# Sixense SDK Reference Guide

Sixense Control System Runtime Library

© 2012 Sixense Entertainment, Inc.
All Rights Reserved
Confidential

# Table of Contents

# Datatypes

# sixenseControllerData

Controller data retrieval structure

## Definition

```
typedef struct _sixenseControllerData {
  float pos[3];
  float rot_mat[3][3];
  float joystick_x;
  float joystick_y;
  float trigger;
  unsigned int buttons;
  unsigned char sequence_number;
  float rot_quat[4];
  unsigned short firmware_revision;
  unsigned short hardware_revision;
  unsigned short packet_type;
  unsigned short magnetic_frequency;
  int enabled;
  int controller_index;
  unsigned char is_docked;
  unsigned char which_hand;
  unsigned char hemi_tracking_enabled;
} sixenseControllerData;
```

## Members

| | |
|---|---|
| pos | The X, Y and Z position of the controller. |
| rot_mat | A 3x3 matrix describing the rotation of the controller. |
| joystick_x | The horizontal position of the joystick. -1.0 is full left, 0 is centered, 1.0 is full right. |
| joystick_y | The vertical position of the joystick. -1.0 is full down, 0 is centered, 1.0 is full up. |
| trigger | The status of the analog trigger. 0 is unpressed, 1.0 is fully pressed. |
| buttons | A bit vector describing the state of the controller buttons. See below for the bit field descriptions. This value can be OR'ed with one of the button macros to check the state of a given button. See the Notes section for a list of these macros. |
| sequence_number | Each subsequent datapoint is stamped with a sequence number designating its order in the data stream. Since the update rate is fixed at 60Hz, each increment in sequence number counts as 16.6ms of system time. |
| rot_quat | The current rotation angles for the controller in quaternion form. |
| firmware_revision | The current firmware revison. |
| hardware_revision | The current revision. |
| packet_type | The type of data packet, currently always 1. |
| magnetic_frequency | Unused |
| enabled | If the controller is connected this value will be 1. Equivalent |

| | |
|---|---|
| | to calling sixenseIsControllerEnabled(). |
| controller_index | The hardware index of the controller referred to by this packet. Note that this is independent of which controller is in the players lef t or right hands. To determine which hardware index is in which hand, the game must ask the user. An example of this is sixenseUtils::controller_manager. |
| is_docked | This will be 1 when the controller is sitting in the dock and 0 otherwise. Games can reference this value for determining when to pause or disable Sixense functionality. |
| which_hand | When the controllers are placed in the dock (or when the controller manager is run) this field will be set to a non-zero value. If the controller is placed in the left side of the dock it will be set to 1, and if it is placed on the right it will be set to 2. When sixenseInit is called this field is initialized to 0, and will remain so until the controllers are docked. See the Overview document for more details. |
| hemi_tracking_enabled | This will be 1 when both controllers have been docked or when the sixenseUtils::controller_manager has successfully completed. |

## Description

This structure is used by the `sixenseGetNewestData()`, `sixenseGetData()`, `sixenseGetAllNewestData()` and `sixenseGetAllData()` call to retrieve the current position data.

## Notes

The following table lists the definitions that can be used to access specific button states.

| |
|---|
| SIXENSE_BUTTON_1 |
| SIXENSE_BUTTON_2 |
| SIXENSE_BUTTON_3 |
| SIXENSE_BUTTON_4 |
| SIXENSE_BUTTON_START |
| SIXENSE_BUTTON_BUMPER |
| SIXENSE_BUTTON_JOYSTICK |

## See Also

`sixenseGetNewestData()`

`sixenseGetData()`

`sixenseGetAllNewestData()`

`sixenseGetAllData()`

Button macro definitions

# sixenseAllControllerData

A convenience structure for querying all controllers at once.

## Definition

```
typedef struct _sixenseAllControllerData {
  sixenseControllerData controllers[4];
} sixenseAllControllerData;
```

## Members

| | |
|---|---|
| `controllers` | An array of 4 `sixenseControllerData` structures. |

## Description

This structure is used by the `sixenseGetAllData()` call to retrieve the current position data for up to 4 controllers in a single call. This is more efficient than calling `sixenseGetData()` multiple times per frame.

## See Also

`sixenseGetAllData()`,

`sixenseGetAllNewestData()`,

button macro definitions.

# Functions

# sixenseInit

Initialize the Sixense library.

## Definition

```
int sixenseInit(void);
```

## Return Values

SIXENSE_SUCCESS is returned if the library is successfully initialized; otherwise, the return value is SIXENSE_FAILURE.

