3DSYSTEMS

OpenHaptics Toolkit

version 3.4.0

API Reference Guide

TABLE OF CONTENTS

COPYRIGHT

WARRANTY

No warranties of any kind are created or extended by this publication. 3D Systems warrants that the Touch haptic device will be free from defects in materials and workmanship, during the applicable warranty period, when used under the normal conditions described in the documentation provided to you, including the respective User Guide. 3D Systems will promptly repair or replace the Touch, if required, to make it free of defects during the warranty period. This warranty excludes repairs required during the warranty period because of abnormal use or conditions (such as riots, floods, misuse, neglect or improper service by anyone except 3D Systems or its authorized service provider). For consumers who are covered by consumer protection laws or regulations in their country of purchase or, if different, their country of residence, the benefits conferred by our standard warranty are in addition to, and operate concurrently with, all rights and remedies conveyed by such consumer protection laws -and regulations, including but not limited to these additional rights.

LIMITATION OF LIABILITY

3D SYSTEMS WILL NOT BE RESPONSIBLE FOR CONSEQUENTIAL, EXEMPLARY OR INCIDENTAL DAMAGES (SUCH AS LOSS OF PROFIT OR EMPLOYEE'S TIME) REGARDLESS OF THE REASON. IN NO EVENT SHALL THE LIABILITY AND/OR OBLIGATIONS OF 3D SYSTEMS ARISING OUT OF THE PURCHASE, LEASE, LICENSE AND/OR USE OF THE EQUIPMENT BY YOU OR OTHERS EXCEED THE PURCHASE PRICE OF THE TOUCH 3D DEVICE.

1 INTRODUCTION

Thank you for downloading the OpenHaptics Toolkit version 3.4.0!

Should you encounter any difficulties while setting up or learning the application, we provide a variety of resources to help you learn the product. These are described below in “Resources for Learning the OpenHaptics Toolkit”.

Resources for Learning the OpenHaptics Toolkit

The following documentation and other materials are provided to assist you in learning about OpenHaptics:

OpenHaptics Installation Guide This guide walks you through installing the toolkit and deploying your haptically enabled application. Detailed instructions for installing the Geomagic® Touch™ haptic device can be found in the Geomagic Touch User Guide that came with your device. This can also be found on the OpenHaptics CD.

OpenHaptics Programmer’s Guide This guide explains the OpenHaptics toolkit (which includes the QuickHaptics™ micro API), and introduces you to the architecture of the toolkit, how it works, and what you can do with it. The guide will also introduce you to the fundamental components of creating haptic environments.

OpenHaptics API Reference This manual is meant to be used as a companion to the OpenHaptics Toolkit Programmer’s Guide. It contains reference pages to all the QuickHaptics micro API and OpenHaptics toolkit HDAPI and HLAPI functions and types as well as appendices with tables that describe all the parameters.

Source Code Examples & Guide Several examples with source code to illustrate commonly used functionality of the HDAPI and HLAPI are installed with the toolkit. These include both console examples and graphics examples. A guide to these examples is located in <OpenHaptics Install directory>\doc.

Developer Support Center This is described in more detail below.

The Developer Support Center

A more recent version of this document may be available for download from the online Developer Support Center (DSC). To access the DSC, go to http://dsc.sensable.com.

The DSC provides customers with 24 x 7 access to the most current information and forums for the OpenHaptics toolkit. Please note that you will be asked to create a registration profile and have your customer information authenticated before you will have access to the DSC.

OVerview

This manual is meant to be used as a companion to the OpenHaptics® Toolkit Programmer’s Guide. It contains reference pages to all the OpenHaptics® toolkit HDAPI and HLAPI functions and types as well as appendices with tables that describe all the parameters. Functions are listed alphabetically by section.

A more recent version of this document may be available for download from the online Developer Support Center (DSC). To access the DSC, go to http://dsc.sensable.com/. The DSC provides 24/7 access to the most current information and forums for the OpenHaptics toolkit. Please note that you will need to register for a user name and password to access the DSC.

For the QuickHaptics™ micro API, a detailed description of classes and functions with links to code examples is provided in online Doxygen format in the DSC and not in this guide.

2 DEVICE ROUTINES

The following are routines for managing the device and forces. This includes all functionality for initializing devices, querying and setting device state, and turning capabilities on or off.

hdBeginFrame

Description:	Begins an haptics frame, which is a block of code within which the device state is guaranteed to be consistent. Updates state from the device for current/last information. All state-related information, such as setting state, should be done within a haptics frame.
Syntax:	`void hdBeginFrame(HHD hHD)` Argument: hHD - The device handle of an initialized device
Returns:	None
Usage:	Typically the first haptics call per device per scheduler tick. For example, if two haptics devices are being managed, hdBeginFrame() should be called for each of them before device-specific calls are made. Only one frame is allowed per device per scheduler tick unless HD_ONE_FRAME_LIMIT is disabled. However, frames for the same device can be nested within a scheduler tick. This function automatically makes the supplied device current.
Example:	`HHD hHD = hdGetCurrentDevice;` `hdBeginFrame(hHD;`
Errors:	HD_ILLEGAL_BEGIN if a frame was already completed for the current device within the current scheduler tick.
See also:	hdMakeCurrentDevice, hdDisable, hdIsEnabled

hdDisable

Description:	Disables a capability.
Syntax:	`void hdDisable(HDenum cap)` Argument: cap - The capability to disable. For a list of the capabilities see "Table A-10: hdEnable, hdDisable Parameters" on page 61.
Returns:	None
Usage:	Capabilities are typically related to safety mechanisms. Use extreme caution when disabling safety features.
Example:	`hdDisable(HD_MAX_FORCE_CLAMPING);`
Errors:	HD_INVALID_ENUM if cap does not support hdEnable() or hdDisable().
See also:	hdEnable, hdIsEnabled

hdDisableDevice

Description:	Disables a device. The handle should not be used afterward.
Syntax:	`void hdDisableDevice(HHD hHD)` Argument: hHD - The device handle of an initialized device.
Returns:	None
Usage:	Call during cleanup when done using a device. Typically the last call after stopping the scheduler and unscheduling all scheduled callbacks.

Example:	`hdStopScheduler();`
	`hdUnschedule(scheduleCallbackHandle);`
	`hdDisableDevice(hdGetCurrentDevice());`

Errors:	None
See also:	hdInitDevice, hdStopScheduler

hdEnable

Description:	Enables a capability.
Syntax:	`void hdEnable(HDenum cap)` Argument: cap - The capability to enable. For a list of the capabilities see "Table A-10: hdEnable, hdDisable Parameters" on page 61.
Returns:	None
Usage:	Capabilities are typically related to safety mechanisms. Most are turned on by default.
Example:	`hdEnable(HD_FORCE_OUTPUT);`
Errors:	HD_INVALID_ENUM if cap does not support enable/disable.
See also:	hdDisable, hdIsEnabled

hdEndFrame

Description:	Ends a haptics frame. Causes forces and other states to be written to the device. An hdBeginFrame() and hdEndFrame() should always be paired within the same scheduler tick.
Syntax:	`void hdEndFrame(HHD hHD)` Argument: hHD - The device handle of an initialized device.
Returns:	None
Usage:	Typically the last haptics call per device, per scheduler tick.
Example:	`hdEndFrame(hdGetCurrentDevice());`
Errors:	HD_ILLEGAL_END if this call is not paired correctly with an hdBeginFrame() of the same device handle.
See also:	hdBeginFrame, hdMakeCurrentDevice

hdGet (Parameter Values)

Description:	Obtains information about the device. There are five query functions for obtaining information about the device associated with the parameter name used.

Syntax:	`void hdGetBooleanv(HDenum pname, HDboolean *params)`
	`void hdGetIntegerv(HDenum pname, HDint *params)`
	`void hdGetFloatv(HDenum pname, HDfloat *params)`
	`void hdGetDoublev(HDenum pname, HDdouble *params)`
	`void hdGetLongv(HDenum pname, HDlong *params)`
	
	Argument: pname - Parameter name to use.
	Argument: params - Array where the results will be returned. For a list of parameters see "Table A-1: hdGet Parameters" on page 56.

Returns:	None
Usage:	Primary function for getting device information. Depending on the parameter, one should use the appropriate params type, and therefore the appropriate function signature. The caller of the function has the responsibility for allocating memory. These functions should only be called within a haptics frame.

Example:	`HDdouble position[3];`
	`hdGetDoublev(HD_CURRENT_POSITION,position);`
	
	`HDint buttons;`
	`hdGetIntegerv(HD_CURRENT_BUTTONS,&buttons);`

Errors:	HD_INVALID_INPUT_TYPE if pname does not support the input type. HD_INVALID_ENUM if pname does not support hdGet().
See also:	hdSet (Parameter Values)

hdGetCurrentDevice

Description:	Gets the handle of the current device.
Syntax:	`HHD hdGetCurrentDevice()`
Returns:	The handle of the current device.
Usage:	Primarily used in multi-device applications to keep track of which device is current, or for calls that require a device handle.
Example:	`HHD hHD = hdGetCurrentDevice();`
Errors:	HD_INVALID_HANDLE if no device is current, for example, if no device has been initiated yet.
See also:	hdMakeCurrentDevice

hdGetError

Description:	Returns errors in order from most recent to least. Each call retrieves and removes one error from the error stack. If no error exists, this function returns a HDErrorInfo with HD_SUCCESS as its code. HDErrorInfo contains the error code from the defines file, the handle of the device that was active when the error occurred, and the device's original internal error code. The internal code can be used for obtaining additional support from the device vendor.
Syntax:	`HDErrorInfo hdGetError()`
Returns:	HDErrorInfo. This data structure is a typedef in hdDefines.h. It exposes information on the handle of the device, an errorCode, and an internal error code. See "Table A-15: Device Error Codes" on page 62 for a list of errorCodes and descriptions.
Usage:	Intersperse in code to occasionally check for errors.

Example:	`HHDErrorInfo error;`
	`error = hdGetError();`
	`if (HD_DEVICE_ERROR(error))`
	`// do error handling`
	`hdGetErrorString(error.errorCode);`

Errors:	None
See also:	hdGetErrorString, HDErrorInfo Type

hdGetErrorString

Description:	Returns information about an error code.
Syntax:	`HDstring hdGetErrorString(HDerror errorCode)` Argument: errorCode - An error code from hdDefines.
Returns:	A readable string representing the explanation of the error code.
Usage:	Obtains useful information about error codes. The return string is static and should not be deallocated using Free or Delete or any similar function.
Example:	`hdGetErrorString(HD_FRAME_ERROR);`
Errors:	None
See also:	hdGetError, HD_DEVICE_ERROR

hdGetString

Description:	Gets a string value for the associated parameter name.
Syntax:	`HDstring hdGetString(HDenum pname)` Argument: pname - Parameter name to use. For a list what types and how many values are used by each parameter name, see "Table A-3: Get Identification Parameters" on page 57. Note that not all parameter names support string values.
Returns:	Requested string associated with the parameter name.
Usage:	Gets readable string information about device properties, such as the device model type. The return string is static and should not be deallocated using Free or Delete or any similar function.
Example:	`HDstring *deviceType = hdGetString(HD_DEVICE_MODEL_TYPE);`
Errors:	HD_INVALID_INPUT_TYPE if pname does not support string as an input type. HD_INVALID_ENUM if pname does not support hdGetString().

hdInitDevice

Description:	Initializes the device. If successful, this returns a handle to the device and sets the device as the current device.
Syntax:	`HHD hdInitDevice(HDstring pConfigName)` Argument: pConfigName **-** The name of the device, such as the name found under the control panel "Geomagic Touch Setup". If HD_DEFAULT_DEVICE is passed in as pConfigName, hdInitiDevice() will initialize the first device that it finds.
Returns:	A handle to the initialized device.
Usage:	Generally the first haptics command that should be issued.
Example:	`HHD hDevice = hdInitDevice("Default Device");`
Errors:	HD_DEVICE_ALREADY_INITIATED if the device was already initiated. HD_DEVICE_FAULT if the device could not be initiated, for example, if pConfigName is invalid.
See also:	hdDisableDevice, hdMakeCurrentDevice, hdStartScheduler

hdIsEnabled

Description:	Checks if a capability is enabled. Capabilities are in hdDefines.h file under the enable/disable category.
Syntax:	`HDboolean hdIsEnabled(HDenum cap)` Argument: cap **-** Capability to check. For a list of the capabilities see "Table A-10: hdEnable, hdDisable Parameters" on page 61.
Returns:	Whether the capability is enabled.
Usage:	Capabilities are typically related to safety mechanisms. Most are turned on by default.
Example:	`HDboolean bForcesOn = hdIsEnabled(HD_FORCE_OUTPUT);`
Errors:	HD_INVALID_ENUM if cap does not support enable/disable.
See also:	hdEnable, hdDisable

hdMakeCurrentDevice

Description:	Makes the device current. All subsequent device-specific actions such as getting and setting state or querying device information will be performed on this device until another is made current.
Syntax:	`void hdMakeCurrentDevice(HHD hHD)` Argument: hHD - The device handle of an initialized device.
Returns:	None
Usage:	Primarily used to switch between devices in multi-device applications.

Example:	`hdBeginFrame(hHD1);`
	`hdGetDoublev(HD_CURRRENT_POSITION, p1);`
	`hdBeginFrame(hHD2);`
	`hdGetDoublev(HD_CURRENT_POSITION, p2);`
	`calculateforce(p1,p2,outforce1,outforce2);`
	`hdMakeCurrentDevice(hHD1);`
	`hdSetDoublev(HD_CURRENT_FORCE,outforce1);`
	`hdEndFrame(hHD1);`
	`hdMakeCurrentDevice(hHD2);`
	`hdSetDoublev(HD_CURRENT_FORCE,outforce2);`
	`hdEndFrame(hHD2);`

Errors:	HD_INVALID_HANDLE if hHD does not refer to an initiated device.
See also:	hdInitDevice

hdSet (Parameter Values)

Description:	Sets information associated with the parameter name.

Syntax:	`void hdSetBooleanv(HDenum pname, const HDboolean *params)`
	`void hdSetIntegerv(HDenum pname, const HDint *params)`
	`void hdSetFloatv(HDenum pname, const HDfloat *params)`
	`void hdSetDoublev(HDenum pname, const HDdouble *params)`
	`void hdSetLongv (HDenum pname,const HDlong *params)`
	
	Argument: pname - Parameter name to set.
	Argument: params - Array of values to set. For a list of parameters see Tables in “Set Parameters” on page 59.

Returns:	None
Usage:	Primary function for sending information to the device or changing device properties. Depending on the parameter, one should use the appropriate params type, and therefore the appropriate function signature. The caller of the function has the responsibility for allocating memory. These functions should only be called within a haptics frame.

Example:	`HDfloat forces[3] = { 2,.5, 0};`
	`hdSetFloatv(HD_CURRENT_FORCE,forces);`
	Or,
	`hduVector3Dd forces(0.2,0.5,0);`
	`hdSetFloatv(HD_CURRENT_FORCE,forces);`

Errors:	HD_INVALID_INPUT_TYPE if pname does not support the input type. HD_INVALID_ENUM if pname does not support hdSet.
See also:	hdGet (Parameter Values).

HD_DEVICE_ERROR

Description:	A macro useful for checking if an error has occurred. #define HD_DEVICE_ERROR(X)(((X).errorCode)!=HD_SUCCESS)
Syntax:	`int HD_DEVICE_ERROR(x)` Argument: X - The error structure to check. Error structure is of type HDErrorInfo.
Returns:	1 if error, 0 if no error
Usage:	Typically, call this with hdGetError() to check if the error stack contains an error.

Example:	`HDErrorInfo error;`
	`if (HD_DEVICE_ERROR(error = hdGetError()))`
	`{`
	`hduPrintError(stderr, &error, "HDAPI device error encountered");`
	`return -1;`
	`}`

Errors:	None
See also:	hdGetError, hdGetErrorString

3 CALIBRATION ROUTINES

This chapter covers the routines used for managing the calibration of the device. Calibration is necessary for accurate position mapping and force/torque rendering of the haptic device; calibration synchronizes the software's notion of the device position with the actual mechanical position. For example, if the device is not calibrated, then the software may believe the device position is in the corner of the workspace while the actual device arm is in the center of its workspace.

A typical application begins by asking the user to put the device in a neutral position or inkwell so that it can be calibrated. Some devices automatically calibrate so that this step is unnecessary.

The calibration interface provides functions for querying the calibration style(s) supported by the device, checking the calibration status, and updating the calibration.

Querying Calibration Styles

Supported calibration styles differ depending on device. The list of calibration styles is:

HD_CALIBRATION_ENCODER_RESET: The device position must be read while the device is in a specified neutral position. For devices that use encoder reset calibration style, typically the user is asked to place the device in the neutral position at the start of the application so that position can be sampled. Touch X haptic device models use encoder reset.

HD_CALIBRATION_INKWELL: The device must be placed into the device's inkwell to be calibrated. The Geomagic Touch haptic device supports inkwell calibration.

HD_CALIBRATION_AUTO: The device automatically calibrates by collecting positional data while the user moves the device arm around. This data persists between sessions so the device does not need to be repeatedly calibrated. The Geomagic Touch X haptic device supports auto calibration.

The calibration styles that a particular device supports can be obtained as follows:

```
HDint supportedStyles;

hdGetIntegerv(HD_CALIBRATION_STYLE,&supportedStyles);
```

It is possible for a device to support more than one calibration style. The supportedStyles value should be masked (&) with HD_CALIBRATION_ENCODER_RESET, HD_CALIBRATION_INKWELL, HD_CALIBRATION_AUTO to determine if each particular style is supported.

hdCheckCalibration

Description:	Checks the calibration status. If the return value is HD_CALIBRATION_OK, the device is already calibrated. If the return value is HD_CALIBRATION_NEEDS_UPDATE, call hdUpdateCalibration() to update the calibration. If the return value is HD_CALIBRATION_NEEDS_MANUAL_INPUT, the user needs to provide additional input particular to the calibration style.
Syntax:	`HDenum hdCheckCalibration()`
Returns:	Possible values are: HD_CALIBRATION_OK HD_CALIBRATION_NEEDS_UPDATE HD_CALIBRATION_NEEDS_MANUAL_INPUT
Usage:	For devices that support auto calibration, call intermittently so that the device continues to update its calibration.

