a Component Based Architecture
for Uniform Access to Haptic Devices
http://www.haptiklibrary.org
designed by
Maurizio de Pascale
mdepascale@dii.unisi.it
developed at
Siena Robotics and Systems Lab
http://sirslab.dii.unisi.it

Siena, Italy
November 17th, 2005
Disclaimer

This documentation is far from complete. While it includes almost all needed information surely there is lot of room for improvements. However you will find both introductory and tutorial sections as well as a complete reference for the library’s API. Please send any suggestions to info@haptiklibrary.org.

Have fun, and enjoy coding!
## Contents

### I ABOUT HAPTIK LIBRARY

1 Introduction ................................................. 11
   1.1 Why another library? .................................. 11
   1.2 The Haptik Library .................................... 12
      1.2.1 Advantages ....................................... 12
      1.2.2 Binary Compatibility ............................... 13
   1.3 Where Applicable ...................................... 13

2 The Haptik Library ........................................ 15
   2.1 Architecture .......................................... 15
      2.1.1 DLLs ............................................... 16
      2.1.2 Components and Interfaces ......................... 16
   2.2 Behavior ............................................... 17
      2.2.1 Internal Recursion ................................ 17
   2.3 Performance Considerations ............................ 18
   2.4 Configuration File and Library Settings ............... 19
   2.5 Bundled Plugins ........................................ 20
      2.5.1 Phantom ........................................... 20
      2.5.2 Delta ............................................... 21
      2.5.3 Freedom ............................................ 21
      2.5.4 Spectre ............................................. 22
      2.5.5 Remote ............................................. 23
      2.5.6 Betamix ............................................. 23
   2.6 Supported Platforms ................................... 24

3 Using and Extending ..................................... 25
   3.1 About 0.Xs Versions .................................. 25
   3.2 Compatibility ......................................... 25
   3.3 License ................................................ 26
   3.4 Supporting new Devices ................................. 26

### II USING HAPTIK LIBRARY

4 Application Development ................................ 29
   4.1 Project/Environment Setup ............................. 29
   4.2 Including Header and Lib .............................. 29
### Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3 Hello Haptik Sample</td>
<td>30</td>
</tr>
<tr>
<td>4.4 Accessing the Library (the easy way)</td>
<td>31</td>
</tr>
<tr>
<td>4.5 Accessing the Library (the hard way)</td>
<td>32</td>
</tr>
<tr>
<td>4.6 Getting interfaces to devices</td>
<td>33</td>
</tr>
<tr>
<td>4.7 Accessing Devices</td>
<td>34</td>
</tr>
<tr>
<td>4.7.1 Using Callback</td>
<td>34</td>
</tr>
<tr>
<td>4.7.1.1 Callbacks</td>
<td>35</td>
</tr>
<tr>
<td>4.7.1.2 User Data</td>
<td>35</td>
</tr>
<tr>
<td>4.7.2 Using Polling</td>
<td>35</td>
</tr>
<tr>
<td>4.8 Using Devices</td>
<td>36</td>
</tr>
<tr>
<td>4.9 Releasing Interfaces</td>
<td>36</td>
</tr>
<tr>
<td>4.10 Exchanging Data with the Device</td>
<td>37</td>
</tr>
<tr>
<td>4.11 Measure Units</td>
<td>38</td>
</tr>
<tr>
<td>4.12 Coordinate Systems</td>
<td>38</td>
</tr>
<tr>
<td>5 PlugIn Development</td>
<td>39</td>
</tr>
<tr>
<td>5.1 Headers &amp; Libs</td>
<td>39</td>
</tr>
<tr>
<td>5.2 Sample Plugin Projects</td>
<td>39</td>
</tr>
<tr>
<td>5.3 Building DLLs</td>
<td>40</td>
</tr>
<tr>
<td>5.4 Building Plugins</td>
<td>41</td>
</tr>
<tr>
<td>5.5 IDs</td>
<td>42</td>
</tr>
<tr>
<td>5.6 Ordinals</td>
<td>42</td>
</tr>
<tr>
<td>5.7 Accessing the Library from Plugins</td>
<td>43</td>
</tr>
<tr>
<td>5.8 Building Devices</td>
<td>44</td>
</tr>
<tr>
<td>5.9 Invoking Callbacks</td>
<td>45</td>
</tr>
<tr>
<td>5.10 Coordinate Systems</td>
<td>45</td>
</tr>
<tr>
<td>III HAPTIK LIBRARY REFERENCE</td>
<td>47</td>
</tr>
<tr>
<td>6 C++ Application Developer’s Reference</td>
<td>49</td>
</tr>
<tr>
<td>6.1 Classes</td>
<td>50</td>
</tr>
<tr>
<td>6.1.1 Haptik</td>
<td>51</td>
</tr>
<tr>
<td>6.1.1.1 GetDeviceInterface</td>
<td>52</td>
</tr>
<tr>
<td>6.1.1.2 RefreshDeviceList</td>
<td>53</td>
</tr>
<tr>
<td>6.2 Functions</td>
<td>54</td>
</tr>
<tr>
<td>6.2.1 GetHaptikLibraryInterface</td>
<td>55</td>
</tr>
<tr>
<td>6.3 Constants</td>
<td>56</td>
</tr>
<tr>
<td>6.3.1 Default Devices</td>
<td>56</td>
</tr>
<tr>
<td>6.3.2 Return Values</td>
<td>57</td>
</tr>
<tr>
<td>6.4 Interfaces</td>
<td>58</td>
</tr>
<tr>
<td>6.4.1 IHaptikLibrary</td>
<td>59</td>
</tr>
<tr>
<td>6.4.1.1 Init</td>
<td>60</td>
</tr>
<tr>
<td>6.4.1.2 Release</td>
<td>61</td>
</tr>
<tr>
<td>6.4.1.3 GetNumberOfDevices</td>
<td>62</td>
</tr>
<tr>
<td>6.4.1.4 GetDeviceInfo</td>
<td>63</td>
</tr>
<tr>
<td>6.4.1.5 GetDeviceInterface</td>
<td>64</td>
</tr>
</tbody>
</table>
Part I

ABOUT HAPTIK LIBRARY
Chapter 1

Introduction

The use of haptic devices allows to add new ways of tactile interaction with virtual reality simulations. The growing power of personal computers makes less and less heavy the computational costs associated with haptic rendering, moving to the world an ever larger diffusion of haptic applications and devices (hopefully with more politically-correct costs than today).

