# Logitech Gaming Steering Wheel SDK V1.00

## Overview and Reference

## Contents

## Overview
The Logitech Gaming Steering Wheel SDK enables applications such as games to control various types of game controllers (USB/gameport wheels/joysticks/game pads, Force Feedback enabled or not).
By using the Steering Wheel SDK you have the guarantee that all wheels and joysticks will function flawlessly. No more situations where force feedback in a game behaves very differently from one wheel/joystick to another, which  in turn results in user frustration and product returns. Please note that the SDK will work properly only if the Logitech Gaming Software is installed. Visit http://www.logitech.com/en-us/gaming for more informations.

### SDK Package
The following files are included:
- LogitechSteeringWheel.h: C/C++ header file containing function prototypes
- LogitechSteeringWheel.lib: companion lib file to access DLL exported functions (32 and 64 bit)
- LogitechSteeringWheel.dll: library of SDK functions (32 and 64 bit)

### Requirements
The Logitech Gaming Steering Wheel SDK can be used on the following platforms:
- Windows XP SP2 (32-bit and 64-bit)
- Windows Vista (32-bit and 64-bit)
- Windows 7 (32-bit and 64-bit)
- Windows 8 (32-bit and 64-bit)

The Logitech Gaming Steering Wheel SDK is a C based interface and is designed for use by C/C++ programmers.  Familiarity with Windows programming is required.

### Interfacing with the SDK

#### Using LogitechSteeringWheel.h and LogitechSteeringWheel.lib to access LogitechSteeringWheel.dll
The application can include LogitechSteeringWheel.h and link to LogitechSteeringWheel.lib (see "Sample usage of the SDK" further below or sample program in Samples folder). Installation folder for the DLL needs to be the same as the main executable, or needs to be part of the Path in the system environment.

#### Using LogitechSteeringWheel.dll directly
Alternatively the game can use the DLL directly by loading it via LoadLibrary, and accessing its functions using GetProcAddress (see "Sample usage of the SDK" further below or sample program in Samples folder).

### Multiple clients using the SDK at the same time
The SDK allows only one client to control a device at any given time.  In case two applications try to initialize the SDK for the device, only the first will succeed. The second application's initialization will fail.

## Do's and Don'ts
These are a few guidelines that may help you implement 'better' support in your game:
- The function LogiSteeringInitialize() will try to get your application/game main window handler. It could fail, because when you are calling the function the main window may not be in the foreground yet. If LogiSteeringInitialize() returns false, the initialization will be attempted in any

of the next LogiUpdate() function calls. When the window will be in foreground and the SDK can be initialized, LogiUpdate() will return true.

## Sample usage of the SDK

### Using header and lib

```
#pragma comment(lib, "LogitechSteeringWheel.lib")

#include "LogiSteeringWheel.h"

//the parameter determine whether you'll use X-input or not
LogiSteeringInitialize(TRUE);

//this is your main loop
while(!done){
        if(LogiUpdate()){
            //the main window handler has been found, you can now call any function
            //from the steering wheel sdk
        }
}
```