Example:	if(hdCheckCalibration()==HD_CALIBRATION_NEEDS_UPDATE)
	{
	hdUpdateCalibration(myStyle);

Errors:	HD_DEVICE_FAULT if the calibration information could not obtained from the device.
See also:	hdUpdateCalibration

hdUpdateCalibration

Description:	Calibrates the device. The type of calibration supported can be queried through getting HD_CALIBRATION_STYLE.
Syntax:	`void hdUpdateCalibration(HDenum style)` Argument: style - The calibration style of the device. For a list of calibration styles see "Table A-3: Get Identification Parameters" on page 57.
Returns:	None
Usage:	Calibrate the device when hdCheckCalibration() returns HD_CALIBRATION_NEEDS_UPDATE.
Example:	`HDint style;` `hdGetIntegerv(HD_CALIBRATION_STYLE,&style); hdUpdateCalibration(style);`
Errors:	HD_DEVICE_FAULT if the calibration type could not be performed on the device.
See also:	hdCheckCalibration

4 SCHEDULER ROUTINES

The following are routines for managing the scheduler and scheduler routines. This includes starting and stopping the scheduler, adding callbacks into the scheduler, and managing those callbacks.

hdGetSchedulerTimeStamp

Description:	Returns the elapsed time since the start of the servo loop tick. This is useful for measuring duty cycle of the servo loop.
Syntax:	`HDdouble hdGetSchedulerTimeStamp()`
Returns:	Number of seconds since the start of the servo loop tick.
Usage:	Used to check how long operations have taken.

Example:	`HDCallbackCode HDCALLBACK schedulerTimeCallback(void *pUserData)`
	`{`
	`HDdouble *schedulerTime = (HDdouble *)pUserData;`
	`*schedulerTime = hdGetSchedulerTimeStamp();`
	`return HD_CALLBACK_DONE;`
	`}`
	`HDdouble schedulerTime;`
	`hdScheduleSynchronous(schedulerTimeCallback,&schedulerTime,`
	`HD_MAX_SCHEDULER_PRIORITY);`

Errors:	None
See also:	None

hdScheduleAsynchronous

Description:	Schedules an operation to be executed by the scheduler in the servo loop, and does not wait for its completion. Scheduler callbacks submitted from the servo loop thread are run immediately regardless of priority.

Syntax:	`HDSchedulerHandle`
	`hdScheduleAsynchronous(HDSchedulerCallback pCallback,`
	`void *pUserData,`
	`HDushort nPriority)`
	
	Argument: `pCallback` - The function callback.
	Argument: `pUserData` - The data to be used by the function.
	Argument: `nPriority` - The priority of the operation. Callbacks are processed once per scheduler tick, in order of decreasing priority.

Returns:	Handle for the operation.
Usage:	Typically used for scheduling callbacks that run every tick of the servo loop. For example, one can run a dynamic simulation within an asynchronous callback and set forces within that simulation.

Example:

```
HDCallbackCode HDCALLBACK mySchedulerCallback(void *pUserData)
{
   HDint buttons;
   hdGetIntegerv(HD_CURRENT_BUTTONS,&buttons);
   if (buttons == 0)
       return HD_CALLBACK_CONTINUE;
   return HD_CALLBACK_DONE;
}
...
HDSchedulerHandle hHandle = hdScheduleAsynchronous(mySchedulerCallback,
(void*)0, HD_DEFAULT_SCHEDULER_PRIORITY);
```

Errors:

HD_SCHEDULER_FULL if the scheduler has reached its upper limit on the number of scheduler operations that it can support at once.

HD_INVALID_PRIORITY if nPriority is out of range, i.e. less than HD_MIN_SCHEDULER_PRIORITY or greater than HD_MAX_SCHEDULER_PRIORITY.

See also: HDSchedulerCallback Type, hdStartScheduler

hdScheduleSynchronous

Description: Schedules an operation to be executed by the scheduler in the servo loop, and waits for its completion. Scheduler callbacks submitted from the servo loop thread are run immediately regardless of priority.

Syntax:

```
void hdScheduleSynchronous(HDSchedulerCallback pCallback,
void *pUserData,
HDushort nPriority)
```

Argument: `pCallback` - The function callback.

Argument: `pUserData` - The data to be used by the function.

Argument: `nPriority` - The priority of the operation. Callbacks are processed once per scheduler tick, in order of decreasing priority.

Returns: None

Usage: Typically used as a synchronization mechanism between the servo loop thread and other threads in the application. For example, if the main application thread needs to access the position or button state, it can do so through a synchronous scheduler call. Can be used for synchronously copying state from the servo loop or synchronously performing an operation in the servo loop.

Example:

```
HDCallbackCode HDCALLBACK buttonCallback(void *pUserData)
{
int *buttons = (int *)pUserData;
hdGetIntegerv(HD_CURRENT_BUTTONS,buttons);
return HD_CALLBACK_DONE;
}
int buttons;
HDSchedulerHandle hHandle =
hdScheduleSynchronous(buttonCallback, &buttons,
HD_DEFAULT_SCHEDULER_PRIORITY);
```

Errors:

HD_SCHEDULER_FULL if the scheduler has reached its upper limit on the number of scheduler operations that it can support at once.

HD_INVALID_PRIORITY if nPriority is out of range, i.e. less than HD_MIN_SCHEDULER_PRIORITY or greater than HD_MAX_SCHEDULER_PRIORITY.

See also: HDSchedulerCallback Type, hdStartScheduler

hdSetSchedulerRate

Description:	Sets the number of times the scheduler ticks its callbacks per second.
Syntax:	`void hdSetSchedulerRate(HDulong nRate)` Argument: `nrate` - The requested rate for the scheduler to run in Hz.
Returns:	None
Usage:	Typically used to control the fidelity of force rendering. Most haptic applications run at 1000Hz. PCI and EPP support 500, 1000, and 2000 Hz. Firewire supports 500, 1000, 1600 Hz, plus some increments in between based on the following expression: floor(8000/N + 0.5). As a word of caution, decreasing the rate can lead to instabilities and kicking. Increasing the servo loop rate can yield stiffer surfaces and better haptic responsiveness, but leaves less time for scheduler operations to complete. When setting the scheduler rate, check hdGetSchedulerTimeStamp() and HD_INSTANTANEOUS_UPDATE_RATE to verify that the servo loop is able to maintain the rate requested. Some devices may support a variety of settings. Certain devices may need to be flashed with the latest firmware to be able to use this feature. You can find current information from the Developer Support Center, see"The Developer Support Center" on page 5 for information about the DSC.
Example:	`// try to set rate of 500 Hz` `hdSetSchedulerRate(500);`
Errors:	HD_INVALID_VALUE if the nRate specified cannot be set.
See also:	hdGetSchedulerTimeStamp

hdStartScheduler

Description:	Starts the scheduler. The scheduler manages callbacks to be executed within the servo loop thread.
Syntax:	`void hdStartScheduler()`
Returns:	None
Usage:	Typically call this after all device initialization routines and after all asynchronous scheduler callbacks have been added. There is only one scheduler, so it needs to be started once, no matter how many devices one is using. Execution of callbacks starts when the scheduler is started. Forces are enabled at this point also.
Example:	`hdStartScheduler();`
Errors:	HD_TIMER_ERROR if the servo loop thread could not be initialized or the servo loop could not be started.
See also:	hdStopScheduler

hdStopScheduler

Description:	Stops the scheduler.
Syntax:	`void hdStopScheduler()`
Returns:	None
Usage:	Typically call this as a first step for cleanup and shutdown of devices.

Example:	`hdStopScheduler();`
	`hdUnschedule();`
	`hdUnschedule(myCallbackHandle);`
	`hdDisableDevice(hHD);`

Errors:	HD_TIMER_ERROR if the servo loop thread could not be initialized.
See also:	hdUnschedule, hdDisableDevice

hdUnschedule

Description:	Un-schedules an operation by removing the associated callback from the scheduler.
Syntax:	`void hdUnschedule(HDSchedulerHandle hHandle)` Argument: `hHandle` - Handle for an active operation to be unscheduled, obtained via hdScheduleAsynchronous() or hdScheduleSynchronous().
Returns:	None
Usage:	Used to stop an active asynchronous operation. For example, if the application thread has created an asynchronous operation that returns HD_CALLBACK_CONTINUE to run indefinitely, the application can call hdUnschedule() to force the callback to terminate.

Example:	`HDSchedulerHandle hHandle =`
	`hdScheduleAsynchronous(mySchedulerCallback,`
	`(void*)0, HD_DEFAULT_SCHEDULER_PRIORITY);`
	`hdUnschedule(hHandle);`

Errors:	HD_INVALID_OPERATION if the scheduler operation associated with the handle has already terminated.
See also:	hdStopScheduler, hdScheduleAsynchronous, hdScheduleSynchronous

hdWaitForCompletion

Description:	Checks if a callback is still scheduled for execution. This can be used as a non-blocking test or can block and wait for the callback to complete.
Syntax:	`HDboolean hdWaitForCompletion(HDSchedulerHandle hHandle, HDWaitCode param)` Argument `hHandle` - Handle of the target active asynchronous operation. Argument `param` - Either HD_WAIT_CHECK_STATUS or HD_WAIT_INFINITE.
Returns:	Whether the callback is still scheduled. If HD_WAIT_CHECK_STATUS is passed into param, returns true if the scheduler operation is still active. If HD_WAIT_INFINITE is passed into param, this function will return true if the wait was successful or false if the wait failed.
Usage:	Can be used on an asynchronous operation to either check the status or to commit to waiting for the operation to finish.

Example:	`HDSchedulerHandle hHandle =`
	`hdSchedulerAsynchronous(mySchedulerCallback,`
	`(void*) 0, HD_DEFAULT_SCHEDULER_PRIORITY);`
	
	`/* Do some other work and then wait for the operation to complete */`
	
	`HDboolean bWaitSuccess = hdWaitForCompletion(hHandle, HD_WAIT_INFINITE);`

Errors:	None
See also:	hdUnschedule, hdScheduleAsynchronous, hdScheduleSynchronous

hdDeploymentLicense

Description:	Activates the deployment license issued to the application developer. Once the deployment license has been validated, it will remain active until the application is exited. It is not an error to call this more than once in an application session.
Syntax:	`HDboolean hdDeploymentLicense (const char* vendorName, const char* applicationName, const char* password)` Argument: `vendor` - The name of the organization to which the license is issued. Argument: `applicationName` - The name of the application the license is tied to. Argument: `password` - The license string which authenticates the deployment license.
Returns:	Non-zero if the validation succeeded, zero if the validation failed.
Usage:	Needs to be executed before hdInitDevice to operate correctly. hdInitDevice will fail if the deployment license is not valid. Note that if a developers license is present hdInitDevice will succeed.
Example:	HDboolean bValid = hdDeploymentLicense (“Haptic Co.”, “Haptic Application”, 6A12...);
Errors:	HD_DEVICE_FAULT if the calibration information could not obtained from the device.
See also:	hdDeploymentLicense

6 HDAPI TYPES

The following are variable and function types used by the HDAPI.

HDSchedulerCallback Type

Description:	Scheduler operation function type, used to define operations to be run in the scheduler thread.
Syntax:	`typedef HDCallbackCode (HDCALLBACK *HDSchedulerCallabck) (void *pUserData)` Argument: `pUserData` - Data used by the operation, created when the operation is scheduled.
Returns:	HD_CALLBACK_DONE or HD_CALLBACK_CONTINUE depending on whether the operation is complete or should be executed again the next tick of the scheduler.
Usage:	Scheduled by hdSchedulerSynchronous() or hdScheduleAsynchronous(), the operation is then run during the next scheduler tick, or immediately for synchronous operations if the scheduler has not been started. The return value controls whether the operation terminates after being run or runs again in the next scheduler tick. A periodic operation, i.e. one that is executed indefinitely, should return HD_CALLBACK_DONE when it is finally to be unscheduled.

Example:	HDCallbackCode HDCALLBACK renderForceCallback(void *pUserData)
	{
	HDdouble beginTime = hdGetSchedulerTimeStamp();
	HHD hHD = hdGetCurrentDevice();
	hdBeginFrame(hHD);
	computeForce();
	hdEndFrame(hHD);
	HDdouble endTime = hdGetSchedulerTimeStamp();
	/* Compute the elasped time within this callback */
	HDdouble callbackTime = endTime - beginTime;
	}

HDSchedulerHandle Type

Description:	A handle to scheduler operations.
Syntax:	`typedef unsigned long HDSchedulerHandle`
Returns:	None
Usage:	A HDSchedulerHandle is returned whenever the user schedules a scheduler callback. It can be used for a variety of operations such as unscheduling the associated callback, checking the status of the callback, and waiting for the callback to complete.
Example:	HDSchedulerHandle handle = hdScheduleAsynchronous(MyServoForceCallbackFunction, 0, HD_MAX_SCHEDULER_PRIORITY); ... hdUnSchedule(handle);
Errors:	None
See also:	HDSchedulerCallback Type, hdScheduleSynchronous, hdScheduleAsynchronous, hdUnschedule, hdWaitForCompletion

hlBeginFrame

Description:	Begins a haptics rendering pass, that is, a block of code that sends primitives to the haptic device to be rendered. hlBeginFrame() updates the current state of the haptic device and clears the current set of haptics primitives being rendered in preparation for rendering a new or updated set of primitives. All haptic primitive rendering functions (that is, shapes and effects) must be made within a begin/end frame pair. hlBeginFrame() also updates the world coordinate reference frame used by the haptic rendering engine. By default, hlBeginFrame() samples the current GL_MODELVIEW_MATRIX from OpenGL® to provide a world coordinate space for the entire haptic frame. All positions, vectors and transforms queried through hlGet*() or hlCacheGet*() in the client or collision threads will be transformed into that world coordinate space. Typically, the GL_MODELVIEW_MATRIX contains just the world to view transform at the beginning of a render pass.
Syntax:	`void hlBeginFrame()`
Returns:	none.
Usage:	Typically, haptic primitives are specified just following (or just before) rendering of graphics primitives. hlBeginFrame() is called at the start of the haptics update, the primitives are specified, and then hlEndFrame() is called.

Example:	`hlBeginFrame();`
	`hlBeginShape(…);`
	…
	`hlEndShape();`
	`hlEndFrame();`

Errors:	HL_INVALID_OPERATION if nested inside another hlBeginFrame() or if no haptic rendering context is active.
See also:	hlEndFrame

hlContextDevice

Description:	Sets the haptic device for the current rendering context.
Syntax:	`void hlContextDevice(HHD hHD)`
Returns:	None
Parameters:	`hHD` - handle to haptic device or HD_INVALID_HANDLE to set no device.
Usage:	Used to change which haptic device will be used with the current haptic rendering context. HD_INVALID_HANDLE can also be used to release the use of the device and deactivate haptic rendering.

Example:	`HHD hHD1, hHD2;`
	…
	`// create context with first haptic device`
	`hHLRC = hlCreateContext(hHD1);`
	`hlMakeCurrent(hHLRC);`
	…
	`// switch to second device`
	`hlContextDevice(hHD2);`

Errors:	HL_INVALID_OPERATION if nested inside another hlBeginFrame() or if no haptic rendering context is active.
See also:	hlMakeCurrent, hlCreateContext, hlDeleteContext, hlGetCurrentContext, hlGetCurrentDevice

hlCreateContext

Description:	Creates a new haptic rendering context. The haptic rendering context stores the current set of haptic primitives to be rendered as well as the haptic rendering state.
Syntax:	`HHLRC hlCreateContext(HHD hHD);` Argument: `hHD` - Device handle to the initialized haptic device.
Returns:	HHLRC handle to haptic rendering context.
Usage:	Called during program initialization after initializing the haptic device. HD_INVALID_HANDLE can be provided as the device handle in order to create the context in suspended state.

Example:	`HDErrorInfo error;`
	`hHD = hdInitDevice(HD_DEFAULT_DEVICE);`
	`if (HD_DEVICE_ERROR(error = hdGetError()))`
	`{`
	`hduPrintError(stderr, &error, "Failed to initialize haptic device");`
	`exit(1);`
	`}`
	`hHLRC = hlCreateContext(hHD);`
	`hlMakeCurrent(hHLRC);`

Errors:	None
See also:	hlMakeCurrent, hlDeleteContext

hlDeleteContext

Description:	Deletes a haptic rendering context. The haptic rendering context stores the current set of haptic primitives to be rendered as well as the haptic rendering state.
Syntax:	`void hlDeleteContext(HHLRC hHLRC);` Argument: `hHLRC` - Handle to haptic rendering context returned by hlCreateContext().
Returns:	None
Usage:	Called at program exit to end haptic rendering and deallocate memory.

Example:	`hHLRC = hlCreateContext(hHD);`
	`hlMakeCurrent(hHLRC);`
	…
	`hlMakeCurrent(NULL);`
	`hlDeleteContext(hHLRC);`

Errors:	None
See also:	hlMakeCurrent, hlCreateContext

hlEndFrame

Description:	Ends a haptics rendering pass, that is a block of code that sends primitives to the haptic device to be rendered. hlEndFrame() flushes the set of haptics primitives (that is, shapes, effects) specified since the last hlBeginFrame() to the haptics device. All haptic primitive rendering functions must be made within a begin/end frame pair.
Syntax:	`void hlEndFrame()`
Returns:	None
Usage:	Typically, haptic primitives are specified just following (or just before) rendering of graphics primitives. hlBeginFrame() is called at the start of the haptics update, the primitives are specified, and then hlEndFrame() is called.

Example:	`hlBeginFrame();`
	`hlBeginShape(…);`
	…
	`hlEndShape();`
	`hlEndFrame();`

Errors:	HL_INVALID_OPERATION if not preceded by an hlBeginFrame(), if inside an hlBeginShape()/ hlEndShape() block, or if no rendering context is active.
See also:	hlBeginFrame

hlGetCurrentContext

Description:	Returns a handle to the currently active haptic rendering context for a given thread.
Syntax:	`HHLRC hlGetCurrentContext(void)`
Returns:	Handle to currently active rendering context for the current thread or NULL if no context is active.
Parameters:	None
Usage:	Used to query which haptic rendering context is active for a given thread.