1.1 Why another library?

Actually there exists two ways to access a haptic device:

- Using the native API typically written by the manufacturer
- Using generic libraries shipped by either oem or third parties

The former are usually very simple APIs, that expose a limited set of features (basically reading the state of the haptic interface and sending forces back), tied to a particular class of devices from the same manufacturer. On the contrary, the latter are large libraries, untied from a particular manufacturer, and thus able to work with more than one kind of device. Besides haptic rendering, they typically address also graphical and audio related aspects.

Unfortunately both these solutions for hardware access present some drawbacks:

- The use of specific APIs bounds applications to a single device type, or compels developers to use many different APIs, typically very far from each other. What normally happens is that developers have to rewrite some code which works as a common interface to devices from different manufacturers. Moreover, each of this API requires its own device-specific drivers and libraries. Even if an application has been written to use many device types, it cannot run on systems that do not have all of the used APIs installed. For example, an application written to use both Phantom API and Delta API will be statically linked¹ to both specific DLLs. For application to be executed both dlls must be present, otherwise the operating system will raise a runtime link error. Therefore even if application can correctly work with only one of the two supported devices, the running system must have installed both APIs. The naïve solution to this problem is to use runtime linking², which is unfortunately quite demanding for procedural C libraries and almost

¹Static linking means that dependencies on a specific DLL are built inside the executable at compile time. However actual loading of DLL is performed transparently to the application by the operating system at run time.

²With runtime linking it’s up to application at runtime to explicitly load a DLL in memory, take addresses of exposed functions and invoke them through function pointers. This is true especially if the two class hierarchies use different coding conventions.
impossible for C++ class libraries. Generic libraries, on the contrary, provide standardized access to different devices. Moreover they typically fulfil even other requirements of visio/haptic applications, providing rendering primitives, collision detection algorithms and physical models. All the best known libraries of this kind ship in the form of C++ class hierarchy, heavily relying on inheritance and polymorphism to guarantee a certain amount of flexibility and to support callbacks. This structure could lead to some problems: adding haptic rendering to an existing application with its own graphic system requires some effort either to adopt the library’s one or to make the library working with a different graphic system. Moreover if application already has its own class hierarchy, it becomes necessary to merge them: C++ class hierarchy merging is pretty easy to implement, but sometimes can make source code less clear and uniform, and additionally may slow down the compiled code, even for parts apparently not directly involved (e.g. using multiple virtual derivation). Obviously these problems do not arise with scratch-written code, and a visio haptic application can be comfortably built up with only few hours of development. Unfortunately, to the best of our knowledge, no one of these libraries directly support last generation graphic technologies such as programmable GPUs. Therefore, sometimes they have too much stuff; sometimes they do not have enough. Moreover there are some other portability and compatibility issues, common to both kinds of libraries, mainly related to software engineering design. For example, potential modifications to the APIs in future releases of these libraries require to recompile or even to modify the application source code, preventing any form of binary compatibility.

1.2 The Haptik Library

The Haptik Library is a small library with a component based architecture that overcomes these limitations, achieving many benefits such as device independency, driver version transparency and application’s binary compatibility with future devices, SDKs and library versions. It does not contain graphic primitives nor physical algorithms or complex class hierarchies, but instead provides a set of interfaces that hide differences between devices to the applications. All of this is built around a flexible infrastructure of dynamically loadable plugins. The library manages the plugins, and establishes the connection with the application. Each plugin effectively implements the routines necessary for a particular class of devices. This behaviour is implemented with the opaque binary standard mechanism of interfaces from the component model. Therefore applications using Haptik are indifferent to the actual implementation of the device with which they are communicating.

1.2.1 Advantages

Due to this architecture several advantages arise from the usage of the Haptik Library. For the end user:

- Available devices are enumerated by the library and applications can easily let users choose
- The loading of specific dlls is performed transparently at runtime, allowing the same executable to run on systems with different hardware configuration, different driver versions, or even without any physical device
1.2.2 Binary Compatibility

- The fallback loading and default device selection allows the same executable to automatically use the most recent hardware and driver version

For the developer:

- Every device exposes a common interface on which operations supported by any device can be invoked. In this way the same executable can run against every actual and future device
- Every device can expose custom interfaces, supporting specific behaviours and features of that device. So an application can be tuned to easily exploit device peculiarity
- The library is non-invasive. It supports both polling and event driven behaviours. As callbacks it can both invoke procedures and non-virtual methods on any object without requiring inheritance
- The library is portable and same code runs unchanged on Windows and GNU/Linux systems
- The library provides bindings for many languages and can be used not only with C/C++ but also in Matlab and Simulink, as well as from Java applications and applets

1.2.2 Binary Compatibility

All these advantages are guaranteed at binary compatibility level: an application written (and compiled) to run on Haptik Library will always work, on every system, no matter what kind of drivers are installed, no matter if devices are present or not. The same executable will continue to run on newer version of library and plugins, even if new completely different interfaces were introduced, and will work on devices not yet released at application build time.