### Using DLL directly

```
#include <dinput.h> //needs dinput.h in order to use the DIJOYSTATE2 struct

#define LOGI_MAX_CONTROLLERS  2
#define LOGI_FORCE_NONE  -1
#define LOGI_FORCE_SPRING  0
#define LOGI_FORCE_CONSTANT  1
#define LOGI_FORCE_DAMPER  2
#define LOGI_FORCE_SIDE_COLLISION  3
#define LOGI_FORCE_FRONTAL_COLLISION  4
#define LOGI_FORCE_DIRT_ROAD  5
#define LOGI_FORCE_BUMPY_ROAD  6
#define LOGI_FORCE_SLIPPERY_ROAD  7
#define LOGI_FORCE_SURFACE_EFFECT  8
#define LOGI_NUMBER_FORCE_EFFECTS  9
#define LOGI_FORCE_SOFTSTOP  10
#define LOGI_FORCE_CAR_AIRBORNE  11
#define LOGI_PERIODICTYPE_NONE  -1
#define LOGI_PERIODICTYPE_SINE  0
#define LOGI_PERIODICTYPE_SQUARE  1
#define LOGI_PERIODICTYPE_TRIANGLE 2

typedef enum
{
        LOGI_DEVICE_TYPE_NONE = -1, LOGI_DEVICE_TYPE_WHEEL, LOGI_DEVICE_TYPE_JOYSTICK,
LOGI_DEVICE_TYPE_GAMEPAD, LOGI_DEVICE_TYPE_OTHER,
        LOGI_NUMBER_DEVICE_TYPES
} LogiDeviceType;

typedef enum
{
        LOGI_MANUFACTURER_NONE = -1, LOGI_MANUFACTURER_LOGITECH, LOGI_MANUFACTURER_MICROSOFT,
LOGI_MANUFACTURER_OTHER
```

```
} LogiManufacturerName;

typedef enum
{
        LOGI_MODEL_G27,
        LOGI_MODEL_DRIVING_FORCE_GT,
        LOGI_MODEL_G25,
        LOGI_MODEL_MOMO_RACING,
        LOGI_MODEL_MOMO_FORCE,
        LOGI_MODEL_DRIVING_FORCE_PRO,
        LOGI_MODEL_DRIVING_FORCE,
        LOGI_MODEL_NASCAR_RACING_WHEEL,
        LOGI_MODEL_FORMULA_FORCE,
        LOGI_MODEL_FORMULA_FORCE_GP,
        LOGI_MODEL_FORCE_3D_PRO,
        LOGI_MODEL_EXTREME_3D_PRO,
        LOGI_MODEL_FREEDOM_24,
        LOGI_MODEL_ATTACK_3,
        LOGI_MODEL_FORCE_3D,
        LOGI_MODEL_STRIKE_FORCE_3D,
        LOGI_MODEL_G940_JOYSTICK,
        LOGI_MODEL_G940_THROTTLE,
        LOGI_MODEL_G940_PEDALS,
        LOGI_MODEL_RUMBLEPAD,
        LOGI_MODEL_RUMBLEPAD_2,
        LOGI_MODEL_CORDLESS_RUMBLEPAD_2,
        LOGI_MODEL_CORDLESS_GAMEPAD,
        LOGI_MODEL_DUAL_ACTION_GAMEPAD,
        LOGI_MODEL_PRECISION_GAMEPAD_2,
        LOGI_MODEL_CHILLSTREAM,
        LOGI_NUMBER_MODELS
} LogiModelName;

typedef struct LogiControllerPropertiesData
{
        bool forceEnable;
        int overallGain;
        int springGain;
        int damperGain;
        bool defaultSpringEnabled;
        int defaultSpringGain;
        bool combinePedals;
        int wheelRange;
        bool gameSettingsEnabled;
        bool allowGameSettings;
}LogiControllerPropertiesData;

typedef struct DIJOYSTATE2ENGINES {
    int                     lX;                 /* x-axis position              */
    int                     lY;                 /* y-axis position              */
    int                     lZ;                 /* z-axis position              */
    int                     lRx;                /* x-axis rotation              */
    int                     lRy;                /* y-axis rotation              */
    int                     lRz;                /* z-axis rotation              */
    int                     rglSlider[2];       /* extra axes positions         */
    unsigned int      rgdwPOV[4];         /* POV directions               */
    unsigned char   rgbButtons[128];    /* 128 buttons                  */
    int                     lVX;                /* x-axis velocity              */
    int                     lVY;                /* y-axis velocity              */
    int                     lVZ;                /* z-axis velocity              */
    int                     lVRx;               /* x-axis angular velocity      */
    int                     lVRy;               /* y-axis angular velocity      */
    int                     lVRz;               /* z-axis angular velocity      */
    int                     rglVSlider[2];      /* extra axes velocities        */
    int                     lAX;                /* x-axis acceleration          */
    int                     lAY;                /* y-axis acceleration          */
    int                         lAZ;                /* z-axis acceleration         */
    int                         lARx;               /* x-axis angular acceleration  */
    int                         lARy;               /* y-axis angular acceleration  */
    int                         lARz;               /* z-axis angular acceleration  */
    int                         rglASlider[2];      /* extra axes accelerations     */
    int                         lFX;                /* x-axis force                 */
    int                         lFY;                /* y-axis force                 */
    int                         lFZ;                /* z-axis force                 */
    int                         lFRx;               /* x-axis torque                */
    int                         lFRy;               /* y-axis torque                */
    int                         lFRz;               /* z-axis torque                */
    int                         rglFSlider[2];      /* extra axes forces            */
} DIJOYSTATE2ENGINES;


typedef bool (* LPFNDLLINIT)(BOOL);
typedef bool (* LPFNDLLUPDATE)();
typedef bool (* LPFNDLLISCONNECTED)(int);
typedef bool (* LPFNDLLISDEVICECONNECTED)( int, int);
typedef bool (* LPFNDLLISMANUFACTURERCONNECTED)( int, int);
typedef bool (* LPFNDLLISMODELCONNECTED)( int, int);
typedef bool (* LPFNDLLGETNONLINVALUES)( int, int);
typedef bool (* LPFNDLLLGETCONTRPROPERTIES)( int, LogiControllerPropertiesData &);
typedef DIJOYSTATE2* (* LPFNDLLLGETSTATE)( int);
typedef DIJOYSTATE2ENGINES* (* LPFNDLLLGETSTATEENGINES)( int);
typedef bool (* LPFNDLLPLAYLEDS)( int, float, float, float);
typedef bool (*LPFNDLLPLAYFORCE)( int, int);
typedef bool (*LPFNDLLISPLAYING)( int, int);
typedef bool (* LPFNDLLPLAYSPRINGFORCE)( int, int, int, int);
typedef bool (* LPFNDLLPLAYSURFACEEFFECT)( int, int, int, int);
typedef bool (* LPFNDLLSTOPFORCE)( int);
typedef bool (* LPFNDLLGETBUTTON)( int, int);
typedef bool (* LPFNDLLSETCONTRPROPERTIES)(LogiControllerPropertiesData);