## Description

This function initializes the Sixense library. It must be called at least one time per application. Subsequent calls will have no effect. Once initialized, the other Sixense function calls will work as described until `sixenseExit()` is called.

## See Also

`sixenseExit()`

# sixenseExit

Shut down the Sixense library.

## Definition

```
int sixenseExit(void);
```

## Return Values

SIXENSE_SUCCESS is returned if the library was successfully shut down; otherwise, the return value is SIXENSE_FAILURE.

## Description

This shuts down the Sixense library. After this function call, all Sixense API calls will return failure until `sixenseInit()` is called again.

## See Also

`sixenseInit ()`

# sixenseGetMaxBases

Returns the maximum number of base units that can be connected to the computer at once. Note that the bases have to have different magnetic frequencies in order to not interfere with each other, and current retail products like the Razer Hydra only support one frequency.

## Definition

```
int sixenseGetMaxBases(void);
```

## Return Values

This call returns the maximum number of base units supported by the Sixense control system. Currently, this number is 4 for all platforms.

## Description

At the current time the Sixense driver supports a maximum of 4 simultaneous base units. Since this number may change in the future, `sixenseGetMaxBases()` should be called when iterating through all bases to ensure compatibility. Note that the bases have to have different magnetic frequencies in order to not interfere with each other, and current retail products like the Razer Hydra only support one frequency.

# sixenseSetActiveBase

Designates which base subsequent API calls are to be directed to.

## Definition

```
int sixenseSetActiveBase( int base_num );
```

## Arguments

| | |
|---|---|
| `base_num` | An integer from 0 to `sixenseGetMaxBases()-1` |

## Return Values

SIXENSE_SUCCESS is returned if the the designated base is active and valid; otherwise, the return value is SIXENSE_FAILURE

## Description

It is possible for the Sixense API to address multiple bases connected to the same computer. This call can be used to designate which base all subsequent API calls are directed towards. sixenseInit and sixenseExit are not affected by this call. Note that the bases have to have different magnetic frequencies in order to not interfere with each other, and current retail products like the Razer Hydra only support one frequency.

# sixenseIsBaseConnected

Used to determine if a base is currently connected to the system.

## Definition

```
int sixenseIsBaseConnected( int base_num );
```

## Arguments

| | |
|---|---|
| `base_num` | An integer from 0 to `sixenseGetMaxBases()-1` |

## Return Values

This call returns 1 if the base is currently plugged in and 0 otherwise.

## Description

This call returns whether or not the designated base unit is attached to the system. For all current consumer applications only one base is supported so the argument should always be 0. Calling sixenseIsBaseConnected(0) can be used by the game to enable or disable Sixense support.

# sixenseGetMaxControllers

Returns the maximum number of controllers supported by the Sixense control system.

## Definition

```
int sixenseGetMaxControllers(void);
```

## Return Values

This call returns the maximum number of controllers supported by the Sixense control system. Currently, this number is 4 for all platforms.

## Description

At the current time the Sixense control system supports a maximum of 4 simultaneous controllers. Since this number may change in the future, `sixenseGetMaxControllers()` should be called when iterating through all controllers to ensure compatibility.

# sixenseGetNumActiveControllers

Reports the number of active controllers.

## Definition

```
int sixenseGetNumActiveControllers(void);
```

## Return Values

This call returns the number of controllers currently linked to the base station.

## Description

At the current time the Sixense API supports a maximum of 4 simultaneous controllers. This call can be used as a quick check to see whether enough controllers are available to play the game. To find which controllers are actually linked, iterate from 0 to `sixenseGetMaxControllers` and call `sixenseIsControllerEnabled` on each index.

## See Also

`sixenseGetMaxControllers, sixenseIsControllerEnabled`

# sixenseIsControllerEnabled

Returns true if the referenced controller is currently connected to the Base Unit.

## Definition

```
int sixenseIsControllerEnabled( int which );
```

## Return Values

This call returns the connection status of the referenced controller. 1 means the controller is enabled, 0 means it is disabled.

## Description

This call is used to determine whether or not a given controller is powered on and connected to the system. The argument is an index between 0 and the maximum number of supported controllers.

## See Also

`sixenseGetMaxControllers()`

# sixenseGetAllNewestData

Get the most recent state of all of the controllers.

## Definition

```
int sixenseGetAllNewestData( sixenseAllControllerData *all_data );
```

## Arguments

| | |
|---|---|
| `all_data` | A pointer to user-allocated memory for returning the controller information. |

## Return Values

SIXENSE_SUCCESS is returned as long as the Sixense system has been initialized, otherwise SIXENSE_FAILURE.