1.3 Where Applicable

The usage of Haptik Library is thus advisable:

- To add haptic rendering to already existing applications
- To develop high-end visio/haptic applications
- To develop research applications that need an own graphic engine, or already have one of their own
- For developers who desire an easy but powerful low-level access to haptic devices
- As the low-level layer for high-level haptic libraries
Chapter 2

The Haptik Library

This section describes the library internal architecture and behavior, besides the main features of bundled plugins.

2.1 Architecture

The architecture of Haptik can be viewed from two different points of view, the first regarding the subdivision in different dynamic linking libraries with different link times and a second regarding subdivision in components and interfaces. The figure below tries to sketch both:
2.1.1 DLLs

In what follows we will use the terminology of Windows OSes, referring to DLLs (Dynamic Link Libraries). On GNU/Linux Haptik is (being) implemented as Shared Objects. Three are the main entities of the library architecture:

- the Application
- the Library
- the Plugins

The Haptik Library is implemented by a dll (HAPTIK.LIBRARY.DLL) usually linked at loadtime by the application (but this is not mandatory), which expose a simple component-based API that the application can use to enumerate available devices and request their interfaces.

The Plugins are a set of DLLs, not directly seen by the application, that implement the functionality needed to use a specific set of haptic devices (typically all the devices of a particular manufacturer for a particular native API). The library itself acts as a manager: it loads the plugins when needed, make them ready for usage by upper levels, and gathers information in order to organically expose them to application. In addition it provides some centralized services to the plugins.

This runtime-loading scheme offers two main advantages:

- Application is not bound to the presence of particular device driver dlls - only the affected plugin will fail to load and not the whole application
- Future binary compatibility with new devices is achieved - simply adding the related plugin automatically adds support for the new device to existing executables.

2.1.2 Components and Interfaces

Besides the advantages given by the division between library and plugins, with their different loading times and linkage, there are some others given by the adoption of the Component Model. The library, the plugins, even the devices, are all implemented as components. All "communications channels" (Application-Library, Application-Devices and Library-Plugins) are all interface-based. In this way the application can handle all devices in a uniform way, being completely transparent to actual implementation. The same executable runs unchanged on hardware from different manufacturers, or perhaps on completely software-based implementations of virtual devices.

Using interfaces even in the library-plugins communication assure that today-written plugin-dlls will work unchanged on future library releases, even in the presence of major changes. Obviously new plugins could use new completely different communication protocols.
2.2 Behavior

When an application firstly obtains an interface to the library (this is opaque to C++ users if the commodity wrapper class is used) the runtime initializes its internal structures and fetch the configuration settings from the configuration file.

The Configuration file is a per-application text-based file that contains library and plugin settings. In this way each application can use its own set of settings such as, for example, callback frequency, window placement, used plugins and fallback chains. The library first looks for the configuration file in the application current directory. If it can’t find then it looks in the same directory of the Haptik.Library.dll. In this way a single default configuration file can be used for applications that do not need custom settings.

Plugins are loaded in fallback-chains: a plugin could be implemented in different versions that use different SDKs/Drivers. For example, the Phantom42 plugin is implemented with the HDAPI, while Phantom40 and 31 are respectively implemented using GHOST 4.0 and GHOST 3.1. Each one of these three different plugins (for the same class of devices) has its own requirements: a particular hardware-driver version and some specific dlls from the used SDKs. By specifying these three plugin as a fallback chain if Phantom42 cannot be loaded (maybe installed driver is a lower version) than the library falls back to Phantom40 and eventually to Phantom31. In this way the application will always use the newest version of device drivers, thus being able to use the most advanced features, and will not be prevented to run if no driver is available.

Once loaded, the plugins are queried for a list of supported devices, each of them univocally identified by a DEVICE ID. This list of information is made available to the application. Afterwards each request for a device interface from the application is routed towards the correct plugin. The returned interface directly connects the application with the code that effectively handles the device, so Haptik is not traversed for each method invocation.

Internally the library handles plugin reference count to keep loaded in memory only used plugins. When all interfaces to library are released unused plugins are unloaded from memory. When all interfaces to a plugin (i.e. to devices exposed by a plugin) are released that plugin gets unloaded. If new interfaces to library are requested unloaded plugins are automatically reloaded. Whenever all the interfaces are released the library saves the configuration, if required, and release all used resources.

2.2.1 Internal Recursion

Services provided to the application are exposed also to plugins, that is a plugin can use devices exposed by others to implement extended behaviors. The library ensures that a plugin will not receive requests for its own devices. Moreover to avoid infinite recursion a plugin can only access devices exposed from already loaded plugins, i.e. plugins listed before in the configuration file. Usually extension plugins are listed as last.
2.3 Performance Considerations

One may question that such a runtime loading architecture and the use of interface invocations may be cause of a performance penalty, however we state that actually there is no overhead with respect to a statically-linked object-based code. Once an application obtains an interface to a device, the library is completely stepped over and not crossed by interface invocations, which instead jumps directly to the code that handle communication with the hardware device. Moreover dynamic-loading allows a low memory footprint with respect to statically linking multiple libraries. Even the low overhead added by interface invocations does not affect the performance of the haptic loop because interfaces are invoked only during initialization while, in the haptic high-priority thread, communication is performed through callbacks exactly as with any other procedural or object-based library.

In addition, using low-level callback techniques, the Haptik Library does not require any changes to the application class hierarchies, and at each cycle performs a direct address invocation instead of a virtual call as in classic inheritance-based callback techniques.

Even when we compare the use of the Haptik library to the direct use of native SDKs, we can point out that Haptik does exactly what any application directly using native SDKs would do, i.e. calling native methods to get the status of the device (orientation, position, velocity) and to send back forces and torques. The additional operation that Haptik performs is only invoking the application callback passing a pointer to the structure where data from the device has been received. Consequently, the additional computational cost paid to get hardware independency is only an additional function/method invocation.