typedef int (* LPFNDLLGETSHIFTERMODE)( int);


LPFNDLLINIT LogiSteeringInitialize = NULL;
LPFNDLLUPDATE LogiUpdate = NULL;
LPFNDLLISCONNECTED LogiIsConnected = NULL;
LPFNDLLISDEVICECONNECTED LogiIsDeviceConnected = NULL;
LPFNDLLISMANUFACTURERCONNECTED LogiIsManufacturerConnected = NULL;
LPFNDLLISMODELCONNECTED LogiIsModelConnected = NULL;
LPFNDLLGETNONLINVALUES LogiGenerateNonLinearValues = NULL;
LPFNDLLLGETCONTRPROPERTIES LogiGetCurrentControllerProperties = NULL;
LPFNDLLLGETSTATE LogiGetState = NULL;
LPFNDLLLGETSTATEENIGNES LogiGetStateENGINES = NULL;
LPFNDLLPLAYLEDS LogiPlayLeds = NULL;
LPFNDLLPLAYFORCE LogiPlayConstantForce = NULL, LogiPlayDamperForce = NULL,
        LogiPlaySideCollisionForce = NULL, LogiPlayFrontalCollisionForce = NULL,
        LogiPlayDirtRoadEffect = NULL, LogiPlayBumpyRoadEffect = NULL,
        LogiPlaySlipperyRoadEffect = NULL, LogiPlaySoftstopForce = NULL;
LPFNDLLISPLAYING LogiIsPlaying = NULL;
LPFNDLLPLAYSPRINGFORCE LogiPlaySpringForce = NULL;
LPFNDLLPLAYSURFACEEFFECT LogiPlaySurfaceEffect = NULL;
LPFNDLLSTOPFORCE LogiStopSpringForce = NULL, LogiStopConstantForce = NULL,
        LogiStopDamperForce = NULL, LogiStopDirtRoadEffect = NULL,
        LogiStopBumpyRoadEffect = NULL, LogiStopSlipperyRoadEffect = NULL,
        LogiStopSurfaceEffect = NULL, LogiStopCarAirborne = NULL,
        LogiStopSoftstopForce = NULL, LogiPlayCarAirborne = NULL;
LPFNDLLGETBUTTON LogiButtonTriggered = NULL, LogiButtonReleased = NULL,
        LogiButtonIsPressed = NULL;
LPFNDLLSETCONTRPROPERTIES LogiSetPreferredControllerProperties = NULL;
LPFNDLLGETSHIFTERMODE LogiGetShifterMode = NULL;

HINSTANCE logiDllHandle = LoadLibrary(L"LogitechSteeringWheel.dll");
if (logiDllHandle != NULL)
```

```
{
        LogiSteeringInitialize = (LPFNDLLINIT)GetProcAddress(logiDllHandle, "LogiSteeringInitialize");
        LogiUpdate = (LPFNDLLUPDATE)GetProcAddress(logiDllHandle, "LogiUpdate");
        LogiIsConnected = (LPFNDLLISCONNECTED)GetProcAddress(logiDllHandle, "LogiIsConnected");
        LogiIsDeviceConnected = (LPFNDLLISDEVICECONNECTED)GetProcAddress(logiDllHandle,
"LogiIsDeviceConnected");
        LogiIsManufacturerConnected = (LPFNDLLISMANUFACTURERCONNECTED)GetProcAddress(logiDllHandle,
"LogiIsManufacturerConnected");
        LogiIsModelConnected = (LPFNDLLISMODELCONNECTED)GetProcAddress(logiDllHandle,
"LogiIsModelConnected");
        LogiGenerateNonLinearValues = (LPFNDLLGETNONLINVALUES)GetProcAddress(logiDllHandle,
"LogiGenerateNonLinearValues");
        LogiGetCurrentControllerProperties = (LPFNDLLLGETCONTRPROPERTIES)GetProcAddress(logiDllHandle,
"LogiGetCurrentControllerProperties");
        LogiGetState = (LPFNDLLLGETSTATE)GetProcAddress(logiDllHandle, "LogiGetState");
        LogiGetStateENGINES = (LPFNDLLLGETSTATEENGINES)GetProcAddress(logiDllHandle,
"LogiGetStateENGINES");
        LogiPlayLeds = (LPFNDLLPLAYLEDS)GetProcAddress(logiDllHandle, "LogiPlayLeds");
        LogiPlayConstantForce = (LPFNDLLPLAYFORCE)GetProcAddress(logiDllHandle, "LogiPlayConstantForce");
        LogiPlayDamperForce = (LPFNDLLPLAYFORCE)GetProcAddress(logiDllHandle, "LogiPlayDamperForce");
        LogiPlaySideCollisionForce = (LPFNDLLPLAYFORCE)GetProcAddress(logiDllHandle,
"LogiPlaySideCollisionForce");
        LogiPlayFrontalCollisionForce = (LPFNDLLPLAYFORCE)GetProcAddress(logiDllHandle,
"LogiPlayFrontalCollisionForce");
        LogiPlayDirtRoadEffect = (LPFNDLLPLAYFORCE)GetProcAddress(logiDllHandle, "LogiPlayDirtRoadEffect");
        LogiPlayBumpyRoadEffect = (LPFNDLLPLAYFORCE)GetProcAddress(logiDllHandle,
"LogiPlayBumpyRoadEffect");
        LogiPlaySlipperyRoadEffect = (LPFNDLLPLAYFORCE)GetProcAddress(logiDllHandle,
"LogiPlaySlipperyRoadEffect");
        LogiPlaySoftstopForce = (LPFNDLLPLAYFORCE)GetProcAddress(logiDllHandle, "LogiPlaySoftstopForce");
        LogiIsPlaying = (LPFNDLLISPLAYING)GetProcAddress(logiDllHandle, "LogiIsPlaying");
        LogiPlaySpringForce = (LPFNDLLPLAYSPRINGFORCE)GetProcAddress(logiDllHandle, "LogiPlaySpringForce");
        LogiPlaySurfaceEffect = (LPFNDLLPLAYSURFACEEFFECT)GetProcAddress(logiDllHandle,
"LogiPlaySurfaceEffect");
        LogiStopSpringForce = (LPFNDLLSTOPFORCE)GetProcAddress(logiDllHandle, "LogiStopSpringForce");
        LogiStopConstantForce = (LPFNDLLSTOPFORCE)GetProcAddress(logiDllHandle, "LogiStopConstantForce");
        LogiStopDamperForce = (LPFNDLLSTOPFORCE)GetProcAddress(logiDllHandle, "LogiStopDamperForce");
        LogiStopDirtRoadEffect = (LPFNDLLSTOPFORCE)GetProcAddress(logiDllHandle, "LogiStopDirtRoadEffect");
        LogiStopBumpyRoadEffect = (LPFNDLLSTOPFORCE)GetProcAddress(logiDllHandle,
"LogiStopBumpyRoadEffect");
        LogiStopSlipperyRoadEffect = (LPFNDLLSTOPFORCE)GetProcAddress(logiDllHandle,
"LogiStopSlipperyRoadEffect");
        LogiStopCarAirborne = (LPFNDLLSTOPFORCE)GetProcAddress(logiDllHandle, "LogiStopCarAirborne");
        LogiStopSoftstopForce = (LPFNDLLSTOPFORCE)GetProcAddress(logiDllHandle, "LogiStopSoftstopForce");
        LogiPlayCarAirborne = (LPFNDLLSTOPFORCE)GetProcAddress(logiDllHandle, "LogiPlayCarAirborne");
        LogiButtonTriggered = (LPFNDLLGETBUTTON)GetProcAddress(logiDllHandle, "LogiButtonTriggered");
        LogiButtonReleased = (LPFNDLLGETBUTTON)GetProcAddress(logiDllHandle, "LogiButtonReleased");
        LogiButtonIsPressed = (LPFNDLLGETBUTTON)GetProcAddress(logiDllHandle, "LogiButtonIsPressed");
        LogiGetShifterMode = (LPFNDLLGETSHIFTERMODE)GetProcAddress(logiDllHandle, "LogiGetShifterMode");
        LogiSetPreferredControllerProperties = (LPFNDLLSETCONTRPROPERTIES)GetProcAddress(logiDllHandle,
"LogiSetPreferredControllerProperties");



        LogiSteeringInitialize(TRUE);

        //this is your main loop
        while(!done){
                if(LogiUpdate()){
                        //the main window handler has been found, you can now call any function
                        //from the steering wheel sdk
                }
        }



}
```