Example:	`HHLRC hHLRC = hlGetCurrentContext();`
	`if (hHLRC != myHLRC)`
	{
	`hlMakeCurrent(myHLRC);`
	}

Errors:	None
See also:	hlMakeCurrent, hlCreateContext, hlDeleteContext, hlContextDevice, hlGetCurrentContext

hlGetCurrentDevice

Description:	Returns a handle to the haptic device for the currently active haptic rendering context.
Syntax:	`HHD hlGetCurrentDevice(void);`
Returns:	Handle to haptic device for currently active rendering context for the current thread or HD_ INVALID_HANDLE if no context is active or no haptic device is selected for the current context.
Parameters:	None
Usage:	Used to query which haptic device is active for the current context.

Example:	`HHHD hHD = hlGetCurrentDevice();`
	`if (hHD == HD_INVALID_HANDLE)`
	{
	`hlContextDevice(myHD);`
	}

Errors:	None
See also:	hlMakeCurrent, hlCreateContext, hlDeleteContext, hlContextDevice, hlGetCurrentContext

hlMakeCurrent

Description:	Makes a haptic rendering context current. The current rendering context is the target for all rendering and state commands. All haptic rendering commands will be sent to the device in the current context until a context with a different device is made current.
Syntax:	`void hlMakeCurrent (HHLRC hHLRC)` Argument: `hHLRC` - The handle to haptic rendering context returned by hlCreateContext().
Returns:	None
Usage:	Called at program startup after creating the rendering context or called during program execution to switch rendering contexts in order to render to multiple haptic devices.

Example:	`HDErrorInfo error;`
	`hHD = hdInitDevice(HD_DEFAULT_DEVICE);`
	`if (HD_DEVICE_ERROR(error = hdGetError()))`
	`{`
	`hduPrintError(stderr, &error, "Failed to initialize haptic device");`
	`exit(1);`
	`}`
	`hHLRC = hlCreateContext(hHD);`
	`hlMakeCurrent(hHLRC);`

Errors:	None
See also:	hlCreateContext, hlDeleteContext

hlEnable, hlDisable

Description:	Enables or disables a capability for the current rendering context.

Syntax:	`void hlEnable(HLenum cap);`
	`void hlDisable(HLenum cap);`
	Argument: `cap` - Capability to enable. For a list of possible capabilities see, "Table B-9: hlEnable, hlDisable, hlIsEnabled" on page 67.

Returns:	none.
Usage:	Enables or disables the capability specified.
Example:	`hlDisable(HL_PROXY_RESOLUTION);` `hlProxydv(HL_PROXY_POSITION, newProxyPos);`
Errors:	HL_INVALID_ENUM if cap is not one of the values listed.
See also:	hlIsEnabled

hlGetBooleanv, hlGetDoublev, hlGetIntegerv

Description:	Allow querying of the different state values of the haptic renderer.

Syntax:	`void hlGetBooleanv(HLenum pname, HLboolean *params)`
	`void hlGetDoublev(HLenum pname, HLdouble *params)`
	`void hlGetIntegerv(HLenum pname, HLint *params)`
	Argument: `pname` - Name of parameter (state value) to query. For a list of parameters see "Table B-1: hlGetBooleanv, hlGetIntegerv, hlGetDoublev" on page 63.
	Argument: `params` - Address to which to return the value of the parameter being queried.

Returns:	None
Parameters:	`hHD` - handle to haptic device or HD_INVALID_HANDLE to set no device.
Usage:	Queries the state of the haptic renderer.

Example:	`hlDouble proxyPos[3];`
	`hlGetDoublev(HL_PROXY_POSITION, proxyPos);`
	`hlBoolean switchDown;`
	`hlGetBooleanv(HL_BUTTON1_STATE, &switchDown);`

Errors:	HL_INVALID_ENUM if pname is not one of listed values.
See also:	hlIsEnabled, hlCacheGetBooleanv, hlCacheGetDoublev

hlGetError

Description:	Returns the error code from the last API command called.
Syntax:	`HLerror hlGetError(void);`
Returns:	An HLerror struct containing an HL error code and an optional device specific error code. The device specific error code is the same as the errorCode in hdGetError(). For an explanation of these device error codes, see "Table A-15: Device Error Codes" on page 62. For a list of possible error codes the errorCode field of the returned HLerror can contain, see "Table B-3: hlGetError" on page 65.
Usage:	hlGetError() functions similarly to hdGetError(), i.e. it returns errors in order from most recent to least. Each call retrieves and removes one error from the error stack. Intersperse hlGetError() in the code to occasionally check for errors. For a list of possible error codes the errorCode field of the returned HLerror can contain, see "Table B-3: hlGetError" on page 65.

Example:	`HLerror error = hlGetError();`
	`if (error.errorCode == HL_DEVICE_ERROR)`
	`{`
	
	`}`

Errors:	None

hlGetString

Description:	Returns a string describing the haptic renderer implementation.
Syntax:	`const HLubyte* hlGetString(HLenum name);` Argument: `name` - Haptic renderer implementation property to describe. For a list of possible names, see "Table B-4: hlGetString" on page 66.
Returns:	A static character string describing the implementation of the haptic renderer.
Usage:	Used to determine which implementation and version of the HL library is being used. This allows for programs to take advantage of functionality in newer implementation of the library while maintaining backward compatibility.

Example:	`const HLubyte* vendor = hlGetString(HL_VENDOR);`
	`const HLubyte* version = hlGetString(HL_VERSION);`
	`cout << "vendor= " << vendor << "version= " << version << endl;`
	`output:`
	`vendor=SensAble Technologies, Inc. version=1.01.23`

Errors:	HL_INVALID_ENUM if name is not one of the values listed.

hlHinti, hlHintb

Description:	Sets parameters that allow the haptic renderer to optionally perform selected optimizations.

Syntax:	`void hlHinti(HLenum target, HLint value)`
	`void hlHintb(HLenum target, HLboolean value)`
	Argument: `target` - Hint property to set. For list of possible values, see "Table B-5: hlHinti, hlHintb" on page 66.
	Argument: `value` - Value of target property to set.

Returns:	None
Usage:	Used to allow control over the optimizations used by the haptics renderer.

Example:	`hlHinti(HL_SHAPE_FEEDBACK_BUFFER_VERTICES, nVertices);`
	`hlBeginShape(HL_SHAPE_FEEDBACK_BUFFER);`
	`glBegin(GL_TRIANGLES);`
	`for (int i = 0; i < nVertices; ++i)`
	`glVertex3f(vertices[i][0], vertices[i][1],`
	`vertices[i][2]);`
	`glEnd();`
	`hlEndShape();`

Errors:	HL_INVALID_ENUM if target is not one of the values listed. HL_INVALID_VALUE if value is out of range. HL_INVALID_OPERATION if no haptic rendering context is current.
See also:	hlBeginShape, hlPushAttrib, hlPopAttrib

hlIsEnabled

Description:	Checks if a capability is enabled or disabled.
Syntax:	`hlBoolean hlIsEnabled(HLenum cap);`
Returns:	HL_TRUE if the capability is enabled in the current rendering context, HL_FALSE if it is disabled. Argument: `cap` - Capability to query. For a list of possible capabilities, see "Table B-9: hlEnable, hlDisable, hlIsEnabled" on page 67.
Parameters:	None
Usage:	Used to query if a particular capability feature is enabled. Capabilities include proxy resolution, haptic camera view, adaptive viewport, and gl modelview.
Example:	`hlDisable(HL_PROXY_RESOLUTION);` `assert(HL_FALSE == hlIsEnabled(HL_PROXY_RESOLUTION));`
Errors:	HL_INVALID_ENUM if cap is not one of the values listed. HL_INVALID_OPERATION if no haptic rendering is current.

hlMakeCurrent

Description:	Makes a haptic rendering context current. The current rendering context is the target for all rendering and state commands. All haptic rendering commands will be sent to the device in the current context until a context with a different device is made current.
Syntax:	`void hlMakeCurrent (HHLRC hHLRC)` Argument: `hHLRC` - The handle to haptic rendering context returned by hlCreateContext().
Returns:	None
Usage:	Called at program startup after creating the rendering context or called during program execution to switch rendering contexts in order to render to multiple haptic devices.

Example:	`HDErrorInfo error;`
	`hHD = hdInitDevice(HD_DEFAULT_DEVICE);`
	`if (HD_DEVICE_ERROR(error = hdGetError()))`
	`{`
	`hduPrintError(stderr, &error, "Failed to initialize haptic device");`
	`exit(1);`
	`}`
	`hHLRC = hlCreateContext(hHD);`
	`hlMakeCurrent(hHLRC);`

Errors:	None
See also:	hlCreateContext, hlDeleteContext

hlCacheGetBooleanv, hlCacheGetDoublev

Description:

These functions allow querying of the different state values from a cached version of the state of the haptic renderer. Cached renderer states are passed into event and effect callback functions and contain the renderer state relevant for use by the callback function. The cache is important due to the multi-threaded implementation of the haptic renderer. The state may change between the time the event occurs and the time the callback is called.

Syntax:

```
void  hlCacheGetBooleanv(HLcache* cache, HLenum pname, HLboolean
*params)
void  hlCacheGetDoublev(HLcache* cache, HLenum pname, HLdouble *params)
```

Argument: `cache` - state cache to query

Argument: `pname` - Name of parameter (state value) to query. For a list of possible pname values, see "Table B-2: hlCacheGetBooleanv, hlCacheGetDoublev" on page 65.

Argument: `params` - Address at which to return the value of the parameter being queried.

Returns:

None.

Usage:

Queries the value of the state of the haptics renderer as stored in the state cache. Note that cached state is a subset of the full renderer state and therefore the possible arguments to the cached state query functions are a subset of those for the non-cached versions. The HLcache container is used by events and custom force effects. All positions, vectors and transforms accessible through hlCacheGet*() are provided in world coordinates for events and workspace coordinates for custom force effects.

Example:

```
void onClickSphere(HLenum event, HLuint object, HLenum thread,

HLcache *cache, void *userdata)
{
hlDouble proxyPos[3];
hlCacheGetDoublev(cache, HL_PROXY_POSITION, proxyPos);
hlBoolean switchDown;
hlCacheGetBooleanv(cache, HL_BUTTON1_STATE, &switchDown);
}
```

Errors:

HL_INVALID_ENUM if param is not one of the values listed.
HL_INVALID_OPERATION if no haptic rendering context is active.

See also:

hlGetBooleanv, hlGetDoublev, hlGetIntegerv, hlAddEventCallback

hlBeginShape

Description:	Indicates that subsequent geometry commands will be sent to the haptic renderer as part of the shape indicated until hlEndShape() is called. Geometric primitives may only be sent to the haptic renderer if they are specified inside an hlBeginShape()/hlEndShape() block. Grouping geometric primitives is important for two reasons: 1. Event callbacks will report the shape id of geometric primitives touched, and 2. Correct haptic rendering of geometric primitives that are dynamically moving is accomplished by looking at frame to frame differences based on shape id.
Syntax:	`void hlBeginShape(HLenum type, HLuint shape)` Argument: `type` - Type of shape to specify. For a list of supported shapes types, see "Table B-6: hlBegin Shape" on page 66. Argument: `shape` - The id of shape to specify returned from an earlier call to hlGenShapes().
Returns:	None.
Usage:	Used before specifying geometry and callback functions to the haptic renderer.

Example:	`hlBeginShape(HL_SHAPE_FEEDBACK_BUFFER, myShapeId);`
	`glBegin(GL_TRIANGLES);`
	`glVertex3f(-1,-1,0);`
	`glVertex3f(-1,1,0);`
	`glVertex3f(1,1,0);`
	`glVertex3f(1,-1,0);`
	`glEnd();`
	`hlEndShape();`

Errors:	HL_INVALID_ENUM if the type is not one of the values listed. HL_INVALID_OPERATION if not inside an hlBeginFrame()/hlEndFrame() block or if already inside an hlBeginShape()/hlEndShape() block.
See also:	hlEndShape, hlHinti, hlHintb, hlCallback

hlDeleteShape

Description:	Deallocates unique identifiers created by hlGenShapes.
Syntax:	`void hlDeleteShapes(HLuint shape, HLsizei range)` Argument: `shape` - ID of first shape to delete. Argument: `range` - Number of consecutive unique identifiers to generate.
Returns:	None.
Usage:	Deletes all consecutive shape identifiers in the range [shape, shape+range-1].

Example:	`HLuint myShapeId = hlGenShapes(1);`
	…
	`hlDeleteShapes(myShapeId, 1);`

Errors:	HL_INVALID_VALUE if any of the identifiers to deallocate were not previously allocated by glGenShapes(). HL_INVALID_OPERATION if no haptic rendering context is active.
See also:	hlGenShapes, hlBeginShape, hlIsShape

hlEndShape

Description:	Completes the shape specified by the last call to hlBeginShape(). Geometry, materials, transforms and other state for the shape are captured and sent to the haptic renderer.
Syntax:	`void hlEndShape()`
Returns:	None.
Usage:	After a call to hlBeginShape().

Example:	`hlBeginShape(HL_SHAPE_FEEDBACK_BUFFER, myShapeId);`
	`glBegin(GL_TRIANGLES);`
	`glVertex3f(-1,-1,0);`
	`glVertex3f(-1,1,0);`
	`glVertex3f(1,1,0);`
	
	`glEnd();`
	`hlEndShape();`

Errors:	HL_INVALID_OPERATION if not inside an hlBeginFrame()/hlEndFrame() or an hlBeginShape()/hlEndShape() block.
See also:	hlBeginShape

hlGenShapes

Description:	For shapes, generates a unique identifier that may be used with hlBeginShape().
Syntax:	`HLuint hlGenShapes(HLsizei range)` Argument: `range` - Number of unique identifiers to generate.
Returns:	A unique integer that may be used as an identifier for a shape. If the range is greater than one, the return value represents the first of a series of range consecutive unique identifiers.
Usage:	Before a call to hlBeginShape() to create a unique identifier for the new shape.

Example:	`HLuint myShapeId = hlGenShapes(1);`
	`hlBeginShape(HL_SHAPE_FEEDBACK_BUFFER, myShapeId);`
	…
	`hlEndShape();`

Errors:	HL_INVALID_OPERATION if no haptic rendering context is active.
See also:	hlBeginShape, hlDeleteShapes, hlIsShape

hlLocalFeature

Description: These functions specify local feature geometry to the haptic renderer.

Syntax:

```
void hlLocalFeature1fv(HLgeom *geom, HLenum type,
                       const HLfloat *v)
void hlLocalFeature1dv(HLgeom *geom, HLenum type,
                       const HLdouble *v)
void hlLocalFeature2fv(HLgeom *geom, HLenum type,
                       const HLfloat *v1,
                       const HLfloat *v2)
void hlLocalFeature2dv(HLgeom *geom, HLenum type,
                       const HLdouble *v1,
                       const HLdouble *v2)
```

Argument: `geom` - Container for local features passed in as parameter to callback shape closest features callback function.

Argument: `ctype` - Type of local feature to create. For a list of supported feature types, see "Table B-8: hlLocalFeature types" on page 67.

Argument: `v, v1, v2` - One or two vectors, given in the local coordinates of the shape, which define the local feature.

Returns: None

Usage:

Called from within the closest feature callback function (HLclosestFeaturesProc) of a callback shape. Used to specify the geometry of whatever local feature of the callback shape is closest to the proxy position. That local feature is then used by the haptic renderer when either rendering the callback shape using the constraint touch model or dynamically deforming the callback shape.

In terms of threading, the haptic renderer calls the closest features callback in the collision thread and stores the local feature that is specified by that callback. The renderer then uses that local features in the servo thread to constrain the proxy and generate forces.

The example below implements a closest feature callback function for a sphere. A sphere can be represented locally as a plane tangent to a point of interest on it. So our callback function finds the closest point on the sphere to the proxy and then generates a plane tangent to the sphere at that point. This might then be used by the servo thread to constrain the proxy to the surface of the sphere.

```
// find the closest surface feature(s) to queryPt for sphere
// callback shape
bool closestSurfaceFeatures(const HLdouble queryPt[3],
                            const HLdouble targetPt[3],
                            HLgeom *geom,
                            void *userdata)
{
   HapticSphere *pThis = static_cast<HapticSphere *>
                         (userdata);
```

Example:

```
// Return a plane tangent to the sphere as the closest
// 2D local feature
   hduVector3Dd normal(queryPt);
   normal.normalize();

   hduVector3Dd point = normal * pThis->getRadius();

   hlLocalFeature2dv(geom, HL_LOCAL_FEATURE_PLANE, normal, point);

   return true;
}
```

Errors: HL_INVALID_ENUM if ctype is not one of the values listed.

See also: hlBeginShape, hlEndShape

hlIsShape

Description: Determines if an identifier is a valid shape identifier.

Syntax:

```
HLboolean hlIsShape(HLuint shape)
```

Argument: `shape` - Identifier of shape.

Returns: Returns HL_TRUE if the shape identifier is a valid allocated value.

Usage: Determines if an identifier is a valid shape identifier.

Example:

```
HLuint myShapeId = hlGenShapes(1);
assert(hlIsShape(myShapeId));
```

Errors: HL_INVALID_OPERATION if called within an hlBeginShape()/ hlEndShape() block.

See also: hlGenShapes, hlBeginShape, hlIsShape

hlGetShapeBooleanv, hlGetShapeDoublev

Description: These functions allow querying of the state of specific shapes.

Syntax:

```
void hlGetShapeBooleanv(HLuint shapeId, HLenum pname, HLboolean
*params)
```

```
void hlGetShapeDoublev(HLuint shapeId, HLenum pname, HLdouble *params)
```

Argument: `geom` - Container for local features passed in as parameter to callback shape closest features callback function.

Argument: `shapeid` - Identifier of shape to query.
Argument: `pname` - Name of parameter (state value) to query.
Argument: `params` - Address at which to return the value of the parameter being queried. For a list of supported shapes types, see “Shape Parameters” on page 66.

Returns: None

Usage:	Queries the state of the shape with identifier shapeid from the haptic rendering engine. This shape must have been rendered during the last frame.