## Description

This function returns the most recent state of all of the Sixense controllers. `sixenseIsControllerEnabled()` should be used to determine whether a given controller's data is valid. This call is currently more efficent than calling `sixenseGetData()` multiple times per frame.

# sixenseGetAllData

Get state of all of the controllers, selecting how far back into a history of the last 10 updates.

## Definition

```
int sixenseGetAllData( int index_back,
                       sixenseAllControllerData *all_data );
```

## Arguments

| | |
|---|---|
| `index_back` | How far back in the history buffer to retrieve data. 0 returns the most recent data, 9 returns the oldest data. Any of the last 10 positions may be queried. |
| `all_data` | A pointer to user-allocated memory for returning the desired controller information. |

## Return Values

SIXENSE_SUCCESS is returned as long as the Sixense system has been initialized, otherwise SIXENSE_FAILURE.

## Description

This function returns the most recent state of all of the Sixense controllers, looking back in history if desired. When used in conjunction with the sequence numbers in the data packets, this function is useful for referencing packets that may have been skipped due to the game's frame rate relative to the 60Hz update rate of the controller.

# sixenseGetNewestData

Get the most recent state of one of the controllers.

## Definition

```
int sixenseGetNewestData( int which, sixenseControllerData *data );
```

## Arguments

| | |
|---|---|
| `which` | The ID of the desired controller. Valid values are from 0 to 3. If the desired controller is not connected, an empty data packet is returned. Empty data packets are initialized to a zero position and the identity rotation matrix. |
| `data` | A pointer to user-allocated memory for returning the desired controller information. |

## Return Values

SIXENSE_SUCCESS is returned if the data was successfully retrieved; otherwise, the return value is SIXENSE_FAILURE. SIXENSE_FAILURE is also returned if the desired controller is not currently connected.

## Description

This function returns the most recent state of one of the connected Sixense controllers.

# sixenseGetData

Get state of one of the controllers, selecting how far back into a history of the last 10 updates.

## Definition

```
int sixenseGetData( int which, int index_back,
                    sixenseControllerData *data );
```

## Arguments

| | |
|---|---|
| `which` | The ID of the desired controller. Valid values are from 0 to 3. If the desired controller is not connected, an empty data packet is returned. Empty data packets are initialized to a zero position and the identity rotation matrix. |
| `index_back` | How far back in the history buffer to retrieve data. 0 returns the most recent data, 9 returns the oldest data. Any of the last 10 positions may be queried. |
| `data` | A pointer to user-allocated memory for returning the desired controller information. |

## Return Values

SIXENSE_SUCCESS is returned if the data was successfully retrieved; otherwise, the return value is SIXENSE_FAILURE. SIXENSE_FAILURE is also returned if the desired controller is not currently connected.

## Description

This function returns the current state of one of the connected Sixense controllers, looking back in history if desired. When used in conjunction with the sequence numbers in the data packets, this function is useful for referencing packets that may have been skipped due to the games frame rate relative to the 60Hz update rate of the controller.

# sixenseGetHistorySize

Get the size of the history buffer.

## Definition

```
int sixenseGetHistorySize();
```

## Return Values

The size of the history buffer is returned. For 3.2 hardware this value is always 10.

## Description

The data access calls `sixenseGetData` and `sixenseGetAllData` take an argument `index_back` that allows for the retrieval of older data packets. This can be used when the application is checking data less frequently than the data is arriving from the USB port. If this is the case, the missed packets can be fetched by using non 0 values of `index_back`. The `sequence_number` element of the `sixenseControllerData` structure can be used to see whether a given data packet has been seen before.

## See Also

`sixenseGetData`, `sixenseGetAllData`, `sixenseControllerData` structure definition

# sixenseSetFilterEnabled

Turn the internal position and orientation filtering on or off.

## Definition

```
int sixenseSetFilterEnabled( int on_or_off );
```

## Arguments

| | |
|---|---|
| on_or_off | The desired state of the filtering. 0 is off, 1 is on. |

## Return Values

SIXENSE_SUCCESS is returned as long as the Sixense system has been initialized, otherwise SIXENSE_FAILURE.

## Description

This call can be used to enable or disable filtering of the controller data. The filter parameters are not affected by this call.

# sixenseGetFilterEnabled

Returns the enable status of the internal position and orientation filtering.