Therefore we can conclude that applications using the Haptik library run at least as fast as applications using other libraries.
2.4 Configuration File and Library Settings

The configuration file Haptik.Config.txt is a simple text file located in the application current directory or in the same directory of Haptik.Library.Dll. It contains pairs of settings in the form

`Setting = Value`

with white spaces simply ignored. The mandatory part of a configuration file is the plugin list:

```
HaptikLibrary.numberOfPlugins = 6
HaptikLibrary.plugin0_0 = Haptik.Phantom42OH.dll
HaptikLibrary.plugin0_1 = Haptik.Phantom42.dll
HaptikLibrary.plugin0_2 = Haptik.Phantom40.dll
HaptikLibrary.plugin0_3 = Haptik.Phantom31.dll
HaptikLibrary.plugin1_0 = Haptik.Delta.dll
HaptikLibrary.plugin2_0 = Haptik.Remote.dll
HaptikLibrary.plugin3_0 = Haptik.Mobile.dll
HaptikLibrary.plugin4_0 = Haptik.Spectre.dll
HaptikLibrary.plugin5_0 = Haptik.Betamax.dll
```

where the second index is used to specify fallbacks.

Two more Library-related settings are:

```
HaptikLibrary.allowMessageBoxes = FALSE
HaptikLibrary.allowLogging = FALSE
```

The first one prevents the use of message boxes to communicate with the user that can cause problems with fullscreen or poorly written applications. Future version of Haptik will include a mechanism to route messages to user through the application.

The second setting can be used to prevent Haptik from logging to the Haptik.Log.txt file. Usually there is no need to prevent logging as there is no performance penalty related to logging: no logging is performed in the high rate loop except for serious errors. On the contrary many small issues can be usually solved by simply taking a look into the log file. Developers using an integrated debugger also receive all info saved to the log file in the debug output of their environment.

The default for both these settings is TRUE so you can use to them only when actually needed.

**Writing a Config File**

A sample file showing all setting is usually contained in the /Docs directory within the Haptik distribution.

The best way to start is usually by copying the configuration file of one the samples.
2.5 Bundled Plugins

A set of plugin are shipped together with Haptik Library. They support the most common haptic hardware besides some software devices to be used on systems with no haptic device.

2.5.1 Phantom

Phantom42OH, Phantom42, Phantom40 and Phantom31 are plugins for all SensAble PHANToM Premium, Desktop and Omni devices. They are written respectively against HD-API 1.02 (OpenHaptics Academic Edtion, driver version 4.2), HD-API 1.0 (driver version 4.2), GHOST SDK 4.0 (driver version 4.0) and GHOST SDK 3.1 (driver version 3.1), and should be used in a fallback chain. Every 6DoF and 3DoF device is supported on all plugins, except for the PHANToM Omni which requires Phantom42OH/Phantom42 because it only runs with the HD-API. Calibration info for Premium devices are persistent through system restarts, so you only have to initialize them once. For Desktop devices auto recalibration can also be performed periodically by the plugins.

**IMPORTANT NOTE:** The Phantom Device Drivers (PDD) 4.0 and 3.1 DO install all files needed for GHOST based applications (Phantom31/40) to run. The PDD 4.2.XX drivers DO NOT install all files needed for HD-API based applications (Phantom42/42OH) to run but only the SensAble private interface PHANTOMIOLib*.DLL. In order to run HD-API based applications you need also the public interface (HD.DLL) which gets installed with the HD-API SDK or OpenHaptics Toolkit. Redistribution of HD.DLL is prohibited by the HD-API license.

The following settings are used by Phantom* plugins:

- Phantom.LeftHanded = TRUE
- Phantom.MaxForce = 10.000000
- Phantom.WatchdogPeriod = 5000
- Phantom.AutoRecalibratePeriod = 5000
- Phantom.License.VendorName = Your Company Name
- Phantom.License.ApplicationName = Your Application Name
- Phantom.License.Password = Password provided by SensAble

The `AutoRecalibratePeriod` can be used to activate autorecalibration with Desktop models: The Phantom* plugins will periodically check for needed calibration.

**IMPORTANT NOTE:** When autorecalibrating forces are turned off for safety reason. If you don’t move your device’s stylus it will not recalibrate and forces will stay off.

The `License` info are used only by Phantom42OH. They can be used to release applications with the Open Haptics Toolkit Commercial Edition.
2.5.2 Delta

The Delta plugins supports 6DoF and 3Dof Delta and Omega devices from ForceDimension.

The following settings are used by the Delta plugin:

- Delta.LeftHanded = TRUE
- Delta.MaxForce = 10.000000
- Delta.MaxTorque = 10.000000
- Delta.GravityCompensation = TRUE
- Delta.GripperMass = 0.13940000534057617

The *Offset settings can be used to center the workspace.

2.5.3 Freedom


**IMPORTANT NOTE:** the Freedom plugin has not been tested and it could show inverted axis/torques.

Please report any issues or successful usage to info@haptiklibrary.org, problems with the Freedom plugin will be addressed ASAP.

The following settings are used by the Freedom plugin:

- Freedom.LeftHanded = TRUE
- Freedom.MaxForce = 10.000000
- Freedom.MaxTorque = 10.000000

The *Offset settings can be used to center the workspace.
2.5.4 Spectre

Spectre is a plugin that exposes some software-only virtual devices. The Mouse Spectre is a mouse-based device that simulates a PHANToM, and can be used when no actual device is available, or for debugging purpose. Using a common 3-buttons wheeled mouse is possible to control position and orientation of the stylus. Through the context-menu (right clicking) one can choose if the feedback forces, graphically shown, are to be ignored or simulated by “pushing” back device position. The Mouse Spectre is pretty configurable using the configuration file (workspace dimension, callback frequency, scale factors for forces and movements as well as many window-related settings) allowing per-application customization. For these reasons the Mouse Spectre is ideal for student, whereas it is not always possible to arrange for a haptic device. It’s also well suited for professional use, such as testing multi device applications with more than two devices or simply to continue using the same unchanged application even if a haptic device is not currently available.