Example:	`HLboolean isTouching;`
	`hlGetShapeBooleanv(myShapeId, HL_PROXY_IS_TOUCHING, &isTouching);`
	`if (isTouching)`
	`{`
	`hduVector3Dd force;`
	`hlGetShapeDoublev(myShapeId, HL_REACTION_FORCE, force);`
	`}`

Errors:	HL_INVALID_ENUM if param is not one of the values listed. HL_INVALID_OPERATION if no haptic rendering context is active.
See also:	hlBeginShape, hlEndShape, hlGetBooleanv, hlGetDoublev, hlGetIntegerv

hlGetMaterialfv

Description:	Gets the current haptic material properties for shapes.
Syntax:	`void hlGetMaterialfv(HLenum face, HLenum pname, HLfloat *param)` Argument: `face` - Face(s) to apply this material to. For a list of the supported face values, see "Table B-12: hlGetMaterialfv - face values" on page 68. Argument: `pname` - Material property to set. For a list of the supported values for pname, see "Table B-11: hlMaterialf - pname values" on page 68. Argument: `paramfac` - New value for material property.
Returns:	None.
Usage:	Used to find the material properties that will be used for rendering shapes.
Example:	`hlFloat stiffness;` `hlGetMaterialfv(HL_FRONT, HL_STIFFNESS, &stiffness);`
Errors:	HL_INVALID_ENUM if the face or pname arguments are not one of the values listed. HL_INVALID_OPERATION if no haptic rendering context is current.
See also:	hlMaterialf, hlPushAttrib, hlPopAttrib

hlMaterialf

Description:	Sets haptic material properties for shapes. Material properties can be set independently for front and back facing triangles of contact shapes. Only front face materials apply to constraints.
Syntax:	`void hlMaterialf(HLenum face, HLenum pname, HLfloat param)` Argument: `face` - Face(s) to apply this material to. For a list of the supported face values, see "Table B-10: hlMaterialf - face values" on page 68. Argument: `pname` - Material property to set. For a list of the supported values for pname, see "Table B-11: hlMaterialf - pname values" on page 68. Argument: `param` - New value for material property.
Returns:	None.
Usage:	Used before defining shapes to set their materials.

Example:	`hlMaterialf(HL_FRONT_AND_BACK, HL_STIFFNESS, 0.8);`
	`hlMaterialf(HL_FRONT_AND_BACK, HL_DAMPING, 0.6);`
	`hlMaterialf(HL_FRONT_AND_BACK, HL_STATIC_FRICTION, 0.5);`
	`hlMaterialf(HL_FRONT_AND_BACK, HL_DYNAMIC_FRICTION, 0.4);`

Errors:	HL_INVALID_ENUM if the face or pname arguments are not one of the values listed. HL_INVALID_OPERATION if no haptic rendering context is current. HL_INVALID_VALUE if param is not between 0 and 1.
See also:	hlGetMaterialfv, hlPushAttrib, hlPopAttrib

hlTouchableFace

Description:	Sets which faces of shapes will be touchable by the haptic device.
Syntax:	`void hlTouchableFace(HLenum mode)` Argument: `mode` - Which face(s) of shapes to make touchable. For a list of mode values, see "Table B-14: hlTouchableFace - mode values" on page 69.
Returns:	None.
Usage:	Shapes may be touchable on one or both sides. Use this function to set which of those sides is touchable. The front side of feedback buffer and depth buffer shapes is defined by the winding order of the vertices as set in OpenGL; that is, triangles considered front facing by OpenGL will also be considered as front facing by HL. When using the HL_CONSTRAINT touch model, all shapes are always touchable from both sides, independent of the touchable face.
Example:	`hlTouchableFace(HL_FRONT);`
Errors:	HL_INVALID_ENUM if the mode is not one of the values listed. HL_INVALID_OPERATION if no haptic rendering context is active.
See also:	hlTouchModel

hlTouchModel

Description:	Sets the touch model to specify shapes as contact shapes or constraints.
Syntax:	`void hlTouchModel(HLenum mode)` Argument: `mode` - Contact or constraint model. For a list of supported mode values, see "Table B-15: hlTouchModel" on page 69.
Returns:	None
Usage:	Used before specifying a shape to set whether it is a contact shape that resists penetration or whether it is a constraint that will force the haptic device to the surface of the shape.
Example:	`hlTouchModel(HL_CONTACT);`
Errors:	HL_INVALID_ENUM if mode is not one of the values listed. HL_INVALID_OPERATION if no haptic rendering context is current.
See also:	hlTouchModelf, hlPushAttrib, hlPopAttrib

hlTouchModelf

Description:	Sets properties of the touch model.
Syntax:	`void hlTouchModelf(HLenum pname, HLfloat param)` Argument: `pname` - Touch model parameter to modify. For a list of supported values for pname, see "Table B-16: hlTouchModelIf" on page 69. Argument: `param` - New value for the parameter.
Returns:	None
Usage:	Used before specifying a shape to set the parameters used by the touch model for that shape.
Example:	`hlTouchModelf(HL_SNAP_DISTANCE, 1.5);`
Errors:	HL_INVALID_ENUM if pname is not one of the values listed. HL_INVALID_OPERATION if no haptic rendering context is current.
See also:	hlTouchModel, hlPushAttrib, hlPopAttrib

hlDeleteEffects

Description:	For effect, deallocates unique identifiers created by hlGenEffects().
Syntax:	`void hlDeleteEffects(HLuint effect, HLsizei range)` Argument: `effect` - Identifier of first effect to delete. Argument: `range` - Number of consecutive unique identifiers to generate.
Returns:	None.
Usage:	Deletes all consecutive effect identifiers starting in the range [effect, effect+range-1].

Example:	`HLuint myEffectId = hlGenEffects(1);`
	…
	`hlDeleteEffects(myEffectId, 1);`

Errors:	HL_INVALID_VALUE if any of the effect identifiers to deallocate were not previously allocated by glGenEffects() HL_INVALID_OPERATION if no haptic rendering context is active.
See also:	hlGenEffects, hlStartEffect, hlStopEffect, hlIsEffect

hlEffectd, hlEffecti, hlEffectdv, hlEffectiv

Description:	Sets the current value of an effect property.

Syntax:	`void hlEffectd (HLenum pname, HLdouble param)`
	`void hlEffecti (HLenum pname, HLint param)`
	`void hlEffectdv (HLenum pname, const HLdouble *params)`
	`void hlEffectiv (HLenum pname, const HLint *params)`
	Argument: `pname` - The name of parameter to set. For a list of possible names, see "Table B-19: hlEffectd, hlEffecti, hlEffectdv, hlEffectiv" on page 70.
	Argument: `params` - The new value of the property.

Returns:	None.
Usage:	Sets the value of an effect property which will be applied to the effect generated by the next call to hlStartEffect() or hlTriggerEffect().

Example:	`hlBeginFrame();`
	`hlEffectd(HL_EFFECT_PROPERTY_DURATION, pulseLength);`
	`hlEffectd(HL_EFFECT_PROPERTY_GAIN, 0.4);`
	`hlEffectd(HL_EFFECT_PROPERTY_MAGNITUDE, 0.4);`
	`hlTriggerEffect(HL_EFFECT_FRICTION);`
	`hlEndFrame();`

Errors:	HL_INVALID_ENUM if the type is not one of the values listed. HL_INVALID_OPERATION if not inside an hlBeginFrame()/hlEndFrame() block or if inside an hlBeginShape()/hlEndShape() block.
See also:	hlStartEffect, hlTriggerEffect, hlStartEffect, hlEffectd, hlEffecti, hlEffectdv, hlEffectiv, hlCallback, hlPushAttrib, hlPopAttrib

hlGenEffects

Description:	Generates unique identifiers for effects that may be used with hlStartEffect().
Syntax:	`HLuint hlGenEffects(HLsizei range)` Argument: `range` - Number of unique identifiers to generate.
Returns:	A unique integer that may be used as an identifier for an effect. If the range argument is greater than one, the return value represents the first of a series of range consecutive unique identifiers.
Usage:	Before a call to hlStartEffect() to create a unique identifier for the new effect.

Example:	`HLUintfriction = hlGenEffects(1);`
	`hlBeginFrame();`
	`hlEffectd(HL_EFFECT_PROPERTY_GAIN, 0.4);`
	`hlEffectd(HL_EFFECT_PROPERTY_MAGNITUDE, 0.4);`
	`hlStartEffect(HL_EFFECT_FRICTION, friction);`
	`hlEndFrame();`

Errors:	HL_INVALID_OPERATION if no haptic rendering context is active or if inside an hlBeginShape()/ hlEndShape() block.
See also:	hlStartEffect, hlStopEffect, hlDeleteEffects, hlIsEffect

hlGetEffectdv, hlGetEffectiv, hlGetEffectbv

Description:	Queries the current value of an effect property.

Syntax:	`void hlGetEffectdv (HLuint effect, HLenum pname,`
	`HLdouble *param)`
	`void hlGetEffectiv (HLuint effect, HLenum pname,`
	`HLint *param)`
	`void hlGetEffectbv (HLuint effect, HLenum pname,`
	`HLboolean *params)`
	Argument: `effect` - Identifier of effect to query.
	Argument: `pname` - Name of parameter to query. For a list of supported name values, see "Table B-20: hlEffectd, hlEffecti, hlEffectdv, hlEffectiv" on page 70.
	Argument: `params` - Receives the value of the property.

Returns:	None
Usage:	Gets the value of an effect property which will be applied to the effect generated by the next call to hlStartEffect() or hlTriggerEffect().

Example:	`hlBeginFrame();`
	`hlEffectd(HL_EFFECT_PROPERTY_GAIN, 0.4);`
	`hlStartEffect(myEffect, HL_EFFECT_FRICTION);`
	`hlEndFrame();`
	
	`HLdouble gain;`
	`hlGetEffectdv(myEffect, HL_EFFECT_PROPERTY_GAIN, &gain);`
	`assert(gain == 0.4);`

Errors:	HL_INVALID_ENUM if the pname is not one of the values listed. HL_INVALID_OPERATION if no haptic rendering context is active.
See also:	hlStartEffect, hlTriggerEffect, hlEffectd, hlEffecti, hlEffectdv, hlEffectiv

hlIsEffect

Description:	Determine if an identifier is a valid effect identifier.
Syntax:	`HLboolean hlIsEffect(HLuint effect)` Argument: `effect` - Identifier of first effect to check.
Returns:	None
Usage:	Returns true if effect is a valid identifier that was previously returned by hlGenEffects().

Example:	`HLuint myEffectId = hlGenEffects(1);`
	…
	`assert(hlIsEffect(myEffectId);`

Errors:	HL_INVALID_OPERATION if no haptic rendering context is active.
See also:	hlGenEffects

hlStartEffect

Description:	Starts an effect that will continue to run until it is terminated by a call to hlStopEffect().
Syntax:	`void hlStartEffect (HLenum type, HLuint effect)` Argument: `type` - Type of effect to start. For a list of supported types, see "Table B-17: hlStartEffect - effect types" on page 69. Argument: `effect` - Identifier of effect to start, generated by a call to hlGenEffects().
Returns:	None
Usage:	Starting an effect will cause it to continue to run until hlStopEffect() is called to terminate it. When using hlStartEffect(), the duration property is ignored.

Example:	`hlBeginFrame();`
	`hlEffectd(HL_EFFECT_PROPERTY_GAIN, 0.4);`
	`hlEffectd(HL_EFFECT_PROPERTY_MAGNITUDE, 0.4);`
	`hlStartEffect(HL_EFFECT_FRICTION);`
	`hlEndFrame();`

Errors:	HL_INVALID_ENUM if the type is not one of the values listed. HL_INVALID_OPERATION if not inside an hlBeginFrame()/hlEndFrame() block or if inside an hlBeginShape()/hlEndShape() block.
See also:	hlStopEffect, hlEffectd, hlEffecti, hlEffectdv, hlEffectiv, hlTriggerEffect, hlCallback

hlStopEffect

Description:	Stops an effect that was started with hlStartEffect().
Syntax:	`void hlStopEffect (HLuint effect);` Argument: `effect` - Identifier of effect to start, generated by a call to hlGenEffects().
Returns:	None
Usage:	Once an effect has been started by a call hlStartEffect(), it will continue running until hlStopEffect() is called.

Example:	`hlBeginFrame();`
	`hlStopEffect(friction);`
	`hlEndFrame();`

Errors:	HL_INVALID_OPERATION if not inside an hlBeginFrame()/hlEndFrame() block or if inside an hlBeginShape()/hlEndShape() block.
See also:	hlStartEffect, hlEffectd, hlEffectdv, hlEffecti, hlEffectiv, hlTriggerEffect, hlCallback

hlTriggerEffect

Description:	Starts an effect which will continue to run for a specified duration.
Syntax:	`void hlTriggerEffect (HLenum type);` Argument: `type` - Type of effect to start. For a list of support value types, see "Table B-18: hlTriggerEffect" on page 70.
Returns:	None
Usage:	Triggering an effect will cause it to continue to run for the amount of time specified by the current effect duration. Unlike effects started with hlStartEffect(), those started with hlTriggerEffect() do not need to be terminated with a call to hlStopEffect(). The effect will be terminated automatically when the time elapsed since the call to hlTriggerEffect() becomes greater than the effect duration. The effect duration will be the value specified by the last call to hlEffectd() with HL_EFFECT_PROPERTY_DURATION.

Example:	`hlBeginFrame();`
	`hlEffectd(HL_EFFECT_PROPERTY_DURATION, pulseLength);`
	`hlEffectd(HL_EFFECT_PROPERTY_GAIN, 0.4);`
	`hlEffectd(HL_EFFECT_PROPERTY_MAGNITUDE, 0.4);`
	`hlTriggerEffect(HL_EFFECT_FRICTION);`
	`hlEndFrame();`

Errors:	HL_INVALID_ENUM if the type is not one of the values listed. HL_INVALID_OPERATION if not inside an hlBeginFrame()/hlEndFrame() block or if inside an hlBeginShape()/hlEndShape() block.
See also:	hlStartEffect, hlEffectd, hlEffecti, hlEffectdv, hlEffectiv, hlCallback

hlUpdateEffect

Description:	Updates the given active effect with the current effect state.
Syntax:	`hlUpdateEffect(HLuint effect)` Argument: `effect` - The id of the effect to be updated
Returns:	None
Usage:	hlUpdateEffect() is used for changing parameters of an effect that is currently active (has been started by hlStartEffect(). For example, the user may want to increase the force of a spring effect each time a button is pressed; hlUpdateEffect() allows the effect to be changed in place without the user stopping the effect, specifying new parameters, and starting it again. The effect is updated with whatever effect properties are in the current state, just as if the effect was specified via hlStartEffect().

Example:	`hlStartEffect(HL_EFFECT_SPRING, springEffect);`
	`...`
	`hlEffectd(HL_EFFECT_PROPERTY_GAIN, 0.1);`
	`hlUpdateEffect(springEffect);`

Errors:	HL_INVALID_VALUE if effect is not an active effect.
See also:	hlStartEffect, hlStopEffect

hlDeleteEffects

Description:	Sets properties of the proxy.
Syntax:	`void hlProxydv (HLenum pname, const HLdouble *params)` Argument: `pname` - Name of parameter to set. For a list of supported pnames, see "Table B-25: hlProxydv" on page 72. Argument: `param` - Value of the property to set.
Returns:	None.
Usage:	Sets properties of the proxy. The proxy is a virtual object that follows the haptic device but is constrained by geometric primitives. The force sent to the haptic device is based in part on the relative positions of the haptic device and proxy. When proxy resolution is enabled, the haptic rendering engine updates the proxy transform automatically as the haptic device moves and geometric primitives change. Users may disable automatic proxy resolution (see hlDisable) and set the proxy transform directly in order to use their own algorithms for proxy resolution.

Example:	`hlBeginFrame();`
	`hlDisable(HL_PROXY_RESOLUTION);`
	`HLdouble newProxyPos[] = {3, 4, 5};`
	`hlProxydv(HL_PROXY_POSITION, newProxyPos);`
	`hlEndFrame();`

Errors:	HL_INVALID_ENUM if the pname is not one of the values listed. HL_INVALID_OPERATION if no haptic rendering context is active or if not within an hlBeginFrame()/hlEndFrame() block.
See also:	hlEnable, hlDisable, hlGetBooleanv, hlGetDoublev, hlGetIntegerv

hlLoadIdentity

Description: Replaces the current matrix on the top of the current matrix stack with the identity matrix.

Syntax:
```
void hlLoadIdentity(void)
```

Returns: None.

Usage: Clears the top of the current matrix and replaces it with the identity matrix:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

This command applies to either the touchworkspace, viewtouch matrix or modelview depending on the current matrix mode.

Example:
```
hlMatrixMode(HL_TOUCHWORKSPACE);
hlPushMatrix(); // save off old touchworkspace matrix
hlLoadIdentity(); // set touchworkspace to identity
hlScaled(wsscale, wsscale, wsscale); // set scale
// render some stuff

…

hlPopMatrix(); // restore old touchworkspace matrix
```

Errors: HL_INVALID_OPERATION if no haptic rendering context is active.

See also: hlMatrixMode, hlPushMatrix, hlPopMatrix, hlTranslated, hlTranslatef, hlRotated, hlRotatef, hlScaled, hlScalef, hlLoadMatrixd, hlLoadMatrixf, hlMultMatrixd, hlMultMatrixf, hlGetBooleanv, hlGetDoublev, hlGetIntegerv

hlLoadMatrixd, hlLoadMatrixf

Description: Replaces the current matrix on the top of the current matrix stack with the 4x4 matrix specified.

Syntax:
```
void hlLoadMatrixd(const HLdouble *m)
void hlLoadMatrixf(const HLfloat *m)
```

Argument: `m` - An array of 16 floating points or double precision values representing a 4x4 transform matrix.

Returns: None.

Usage: Clears the top of the current matrix and replaces it with the 4x4 matrix constructed from the values in m as follows:

$$\begin{pmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{pmatrix}$$

This command applies to either the touchworkspace, viewtouch matrix or modelview depending on the current matrix mode.

Example:
```
HLdouble myShapeXfm[16];

…

hlPushMatrix(); // save off old matrix
hlLoadMatrixd(myShapeXfm); // set xfm
// render some stuff

…

hlPopMatrix(); // restore old matrix
```

Errors: HL_INVALID_OPERATION if no haptic rendering context is active.