## Reference

### LogiSteeringInitialize

The **LogiSteeringInitialize**() function makes sure the main window has come to foreground. Then, if there isn't already another instance running makes necessary initializations.

```
bool LogiSteeringInitialize(CONST bool ignoreXInputControllers)
```

#### *Parameters*

- ignoreXInputControllllers: if set to true, the SDKs will ignore any X-input controller

#### *Return value*

If the function succeeds, it returns true, otherwise false.
If it returns false, it is because the main window of your application has not come to foreground yet. This means that the window handler cannot be retrieved yet.

### LogiUpdate

The **LogiUpdate**() looks for the main window handler, if it has been found keeps forces and controller connections up to date. It has to be called every frame of your application.

```
bool LogiUpdate();
```

#### *Return value*

If the function succeeds, it returns true. Otherwise false.
The function will return false if **LogiSteeringInitialize** () hasn't been called or it has been unable to find the main window handler.

### LogiGetState

The **LogiGetState**() returns the state of the controller in the struct DIJOYSTATE2. Use this if working with DirectInput from Microsoft Windows, it requires dinput.h.

```
DIJOYSTATE2* LogiGetState(const int index);
```

#### *Parameters*

- index : index of the game controller.  Index 0 corresponds to the first game controller connected. Index 1 to the second game controller.

#### *Return value*

DIJOYSTATE2 structure containing the device's positional information for axes, POVs and buttons.

#### *Notes*

If not working with DirectInput, or can't include dinput.h in your game/project, please have a look at the following function : LogiGetStateENGINES

## LogiGetStateENGINES

The **LogiGetStateENGINES**() is a simplified version of the function LogiGetState. Use this if not working with DirectInput. It returns a simplified version of the DIJOYSTATE2 struct, called DIJOYSTATE2ENGINES.

```
DIJOYSTATE2ENGINES* LogiGetStateENGINES(const int index);
```

### *Parameters*

- index : index of the game controller. Index 0 corresponds to the first game controller connected. Index 1 to the second game controller.

### *Return value*

DIJOYSTATE2ENGINES structure containing the device's positional information for axes, POVs and buttons. For details, have a look at the comments in the header file LogiSeeringWheel.h.

## LogiGetFriendlyProductName

The **LogiGetFriendlyProductName** () function gets the device's friendly name

```
wchar_t* LogiGetFriendlyProductName(const int index);
```

### *Parameters*

- index : index of the game controller. Index 0 corresponds to the first game controller connected. Index 1 to the second game controller.

### *Return value*

String containing device friendly product name.

## LogiIsConnected

The **LogiIsConnected** () function checks if a game controller is connected at the specified index

```
bool LogiIsConnected(const int index);
```

### *Parameters*

- index : index of the game controller. Index 0 corresponds to the first game controller connected. Index 1 to the second game controller.

### *Return value*

True if a device is connected at the specified index, false otherwise.