The Mouse Spectre can use transparency and be overlaid above other windows (usually the application ones). Unfortunately transparency support by both operating systems and videoboard is not as robust as one would expect, therefore some issues must be taken into account when using this feature: When a transparent window refresh it compels all underlying windows to refresh, therefore causing performance hits. Moreover when a transparent window is overlaid above a 3D rendered one a visible flickering can occur. This happens if the client area of the 3D window differs in size from the render target dimension, causing the operating system to perform resampling.

Controls:
LEFT BUTTON - Move on the XY plane (WHEEL move on the XZ plane)
RIGHT BUTTON - Move on the XZ plane (WHEEL move on the XY plane)
MIDDLE BUTTON - stylus button
LEFT + RIGHT BUTTON - Rotate around X and Y axis (WHEEL rotate around Z axis)

The following settings are used by the Spectre plugin:

MouseSpectre.Scale.force = 1.000000
MouseSpectre.Scale.torque = 1.000000
MouseSpectre.Scale.moveX = 1.000000
MouseSpectre.Scale.moveY = 1.000000
MouseSpectre.Scale.moveZ = 1.000000
MouseSpectre.Scale.rotateX = 1.000000
MouseSpectre.Scale.rotateY = 1.000000
MouseSpectre.Scale.rotateZ = 1.000000
2.5.5 Remote

The Remote plugin allows the library to use haptic devices physically connected to remote computers (on which a simple server application must be running, of course) and reachable through TCP/IP based networks. The Remote plugin can be very useful with PCI-based devices which are unpractical (maybe impossible) to move away from their default computer. Using this plugin these devices can be easily made available to other PCs, and the most benefits can be seen using notebooks with low latency WiFi support.

The following settings are used by the Remote plugin:

Remote.Port = 60000
Remote.Address = HapticServer

If the Address setting is not specified the plugin will use a broadcast to search for running server.

2.5.6 Betamax

The Betamax plugin uses internal recursion to add recording and playback capabilities to every device. Once a recording session has been made, timestamped info gets saved to a human readable text file. This file can then be used by a virtual device which reproduces the recorded path.
2.6 Supported Platforms

Haptik Library is developed and tested on Windows XP but starting from version 1.0final it works also on Windows 2000 with minor limitations (lazy unloading of plugins).
Admin privileges are not needed by the library even if some plugins which uses hardware devices could eventually require admin privileges to run.
The Linux port is being developed on a Kubuntu Linux running KDE.
The Java JNI interface has been tested against JRE 1.5.0.
The Matlab mex and the Simulink S-Function have been developed on Matlab 6.5 R13 but users have reported it working also with Matlab 7 R14.
The "(HKLC/HKCU)\Software\Siena Robotics and Systems Lab\Haptik Library" registry key is used (and removed on uninstall). Depending on the type of installation it will be created under the LOCAL_MACHINE or CURRENT_USER root key.
Chapter 3

Using and Extending

3.1 About 0.Xs Versions

Interfaces and structures in Haptik 0.X releases where not final (as clearly stated in the documentation) and they’ve been changed in 1.0. All changes that have been made were aimed to increase overall application robustness: Many devices proved to be not enough robust with regards to calibration and most old devices easily lost calibration. Therefore explicit recalibration facilities has been introduced in the IHaptikDevice interface to allow developers to control calibrations. Luckily enough auto-calibration carried on by plugins is usually enough. A more explicit error management has been added to cope with devices and SDKs faults. Moreover the callback target can now be changed at runtime to allow passing used interfaces to functions for complex operations (for example automatic reference frame alignment on multi-contact applications).

Interfaces and structures in 1.0 are final, that is they will never change. Nevertheless runtime mechanisms have been implemented in the library and structures have been designed in order to allow minor modifications to be included in 1.X versions without breaking code and binary compatibility. Major modifications will be released with new sets of structures and interfaces (i.e. IHaptikDevice2) that will be supported side by side with older ones.

By the way, recompiling 0.X applications is usually enough. The only breaking mod is the Rate() method renamed SetRate() for naming consistency with newly introduced methods.

3.2 Compatibility

The use of Haptik ensures both backward and future binary compatibility:

Backward Compatibility means that applications compiled with a certain version of Haptik will work with every newer version of the library. New APIs will be introduced on new interfaces that will coexist side-by-side with current ones. Applications compiled with a certain version of Haptik “could” eventually run with an older version of the library but this is not guaranteed.

Same rules applies to plugins. Therefore its a good practice to always use the latest release of the library.

Future Compatibility means that any new devices will be supported in old (compiled) applications by just adding the related plugin.
3.3 License

Haptik is free software: it can be used for both academic and commercial purposes at absolutely no cost. It is not in the public domain and the original author keeps its copyright. Haptik is currently released in two separated distributions: Source and Binary.

- The Source Distribution is released under GNU GPL version 2 in the form of C++ sources and is both accessible as milestone snapshots from the haptik library web site or directly from the public subversion repository hosting the development.

- The Binary Distribution is released under a modified MIT license in the form of precompiled binaries (no sources included) downloadable from the haptik web site. It can be used in both commercial and non-commercial products without any restriction and freely redistributed with them.

There is absolutely no difference in functionalities or whatsoever between the two distributions and the spirit of this double form is just to allow using Haptik with projects that can’t be mixed with the GNU GPL license. Eventually Haptik will move to the GNU LGPL license in the future.