See also:	hlMatrixMode, hlPushMatrix, hlPopMatrix, hlTranslated, hlTranslatef, hlRotated, hlRotatef, hlScaled, hlScalef, hlLoadIdentity, hlMultMatrixd, hlMultMatrixf, hlGetBooleanv, hlGetDoublev, hlGetIntegerv

hlMultMatrixd, hlMultMatrixf

Description:	Multiplies the current matrix on the top of the current matrix stack with the 4x4 matrix specified.

Syntax:	`void hlMultMatrixd(const HLdouble *m)`
	`void hlMultMatrixf(const HLfloat *m)`
	Argument: `m` - An array of 16 floating points or double precision values representing a 4x4 transform matrix.

Returns:	None
Usage:	Replaces the top of the current matrix stack with the product of the top of the stack and the matrix constructed from the values in m as follows::  This command applies to either the touchworkspace, viewtouch matrix or modelview depending on the current matrix mode.

Example:	`HLdouble myShapeXfm[16];`
	…
	`hlPushMatrix(); // save off old matrix`
	`hlMultMatrixd(myShapeXfm); // set xfm`
	`// render some stuff`
	…
	`hlPopMatrix(); // restore old matrix`

Errors:	HL_INVALID_OPERATION if no haptic rendering context is active.
See also:	hlMatrixMode, hlPushMatrix, hlPopMatrix, hlTranslated, hlTranslatef, hlRotated, hlRotatef, hlScaled, hlScalef, hlLoadIdentity, hlLoadMatrixd, hlLoadMatrixf, hlGetBooleanv, hlGetDoublev, hlGetIntegerv

hlMatrixMode

Description:	Sets which matrix stack is the target for future calls to matrix manipulation commands.
Syntax:	`void hlMatrixMode(HLenum mode);` Argument: `mode` - Which matrix stack to apply matrix manipulation commands to. See "Table B-26: hlMatrix - mode values" on page 72 for a list of valid modes.
Returns:	None
Usage:	The HLAPI maintains two transforms that, when multiplied together, determine the mapping from the local coordinate system of the haptic device (workspace coordinates) to graphical view coordinates. The first transform maps from touch coordinates to workspace coordinates and the second maps from view coordinates to touch coordinates. The API maintains a stack of matrices for each of these of two transforms where the top of the stack represents the current matrix to use for the transform. HLAPI also maintains an optional modelview matrix stack. The modelview maps from model coordinates to view coordinates. By default, HLAPI ignores this stack and instead uses the current OpenGL modelview matrix. This allows easier reuse of OpenGL code by applying OpenGL transformations to geometry for haptic rendering. In some cases you may not want to use the OpenGL modelview matrix (for example, if you have no OpenGL rendering context.) In this case you can disable the use of the OpenGL modelview matrix by calling hlDisable(HL_USE_GL_MODELVIEW) and HLAPI will use its own modelview matrix instead. All matrix manipulation commands target the top matrix on one of these two stacks. Setting the matrix mode controls which of the three stacks is targeted.

Example:

```
hlMatrixMode(HL_TOUCHWORKSPACE);
hlPushMatrix(); // save off old touchworkspace matrix
hlLoadIdentity(); // set touchworkspace to identity
hlScaled(wsscale, wsscale, wsscale); // set scale
// render some stuff
…
hlPopMatrix(); // restore old touchworkspace matrix
```

Errors: HL_INVALID_ENUM if the mode is not one of the values listed.
HL_INVALID_OPERATION if no haptic rendering context is active.

See also: hlPushMatrix, hlPopMatrix, hlTranslatef, hlTranslated, hlRotatef, hlRotated, hlScalef, hlScaled, hlWorkspace, hlLoadMatrixd, hlLoadMatrixf hlMultMatrixd, hlMultMatrixf, hlGetBooleanv, hlGetDoublev, hlGetIntegerv

hlOrtho

Description: Sets up the haptic view volume which determines how the workspace of the haptic device will be mapped into the graphical view volume.

Syntax:

```
void hlOrtho (HLdouble left, HLdouble right,
HLdouble bottom, HLdouble top,
HLdouble zNear, HLdouble zFar);
```

Argument: `left, right` - The coordinates of the left and right boundaries of the haptic view volume.
Argument: `top, bottom` - The coordinates of the top and bottom boundaries of the haptic view volume.
Argument: `zNear, zFar` - The coordinates of the front and back boundaries of the haptic view volume.

Returns: None

Usage: The hlOrtho() function is used in conjunction with hlWorkspace() to determine the mapping between the physical workspace of the haptic device and the graphical view. hlOrtho() defines an orthogonal view volume (that is an axis oriented bounding box) in the graphical view coordinate system. hlWorkspace() defines a corresponding box in the physical coordinates of the haptic device. The haptic rendering engine uses these to determine a transformation between the two coordinate spaces. Specifically, the API creates a transform consisting of a uniform or non-uniform scale about the center of the workspace box and a translation. The product of this transform and the top of the matrix stack replaces the current top of the matrix stack. This function is generally used with the

HL_TOUCHWORKSPACE matrix mode.

Example:

```
hlMatrixMode(HL_TOUCHWORKSPACE);

// clear the matrix stack
hlLoadIdentity();

// specify the boundaries for the workspace of the haptic
// device in millimeters in the cordiantes of the haptic
// device the haptics engine will map the view volume to
// this workspace
hlWorkspace (-80, -80, -70, 80, 80, 20);
// specify the haptic view volume
hlOrtho (0.0, 1.0, 0.0, 1.0, -1.0, 1.0);
```

Errors: HL_INVALID_OPERATION if no haptic rendering context is active.

See also: hlWorkspace, hlMatrixMode

hlPushAttrib, hlPopAttrib

Description:	hlPushAttrib() pushes a set of attributes onto the top of the current attribute matrix stack. hlPopAttrib() removes the top of the current attribute stack.

Syntax:	`void hlPushAttrib(HLbitfield mask)`
	`void hlPopAttrib()`
	Argument: `mask` - The type of attributes to push. See "Table B-27: hlPushAttrib, hlPopAttrib" on page 72 for a list of attribute bits.

Returns:	None
Usage:	hlPush()/PopAttrib() allows the user to save and restore rendering state attributes. It works similarly to hlPush()/PopMatrix(), except that it allows the user to selectively push attributes onto the stack.
Example:	`hlPushAttrib(HL_HINT_BIT | HL_TRANSFORM_BIT);`
Errors:	HL_INVALID_VALUE if mask is not an attribute bit or set of attribute bits.
See also:	hlPushMatrix, hlPopMatrix

hlPushMatrix, hlPopMatrix

Description:	hlPushMatrix() pushes a new matrix onto the top of the current matrix stack. hlPopMatrix() removes the top of the current matrix stack.
Syntax:	`void hlPushMatrix()` `void hlPopMatrix()`
Returns:	None
Usage:	The HLAPI maintains matrix stacks for the viewtouch matrix, the touchworkspace matrix, and optionally for the modelview matrix. hlPushMatrix() and hlPopMatrix() allow these matrix stacks to be changed temporarily and then restored. hlPushMatrix() pushes a new matrix onto the top of the current matrix stack. The new matrix is initially a duplicate of the existing matrix on the top of the stack. hlPopMatrix() removes the matrix on the top of the stack, leaving the old top of the stack as the current matrix. This command applies to either the touchworkspace, viewtouch matrix, or modelview depending on the current matrix mode.

Example:	`hlPushMatrix(); // save off old matrix`
	`hlTranslate(0, 20, 0); // move geometry up 20`
	`// draw some stuff`
	`hlPopMatrix(); // restore old matrix`

Errors:	HL_INVALID_OPERATION if no haptic rendering context is active. HL_STACK_OVERFLOW if hlPushMatrix() is called when the matrix stack is already full. HL_STACK_UNDERFLOW if hlPopMatrix() is called when the matrix stack is only one deep.
See also:	hlMatrixMode, hlTranslated, hlTranslatef, hlRotated, hlRotatef, hlScaled, hlScalef, hlLoadIdentity, hlLoadMatrixd, hlLoadMatrixf, hlMultMatrixd, hlMultMatrixf, hlGetBooleanv, hlGetDoublev, hlGetIntegerv, hlPushAttrib, hlPopAttrib

hlRotatef, hlRotated

Description: Multiplies the current matrix on the top of the current matrix stack by a 4x4 rotation matrix.

Syntax:

```
void hlRotated (HLdouble angle, HLdouble x, HLdouble y,
HLdouble z)
void hlRotatef (HLfloat angle, HLfloat x,
HLfloat y,HLfloatz)
```

Argument: `angle` - Angle, in degrees, to rotate.
Argument: `x, y, z` - Coordinates of a vector representing the axis about which to rotate.

Returns: None

Usage:

Creates a 4x4 rotation matrix that represents a rotation of angle degrees about the given axis. Replaces the top of the current matrix stack with the product of the top of the matrix stack and the rotation matrix. The rotation matrix created is of the form:

$$\begin{pmatrix} x^2+\cos(angle)(1-x^2) & xy[1-\cos(angle)]-z\sin(angle) & zx[1-\cos(angle)]+y\sin(angle) & 0 \\ xy[1-\cos(angle)]+z\sin(angle) & y^2+\cos(angle)(1-y^2) & yz[1-\cos(angle)]-x\sin(angle) & 0 \\ zx[1-\cos(angle)]-y\sin(angle) & yz[1-\cos(angle)]+x\sin(angle) & z^2+\cos(angle)(1-z^2) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

This command applies to either the touchworkspace, viewtouch matrix, or modelview matrix depending on the current matrix mode.

Example:

```
hlPushMatrix(); // save off old matrix
hlRotate(45, 0, 0, 1); // rotate 45 degrees about z axis
// draw some stuff
hlPopMatrix(); // restore old matrix
```

Errors: HL_INVALID_OPERATION if no haptic rendering context is active.

See also: hlMatrixMode, hlPushMatrix, hlPopMatrix, hlTranslated, hlTranslatef, hlScaled, hlScalef, hlLoadIdentity, hlLoadMatrixd, hlLoadMatrixf, hlMultMatrixd, hlMultMatrixf, hlGetBooleanv, hlGetDoublev, hlGetIntegerv

hlScalef, hlScaled

Description: Multiplies the current matrix on the top of the current matrix stack by a 4x4 scale matrix.

Syntax:

```
void hlScaled(HLdouble x, HLdouble y, HLdouble z);
void hlScalef(HLfloat x, HLfloat y, HLfloat z);
```

Argument: `x, y, z` - Scale factors about the x, y and z axes respectively.

Returns: None

Usage:

Creates a 4x4 scale matrix that scales about the three coordinate axes. Replaces the top of the current matrix stack with the product of the top of the matrix stack and the scale matrix. The scale matrix created is of the form:

$$\begin{pmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

This command applies to either the touchworkspace, viewtouch matrix, or modelview matrix depending on the current matrix mode.

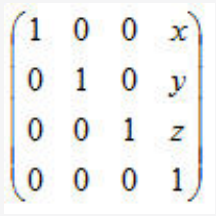
Example:	`hlPushMatrix(); // save off old matrix`
	`hlScale(1, 2, 1); // scale 2x along y axis`
	`// draw some stuff`
	`hlPopMatrix(); // restore old matrix`

Errors:	HL_INVALID_OPERATION if no haptic rendering context is active.
See also:	hlMatrixMode, hlPushMatrix, hlPopMatrix, hlTranslated, hlTranslatef, hlRotated, hlRotatef, hlLoadIdentity, hlLoadMatrixd, hlLoadMatrixf, hlMultMatrixd, hlMultMatrixf, hlGetBooleanv, hlGetDoublev, hlGetIntegerv

hlTranslatef, hlTranslated

Description:	Multiplies the current matrix on the top of the current matrix stack by a 4x4 translation matrix.

Syntax:	`void hlTranslated (HLdouble x, HLdouble y, HLdouble z)`
	`void hlTranslatef (HLfloat x, HLfloat y, HLfloat z)`
	Argument: `x, y, z` - Coordinates of translation vector.

Returns:	None
Usage:	Creates a 4x4 translation matrix based on the vector (x, y, z) and replaces the top of the current matrix stack with the product of the top of the matrix stack and the translation matrix. The translation matrix created is of the form::  This command applies to either the touchworkspace, viewtouch matrix, or modelview matrix depending on the current matrix mode.

Example:	`hlPushMatrix(); // save off old matrix`
	`hlTranslate(0, 20, 0); // move geometry up 20`
	`// draw some stuff`
	`hlPopMatrix(); // restore old matrix`

Errors:	HL_INVALID_OPERATION if no haptic rendering context is active.
See also:	hlMatrixMode, hlPushMatrix, hlPopMatrix, hlRotated, hlRotatef, hlScaled, hlScalef, hlLoadIdentity, hlLoadMatrixd, hlLoadMatrixf, hlMultMatrixd, hlMultMatrixf, hlGetBooleanv, hlGetDoublev, hlGetIntegerv

hlWorkspace

Description:	Defines the extents of the workspace of the haptic device that will be used for mapping between the graphics coordinate system and the physical units of the haptic device.
Syntax:	`void hlWorkspace (HLdouble left, HLdouble bottom, HLdouble back, HLdouble right, HLdouble top, HLdouble front);` Argument: `left, right` - The coordinates of the left and right boundaries of the haptic device workspace. Argument: `top, bottom` - The coordinates of the top and bottom boundaries of the haptic view volume. Argument: `from, back` - The coordinates of the front and back boundaries of the haptic view volume.
Returns:	None

Usage:	The hlWorkspace() function is used in conjunction with hlOrtho() to determine the mapping between the physical workspace of the haptic device and the graphical view. hlOrtho() defines an orthogonal view volume (that is, an axis oriented bounding box) in the graphical view coordinate system. hlWorkspace() defines a corresponding box in the physical coordinates of the haptic device. The haptic rendering engine uses these to determine a transformation between the two coordinate spaces. Specifically, the API creates a transform consisting of a uniform or non-uniform scale about the center of the workspace box and a translation. The product of this transform and the top of the matrix stack replaces the current top of the matrix stack. This function is generally used with the HL_TOUCHWORKSPACE matrix mode.

Example:	`hlMatrixMode(HL_TOUCHWORKSPACE);`
	
	`// clear the matrix stack`
	`hlLoadIdentity();`
	
	`// specify the boundaries for the workspace of the haptic`
	`// device in millimeters in the coordinates of the haptic`
	`// device the haptics engine will map the view volume to`
	`// this workspace`
	`hlWorkspace (-80, -80, -70, 80, 80, 20);`
	
	`// specify the haptic view volume`
	`hlOrtho (0.0, 1.0, 0.0, 1.0, -1.0, 1.0);`

Errors:	HL_INVALID_OPERATION if no haptic rendering context is active.
See also:	hlOrtho, hlMatrixMode

hlCallback

Description:	Sets a user callback function.
Syntax:	`void hlCallback (HLenum type, HLcallbackProc fn, void *userdata)` Argument: `type` - Name of callback to set. For a list of supported type values, see "Table B-21: hlCallback" on page 71. Argument: `fn` - Pointer to callback function. Argument: `userdata` - Pointer to client specific data that will be passed to the callback function.
Returns:	None.
Usage:	Used to set callback functions for custom shapes and custom effect types.

Example:	`hlBeginShape(HL_SHAPE_CALLBACK, myshape);`
	`hlCallback(HL_SHAPE_INTERSECT_LS, HLcallbackProc)`
	`&intersectSurface, (void*) &myShapeData);`
	`hlCallback(HL_SHAPE_CLOSEST_POINT, (HLcallbackProc)`
	`&closestPointSurface, (void*) &myShapeData);`
	`hlEndShape();`

Errors:	HL_INVALID_ENUM if type is not one of the values listed. HL_INVALID_OPERATION if no haptic rendering context is current.
See also:	hlBeginShape

hlAddEventCallback

Description:

Adds a user defined event handling function to the list of callback functions for an event.

Syntax:

```
void hlAddEventCallback (HLenum event, HLuint shapeId,
HLenum thread, HLeventProc fn,
void *userdata)
```

Argument: `event` - Event to which to subscribe. For a list of supported event values, see "Table B-22: hlAddEventCallback, hlRemoveEventCallBack - event values" on page 71.

Argument: `shapeID` - Identifier of shape. Callback will only be called if the event occurs on the shape with this identifier unless this argument is set HL_OBJECT_ANY in which case the callback will be called independent of any objects.

Argument: `thread` - Thread to have callback function called in. For a list of support thread values, see "Table B-23: hlAddEventCallback, hlRemoveEventCallback - thread values" on page 72.

Argument: `fn` - Pointer to callback function.

Argument: `userdata` - Pointer to client specific data that will be passed to the callback function.

Returns:

None.

Usage:

Event callbacks are used to inform programs about occurrences in the haptic renderer such as an object being touched.

Example:

```
void HLCALLBACK onClickSphere(HLenum event, HLuint object,
HLenum thread,
HLcache *cache,
void *userdata)
{
        // handle event
}


hlAddEventCallback(HL_EVENT_1BUTTONDOWN, mySphereId,
HL_CLIENT_THREAD, &onClickSphere,
(void*) &mydata);
```

Errors:

HL_INVALID_ENUM if event and thread are not among the values listed.
HL_INVALID_OPERATION if no haptic rendering context is current.

See also:

hlRemoveEventCallback, hlEventd, hlCheckEvents

hlCheckEvents

Description:

Calls callback functions for all events that are subscribed to and that have occurred since the last call to hlCheckEvents().

Syntax:

```
void hlCheckEvents()
```

Returns:

None.

Usage:

Should be called on a regular basis in the client thread. This is often called as part of the main rendering loop or main event loop. This function checks if any subscribed events have been triggered since the last call to hlCheckEvents(). If any events have been triggered, then event callbacks for these events will be called synchronously before the call to hlCheckEvents() returns. This only applies to event callbacks in the client thread

(HL_THREAD_CLIENT), collision thread events will be called from within the collision thread, outside of call to the hlCheckEvents().

hlCheckEvents() can be called outside of a begin/endFrame pair. It can be called in place of a begin/end frame to update just device and event state. For example, if the user is feeling a static scene and only needs to have updated device state information periodically, he can forego calling hlBegin()/EndFrame() and instead just call hlCheckEvents() periodically.

