds, decays after about a minute
    llSetVehicleVectorParam( VEHICLE_LINEAR_MOTOR_DIRECTION, <0, 0, 0> );
    llSetVehicleFloatParam( VEHICLE_LINEAR_MOTOR_TIMESCALE, 5 );
    llSetVehicleFloatParam( VEHICLE_LINEAR_MOTOR_DECAY_TIMESCALE, 60 );

    // agular motor wins after four seconds, decays in same amount of time
    llSetVehicleVectorParam( VEHICLE_ANGULAR_MOTOR_DIRECTION, <0, 0, 0> );
    llSetVehicleFloatParam( VEHICLE_ANGULAR_MOTOR_TIMESCALE, 4 );
    llSetVehicleFloatParam( VEHICLE_ANGULAR_MOTOR_DECAY_TIMESCALE, 4 );

    // hover
    llSetVehicleFloatParam( VEHICLE_HOVER_HEIGHT, 0 );
    llSetVehicleFloatParam( VEHICLE_HOVER_EFFICIENCY, 0.5 );
    llSetVehicleFloatParam( VEHICLE_HOVER_TIMESCALE, 2.0 );
    llSetVehicleFloatParam( VEHICLE_BUOYANCY, 1 );

    // halfway linear deflection with timescale of 3 seconds
    llSetVehicleFloatParam( VEHICLE_LINEAR_DEFLECTION_EFFICIENCY, 0.5 );
    llSetVehicleFloatParam( VEHICLE_LINEAR_DEFLECTION_TIMESCALE, 3 );

    // angular deflection
    llSetVehicleFloatParam( VEHICLE_ANGULAR_DEFLECTION_EFFICIENCY, 0.5 );
    llSetVehicleFloatParam( VEHICLE_ANGULAR_DEFLECTION_TIMESCALE, 5 );

    // somewhat bounscy vertical attractor
    llSetVehicleFloatParam( VEHICLE_VERTICAL_ATTRACTION_EFFICIENCY, 0.5 );
    llSetVehicleFloatParam( VEHICLE_VERTICAL_ATTRACTION_TIMESCALE, 5 );

    // weak negative damped banking
    llSetVehicleFloatParam( VEHICLE_BANKING_EFFICIENCY, -0.3 );
    llSetVehicleFloatParam( VEHICLE_BANKING_MIX, 0.8 );
    llSetVehicleFloatParam( VEHICLE_BANKING_TIMESCALE, 1 );

    // default rotation of local frame
    llSetVehicleRotationParam( VEHICLE_REFERENCE_FRAME, <0, 0, 0, 1> );

    // remove these flags
```

```
llRemoveVehicleFlags( VEHICLE_FLAG_HOVER_TERRAIN_ONLY
                      | VEHICLE_FLAG_LIMIT_ROLL_ONLY
                      | VEHICLE_FLAG_HOVER_GLOBAL_HEIGHT);

// set these flags
llSetVehicleFlags( VEHICLE_FLAG_NO_DEFLECTION_UP
                   | VEHICLE_FLAG_HOVER_WATER_ONLY
                   | VEHICLE_FLAG_HOVER_UP_ONLY
                   | VEHICLE_FLAG_LIMIT_MOTOR_UP );
```

VEHICLE_TYPE_AIRPLANE

Uses linear deflection for lift, no hover, and banking to turn.

```
// very little friction along forward-back axis
llSetVehicleVectorParam( VEHICLE_LINEAR_FRICTION_TIMESCALE, <200, 10, 5> );

// uniform angular friction
llSetVehicleFloatParam( VEHICLE_ANGULAR_FRICTION_TIMESCALE, 20 );

// linear motor
llSetVehicleVectorParam( VEHICLE_LINEAR_MOTOR_DIRECTION, <0, 0, 0> );
llSetVehicleFloatParam( VEHICLE_LINEAR_MOTOR_TIMESCALE, 2 );
llSetVehicleFloatParam( VEHICLE_LINEAR_MOTOR_DECAY_TIMESCALE, 60 );

// agular motor
llSetVehicleVectorParam( VEHICLE_ANGULAR_MOTOR_DIRECTION, <0, 0, 0> );
llSetVehicleFloatParam( VEHICLE_ANGULAR_MOTOR_TIMESCALE, 4 );
llSetVehicleFloatParam( VEHICLE_ANGULAR_MOTOR_DECAY_TIMESCALE, 8 );

// no hover
llSetVehicleFloatParam( VEHICLE_HOVER_HEIGHT, 0 );
llSetVehicleFloatParam( VEHICLE_HOVER_EFFICIENCY, 0.5 );
llSetVehicleFloatParam( VEHICLE_HOVER_TIMESCALE, 1000 );
llSetVehicleFloatParam( VEHICLE_BUOYANCY, 0 );

// linear deflection
llSetVehicleFloatParam( VEHICLE_LINEAR_DEFLECTION_EFFICIENCY, 0.5 );
llSetVehicleFloatParam( VEHICLE_LINEAR_DEFLECTION_TIMESCALE, 0.5 );