## LogiIsDeviceConnected

The **LogiIsConnected** () function checks if the specified device is connected at that index

```
bool LogiIsDeviceConnected(const int index, const int deviceType);
```

### *Parameters*

- index : index of the game controller. Index 0 corresponds to the first game controller connected. Index 1 to the second game controller.
- deviceType : type of the device to check for. Possible types are :
  - `LOGI_DEVICE_TYPE_WHEEL`

  - `LOGI_DEVICE_TYPE_JOYSTICK`
  - `LOGI_DEVICE_TYPE_GAMEPAD`
  - `LOGI_DEVICE_TYPE_OTHER`

***Return value***

True if the specified device is connected at that index, false otherwise.

## LogiIsManufacturerConnected

The **LogiIsManufacturerConnected** () function checks if the device connected at index is made from the manufacturer specified by manufacturerName

```
bool LogiIsManufacturerConnected(const int index, const int manufacturerName);
```

***Parameters***

- index : index of the game controller. Index 0 corresponds to the first game controller connected. Index 1 to the second game controller.
- manufacturerName: name of the manufacturer the device has been made by. Possible types are:
  - `LOGI_ MANUFACTURER_LOGITECH`
  - `LOGI_ MANUFACTURER_MICROSOFT`
  - `LOGI_ MANUFACTURER_OTHER`

***Return value***

True if a PC controller of specified manufacturer is connected, false otherwise.

## LogiIsModelConnected

The **LogiIsModelConnected** () function checks if the device connected at index is the model specified

```
bool LogiIsModelConnected(const int index, const int modelName);
```

***Parameters***

- index : index of the game controller. Index 0 corresponds to the first game controller connected. Index 1 to the second game controller.
- modelName: name of the model of the device. Possible types are :
  - `LOGI_MODEL_G27`
  - `LOGI _MODEL_G25`
  - `LOGI _MODEL_MOMO_RACING`
  - `LOGI _MODEL_MOMO_FORCE`
  - `LOGI _MODEL_DRIVING_FORCE_PRO`
  - `LOGI _MODEL_DRIVING_FORCE`
  - `LOGI _MODEL_NASCAR_RACING_WHEEL`
  - `LOGI _MODEL_FORMULA_FORCE`
  - `LOGI _MODEL_FORMULA_FORCE_GP`
  - `LOGI _MODEL_FORCE_3D_PRO`
  - `LOGI _MODEL_EXTREME_3D_PRO`
  - `LOGI _MODEL_FREEDOM_24`
  - `LOGI _MODEL_ATTACK_3`
  - `LOGI _MODEL_FORCE_3D`
  - `LOGI _MODEL_STRIKE_FORCE_3D`
  - `LOGI _MODEL_RUMBLEPAD`
  - `LOGI _MODEL_RUMBLEPAD_2`
  - `LOGI _MODEL_CORDLESS_RUMBLEPAD_2`

- `LOGI _MODEL_CORDLESS_GAMEPAD`
- `LOGI _MODEL_DUAL_ACTION_GAMEPAD`
- `LOGI _MODEL_PRECISION_GAMEPAD_2`
- `LOGI _MODEL_CHILLSTREAM`

#### *Return Value*
True if a controller of the specified model is connected, false otherwise.

## LogiButtonTriggered
The **LogiButtonTriggered** () function checks if the device connected at index is currently triggering the button specified

```
bool LogiButtonTriggered(const int index, const int buttonNbr);
```

#### *Parameters*
- index : index of the game controller.  Index 0 corresponds to the first game controller connected. Index 1 to the second game controller.
- buttonNbr : the number of the button that we want to  check. Possible numbers are: 0 to 127

#### *Return value*

True if the button was triggered, false otherwise.

## LogiButtonReleased
The **LogiButtonReleased** () function checks if on the device connected at index has been released the button specified

```
bool LogiButtonReleased(const int index, const int buttonNbr);
```

#### *Parameters*
- index : index of the game controller.  Index 0 corresponds to the first game controller connected. Index 1 to the second game controller.
- buttonNbr : the number of the button that we want to  check. Possible numbers are: 0 to 127

#### *Return value*

True if the button was triggered, false otherwise.

## LogiButtonIsPressed
The **LogiButtonIsPressed** () function checks if on the device connected at index is currently being pressed the button specified

```
bool LogiButtonIsPressed(const int index, const int buttonNbr);
```

#### *Parameters*
- index : index of the game controller.  Index 0 corresponds to the first game controller connected. Index 1 to the second game controller.
- buttonNbr : the number of the button that we want to  check. Possible numbers are: 0 to 127

### *Return value*

True if the button was triggered, false otherwise.