Example:

```
void HLCALLBACK onMotion(HLenum event, HLuint object,
HLenum thread, HLcache *cache,
void *userdata)
{
hlBeginFrame();
drawScene();
hlEndFrame();
}

void setup()
{
hlAddEventCallback(HL_EVENT_MOTION, HL_OBJECT_ANY,
HL_CLIENT_THREAD, &onMotion, NULL);
}

void onIdle()
{
hlCheckEvents();
}
```

Errors: HL_INVALID_OPERATION if no haptic rendering context is current.

See also: hlAddEventCallback, hlRemoveEventCallback, hlEventd

hlEventd

Description:	Sets parameters that influence how and when event callbacks are called.
Syntax:	`void hlEventd(HLenum pname, HLdouble param)` Argument: `pname` - Name of event parameter to set. For a list of pname parameters, see "Table B-24: hlEventd - pname values" on page 72. Argument: `param` - Value of parameter to set.
Returns:	None.
Usage:	Used at program startup to tailor the event management of the haptic renderer to the specific application.
Example:	`hlEventd(HL_EVENT_MOTION_LINEAR_TOLERANCE, 10);`
Errors:	HL_INVALID_ENUM if pname is not one of the values listed. HL_INVALID_VALUE if param value is out of range. HL_INVALID_OPERATION if no haptic rendering context is current.
See also:	hlAddEventCallback; hlPushAttrib, hlPopAttrib

hlRemoveEventCallback

Description:	Removes an existing user defined event handling function from the list of callback functions for an event.
Syntax:	`void hlRemoveEventCallback (HLenum event, HLuint shapeId, HLenum thread, HLeventProc fn)` Argument: `event` - Subscribed event. For a list of event parameters, see "Table B-22: hlAddEventCallback, hlRemoveEventCallBack - event values" on page 71. Argument: `shapeID` - Identifier of shape. Callback will only be called if the event occurs on the shape with this identifier unless this argument is set: HL_OBJECT_ANY. In which case the callback will be called regardless of what object is being touched. Argument: `thread` - Thread in which to have callback function called. For a list of thread parameters, see "Table B-23: hlAddEventCallback, hlRemoveEventCallback - thread values" on page 72. Argument: `fn` - Pointer to callback function.
Returns:	None
Usage:	Removes any event callback that was added to list of subscribed events with a call to hlAddEventCallback() with the same event, thread, shapeId and fn pointer.

Example:	`void HLCALLBACK onClickSphere(HLenum event, HLuint object, HLenum thread, HLcache *cache, void *userdata)`
	`{`
	`// handle event`
	`}`
	
	`hlAddEventCallback(HL_EVENT_1BUTTONDOWN, mySphereId, HL_CLIENT_THREAD, &onClickSphere, void*) &mydata);`
	`hlRemoveCallback(HL_EVENT_1BUTTONDOWN, mySphereId, HL_CLIENT_THREAD, &onClickSphere);`

Errors:	HL_INVALID_ENUM if event and thread are not among the values listed. HL_INVALID_OPERATION if no haptic rendering context is current.
See also:	hlAddEventCallback, hlCheckEvents, hlEventd

hlIsEffect

Description:	Determine if an identifier is a valid effect identifier.
Syntax:	`HLboolean hlIsEffect(HLuint effect)` Argument: `effect` - Identifier of first effect to check.
Returns:	None
Usage:	Returns true if effect is a valid identifier that was previously returned by hlGenEffects().

Example:	`HLuint myEffectId = hlGenEffects(1);`
	…
	`assert(hlIsEffect(myEffectId);`

Errors:	HL_INVALID_OPERATION if no haptic rendering context is active.
See also:	hlGenEffects

hlStartEffect

Description:	Starts an effect that will continue to run until it is terminated by a call to hlStopEffect().
Syntax:	`void hlStartEffect (HLenum type, HLuint effect)` Argument: `type` - Type of effect to start. For a list of supported types, see "Table B-17: hlStartEffect - effect types" on page 69. Argument: `effect` - Identifier of effect to start, generated by a call to hlGenEffects().
Returns:	None
Usage:	Starting an effect will cause it to continue to run until hlStopEffect() is called to terminate it. When using hlStartEffect(), the duration property is ignored.

Example:	`hlBeginFrame();`
	`hlEffectd(HL_EFFECT_PROPERTY_GAIN, 0.4);`
	`hlEffectd(HL_EFFECT_PROPERTY_MAGNITUDE, 0.4);`
	`hlStartEffect(HL_EFFECT_FRICTION);`
	`hlEndFrame();`

Errors:	HL_INVALID_ENUM if the type is not one of the values listed. HL_INVALID_OPERATION if not inside an hlBeginFrame()/hlEndFrame() block or if inside an hlBeginShape()/hlEndShape() block.
See also:	hlStopEffect, hlEffectd, hlEffecti, hlEffectdv, hlEffectiv, hlTriggerEffect, hlCallback

hlStopEffect

Description:	Stops an effect that was started with hlStartEffect().
Syntax:	`void hlStopEffect (HLuint effect);` Argument: `effect` - Identifier of effect to start, generated by a call to hlGenEffects().
Returns:	None
Usage:	Once an effect has been started by a call hlStartEffect(), it will continue running until hlStopEffect() is called.

Example:	`hlBeginFrame();`
	`hlStopEffect(friction);`
	`hlEndFrame();`

Errors:	HL_INVALID_OPERATION if not inside an hlBeginFrame()/hlEndFrame() block or if inside an hlBeginShape()/hlEndShape() block.
See also:	hlStartEffect, hlEffectd, hlEffectdv, hlEffecti, hlEffectiv, hlTriggerEffect, hlCallback

hlUpdateCalibration

Description: Causes the haptic device to recalibrate.

Syntax: `void hlUpdateCalibration(void)`

Returns: None.

Usage:

Calibration of the haptic device is a two-stage process. First, new calibration data must be found. How this is done depends on the device hardware.

- For the Geomagic Touch, new calibration data is found when the stylus is inserted into the inkwell.
- For the Geomagic Touch X, calibration data is found as the device is moved about the extents of the workspace.

A program may subscribe to the HL_EVENT_CALIBRATION_UPDATE event to be notified when valid calibration data has been found. It may also subscribe to the

HL_EVENT_CALIBRATION_INPUT event when to be notified when the device requires user input (such as inserting the stylus into the inkwell) in order to obtain valid data.

Once valid calibration has been found, next `hlUpdateCalibration()` is called to flush that data to the haptic device. Note that the change in calibration may cause a large discontinuity in the positions reported by the device. For this reason, `hlUpdateCalibration()` should not be called while the user is in the middle of an operation that relies on continuity of reported device positions (for example, drawing or manipulating an object). For a list of calibration event types see "Table B-28: Calibration Event Types" on page 73.

Example:

```
void HLCALLBACK calibrationCallback(HLenum event, HLuint object,
HLenum thread, HLcache *cache,
void *userdata)
{
    if (event == HL_EVENT_CALIBRATION_UPDATE)
    {
     std::cout << "Device requires calibration update..." << std::endl;
hlUpdateCalibration();
    }
    else if (event == HL_EVENT_CALIBRATION_INPUT)
    {
    std::cout << "Device requires calibration.input..." << std::endl;
    }
}
…
hlAddEventCallback(HL_EVENT_CALIBRATION_UPDATE, HL_OBJECT_ANY,
                   HL_CLIENT_THREAD, &calibrationCallback, NULL);
hlAddEventCallback(HL_EVENT_CALIBRATION_INPUT, HL_OBJECT_ANY,
                   HL_CLIENT_THREAD, &calibrationCallback, NULL);
```

Errors:
HL_INVALID_ENUM if pname is not one of the values listed.
HL_INVALID_VALUE if value is out of range.
HL_INVALID_OPERATION if no haptic rendering context is current.

See also: hlAddEventCallback

hlDeploymentLicense

Description:	Activates the deployment license issued to the application developer. Once the deployment license has been validated, it will remain active until the application is exited. It is not an error to call this more than once in an application session.
Syntax:	`HLboolean hlDeploymentLicense (const char* vendorName, const char* applicationName, const char* password)` Argument: `vendor` - The name of the organization to which the license is issued. Argument: `applicationName` - The name of the application the license is tied to. Argument: `password` - The license string which authenticates the deployment license.
Returns:	Non-zero if the validation succeeded, zero if the validation failed.
Usage:	Needs to be executed before hlCreateContext to operate correctly. hlCreateContext will fail if the deployment license is not valid. Note that if a developers license is present, hlCreateContext will succeed.
Example:	`HLboolean bValid = hlDeploymentLicense ("Haptic Co.", "Haptic Application", 6A12...);`

20 HAPTIC DEVICE API PARAMETERS

The following pages contain tables that list the names of the HDAPI parameters. For each parameter the table lists a description and what types and how many values are used by each parameter name. The developer is responsible for supplying the correct number of parameters for the particular parameter name. Not all parameter names support every type.

The parameters are grouped as follows:

Get Parameters

Table A-1: hdGet Parameters

Parameter Name	Description	Number of Arguments	Allowed Types	Units
HD_CURRENT_BUTTONS	Get the button state. Individual button values can be retrieved by doing bitwise & with HD_DEVICE_BUTTON_N (N=1,2,3, or 4).	1	HDint	
HD_CURRENT_SAFETY_SWITCH	Get whether the safety switch, if one exists, is active. For example, the Geomagic Touch X safety switch is true if the conductive portion of the stylus is being held.	1	HDboolean	
HD_CURRENT_INKWELL_SWITCH	Get whether the inkwell switch, if one exists is active.	1	HDboolean	
HD_CURRENT_ENCODER_VALUES	Get the raw encoder values.	6	HDlong	
HD_CURRENT_POSITION	Get the current position of the device facing the device base. Right is + x, up is +y, toward user is +z.	3	HDdouble, HDfloat	mm
HD_CURRENT_VELOCITY	Get the current velocity of the device. Note: This value is smoothed to reduce high frequency jitter.	3	HDdouble, HDfloat	mm/s
HD_CURRENT_TRANSFORM	Get the column-major transform of the device end-effector.	16	HDdouble, HDfloat	
HD_CURRENT_ANGULAR_VELOCITY	Gets the angular velocity of the device gimbal.	3	HDdouble, HDfloat	rad/s
HD_CURRENT_JOINT_ANGLES	Get the joint angles of the device. These are joint angles used for computing the kinematics of the armature relative to the base frame of the device. For Touch devices: Turet Left +, Thigh Up +, Shin Up +	3	HDdouble, HDfloat	rad
HD_CURRENT_GIMBAL_ANGLES	Get the angles of the device gimbal. For Touch devices: From Neutral position Right is +, Up is -, CW is +	3	HDdouble, HDfloat	rad
HD_USER_STATUS LIGHT	Get the user setting of the LED status light. See hdDefine.h for constant settings.	1	HDint	
HD_CURRENT_PINCH_VALUE	Get the current normalized pinch encoder value (Windows only).	1	HDdouble HDfloat	
HD_LAST_PINCH_VALUE	Get the last normalized pinch encoder value (Windows only).	1	HDdouble HDfloat	

Table A-2: Get Forces Parameters

Parameter Name	Description	Number of Arguments	Allowed Types	Units
HD_CURRENT_FORCE	Get the current force, i.e. the force that the user is commanding to the device during the frame in which this is called. Returns 0 if no force has been commanded yet in the frame.	3	HDfloat, HDdouble	N
HD_CURRENT_TORQUE	Get the current torque, i.e. the torque that the user is commanding to the device during the frame in which this is called. Returns 0 if no torque has been commanded yet in the frame.	3	HDfloat, HDdouble	mNm
HD_CURRENT_MOTOR_DAC_ VALUES	Get the current motor DAC, i.e. the motor value that the user is commanding to the device during the frame in which this is called. Returns 0 if no DAC has been commanded yet in the frame.	3 or 6	HDlong	Inclusive range [-32767, 32766]
HD_CURRENT_ENCODER_VALUES	Get the raw encoder values.	6	HDlong	
HD_CURRENT_JOINT_TORQUE	Get the current joint torque, i.e. the torque the user is commanding to the first 3 joints of the device during the frame in which this is called. Returns 0 if no joint torque has been commanded yet in the frame.	3	HDfloat, HDdouble	mNm
HD_CURRENT_GIMBAL_TORQUE	Get the current gimbal torque, i.e. the gimbal torque the user is commanding to the device during the frame in which this is called. Returns 0 if no gimbal torque has been commanded yet in the frame.	3	HDfloat, HDdouble	mNm

Table A-3: Get Identification Parameters

Parameter Name	Description	Number of Arguments	Allowed Types	Units
HD_FIRMWARE_REVISION	Get the firmware revision.	1	HDdouble	
HD_VERSION	Get the HDAPI software version, in the form of major.minor.build. This is taken from the HD_ VERSION_ definitions.	1	HDstring	
HD_DEVICE_MODEL_TYPE	Get a readable string of the device model type	1	HDstring	
HD_DEVICE_DRIVER_VERSION	Get a readable string of the driver version.	1	HDstring	
HD_DEVICE_VENDOR	Get a readable string of the device vendor..	1	HDstring	
HD_DEVICE_SERIAL_NUMBER	Gets a readable string of the device serial number.	1	HDstring	
HD_MAX_WORKSPACE_ DIMENSIONS	Get the maximum workspace dimensions of the device, i.e. the maximum mechanical limits of the device, as (minX, minY, minZ, maxX, maxY, maxZ).	6	HDfloat, HDdouble	mm
HD_USABLE_WORKSPACE_ DIMENSIONS	Get the usable workspace dimensions of the device, i.e. the practical limits for the device, as (minX, minY, minZ, maxX, maxY, maxZ). It is guaranteed that forces can be reliably rendered within the usable workspace dimensions.	6	HDfloat, HDdouble	mm
HD_TABLETOP_OFFSE	Get the mechanical offset of the device end-effector in Y from the table top.	1	HDfloat	mm
HD_INPUT_DOF	Get the number of input degrees of freedom. (i.e. the number of independent position variables needed to fully specify the end-effector location for Touch device) 3DOF input means xyz tranlsational sensing-only. 6DOF means 3 translation and 3 rotation.	1	HDint	
HD_OUTPUT_DOF	Get the number of output degrees of freedom, i.e. the number of independent actuation variable. For Touch devices 3DOF means XYZ linear force output whereas 6DOF means xyz linear forces and roll, pitch, yaw, torques about gimbal.	1	HDint	

Parameter Name	Description	Number of Arguments	Allowed Types	Units
HD_CALIBRATION_STYLE	The style(s) of calibration supported by the device. Can be one or more bitwise of the following HD_CALIBRATION_AUTO, HD_CALIBRATION_ENCODER_RESET, or HD_CALIBRATION_INKWELL	1	HDint	

Table A-4: Get Last Values Parameters

Parameter Name	Description	Number of Arguments	Allowed Types	Units
HD_LAST_BUTTONS	Get last frame's button state. Individual button values can be retrieved by doing bitwise & with HD_DEVICE_BUTTON_N (N=1,2,3, or 4).	1	HDint	
HD_LAST_SAFETY_SWITCH	Get last frame's safety switch active status.	1	HDboolean	
HD_LAST_INKWELL_SWITCH	Get last frame's inkwell switch active status.	1	HDboolean	
HD_LAST_ENCODER_VALUES	Get last frame's raw encoder values.	6	HDlong	
HD_LAST_POSITION	Get the last position of the device facing the device base. Right is + x, up is +y, toward user is +z.	3	HDdouble, HDfloat	mm
HD_LAST_VELOCITY	Get the last velocity of the device. Note: this value is smoothed to reduce high frequency jitter.	3	HDdouble, HDfloat	mm/s
HD_LAST_TRANSFORM	Get the last row-major transform of the device end-effector.	16	HDdouble, HDfloat	
HD_LAST_ANGULAR_VELOCITY	Get the last Cartesian angular velocity of the device gimbal.	3	HDdouble, HDfloat	rad
HD_LAST_JOINT_ANGLES	Get the last joint angles of the device. These are joint angles used for computing the kinematics of the armature relative to the base frame of the device. For Touch devices: Turet Left +, Thigh Up +, Shin Up +	3	HDdouble, HDfloat	rd
HD_LAST_GIMBAL_ANGLES	Get the last angles of the device gimbal. For Touch devices: From Neutral position Right is +, Up is -, CW is +	3	HDdouble, HDfloat	rad/s

Table A-5: Get Safety Parameters

Parameter Name	Description	Number of Arguments	Allowed Types	Units
HD_NOMINAL_MAX_STIFFNESS	Get the maximum closed loop stiffness that is recommended for the device, i.e. the spring constant used in F=kx output where k is the spring constant and x is the spring length.	1	HDfloat, HDdouble	0 to 1
HD_NOMINAL_MAX_DAMPING	Get the maximum level of damping that is recommended for the device, i.e. the damping constant used in F=kx-dv where d is the damping constant and v is the velocity of the device.	1	HDfloat, HDdouble	0 to 1
HD_NOMINAL_MAX_FORCE	Get the nominal maximum force, i.e. the amount of force that the device can sustain when the motors are at room temperature (optimal).	1	HDfloat, HDdouble	N
HD_NOMINAL_MAX_CONTINUOUS_ FORCE	Get the nominal maximum continuous force, i.e. the amount of force that the device can sustain through a period of time.	1	HDfloat, HDdouble	N
HD_MOTOR_TEMPERATURE	Get the motor temperature, which is the predicted temperature of all of the motors.	3 or 6	HDfloat, HDdouble	0(coldest) to 1(warmest)
HD_SOFTWARE_VELOCITY_LIMIT	Get the software maximum velocity limit. This does not replace the hardware limit of the device.	1	HDfloat, HDdouble	mm/s
HD_SOFTWARE _FORCE_IMPULSE_ LIMIT	Get the software maximum force impulse limit. This does not replace the hardware limit of the device.	1	HDfloat	N
HD_FORCE_RAMPING_RATE	Get the force ramping rate, which is the rate that the device ramps up forces when the scheduler is started or after an error.	1	HDfloat, HDdouble	N/s

Table A-6: Get Scheduler Update Codes Parameters

Parameter Name	Description	Number of Arguments	Allowed Types	Units
HD_UPDATE_RATE	Get the average update rate of the device, i.e. the number of updates that the scheduler performs per second.	1	HDint	Hz
HD_INSTANTANEOUS_UPDATE_ RATE	Get the instantaneous update rate of the device, i.e. I/T where T is the time in seconds since the last update.	1	HDint	Hz