// angular deflection
llSetVehicleFloatParam( VEHICLE_ANGULAR_DEFLECTION_EFFICIENCY, 1.0 );
llSetVehicleFloatParam( VEHICLE_ANGULAR_DEFLECTION_TIMESCALE, 2.0 );

// vertical attractor
llSetVehicleFloatParam( VEHICLE_VERTICAL_ATTRACTION_EFFICIENCY, 0.9 );
llSetVehicleFloatParam( VEHICLE_VERTICAL_ATTRACTION_TIMESCALE, 2 );

// banking
llSetVehicleFloatParam( VEHICLE_BANKING_EFFICIENCY, 1 );
llSetVehicleFloatParam( VEHICLE_BANKING_MIX, 0.7 );
llSetVehicleFloatParam( VEHICLE_BANKING_TIMESCALE, 2 );

// default rotation of local frame
llSetVehicleRotationParam( VEHICLE_REFERENCE_FRAME, <0, 0, 0, 1> );
```

```
// remove these flags
llRemoveVehicleFlags( VEHICLE_FLAG_NO_DEFLECTION_UP
                      | VEHICLE_FLAG_HOVER_WATER_ONLY
                      | VEHICLE_FLAG_HOVER_TERRAIN_ONLY
                      | VEHICLE_FLAG_HOVER_GLOBAL_HEIGHT
                      | VEHICLE_FLAG_HOVER_UP_ONLY
                      | VEHICLE_FLAG_LIMIT_MOTOR_UP );

// set these flags
llSetVehicleFlags( VEHICLE_FLAG_LIMIT_ROLL_ONLY );
```

VEHICLE_TYPE_BALLOON

Hover, and friction, but no deflection.

```
// uniform linear friction
llSetVehicleFloatParam( VEHICLE_LINEAR_FRICTION_TIMESCALE, 5 );

// uniform angular friction
llSetVehicleFloatParam( VEHICLE_ANGULAR_FRICTION_TIMESCALE, 10 );

// linear motor
llSetVehicleVectorParam( VEHICLE_LINEAR_MOTOR_DIRECTION, <0, 0, 0> );
llSetVehicleFloatParam( VEHICLE_LINEAR_MOTOR_TIMESCALE, 5 );
llSetVehicleFloatParam( VEHICLE_LINEAR_MOTOR_DECAY_TIMESCALE, 60 );

// agular motor
llSetVehicleVectorParam( VEHICLE_ANGULAR_MOTOR_DIRECTION, <0, 0, 0> );
llSetVehicleFloatParam( VEHICLE_ANGULAR_MOTOR_TIMESCALE, 6 );
llSetVehicleFloatParam( VEHICLE_ANGULAR_MOTOR_DECAY_TIMESCALE, 10 );

// hover
llSetVehicleFloatParam( VEHICLE_HOVER_HEIGHT, 5 );
llSetVehicleFloatParam( VEHICLE_HOVER_EFFICIENCY, 0.8 );
llSetVehicleFloatParam( VEHICLE_HOVER_TIMESCALE, 10 );
llSetVehicleFloatParam( VEHICLE_BUOYANCY, 1 );

// no linear deflection
llSetVehicleFloatParam( VEHICLE_LINEAR_DEFLECTION_EFFICIENCY, 0 );
llSetVehicleFloatParam( VEHICLE_LINEAR_DEFLECTION_TIMESCALE, 5 );

// no angular deflection
llSetVehicleFloatParam( VEHICLE_ANGULAR_DEFLECTION_EFFICIENCY, 0 );
llSetVehicleFloatParam( VEHICLE_ANGULAR_DEFLECTION_TIMESCALE, 5 );

// no vertical attractor
llSetVehicleFloatParam( VEHICLE_VERTICAL_ATTRACTION_EFFICIENCY, 1 );
llSetVehicleFloatParam( VEHICLE_VERTICAL_ATTRACTION_TIMESCALE, 1000 );

// no banking
llSetVehicleFloatParam( VEHICLE_BANKING_EFFICIENCY, 0 );
llSetVehicleFloatParam( VEHICLE_BANKING_MIX, 0.7 );
llSetVehicleFloatParam( VEHICLE_BANKING_TIMESCALE, 5 );
```

```
// default rotation of local frame
llSetVehicleRotationParam( VEHICLE_REFERENCE_FRAME, <0, 0, 0, 1> );

// remove all flags
llRemoveVehicleFlags( VEHICLE_FLAG_NO_DEFLECTION_UP
                      | VEHICLE_FLAG_HOVER_WATER_ONLY
                      | VEHICLE_FLAG_LIMIT_ROLL_ONLY
                      | VEHICLE_FLAG_HOVER_TERRAIN_ONLY
                      | VEHICLE_FLAG_HOVER_GLOBAL_HEIGHT
                      | VEHICLE_FLAG_HOVER_UP_ONLY
                      | VEHICLE_FLAG_LIMIT_MOTOR_UP );
```