3.4 Supporting new Devices

Adding support for a new device (as well as rewriting a better support for current ones) is just a matter of writing and releasing a plugin. Section 5 of this manual contains all info needed to code a new plugin using the C++ language (but almost any language can be used).
Part II

USING HAPTIK LIBRARY
Chapter 4

Application Development

The following topics describe how to use the Haptik Library API with the C++ programming language to develop and use Haptik-enabled applications.

4.1 Project/Environment Setup

Haptik Library ships in the form of a main dll (Haptik.Library.dll), a series of plugins (usually Haptik.PluginName.dll), plus some headers file (application developers are only concerned with Haptik.hpp) and a .lib file (Haptik.Library.lib) built for Visual C++ compilers. Other compilers users can use their instruments to extract a .lib files directly from the DLL (other compilers .lib will be eventually shipped directly in future releases). Borland users, for example, can use the IMPLIB, IMPDEF and COFF2OMF command line utilities.

In order to use Haptik both the include and lib directories from Haptik should be added to the default search directories for the project. It is also safe to add these directories to your IDE global settings because Haptik headers are protected by namespaces and the lib file has a hopefully unique name (Haptik.Library.lib). The installer can perform this steps for you.

The Library and plugins dlls could be put in any directory on the system/user path in order to make them accessible for any application, or released in the application directory. The installer could add the /RSLib/bin directory to the system PATH.

4.2 Including Header and Lib

Application should include the Haptik.hpp header in the following way:

```cpp
#include <RSLib\Haptik.hpp>
using namespace RSLib;
```

All code in the Haptik.hpp header is behind the RSLib namespace, to prevent clashing of names. As explained above the /Include directory must be in the compiler/project default include directories list in order to successfully compile code.

Haptik.hpp also throws-in some fixed-size types (like UINT32 or FLT32). They’re not behind the RSLib namespace as they reflect some already defined windows.h types.
Under Visual Studio the required Haptik.Library.lib is automatically included by a pragma, and does not need to be explicitly added to the project files, as long as you have added the /Lib dir to your compiler/project default lib directories list.

4.3 Hello Haptik Sample

In the /Examples directory of the Haptik Library distribution you can find the sources for a working Hello-World-like sample using the Haptik class and callbacks. You can use this source as a starting point. The following code (a simple command line gravity well) is reported here to give an overview (error handling has been ignored for simplicity) of what a basic Haptik application looks like:

```c
void Callback(HaptikData& data)
{
    data.forceFeedback = - data.position;
}

void main()
{
    Haptik haptik;
    IHaptikDeviceInterface device;

    device = (IHaptikDeviceInterface) haptik.GetDeviceInterface();
    device->Init(Callback);
    device->Start();
    getch();
    device->Release();
}
```
4.4 Accessing the Library (the easy way)

The Haptik Library has a single entry point which is the C function `GetHaptikLibraryInterface(...)` which can be used to retrieve an interface to the library. All the rest of the API is interfaces-based. For C++ developers the library functionalities are wrapped also through the inline Haptik class. This class is the preferred way to use Haptik and there is no reason to do differently.

```cpp
Haptik haptik;
```

Haptik is an inline header-only class which automatically obtains and initializes an interface to the library and stores information on available devices in dynamically allocated local data.

```cpp
for(UINT32 i = 0 ; i<haptik.numberOfDevices ; i++)
{
    // access information contained in haptik.device[i];
}
```

```cpp
IHaptikDeviceInterface device = (IHaptikDeviceInterface) haptik.GetDeviceInterface();
```

The Haptik class is composed of only few lines of code which perform the needed operations that need to be used, and is the preferred way to use the library. Defining Haptik as an inline wrapper class allows keeping the library’s API completely interface-based at binary level while taking advantages of C++ classes.

If the Haptik class is used than there is no need to use the `GetHaptikLibraryInterface` function and the IHaptikLibrary interface.

4.5 Accessing the Library (the hard way)

The entry point to the library is the C function `GetHaptikLibraryInterface(...)` which can be used to obtain an interface to the library component.

```cpp
IHaptikLibraryInterface haptikLibrary;
haptikLibrary = (IHaptikLibraryInterface) GetHaptikLibraryInterface();
```

To initialize internal structures of the library the method `Init(...)` must be invoked.

The two methods `GetNumberOfDevices()` and `GetDeviceInfo(...)` can be used to gather information about all exposed devices.

The method `GetDeviceInterface(deviceld,interfaceld)` can be used to obtain an interface to an actual device.

The `Release()` method must be called when the interface to the library is no more used. Release can be safely called even if some other interfaces to devices are yet in use by the application.

The use of `GetHaptikLibraryInterface` function and the IHaptikLibrary interface is needed only if the C++ Haptik wrapper class is not used. There is no reason for C++ user to not use the Haptik class.
4.6 Getting interfaces to devices

As already sketched out the GetDeviceInterface(...) method is used to request an interface to a particular device. Devices in Haptik library are univocally identified by deviceids. The application can query for a specific device in many ways.

Using the id reported in the device info:

```csharp
IHaptikDeviceInterface device;
device = (IHaptikDeviceInterface) haptik.GetDeviceInterface(id);
if (device == NULL)
{
    //handle error
}
```

By ordinal:

```csharp
device = (IHaptikDeviceInterface) haptik.GetDeviceInterface(i);
```

Using the constant ids defined in haptik.hpp:

```csharp
device = (IHaptikDeviceInterface) haptik.GetDeviceInterface(HAPTIK_MOUSE_SPECTRE);
```

The Haptik.hpp header defines some special ids which can be used to query for the first available device in a certain class. For example the HAPTIK_PHANTOM_DESKTOP id can be specified to obtain access to a PHANToM Desktop device (more ids are described in section 6.3.1).

Simple applications could request HAPTIK_DEFAULTDEVICE (or simply calling GetDeviceInterface with no parameters), which search for the first usable device. In this way if a hardware device cannot be found application could obtain an interface to some always-ready software device with no-user intervention.