Set Parameters

Table A-7: Set Parameters

Parameter Name	Description	Number of Arguments	Allowed Types	Units
HD_USER_STATUS_LIGHT	Allows the user to set the LED status light.	1	HDint	

Table A-8: Set Forces Parameters

Parameter Name	Description	Number of Arguments	Allowed Types	Units
HD_CURRENT_FORCE	Set the current force as Cartesian coordinated vector. This is the primary method for sending forces to the device. Setting the current force causes that force to be commanded at end of the frame.	3	N	Hz
HD_CURRENT_TORQUE	Set the current torque Cartesian coordinated vector, for 6DOF devices This is the primary method for sending torques to the device. Setting the current torque causes that torque to be commanded at end of the frame.	3	Nm	Hz
HD_CURRENT_MOTOR_DAC_ VALUES	Set the current motor DAC values. This is the primary method for commanding the amount of motor torque counts. This method cannot presently be used in combination with Cartesian force or torque commands.	3 or 6	Inclusive range of “[-32767, 32766]”	
HD_CURRENT_JOINT_TORQUE	Set the current torque as a joint coordinated vector for the first three joints of the Touch device. Setting the current joint torque causes that torque to be commanded at end of the frame.	3	mNm	
HD_CURRENT_GIMBAL_TORQUE	Set current gimbal torque as a joint coordinated vector for the three gimbal joints of the Touch 6DOF device. Setting the current gimbal torque causes that torque to be commanded at end of the frame.	3	mNm	

Table A-9: Set Safety Parameters

Parameter Name	Description	Number of Arguments	Allowed Types	Units
HD_SOFTWARE_VELOCITY_LIMIT	Set the software maximum velocity limit. This does not replace the hardware limit of the device.	1	HDfloat, HDdouble	mm/s
HD_SOFTWARE_FORCE_IMPULSE_ LIMIT	Sets the software maximum force impulse limit, which is the maximum change in magnitude and direction calculated by taking the difference between the current and last commanded force.	1	HDfloat, HDdouble	N/ms
HD_FORCE_RAMPING_RATE	Set the force ramping rate, which is the rate that the device ramps up forces when the scheduler is started or after an error.	1	HDfloat, HDdouble	N/s

Capability Parameters

Table A-10: hdEnable, hdDisable Parameters

Parameter Name	Description
HD_FORCE_OUTPUT	Enables or disables force output for the device. All motors are turned on or off.
HD_MAX_FORCE_CLAMPING	Enables or disables max force clamping for the device. If enabled, this will clamp the overall force output magnitude to the maximum achievable.
HD_FORCE_RAMPING	Enables or disables force ramping, i.e. whether the device ramps up forces when the scheduler is turned on.
HD_SOFTWARE_FORCE_LIMIT	Enables or disables the software maximum force check, which returns an error if the force magnitude commanded to the device exceeds the maximum force limit. This is primarily to disable force kicking. Disable this at your own risk.
HD_SOFTWARE_VELOCITY_LIMIT	Enables or disables the software maximum velocity check, which returns an error if the velocity magnitude commanded to the device exceeds the maximum velocity limit. This is primarily to disable force kicking. Disable this at your own risk.
HD_SOFTWARE_FORCE_IMPULSE_LIMIT	Enables or disables the software maximum force impulse check, which prevents changes in force impulse and direction that exceed the change of impulse limit. This is primarily to prevent device kicking. Disable this at your own risk.
HD_ONE_FRAME_LIMIT	Enables or disables the one frame limit check, which restricts the application to one haptics frame per scheduler tick This should only be disabled if the developer is running his own scheduler.
HD_USER_STATUS_LIGHT	Enables or disables user specified setting of the LED status light on the device.

Codes

Table A-11: Calibration Return Codes

Parameter Name	Description
HD_CALIBRATION_OK	Calibration is accurate.
HD_CALIBRATION_NEEDS_UPDATE	Calibration needs an update. Call **hdUpdateCalibration()** to have the device update its calibration.
HD_CALIBRATION_NEEDS_MANUAL_INPUT	Calibration needs manual input, such as having the user put the device in a reset position or inkwell. Once the user does this, further calls should no longer return that the calibration needs manual input.

Table A-12: Calibration Styles

Parameter Name	Description
HD_CALIBRATION_ENCODER_RESET	The device needs to be put in a reset position to be calibrated.
HD_CALIBRATION_AUTO	The device will gather new calibration information as the armature is moved.
HD_CALIBRATION_INKWELL	Inkwell calibration. The device needs to put into a fixture, i.e. inkwell, before calibration can be performed.

Table A-13: Device Button Codes

Parameter Name	Description
HD_DEVICE_BUTTON_N (N=1,2,3, or 4)	Button states for current and last buttons. Use bitwise & to extract individual button information.

Table A-14: Scheduler Priority Codes

Parameter Name	Description
HD_MAX_SCHEDULER_PRIORITY	Maximum priority for a scheduler callback. Setting a callback with this priority means that the callback will be run first every scheduler tick in relation to other callbacks.
HD_MIN_SCHEDULER_PRIORITY	Minimum priority for a scheduler callback. Setting a callback with this priority means that the callback will be run last every scheduler tick in relation to other callbacks.
HD_DEFAULT_SCHEDULER_PRIORITY	Default scheduler priority. Set this for callbacks that do not care about when they are executed during the scheduler tick. This value is between min and max priority.

Table A-15: Device Error Codes

Parameter Name	Description
HD_SUCCESS	No error
HD_INVALID_ENUM	Invalid parameter of capability specified for a function that requires an HDenum as one of its inputs.
HD_INVALID_VALUE	Invalid value specified for a function that is attempting to set a value.
HD_INVALID_OPERATION	The operation could not be performed.
HD_INVALID_INPUT_TYPE	The parameter or capability does not support the input type.
HD_BAD_HANDLE	The device handle passed into an operation is not valid.
HD_WARM_MOTORS	Device Warning: One or more of the device motors are warm. Wait until the motors cool before commanding forces again.
HD_EXCEEDED_MAX_FORCE	Device warning: Device exceeded the maximum force limit.
HD_EXCEEDED_MAX_FORCE_IMPULSE	Device warning: Change in force magnitude and direction exceeds the maximum force impulse limit.
HD_EXCEEDED_MAX_VELOCITY	Device warning: Device exceeded the maximum velocity limit.
HD_FORCE_ERROR	The device experienced an error in sending forces because forces of incompatible types such as DAC and Cartesian were commanded simultaneously.
HD_DEVICE_FAULT	Device fault. Check that the device is connected properly and activated.
HD_DEVICE_ALREADY_INITIATED	The device name was already initialized.
HD_COMM_ERROR	Communication Error: Check the device connection and configuration.
HD_COMM_CONFIG_ERROR	Configuration Error: Check the base address and port/adapter number. Consult the appropriate device drivers documentation.
HD_TIMER_ERROR	Unable to start or maintain the servo loop timer.
HD_ILLEGAL_BEGIN	**hdBeginFrame()** for a device was called when already in that device's frame, or more than once during a single servo loop tick.
HD_ILLEGAL_END	**hdEndFrame()** for a device was called without a matching **hdBeginFrame()** for that device beforehand.
HD_FRAME_ERROR	A function that is restricted to being called within a frame was called outside of one.
HD_INVALID_PRIORITY	The priority for a scheduler callback does not fall within the proper numerical bounds defined by MIN and MAX scheduler priority.
HD_SCHEDULER_FULL	The scheduler has reached its limit of callbacks and cannot add the new callback.
HD_INVALID_LICENSE	The required license is invalid or missing.

21 HAPTIC LIBRARY API PARAMETERS

The following pages contain tables that list the names of the HLAPI parameters. For each parameter the table lists a description.

The parameters are grouped as follows:

State Maintenance Parameters

Table B-1: hlGetBooleanv, hlGetIntegerv, hlGetDoublev

Parameter Name	Description
HL_BUTTON1_STATE	Single Boolean value representing the first button on the haptic device. A value of true means that the button is depressed.
HL_BUTTON2_STATE	Single Boolean value representing the second button on the haptic device. A value of true means that the button is depressed.
HL_SAFETY_STATE	Single Boolean value representing the safety switch on the haptic device, if on exists. A value of true means that the switch is depressed.
HL_INKWELL_STATE	Single Boolean value representing the inkwell switch on the haptic device, if on exists. A value of true means that the switch is depressed.
HL_DEPTH_OF_PENETRATION	Double precision value representing the depth of the penetration of the device in world coordinates. This will return the depth of penetration for the current object being touched by the proxy.
HL_DEVICE_FORCE	Vector of 3 doubles representing the last force, in world coordinates, sent to the haptic device.
HL_DEVICE_POSITION	Vector of 3 doubles representing the position of the haptic device in world coordinates.
HL_DEVICE_ROTATION	Vector of 4 doubles representing a quaternion that specifies the rotation of the haptic device in world coordinates.
HL_DEVICE_TORQUE	Vector of 3 doubles representing the last torque, in world coordinates, sent to the haptic device.
HL_DEVICE_TRANSFORM	Vector of 16 doubles representing a 4x4 transform matrix in column major order that specifies the transform of the haptic device relative to world coordinates.
HL_EVENT_MOTION_ANGULAR_TOLERANCE	Double precision value representing the minimum rotation, in radians, that the proxy must move before a motion event is triggered.
HL_EVENT_MOTION_LINEAR_TOLERANCE	Double precision value representing the minimum distance, in device workspace coordinates, that the proxy must move before a motion event is triggered.
HL_GOLDEN_POSITION	A vector of 3 doubles representing the golden sphere center position in world coordinates. The golden sphere is a bounding volume for the proxy and designates the space of allowed proxy movement for a particular HL frame. The client is responsible for providing geometry within this bounding volume. Typically, this bounding volume gets used for sizing the graphic view volume when the haptic camera view is enabled. The adaptive viewport optimization also references this golden sphere for determining how much of the depth buffer to read back. Additionally, the golden sphere can be useful when implementing a callback shape or performing bounding volume culling. The radius of the sphere may be queried using HL_GOLDEN_RADIUS.

Parameter Name	Description
HL_GOLDEN_RADIUS	Double precision value representing the radius of the golden sphere in world coordinates. The golden sphere is a bounding volume for the proxy and designates the space of allowed proxy movement for a particular HL frame. The client is responsible for providing geometry within this bounding volume. Typically, this bounding volume gets used for sizing the graphic view volume when the haptic camera view is enabled. The adaptive viewport optimization also references this golden sphere for determining how much of the depth buffer to read back. Additionally, the golden sphere can be useful when implementing a callback shape or performing bounding volume culling. The center of the sphere may be queried using HL_GOLDEN_POSITION.
HL_MODELVIEW	Array of sixteen doubles representing a 4x4 transform matrix in column major order that specifies the transform from model coordinates to view coordinates. This matrix is only applied when not using the OpenGL modelview (when HL_USE_GL_MODELVIEW is disabled). When the OpenGL modelview is used, you should use OpenGL functions to query the modelview matrix.
HL_PROXY_IS_TOUCHING	Single Boolean value set to true when the proxy is in contact with one or more shapes.
HL_PROXY_POSITION	Vector of 3 doubles representing the position of the proxy in world coordinates.
HL_PROXY_ROTATION	Vector of 4 doubles representing a quaternion that specifies the rotation of the proxy in world coordinates.
HL_TOUCHABLE_FACE	Get the touchable face for the model, which represents which side of the model can be felt. See “Table B-10: hlMaterialf - face values” for a list of return values.
	Gets the integer touch type, which represents how the proxy interacts with the model. See “Table B-15: hlTouchModel” for a list of return values.
HL_PROXY_TOUCH_NORMAL	A vector of 3 doubles representing the surface normal at the point of contact with the set of shapes in contact with the proxy. Only valid if HL_PROXY_IS_TOUCHING is true.
HL_PROXY_TRANSFORM	Vector of 16 doubles representing a 4x4 transform matrix in column major order that specifies the transform of the proxy relative to world coordinates
HL_TOUCHWORKSPACE_MATRIX	Array of sixteen doubles representing a 4x4 transform matrix in column major order that specifies the transform from touch coordinates to workspace coordinates.
HL_VIEWTOUCH	Array of sixteen doubles representing a 4x4 transform matrix in column major order that specifies the transform from view coordinates to touch coordinates.

Table B-2: hlCacheGetBooleanv, hlCacheGetDoublev

Parameter Name	Description
HL_BUTTON1_STATE	Single Boolean value representing the first button on the haptic device. A value of true means that the button is depressed.
HL_BUTTON2_STATE	Single Boolean value representing the second button on the haptic device. A value of true means that the button is depressed.
HL_SAFETY_STATE	Single Boolean value representing the safety switch on the haptic device, if on exists. A value of true means that the switch is depressed.
HL_INKWELL_STATE	Single Boolean value representing the inkwell switch on the haptic device, if on exists. A value of true means that the switch is depressed.
HL_DEVICE_FORCE	Vector of 3 doubles representing the last force, in world coordinates, sent to the haptic device.
HL_DEVICE_POSITION	Vector of 3 doubles representing the position of the haptic device in world coordinates.
HL_DEVICE_ROTATION	Vector of 4 doubles representing a quaternion that specifies the rotation of the haptic device in world coordinates.
HL_DEVICE_TORQUE	Vector of 3 doubles representing the last torque, in world coordinates, sent to the haptic device.
HL_DEVICE_TRANSFORM	Vector of 16 doubles representing a 4x4 transform matrix in column major order that specifies the transform of the haptic device relative to world coordinates.
HL_PROXY_IS_TOUCHING	Single Boolean value set to true when the proxy is in contact with one or more shapes.
HL_PROXY_POSITION	Vector of 3 doubles representing the position of the proxy in world coordinates.
HL_PROXY_ROTATION	Vector of 4 doubles representing a quaternion that specifies the rotation of the proxy in world coordinates.
HL_PROXY_TOUCH_NORMAL	A vector of 3 doubles representing the surface normal at the point of contact with the set of shapes in contact with the proxy. Only valid if HL_PROXY_IS_TOUCHING is true.
HL_PROXY_TRANSFORM	Vector of 16 doubles representing a 4x4 transform matrix in column major order that specifies the transform of the proxy relative to world coordinates.

Table B-3: hlGetError

Parameter Name	Description
HL_DEVICE_ERROR	An error occurred with the haptic device. The errorInfo field of the HLerror struct will contain a device specific error code. See hddefine.h for a list of device errors. This error may be signaled even outside any API function calls if the device error is detected in the haptic rendering engine running in a separate thread. The haptic device recovers in most cases.
HL_INVALID_ENUM	An invalid value for an enumerated type was passed to an API function. The function call will have no effect other than to set the error.
HL_INVALID_OPERATION	An API function was called when the renderer was not in an appropriate state for that function call. For example, **hlBeginShape()** was called outside an **hlBeginFrame()**/ **hlEndFrame()** pair, or **hlBeginFrame()** when no haptic rendering context was active. The function call will have no effect other than to set the error.
HL_INVALID_VALUE	A value passed to an API function is outside the valid range for that function. The function call will have no effect other than to set the error.
HL_NO_ERROR	No error. There were no errors from API function calls since the last time the error stack was empty; the error stack can be empty if either no errors have ever occurred or all errors have been already queried from the stack.
HL_OUT_OF_MEMORY	There is not enough memory to complete the last API function called. This function may have partially completed leaving the haptic renderer in an undefined state.
HL_STACK_OVERFLOW	An API function was called that would have caused an overflow of the matrix stack. The function call will have no effect other than to set the error.
HL_STACK_UNDERFLOW	An API function was called that would have caused an underflow of the matrix stack, i.e. a call to **hlPopMatrix()** when the stack is empty. The function call will have no effect other than to set the error.

Table B-4: hlGetString

Parameter Name	Description
HL_VENDOR	Returns the name of the company responsible for the haptic renderer implementation.
HL_VERSION	Returns the version number of the haptic rendering library as a string of the form: major_number.minor_number.build_number

Table B-5: hlHinti, hlHintb

Parameter Name	Description
HL_SHAPE_DYNAMIC_SURFACE_CHANGE	A single Boolean value indicating whether or not shapes that follow should be treated as dynamically changing surfaces. A value of true tells the haptic rendering engine to do extra processing to ensure that the proxy does not fall through a dynamically changing surface. A value of false tells the rendering engine to optimize haptic rendering for surfaces which will not change.
HL_SHAPE_FEEDBACK_BUFFER_VERTICES	A single integer value indicating to the haptic rendering engine approximately how many vertices will be in the next feedback buffer shape so that it may reserve the correct amount of memory.
HL_SHAPE_PERSISTENCE	A single boolean value indicating whether or not shapes that follow should persistent beyond the current frame. A value of true tells the haptic renderer that the shape does not need to be respecified during the next frame, but that it should be automatically included. This is useful particularly for background objects, i.e. static objects that change rarely in the scene, since those objects can just be specified once and left in the scene versus having the scene recompute the object for each frame. To clear a persistent shape, set it as non-persistent during any frame afterward. **LIMITATION: This feature will not work if HL_HAPTIC_CAMERA_VIEW is enabled.** **NOTE: This is a minimally supported experimental feature for v2.0. It will most likely be deprecated, removed, or replaced by alternate API command in the next release.**

Shape Parameters

Table B-6: hlBegin Shape

Parameter Name	Description
HL_SHAPE_CALLBACK	Allows clients to create custom shapes not supported using the other shape types. OpenGL commands are ignored for this shape type. No geometry is specified. Instead, the current shape intersection and shape closest point callbacks are used for haptic rendering of this shape.
HL_SHAPE_DEPTH_BUFFER	OpenGL commands used following the **hlBeginShape()** call will generate an image in the OpenGL depth buffer. When **hlEndShape()** is called, this image will be read out of the depth buffer and sent to the haptic renderer. Any OpenGL commands that modify the depth buffer may be used. Point and line constraints may not be specified using a depth buffer shape.
HL_SHAPE_FEEDBACK_BUFFER	OpenGL commands used following the **hlBeginShape()** call will generate a set of geometric primitives (quads, triangles, points and lines) that will be sent to the OpenGL feedback buffer. When **hlEndShape()** is called, these primitives will be read out of the feedback buffer and sent to the haptic renderer. Feedback buffer shapes can be used to specify triangles meshes as well as points and lines for constraints. For optimal memory usage and performance, use **hlHint()** with HL_SHAPE_FEEDBACK_BUFFER_VERTICES before **hlBeginShape()**, so that the size of the feedback buffer may be optimally allocated.