## LogiGenerateNonLinearValues

The **LogiGenerateNonLinearValues** () function generate non-linear values for the game controller's axis. Gaming wheels/joysticks/game pads have very different behavior from real steering wheels. The reason for single-turn wheels is that they only do up to three quarters of a turn lock to lock, compared to about 3 turns for a real car.
This directly affects the steering ratio (15:1 to 20:1 for a real car, but only 4:1 for a gaming wheel!). Joysticks and game pads have a much shorter range of movement than a real steering wheel as well. Because of this very short steering ratio or short range, the gaming wheel/joystick/game pad will feel highly sensitive which may make game play very difficult. Especially it may be difficult to drive in a straight line at speed (tendency to swerve back and forth). One way to get around this problem is to use a sensitivity curve. This is a curve that defines the sensitivity of the game controller depending on speed. This type of curve is usually used for game pads to make up for their low physical range. The result of applying such a curve is that at high speed the car's wheels will physically turn less than if the car is moving very slowly. For example the car's wheels may turn 60 degrees lock to lock at low speed but only 10 degrees lock to lock at higher speeds.  If you calculate the resulting steering ratio for 10 degrees lock to lock you find that if you use a steering wheel that turns 180 degrees lock to lock the ratio is equal to 180/10 = 18, which corresponds to a real car's steering ratio. If the sensitivity curve has been implemented for the wheel/joystick, adding a non-linear curve probably is not necessary. But you may find that even after applying a sensitivity curve, the car still feels a little twitchy on a straight line when driving fast. This may be because in your game you need more than 10 degrees lock to lock even at high speeds. Or maybe the car is moving at very high speeds where even a normal steering ratio is not good enough to eliminate high sensitivity. The best way at this point is to add a non-linear curve on top of the sensitivity curve. The effect of the non-linear curve with positive nonLinCoeff is that around center position the wheel/joystick will be less sensitive.  Yet at locked position left or right the car's wheels will turn the same amount of degrees as without the non-linear response curve.  Therefore the car will become more controllable on a straight line and game-play will be improved.
There can sometimes be cases where the wheel does not feel sensitive enough. In that case it is possible to add a non-linear curve with the inverse effect (makes the steering more sensitive around center position) by using negative values for nonLinCoeff. This method lets you define a non-linearity coefficient which will determine how strongly non-linear the curve will be. When running the method it will generate a mapping table in the form of an array. For each of the 1024 entries in this array there will be a corresponding non-linear value which can be used as the wheel/joystick's axis position instead of the original value.

```
bool LogiGenerateNonLinearValues(const int index, const int nonLinCoeff);
```

### *Parameters*

- index : index of the game controller.  Index 0 corresponds to the first game controller connected. Index 1 to the second game controller.
- nonLinCoeff: value representing how much non-linearity should be applied. Range is -100 to 100. 0 = linear curve, 100 = maximum non-linear curve with less sensitivity around center, -100 = maximum non-linearity with more sensitivity around center position.

### *Return value*

True if successful, false otherwise.

## LogiGetNonLinearValue

The **LogiGetNonLinearValue** () function returns a non-linear value from a table previously generated. This can be used for the response of a steering wheel.

```
int LogiGetNonLinearValue(const int index, const int inputValue);
```

### *Parameters*

- index : index of the game controller. Index 0 corresponds to the first game controller connected. Index 1 to the second game controller.
- inputValue: value between -32768 and 32767 corresponding to original value of an axis.

### *Return value*

Value between -32768 and 32767. corresponding to the level of non-linearity previously set with GenerateNonLinearValues().

## LogiHasForceFeedback

The **LogiHasForceFeedback** () function checks if the controller at index has force feedback

```
bool LogiHasForceFeedback(const int index);
```

### *Parameters*

- index : index of the game controller. Index 0 corresponds to the first game controller connected. Index 1 to the second game controller.

### *Return value*

True if the LogiIsPlaying device can do force feedback, false otherwise.

## LogiIsPlaying

The **LogiIsPlaying** () function heck if a certain force effect is currently playing.

```
bool LogiIsPlaying(const int index, const int forceType);
```

***Parameters***

- index : index of the game controller. Index 0 corresponds to the first game controller connected. Index 1 to the second game controller.
- forceType : the type of the force that we want to check to see if it is playing. Possible types are:
    - - LOGI_FORCE_SPRING
    - - LOGI_FORCE_CONSTANT
    - - LOGI_FORCE_DAMPER
    - - LOGI_FORCE_SIDE_COLLISION
    - - LOGI_FORCE_FRONTAL_COLLISION
    - - LOGI_FORCE_DIRT_ROAD
    - - LOGI_FORCE_BUMPY_ROAD
    - - LOGI_FORCE_SLIPPERY_ROAD
    - - LOGI_FORCE_SURFACE_EFFECT
    - - LOGI_FORCE_CAR_AIRBORNE

***Return value***

True if the force is playing, false otherwise.

## LogiPlaySpringForce

The **LogiPlaySpringForce** () function plays the spring force.

```
bool LogiPlaySpringForce(const int index, const int offsetPercentage, const int
saturationPercentage, const int coefficientPercentage);
```

***Parameters***

- index : index of the game controller. Index 0 corresponds to the first game controller connected. Index 1 to the second game controller.
- offsetPercentage: Specifies the center of the spring force effect. Valid range is -100 to 100. Specifying 0 centers the spring. Any values outside this range are silently clamped.
- saturationPercentage: Specify the level of saturation of the spring force effect. The saturation stays constant after a certain deflection from the center of the spring. It is comparable to a magnitude. Valid ranges are 0 to 100. Any value higher than 100 is silently clamped.
- coefficientPercentage - Specify the slope of the effect strength increase relative to the amount of deflection from the center of the condition. Higher values mean that the saturation level is reached sooner. Valid ranges are -100 to 100. Any value outside the valid range is silently clamped.

***Return value***

True if success, false otherwise.

***Notes***

The dynamic spring force gets played on the X axis. If a joystick is connected, all forces generated by the Steering Wheel SDK will be played on the X axis. And in addition there will be a constant spring on the Y axis.

## LogiStopSpringForce

The **LogiStopSpringForce** () stops the spring force.

```
bool LogiStopSpringForce(const int index);
```

#### *Parameters*

- index : index of the game controller.  Index 0 corresponds to the first game controller connected. Index 1 to the second game controller.

#### *Return value*

True if success, false otherwise.