## Definition

```
int sixenseGetFilterEnabled( int *on_or_off );
```

## Arguments

| | |
|---|---|
| `on_or_off` | Pointer to variable in which to store the result. |

## Return Values

SIXENSE_SUCCESS is returned as long as the Sixense system has been initialized, otherwise SIXENSE_FAILURE.

## Description

This call is used to determine whether or not the controllers are filtering their positions and orientations.

## See Also

`sixenseSetFilterEnabled()`

# sixenseSetFilterParams

Set the parameters that control the position and orientation filtering level.

## Definition

```
int sixenseSetFilterParams( float near_range, float near_val,
                            float far_range, float far_val );
```

## Arguments

| | |
|---|---|
| near_range | The range from the Sixense Base Unit at which to start increasing the filtering level from the near_val to far_val. Between near_range and far_range, the near_val and far_val are linearly interpolated. |
| near_val | The minimum filtering value. This value is used for when the controller is between 0 and near_val millimeters from the Sixense Base Unit. Valid values are between 0 and 1. |
| far_range | The range from the Sixense Base Unit after which to stop interpolating the filter value from the near_val, and after which to simply use far_val. |
| far_val | The maximum filtering value. This value is used for when the controller is between far_val and infinity. Valid values are between 0 and 1. |

## Return Values

SIXENSE_SUCCESS is returned as long as the Sixense system has been initialized, otherwise SIXENSE_FAILURE.

## Description

The Sixense controllers have built-in filtering capabilities. The filter is an Exponentially Weighted Moving Average Filter, where each sample is averaged with previous samples via the formula p(n) = f * p(n-1) + (1 – f) * p(n), where p(n) is the nth position in the series, and f is the filter value. The filter value used is linearly interpolated from near_val to far_val over the range [near_range,far_range]. near_val is used for [0, near_range), and far_val is used for (far_range,infinity].

# sixenseGetFilterParams

Returns the current filtering parameter values.

## Definition

```
int sixenseGetFilterParams( float *near_range, float *near_val, float
*far_range, float *far_val );
```

## Arguments

| | |
|---|---|
| near_range | Pointer to variable in which to store the result. |
| near_val | Pointer to variable in which to store the result. |
| far_range | Pointer to variable in which to store the result. |
| far_val | Pointer to variable in which to store the result. |

## Return Values

SIXENSE_SUCCESS is returned as long as the Sixense system has been initialized, otherwise SIXENSE_FAILURE.

## Description

Returns the current values used by the filtering algorithm.

## See Also

sixenseSetFilterParams()

# sixenseTriggerVibration

Enable a period of tactile feedback using the vibration motor. Note the Razer Hydra does not support vibration.

## Definition

```
int sixenseTriggerVibration( int controller_id, int duration_100ms,
                             int pattern_id );
```

## Arguments

| | |
|---|---|
| `controller_id` | The id of the controller to vibrate. Valid values are 0 through `sixenseGetMaxControllers`. |
| `duration_100ms` | The duration of the vibration, in 100 millisecond units. For example, a value of '5' will vibrate for half a second. |
| `pattern_id` | Future SDK's will support different pulsing patterns for the vibration. Currently, this argument is ignored and the vibration motor is engaged for the full duration. |

## Return Values

SIXENSE_SUCCESS is returned as long as the Sixense system has been initialized, otherwise SIXENSE_FAILURE.

## Description

This function triggers the vibration function. Each call triggers a single period of vibration. The duration of the variation is programmable via the `duration_100ms` argument.

# sixenseAutoEnableHemisphereTracking

Enable Hemisphere Tracking when the controller is aiming at the base. This call is deprecieated, as hemisphere tracking is automatically enabled when the controllers are in the dock or by the sixenseUtils::controller_manager. See the Sixense API Overivew for more information.

## Definition

```
int sixenseAutoEnableHemisphereTracking( int which_controller );
```

## Arguments

| | |
|---|---|
| `which_controller` | The 0 based index of the desired controller. |

## Return Values

SIXENSE_SUCCESS is returned as long as the Sixense system has been initialized, otherwise SIXENSE_FAILURE.

## Description

This call does not need to be called directly. When the controllers are placed in their docks, hemisphere tracking is automatically enabled. Also, the sixenseUtils::controller_manager automatically enables hemisphere tracking when the left and right controllers are designated.

The Sixense SDK consistently track positions when the controller is held above the Base Unit. When passing below the Base Unit, an inconsistency in the positions and rotations will be seen. This inconsistency will be that the X and Y values will flip sign, the Z value will begin increasing instead of decreasing, and the rotation matrix will flip by 180 degrees. When hemisphere tracking is enabled, the controller works through the entire tracking space.