GetDeviceInterface currently accepts two parameters: the first is the just explained deviceid/ordinal. The second one is the interface id which allows specifying which kind of interface the application would use to interact with devices. At this time HAPTIK_IHAPTIKDEVICE is the only interface id defined and is passed as default argument.
4.7 Accessing Devices

Data from devices can be obtained in two ways: by callback or polling. Using the callback system the application can specify a procedure or an object/method pair that will be invoked by the device at a constant rate. Using polling the application uses Read(...) and Write(...) methods to get data from device and to send forces back.

4.7.1 Using Callback

If you chose to use the event driven way when you init the device you should configure the callback system. You can use a classic procedural scheme:

```c
void Callback(HaptikData& data)
{
    if (data.button)
    {
        data.forceFeedback = - data.position;
    }
}
```

```c
device->Init(Callback);
```

or an object/method invocation scheme:

```c
class MyObject
{
    private:
    INT32 someValue;
    ...

    public:
    VOID HaptikCallback (HaptikData& data)
    {
        if (data.button)
        {
            data.forceFeedback = - data.position;
        }
    }
    INT32 OtherMethod(...);
    ...
};

MyObject object;
device->Init(METHOD_ADDRESS(MyObject,HaptikCallback),
                OBJECT_ADDRESS(MyObject,object));
```

The callback procedure/method receive from the device the address of a HaptikData structure. To get the address of a method you can use the METHOD_ADDRESS macro defined in the RSLib/LowLevel.hpp header shipped with the Haptik Library. It compiles easily on Visual C++.
4.7.1.1 Callbacks

The usage of direct method invocations gives the advantage of not requiring inheritance and polymorphism to use methods as callbacks. On the other side, completely overtaking the language type checking requires particular attention: when calling a procedure the caller must clean the stack (stdcall), while when calling a method the this pointer should be passed in the ecx register and parameters on the stack (thiscall). Luckily enough these are the default behavior of almost all C++ compilers and so usually there is no need to care about these details except for very old compilers.

4.7.1.2 User Data

Beginning with version 1.0 the Callback can take an additional argument containing user data and asynchronous error reports.

VOID HaptikCallback (HaptikData& data, HaptikUserData& udata);

You can use the SetUserData() method of the IHaptikDevice interface to pass a 32bit value (usually a pointer to something else) to your callback. While this is usually unneeded for object/method callbacks (which implicitly carries additional information) this allows to use a single procedural callback with multiple devices, performing different actions based on the content of the user data. The use of user data is optional, and old style single-argument callbacks will continue working. The HaptikUserData struct is also used to report errors that could happen asynchronously. Non-naive applications must check the lastError field before any other operation.

4.7.2 Using Polling

If you are not going to use the event driven callback system, typically because you already have a haptic thread, you should call Init with no arguments and use Read and Write methods:

device->Init();

...

VOID HapticThreadCode()
{
    HaptikData data;
    device->Read(data);

    // COMPUTE FORCE FEEDBACK
    ...

    device->Write(data);
}
4.8 Using Devices

To begin using a device, or stopping it temporarily, the methods Start() and Stop() are called. Using the event driven way, when you call Start the callback begins being called at constant rate. Please remember that even if you’re using polling to access device you should call Start() and Stop(). Start and Stop enforce internal ordering so you can even call methods not in pairs, and redundant calls are simply ignored.

4.9 Releasing Interfaces

When your application have terminated using the device it should release the interface by calling Release() to free any used resource and make device available again.

device->Release();
device = NULL;

Once released an interface should not be referred any more because it can point to unallocated memory, so it is a good practice to set it to NULL. You can use the RELEASE_HAPTIK_INTERFACE macro, from the Haptik headers, to safely release and null an interface.

#define RELEASE_HAPTIK_INTERFACE(ptr)\  
  if (ptr != NULL){ptr->Release(); ptr = NULL;}

PLEASE NOTE: ANY INTERFACE MUST ALWAYS BE RELEASED.
4.10 Exchanging Data with the Device

The HaptikData struct is used to exchange data with the device.

```cpp
void Callback(HaptikData& data)
{
    data.forceFeedback = - data.position;
}
```

Some fields of HaptikData are for reading info such as position, velocity, orientation; some others can be used to send data to the device, typically the force feedback and torque. Other fields are reserved. These are currently used for alignment and will be eventually used in future minor releases to extend the API (for example adding angular velocities to be passed back) without breaking binary compatibility.

The HaptikData struct is not mandatory and can be seen as a template. Because callbacks are not type checked you can define your own version of HaptikData (obviously preserving fields’ order and memory layout) to use matrix/vector classes of your choice. Haptik is not intended to provide math support, and the include /Math/3D.hpp header contains only basic functionalities. Non naïve applications should not rely on these classes. More info on this subject can be found in HaptikData Struct section (6.5.2) of the Haptik Library Reference.

```cpp
void CallbackProc(MyHaptikData& data);
class MyHaptikData
{
    public:
    // Output from Device
    MyMatrix4x4 matrix;
    MyVector3 position;
    MyVector3 velocity;
    UINT32 buttonMask;

    private:
    DWORD reserved0[1];

    // Input to Device
    MyVector3 forceFeedback;
    MyVector3 torqueFeedback;

    private:
    DWORD reserved1[2];

    public:
    operator HaptikData&(){return *(HaptikDataPtr)this;}
};
```

A HaptikData reference operator allows to use the class also with the type checked Read and Write methods.
4.11 Measure Units

All vector data are specified in the coordinate system configured for that plugin (see 4.12). The matrix field is row-major (just like standard C/C++ matrices), with translation (i.e. position) specified in the last row \((m_{30}, m_{31}, m_{32})\), therefore it can be used directly both with DirectX and OpenGL (they follow different conventions that luckily result in the same binary layout for matrices) as the world matrix for the stylus.

\[
\begin{pmatrix}
axis X \\
axis Y \\
axis Z \\
\text{position}
\end{pmatrix}
\begin{pmatrix}
0 \\
0 \\
0 \\
1
\end{pmatrix}
\]

4.11 Measure Units

All length units are in millimeters, while forces are expressed in Newtons. Torques are in Newton * millimeters.