## LogiPlayConstantForce

The **LogiPlayConstantForce** () function plays the constant force.

```
bool LogiPlayConstantForce(const int index, const int magnitudePercentage);
```

#### *Parameters*

- index : index of the game controller.  Index 0 corresponds to the first game controller connected. Index 1 to the second game controller.
- magnitudePercentage : Specifies the magnitude of the constant force effect. A negative value reverses the direction of the force. Valid ranges for magnitudePercentage are -100 to 100. Any values outside the valid range are silently clamped.

#### *Return value*

True if success, false otherwise.

#### *Notes*

A constant force works best when continuously updated with a value tied to the physics engine. Tie the steering wheel/joystick to the car's physics engine via a vector force. This will create a centering spring effect, a sliding effect, a feeling for the car's inertia, and depending on the physics engine it should also give side collisions (wheel/joystick jerks in the opposite way of the wall the car just touched). The vector force could for example be calculated from the lateral force measured at the front tires. This vector force should be 0 when at a stop or driving straight. When driving through a turn or when driving on a banked surface the vector force should have a magnitude that grows in a proportional way.

## LogiStopConstantForce

The **LogiStopConstantForce** () stops the constant force.

```
bool LogiStopConstantForce(const int index);
```

#### *Parameters*

- index : index of the game controller.  Index 0 corresponds to the first game controller connected. Index 1 to the second game controller.

### *Return value*

True if success, false otherwise.

## LogiPlayDamperForce

The **LogiPlayDamperForce** () function plays the constant force.

```
bool LogiPlayDamperForce(const int index, const int coefficientPercentage);
```

### *Parameters*

- index : index of the game controller. Index 0 corresponds to the first game controller connected. Index 1 to the second game controller.
- coefficientPercentage : specify the slope of the effect strength increase relative to the amount of deflection from the center of the condition. Higher values mean that the saturation level is reached sooner. Valid ranges are -100 to 100. Any value outside the valid range is silently clamped. -100 simulates a very slippery effect, +100 makes the wheel/joystick very hard to move, simulating the car at a stop or in mud.

### *Return value*

True if success, false otherwise.

### *Notes*

Simulate surfaces that are hard to turn on (mud, car at a stop) or slippery surfaces (snow, ice).

## LogiStopDamperForce

The **LogiStopDamperForce** () stops the damper force.

```
bool LogiStopDamperForce(const int index);
```

### *Parameters*

- index : index of the game controller. Index 0 corresponds to the first game controller connected. Index 1 to the second game controller.

### *Return value*

True if success, false otherwise.

## LogiPlaySideCollisionForce

The **LogiPlaySideCollisionForce** () function plays the side collision force on the controller at index

```
bool LogiPlaySideCollisionForce(const int index, const int magnitudePercentage);
```

### Parameters

- index : index of the game controller. Index 0 corresponds to the first game controller connected. Index 1 to the second game controller.
- magnitudePercentage : Specifies the magnitude of the side collision force effect. A negative value reverses the direction of the force. Valid ranges for magnitudePercentage are -100 to 100. Any values outside the valid range are silently clamped.

### Return value

True if success, false otherwise.

### Notes

If you are already using a constant force tied to a vector force from the physics engine, then you may not need to add side collisions since depending on your physics engine the side collisions may automatically be taken care of by the constant force.

## LogiPlayFrontalCollisionForce

The **LogiPlayFrontalCollisionForce** () function plays the frontal collision force on the controller at index

```
bool LogiPlayFrontalCollisionForce(const int index, const int magnitudePercentage);
```

### Parameters

- index : index of the game controller. Index 0 corresponds to the first game controller connected. Index 1 to the second game controller.
- magnitudePercentage : specifies the magnitude of the frontal collision force effect. Valid ranges for magnitudePercentage are 0 to 100. Values higher than 100 are silently clamped.

### Return value

True if success, false otherwise.

## LogiPlayDirtRoadEffect

The **LogiPlayDirtRoadEffect** () function plays the dirt road effect.

```
bool LogiPlayDirtRoadEffect(const int index, const int magnitudePercentage);
```

### Parameters

- index : index of the game controller. Index 0 corresponds to the first game controller connected. Index 1 to the second game controller.
- magnitudePercentage : Specifies the magnitude of the dirt road effect. Valid ranges for magnitudePercentage are 0 to 100. Values higher than 100 are silently clamped.

#### *Return value*

True if success, false otherwise.

## LogiStopDirtRoadEffect

The **LogiStopDirtRoadEffect** () stops the dirt road effect.

```
bool LogiStopDirtRoadEffect(const int index);
```

#### *Parameters*

- index : index of the game controller. Index 0 corresponds to the first game controller connected. Index 1 to the second game controller.

#### *Return value*

True if success, false otherwise.

## LogiPlayBumpyRoadEffect

The **LogiPlayBumpyRoadEffect** () function plays the bumpy road effect.

```
bool LogiPlayBumpyRoadEffect(const int index, const int magnitudePercentage);
```

#### *Parameters*

- index : index of the game controller. Index 0 corresponds to the first game controller connected. Index 1 to the second game controller.
- magnitudePercentage : Specifies the magnitude of the bumpy road effect. Valid ranges for magnitudePercentage are 0 to 100. Values higher than 100 are silently clamped.

#### *Return value*

True if success, false otherwise.

## LogiStopBumpyRoadEffect

The **LogiStopBumpyRoadEffect** () stops the bumpy road effect.

```
bool LogiStopBumpyRoadEffect (const int index);
```

#### *Parameters*

- index : index of the game controller. Index 0 corresponds to the first game controller connected. Index 1 to the second game controller.