To use this call to enable hemisphere tracking, prompt the user to point the controller at the base unit and then make this function call. As long as the controller is pointing more towards the base than away, hemisphere tracking will be enabled and the controller will work within the full tracking sphere.

# sixenseSetHighPriorityBindingEnabled

This function enables and disables High Priority RF binding mode. This call is only used with the wireless Sixense devkits.

## Definition

```
int sixenseSetHighPriorityBindingEnabled( int on_or_off );
```

## Arguments

| | |
|---|---|
| on_or_off | 1 enables High Priority binding, 0 disables it. |

## Return Values

SIXENSE_SUCCESS is returned as long as the Sixense system has been initialized, otherwise SIXENSE_FAILURE.

## Description

By default, a Base Unit's RF link is in low priority binding mode. This means that any controllers that are powered up nearby will bind to whichever Base Unit they communicate with first. In cases where there is only one base unit nearby, this will be the correct behavior. If there are multiple Base Units within RF range (within about 10 meters), a controller in binding mode may link to any of the in-range Base Units. To prevent this, the game application can put the Base Unit into High Priority Binding mode, which will cause any controllers to link to that specific unit.

In a typical application, High Priority binding mode is enabled during the player select mode. For example, when starting a game, High Priority Binding would be enabled, then the screen would say "Player 1 press a button to continue".

High Priority binding should only be left enabled for as long as necessary. The game should monitor the number of controllers currently linked, then disable high priority binding as soon as the desired number of controllers are available. This will make it less likely that multiple bases will be in this mode at the same time.

# sixenseGetHighPriorityBindingEnabled

This function returns the current state of High Priority RF binding mode. This call is only used with the wireless Sixense devkits.

## Definition

```
int sixenseGetHighPriorityBindingEnabled( int *on_or_off );
```

## Arguments

| | |
|---|---|
| `on_or_off` | The current state of the High Priority binding mode is stored in the variable pointed to by this argument. 1 means it is enabled, 0 is disabled. |

## Return Values

SIXENSE_SUCCESS is returned as long as the Sixense system has been initialized, otherwise SIXENSE_FAILURE.

## Description

For a detailed description of High Priority Binding mode, see `sixenseSetHighPriorityBindingEnabled`.

# sixenseSetBaseColor

Sets the color of the LED on the Sixense wireless devkits. The Razer Hydra colors cannot be changed.

## Definition

```
int sixenseSetBaseColor( unsigned char red,
                         unsigned char green,
                         unsigned char blue );
```

## Arguments

| | |
|---|---|
| `red` | Red component of the led color. 0 is off and 255 is fully red. |
| `green` | Green component of the led color. |
| `blue` | Blue component of the led color. |

## Return Values

SIXENSE_SUCCESS is returned as long as the Sixense system has been initialized, otherwise SIXENSE_FAILURE.

## Description

This call sets the RGB value of the LED on the base unit. The 3.2 devkit is limited to approximately 64 different colors.

# sixenseGetBaseColor

Gets the color of the LED on the Sixense wireless devkits. The Razer Hydra colors cannot be changed.

## Definition

```
int sixenseGetBaseColor( unsigned char *red,
                         unsigned char *green,
                         unsigned char *blue );
```

## Arguments

| | |
|---|---|
| `red` | Red component of the led color. 0 is off and 255 is fully red. |
| `green` | Green component of the led color. |
| `blue` | Blue component of the led color. |

## Return Values

SIXENSE_SUCCESS is returned as long as the Sixense system has been initialized, otherwise SIXENSE_FAILURE.

## Description

This returns the current RGB value of the LED on the base unit.

# Constants

# Return Codes Returned by libsixense

List of return codes returned by libsixense

## Definition

| | |
|---|---|
| SIXENSE_SUCCESS | Function call completed successfully. |
| SIXENSE_FAILURE | An Error occurred during function call. |

# Button Macros

The Sixense SDK defines a set of macros for easily checking the state of a button. These macros can be AND'ed with the `buttons` value in the `sixenseControllerData` structure to check the state of the desired button.

## Definition

| |
|---|
| SIXENSE_BUTTON_1 |
| SIXENSE_BUTTON_2 |
| SIXENSE_BUTTON_3 |
| SIXENSE_BUTTON_4 |
| SIXENSE_BUTTON_START |
| SIXENSE_BUTTON_BUMPER |
| SIXENSE_BUTTON_JOYSTICK |

## Example

```
if( ssdata.buttons0 & SIXENSE_BUTTON_1 ) {
      printf("1 button pressed\n);
}
```