Table B-7: hlGetShapeBooleanv, hlGetShapeDoublev

Parameter Name	Description
HL_PROXY_IS_TOUCHING	Single Boolean value. True means that the proxy is currently in contact with the shape.
HL_REACTION_FORCE	Vector of 3 doubles representing the contribution of this shape to the overall reaction force that was sent to the haptic device during the last frame. If the proxy was not in contact with this shape last frame, this vector will be zero.

Table B-8: hlLocalFeature types

Parameter Name	Description
HL_LOCAL_FEATURE_LINE	The local feature is a line segment whose start point and end point are given by the vectors v1 and v2 respectively.
HL_LOCAL_FEATURE_PLANE	The local feature is a plane whose normal is given by the vector v1 and that passes through the point v1.
HL_LOCAL_FEATURE_POINT	The local feature is a single point whose position is given by the vector v.

Capability Parameters

Table B-9: hlEnable, hlDisable, hlIsEnabled

Parameter Name	Description
HL_ADAPTIVE_VIEWPORT	If enabled, this feature serves as an optimization for single pass rendering of depth buffer shapes. This feature adaptively sizes the read-back pixel dimensions of the viewport based on the location and motion of the proxy. This feature is incompatible with haptic camera view. This is disabled by default.
HL_HAPTIC_CAMERA_VIEW	Enhances shape rendering by modifying the shape viewing transform based on the motion of the proxy. This allows the user to feel depth buffer features that are occluded in the primary camera view. The haptic camera view also serves as an optimization by sizing the viewing volume and viewport so that the graphics pipeline only processes geometry that is within a proximity to the proxy. This is disabled by default.
HL_PROXY_RESOLUTION	If enabled, use shape geometry to automatically update the proxy position. The computed proxy position will be valid for all shapes specified each frame. If disabled, shapes will be ignored and the proxy position will be set by the client program. This is enabled by default.
HL_USE_GL_MODELVIEW	If enabled, apply the OpenGL model view matrix to all geometry for haptic rendering and ignore the current HL_MODELVIEW setting. This should be disabled in HLAPI programs that do not have a a valid OpenGL rendering context query. It is enabled by default.

Material and Surface Parameters

Table B-10: hlMaterialf - face values

Parameter Name	Description
HL_FRONT	Apply the material property only to the front face of shapes and to all faces of constraints.
HL_BACK	Apply the material property only to the back face of shapes and to all faces of constraints.
HL_FRONT_AND_BACK	Apply the material property to both front and back faces.
HL_POPTHROUGH	Popthrough controls the amount of force the user must apply to a shape before the device pops through the shape to the other side. The larger the param value, the higher the force required. A param value of o disables popthrough.

Table B-11: hlMaterialf - pname values

Parameter Name	Description
HL_STIFFNESS	Stiffness controls how hard surfaces feel. Param must be a value between 0 and 1 where 1 represents the hardest surface the haptic device is capable of rendering and 0 represents the most compliant surface that can be rendered.
HL_DAMPING	Damping reduces the springiness of the surface. Param must be between 0 and 1 where 0 represents no damping, i.e. a highly springy surface and 1 represents the maximum level of damping possible
HL_STATIC_FRICTION	Static friction controls the resistance of a surface to tangential motion when the proxy position is not changing i.e. how hard it is to slide along the surface starting from a complete stop. A param value of 0 is a completely frictionless surface and a value of 1 is the maximum amount of static friction the haptic device is capable of rendering.
HL_DYNAMIC_FRICTION	Dynamic friction controls the resistance of a surface to tangential motion when the proxy position is changing i.e. how hard it is to slide along the surface once already moving. A param value of 0 is a completely frictionless surface and a value of 1 is the maximum amount of dynamic friction the haptic device is capable of rendering.

Table B-12: hlGetMaterialfv - face values

Parameter Name	Description
HL_FRONT	Query the material property of the front face of shapes and all constraints.
HL_BACK	Query the material property of the back face of shapes and all constraints.

Table B-13: hlGetMaterialfv - pname values

Parameter Name	Description
HL_STIFFNESS	Stiffness controls how hard surfaces feel. Param must be a value between 0 and 1 where 1 represents the hardest surface the haptic device is capable of rendering and 0 represents the most compliant surface that can be rendered.
HL_DAMPING	Damping reduces the springiness of the surface. Param must be between 0 and 1 where 0 represents no damping, i.e. a highly springy surface and 1 represents the maximum level of damping possible.
HL_STATIC_FRICTION	Static friction controls the resistance of a surface to tangential motion when the proxy position is not changing i.e. how hard it is to slide along the surface starting from a complete stop. A param value of 0 is a completely frictionless surface and a value of 1 is the maximum amount of static friction the haptic device is capable of rendering.
HL_DYNAMIC_FRICTION	Dynamic friction controls the resistance of a surface to tangential motion when the proxy position is changing i.e. how hard it is to slide along the surface once already moving. A param value of 0 is a completely frictionless surface and a value of 1 is the maximum amount of dynamic friction the haptic device is capable of rendering.

Parameter Name	Description
HL_POPTHROUGH	Popthrough controls the amount of force the user must apply to a shape before the device pops through the shape to the other side. The larger the param value, the higher the force required. Param must be between 0 and 1 where a value of 0 disables popthrough.

Table B-14: hlTouchableFace - mode values

Parameter Name	Description
HL_FRONT	Only front faces will be touchable.
HL_BACK	Only back faces will be touchable.
HL_FRONT_AND_BACK	All faces, both front and back, will be touchable.

Table B-15: hlTouchModel

Parameter Name	Description
HL_CONTACT	The proxy position is not allowed to pass through geometric primitives (triangles). The proxy may move off the surface but it will remain on one side of it.
HL_CONSTRAINT	The proxy position is constrained to remain exactly on the surface of geometric primitives and it may only be moved off of the surface if the distance between the device position and the proxy is greater than the snap distance.

Table B-16: hlTouchModelIf

Parameter Name	Description
HL_SNAP_DISTANCE	Distance between the proxy position and the surface that must be exceeded to pull off a constraint. Param should be a floating point value representing the distance in millimeters in workspace coordinates. The default value is FLT_MAX to always be active.

Force Effect Parameters

Table B-17: hlStartEffect - effect types

Parameter Name	Description
HL_EFFECT_CONSTANT	Adds a constant force vector to the total force sent to the haptic device. The effect property HL_EFFECT_PROPERTY_DIRECTION specifies the direction of the force vector. The effect property HL_EFFECT_PROPERTY_MAGNITUDE specifies the magnitude of the force vector.
HL_EFFECT_SPRING	Adds a spring force to the total force sent to the haptic device. The spring force pulls the haptic device towards the effect position and is proportional to the product of the gain and the distance between the effect position and the device position. Specifically, the spring force is calculated using the expression F = k(P-X) where F is the spring force, P is the effect position, X is the current haptic device position and k is the gain. The effect position is specified by the property HL_EFFECT_PROPERTY_POSITION. The gain is specified by the property HL_EFFECT_PROPERTY_GAIN. The magnitude of the effect force is capped at the value of the effect property HL_EFFECT_MAGNITUDE.
HL_EFFECT_VISCOUS	Adds a viscous force to the total force sent to the haptic device. The viscous force is based on the current velocity of the haptic device and is calculated to resist the motion of the haptic device. Specifically the force is calculated using the expression F = -kV where f is the spring force, V is the velocity and k is the gain. The gain is specified by the property HL_EFFECT_PROPERTY_GAIN. The magnitude of the effect force is capped at the value of the effect property HL_EFFECT_MAGNITUDE.
HL_EFFECT_FRICTION	Adds a friction force to the total force sent to the haptic device. Unlike friction specified via **hlMaterial()** calls, this is friction both while touching objects and in free space. The gain of the friction force is specified by the property HL_EFFECT_PROPERTY_GAIN. The magnitude of the effect force is capped at the value of the effect property HL_EFFECT_MAGNITUDE.

Parameter Name	Description
HL_EFFECT_CALLBACK	Allows for the user to create a custom effect by setting effect callbacks using **hlCallback()**.

Table B-18: hlTriggerEffect

Parameter Name	Description
HL_EFFECT_CONSTANT	Adds a constant force vector to the total force sent to the haptic device. The effect property HL_EFFECT_PROPERTY_DIRECTION specifies the direction of the force vector. The effect property HL_EFFECT_PROPERTY_MAGNITUDE specifies the magnitude of the force vector.
HL_EFFECT_SPRING	Adds a spring force to the total force sent to the haptic device. The spring force pulls the haptic device towards the effect position and is proportional to the product of the gain and the distance between the effect position and the device position. Specifically, the spring force is calculated using the expression F = k(P-X) where F is the spring force, P is the effect position, X is the current haptic device position and k is the gain. The effect position is specified by the property HL_EFFECT_PROPERTY_POSITION. The gain is specified by the property HL_EFFECT_PROPERTY_GAIN. The magnitude of the effect force is capped at the value of the effect property HL_EFFECT_MAGNITUDE.
HL_EFFECT_VISCOUS	Adds a viscous force to the total force sent to the haptic device. The viscous force is based on the current velocity of the haptic device and is calculated to resist the motion of the haptic device. Specifically the force is calculated using the expression F = -kV where f is the spring force, V is the velocity and k is the gain. The gain is specified by the property HL_EFFECT_PROPERTY_GAIN. The magnitude of the effect force is capped at the value of the effect property HL_EFFECT_MAGNITUDE.
HL_EFFECT_FRICTION	Adds a friction force to the total force sent to the haptic device. Unlike friction specified via **hlMaterial()** calls, this is friction both while touching objects and in free space. The gain of the friction force is specified by the property HL_EFFECT_PROPERTY_GAIN. The magnitude of the effect force is capped at the value of the effect property HL_EFFECT_MAGNITUDE.
HL_EFFECT_CALLBACK	Allows for the user to create a custom effect by setting effect callbacks using **hlCallback()**.

Table B-19: hlEffectd, hlEffecti, hlEffectdv, hlEffectiv

Parameter Name	Description
HL_EFFECT_PROPERTY_GAIN	Used by spring, friction and viscous effect types. Higher gains will cause these effects to generate larger forces.
HL_EFFECT_PROPERTY_MAGNITUDE	Used by constant, spring, friction and viscous effect types. Represents a cap on the maximum force generated by these effects.
HL_EFFECT_PROPERTY_FREQUENCY	Reserved for use by future effect types and callback effects.
HL_EFFECT_PROPERTY_DURATION	Used for all effects started with a call to **hlTriggerEffect()**. The effect will automatically be terminated when the duration has elapsed. Duration is specified in milliseconds.
HL_EFFECT_PROPERTY_POSITION	Used by spring effect. Represents the anchor position of the spring.
HL_EFFECT_PROPERTY_DIRECTION	Used by constant effect. Represents direction of constant force vector.

Table B-20: hlEffectd, hlEffecti, hlEffectdv, hlEffectiv

Parameter Name	Description
HL_EFFECT_PROPERTY_GAIN	Used by spring, friction and viscous effect types. Higher gains will cause these effects to generate larger forces.
HL_EFFECT_PROPERTY_MAGNITUDE	Used by constant, spring, friction and viscous effect types. Represents a cap on the maximum force generated by these effects.
HL_EFFECT_PROPERTY_FREQUENCY	Reserved for use by future effect types and callback effects.
HL_EFFECT_PROPERTY_DURATION	Used for all effects started with a call to **hlTriggerEffect()**. The effect will automatically be terminated when the duration has elapsed. Duration is specified in milliseconds.
HL_EFFECT_PROPERTY_POSITION	Used by spring effect. Represents the anchor position of the spring.
HL_EFFECT_PROPERTY_DIRECTION	Used by constant effect. Represents direction of constant force vector.

Parameter Name	Description
HL_EFFECT_PROPERTY_ACTIVE	Used by all effects. Indicates whether the effect is currently running, i.e. has been started via **hlStartEffect()**.
HL_EFFECT_PROPERTY_TYPE	Used by all effect. Represents the type of effect, such as HL_EFFECT_CALLBACK, HL_EFFECT_SPRING.

Callback Parameters

Table B-21: hlCallback

Parameter Name	Description
HL_SHAPE_INTERSECT_LS	Callback to intersect a line segment with a user defined custom shape. The callback function must have the following signature: HLboolean HLCALLBACK fn(const HLdouble startPt[3], const HLdouble endPt[3], HLdouble intersectionPt[3], HLdouble intersectionNormal[3], void *userdata) Where input parameters startPt and endPt are the endpoints of a line segment to intersect with the custom shape, intersectionPt is used to return the point of intersection of the line segment with the shape. In the case of multiple intersections, intersectionPt should be the closest intersection to startPt. IntersectionNormal is the surface normal of the custom shape at intersectionPt. Userdata is the same userdata pointer passed to **hlCallback()**. The callback function should return true if the line segment intersects the shape. All data is in the local coordinate system of the custom shape.
HL_SHAPE_CLOSEST_FEATURES	Callback to find the closest point on the surface to an input point as well as one ore more local features that approximate the surface in the vicinity of that point. The callback function must have the following signature: HLboolean HLCALLBACK fn(const HLdouble queryPt[3], const HLdouble targetPt[3], HLgeom *geom, HLdouble closestPt[3], void* userdata) Where closest point should be set to the closest point on the surface of the custom shape closest to the input parameter queryPt. Normal is the surface normal of the custom shape at closestPt. Userdata is the same userdata pointer passed to **hlCallback()**. All data is in the local coordinate system of the custom shape.

Event Parameters

Table B-22: hlAddEventCallback, hlRemoveEventCallBack - event values

Parameter Name	Description
HL_EVENT_MOTION	Callback will be called when either the proxy has moved more than the HL_EVENT_MOTION_LINEAR_TOLERANCE millimeters in workspace coordinates from the position when the last motion event was triggered or when the proxy has been rotated more than HL_EVENT_MOTION_ANGULAR_TOLERANCE radians from the rotation of the proxy last time a motion event was triggered.
HL_EVENT_1BUTTONDOWN	Callback will be called when the first button on the haptic device is depressed.
HL_EVENT_1BUTTONUP	Callback will be called when the first button on the haptic device is released.
HL_EVENT_2BUTTONDOWN	Callback will be called when the second button on the haptic device is depressed.
HL_EVENT_2BUTTONUP	Callback will be called when the second button on the haptic device is released.
HL_EVENT_SAFETYDOWN	Callback will be called when the safety switch, if available, is depressed.
HL_EVENT_SAFETYUP	Callback will be called when the safety switch, if available, is released.
HL_EVENT_INKWELLDOWN	Callback will be called when the inkwell switch, if available, is depressed.
HL_EVENT_INKWELL_UP	Callback will be called when the inkwell switch, if available, is released.
HL_EVENT_TOUCH	Callback will be called when a shape in the scene has been touched (the proxy is in contact with the shape).

Table B-23: hlAddEventCallback, hlRemoveEventCallback - thread values

Parameter Name	Description
HL_CLIENT_THREAD	Callback function will be called from client thread, when the client program calls **hlCheckEvents()**.
HL_COLLISION_THREAD	Callback function will be called from the internal collision thread running in the haptic rendering engine. Most event callbacks should be handled in the client thread, however there are some cases where collision thread callbacks are needed due to timing requirements.

Table B-24: hlEventd - pname values

Parameter Name	Description
HL_EVENT_MOTION_LINEAR_TOLERANCE	Sets the minimum distance in device workspace coordinates, that the linear translation of the proxy must change before a motion event is triggered. By default this value is one millimeter.
HL_EVENT_MOTION_ANGULAR_TOLERANCE	Sets the minimum angular distance in, that orientation of the proxy must change before a motion event is triggered. By default this value is 0.02 radians.

Proxy Parameters

Table B-25: hlProxydv

Parameter Name	Description
HL_PROXY_POSITION	Sets the position of the proxy in world coordinates. If proxy resolution is enabled, this call will have no effect.

Transform Parameters

Table B-26: hlMatrix - mode values

Parameter Name	Description
HL_VIEWTOUCH	All matrix manipulation commands will target the transform from view coordinates to touch coordinates.
HL_TOUCHWORKSPACE	All matrix manipulation commands will target the transform from touch coordinates to the local coordinates of the haptic device.
HL_MODELVIEW	All matrix manipulation commands will target the transform from model coordinates to view coordinates. The HL_MODELVIEW is only applied when HL_USE_GL_MODELVIEW is disabled, otherwise the OpenGL modelview matrix is used.

Table B-27: hlPushAttrib, hlPopAttrib

Parameter Name	Description
HL_HINT_BIT	Affects all properties that are set by **hlInt()** command such as HL_SHAPE_FEEDBACK_BUFFER.
HL_MATERIAL_BIT	Affects all properties that are specified by **hlMaterial()** commands such as HL_STIFFNESS, HL_DYNAMIC_FRICTION.
HL_TOUCH_BIT	Affects all properties that are specified by **hlTouch()** commands, such as HL_SNAP_DISTANCE, HL_TOUCH_MODEL.
HL_TRANSFORM_BIT	Affects all transform and camera attributes, such as HL_HAPTIC_CAMERA_VIEW, HL_ADAPTIVE_VIEWPORT.
HL_EFFECT_BIT	Affects all properties that are specified by **hlEffect()** commands, such as HL_EFFECT_PROPERTY_GAIN, HL_EFFECT_PROPERTY_DURATION.
HL_EVENTS_BIT	Affects all properties that are specified by event commands, such as HL_EVENT_MOTION_LINEAR_THRESHOLD, HL_EVENT_MOTION_ANGULAR_TOLERANCE.

Table B-28: Calibration Event Types

Parameter Name	Description
HL_EVENT_CALIBRATION_UPDATE	Triggers when valid calibration data has been found.
HL_EVENT_CALIBRATION_INPUT	Triggers when the device requires user input, such as inserting the stylus into the inkwell.

3D Systems, Inc.
333 Three D Systems Circle | Rock Hill, SC | 29730
www.3dsystems.com

pn 391572-00Rev. A