#### *Return value*

True if success, false otherwise.

## LogiPlaySlipperyRoadEffect

The **LogiPlaySlipperyRoadEffect** () function plays the slippery road effect.

```
bool LogiPlaySlipperyRoadEffect(const int index, const int magnitudePercentage);
```

#### *Parameters*

- index : index of the game controller. Index 0 corresponds to the first game controller connected. Index 1 to the second game controller.
- magnitudePercentage : Specifies the magnitude of the slippery road effect. Valid ranges for magnitudePercentage are 0 to 100. 100 corresponds to the most slippery effect.

#### *Return value*

True if success, false otherwise.

## LogiStopSlipperyRoadEffect

The **LogiStopSlipperyRoadEffect** () stops the slippery road effect.

```
bool LogiStopSlipperyRoadEffect (const int index);
```

#### *Parameters*

- index : index of the game controller. Index 0 corresponds to the first game controller connected. Index 1 to the second game controller.

#### *Return value*

True if success, false otherwise.

## LogiPlaySurfaceEffect

The **LogiPlaySurfaceEffect** () function plays the surface effect.

```
bool LogiPlaySurfaceEffect(const int index, const int type, const int magnitudePercentage,
const int period);
```

#### *Parameters*

- index : index of the game controller. Index 0 corresponds to the first game controller connected. Index 1 to the second game controller.
- type : Specifies the type of force effect. Can be one of the following values:
    - `LOGI_PERIODICTYPE_SINE`
    - `LOGI_PERIODICTYPE_SQUARE`
    - `LOGI_PERIODICTYPE_TRIANGLE`
- magnitudePercentage - Specifies the magnitude of the surface effect. Valid ranges for magnitudePercentage are 0 to 100. Values higher than 100 are silently clamped.

- period - Specifies the period of the periodic force effect. The value is the duration for one full cycle of the periodic function measured in milliseconds. A good range of values for the period is 20 ms (sand) to 120 ms (wooden bridge or cobblestones). For a surface effect the period should not be any bigger than 150 ms.

#### *Return value*

True if success, false otherwise.

## LogiStopSurfaceEffectEffect

The **LogiStopSurfaceEffectEffect** () stops the surface effect.

```
bool LogiStopSurfaceEffect (const int index);
```

#### *Parameters*

- index : index of the game controller.  Index 0 corresponds to the first game controller connected. Index 1 to the second game controller.

#### *Return value*

True if success, false otherwise.

## LogiPlayCarAirborne

The **LogiPlayCarAirborne** () function plays the car airborne effect.

```
bool LogiPlayCarAirborne(const int index);
```

#### *Parameters*

- index : index of the game controller.  Index 0 corresponds to the first game controller connected. Index 1 to the second game controller.

#### *Return value*

True if success, false otherwise.

## LogiStopCarAirborne

The **LogiStopCarAirborne** () stops the car ariborne road effect.

```
bool LogiStopCarAirborne(const int index);
```

#### *Parameters*

- index : index of the game controller.  Index 0 corresponds to the first game controller connected. Index 1 to the second game controller.

### *Return value*

True if success, false otherwise.

## LogiPlaySoftstopForce

The **LogiPlaySoftstopForce** () function plays the soft stop force.

```
bool LogiPlaySoftstopForce(const int index, const int usableRangePercentage);
```

### *Parameters*

- index : index of the game controller. Index 0 corresponds to the first game controller connected. Index 1 to the second game controller.
- usableRangePercentage : specifies the deadband in percentage of the softstop force effect.

### *Return value*

True if success, false otherwise.

## LogiStopSoftstopForce

The **LogiStopSoftstopForce** () stops the soft stop force.

```
bool LogiStopSoftstopForce(const int index);
```

### *Parameters*

- index : index of the game controller. Index 0 corresponds to the first game controller connected. Index 1 to the second game controller.

### *Return value*

True if success, false otherwise.

## LogiSetPreferredControllerProperties

The **LogiSetPreferredControllerProperties** () sets preferred wheel properties.

```
bool LogiSetPreferredControllerProperties(const LogiControllerPropertiesData properties);
```

### *Parameters*

- properties : structure containing all the fields to be set.

### *Return value*

True if success, false otherwise.

## LogiGetCurrentControllerProperties

The **LogiGetCurrentControllerProperties** () fills the properties parameter with the current controller properties

```
bool LogiGetCurrentControllerProperties(const int index, LogiControllerPropertiesData&
properties);
```

### *Parameters*

- index : index of the game controller
- properties : structure to receive current properties

### *Return value*

True if success, false otherwise.

## LogiGetShifterMode

The **LogiGetShifterMode** () gets current shifter mode (gated or sequential)

```
int LogiGetShifterMode(const int index);
```

### *Parameters*

- index : index of the game controller

### *Return value*

1 if shifter is gated, 0 if shifter is sequential, -1 if unknown

## LogiPlayLeds

The **LogiPlayLeds** () plays the leds on the controller

```
bool LogiPlayLeds(const int index, const float currentRPM, const float rpmFirstLedTurnsOn,
const float rpmRedLine);
```

### *Parameters*

- index : index of the game controller
- currentRPM : current RPM.

- rpmFirstLedTurnsOn : RPM when first LEDs are to turn on.
- rpmRedLine : just below this RPM, all LEDs will be on. Just above,  all LEDs will start flashing.

### *Return value*

True if success, false otherwise.