4.12 Coordinate Systems

In order to being easily usable with any other libraries both coordinate systems used by the major graphics APIs are usable. Each plugin can be configured for the coordinate system to use in the configuration file (in a future version this will be accessible directly from the code):

Right Handed (X axis goes right, Y axis goes up, Z axis goes from screen to user) as in OpenGL.

Left Handed (X axis goes right, Y axis goes up, Z axis goes from user to screen) as in DirectX.
Chapter 5

PlugIn Development

The following topics describe how to use the internal Haptik Library PlugIn API with the C++ programming language, to develop device plugins for Haptik Library. The examples are referred to Windows DLL development.

5.1 Headers & Libs

Plugins includes the HaptikPlugin.hpp header in the following way:

#include <RSLib/HaptikPlugin.hpp>
using namespace RSLib;

This file contains interface definitions, defined device constants and commodity macros. It also throws in Haptik.hpp in which are defined the user level interfaces such as IHaptikDevice, and the structures to be exposed to the user application.

5.2 Sample Plugin Projects

The /Examples directory in the Haptik Library distribution contains Visual Studio .NET sample plugin projects which can be used as a template. After reading this section you can take a look directly at sources for bundled plugins or start by recompiling the sample plugins.
5.3 Building DLLs

A Haptik Plugin DLL must export a single C-linkage function named GetInterface. This is the entry procedure that Haptik Library calls from a plugin dll to request an interface to the plugin component. Following is Visual C++ Example Code:

```c
extern "C" __declspec(dllexport) PVOID GetInterface(UINT32 interfaceId)
{
    if (interfaceId == HAPTIK_IHAPTIKPLUGIN) return &pluginComponent;
    else return NULL;
}
```

This function should return an interface of the request type.
5.4 Building Plugins

In order to support future compatibility and os-transparency, plugins must be built as components. In this way the DLL windows-specific mechanism can be replaced by any other os-specific way such as shared objects under UNIX like systems. After loading the dll the library will use the GetInterface function to request an IHaptikPlugin interface through which actual communication with plugin is carried on. Therefore the component plugin must expose the IHaptikPlugin interface. The easiest way to implement this is through a singleton instance of an IHaptikPlugin derived class, as in the following sample code:

```cpp
class PhantomPlugin : public IHaptikPlugin
{
  private:
    IHaptikLibraryLowerInterface iHaptikLibrary;
    UINT32 pluginId;
    //more plugin persistent state data...

  // IHaptikPlugin interface
  public:
    UINT32 Init(IHaptikLibraryLowerInterface inInterface, UINT32 inPluginId);
    UINT32 Release();
    PVOID GetDeviceInterface(UINT32 deviceId, UINT32 interfaceId);
    UINT32 GetDeviceInfo(UINT32 deviceId, HaptikDeviceInfoPtr pDev);
    UINT32 GetNumberOfDevices();
    UINT32 GetPluginInfo(HaptikPluginInfoPtr pInfo);

  private:
    //plugin internal methods
};
```

After loading the plugin, the library will call Init passing an interface to itself and a runtime plugin identifier. This id must be used by the plugin to build device ids. In this way more than one plugin could expose the same actual device but using each one its unique device id. Haptik library routes device requests based on this plugin id so it’s mandatory that a plugin correctly use it.
5.5 IDs

To the application devices exposed by the Library are univocally identified by a `device id`. A device id is composed (from least significant bits to most significant) by a 16-bit hardware id, an 8-bit plugin id, a 5-bit number, a 3-bit flags:

- The hardware id univocally identifies a particular device. For example all phantom desktop devices have the same hardware id.
- As previously stated the plugin id is used to differentiate between plugins exposing devices from the same class.
- The 5-bit number can be used by the plugin to discriminate between its devices from the same classes.
- The first (most significant) flags bit is used to distinguish real device ids from simple ordinal requests. The other two are reserved for future use.

Plugin developers could use the `DEVICEID(number,pluginid,hardwareid)` macro to build an actual device id.

5.6 Ordinals

All methods that accept a device id must also accept an ordinal. This is needed to request info (or interfaces) during enumeration, when the devices' ids are not yet known.
5.7 Accessing the Library from Plugins

The Haptik Library exposes itself to lower levels (plugins) through the IHaptikLibraryLower interface. This interface exposes three sets of methods.

PCHAR8 GetVariable(PCHAR8 name);
UINT32 SetVariable(PCHAR8 name, PCHAR8 value);

The first set can be used by the plugin to access stored configuration values both from configuration file and centralized registry.

UINT32 Lock();
UINT32 Unlock();

The second set of methods must be used by the plugin to communicate to library when it can be unloaded or not. Plugins should maintain internal reference count. When they gave the first interface to a device they should lock themselves to the library. Once the last interface to a device is released the plugin should unlock itself thus allowing the library to unload it from memory.

UINT32 GetNumberOfDevices();
UINT32 GetDeviceInfo(UINT32 deviceId, HaptikDeviceInfoPtr pDev);
PVOID GetDeviceInterface(UINT32 deviceId, UINT32 interfaceId);

The third set of methods allows for internal recursion, that is a plugin can access and re-expose devices from other plugins. These methods behaves exactly as their counterparts to the upper level (application) but the calling plugin is automatically excluded by the library, that is a plugin will never receive a request it has made. Using these methods a plugin could extend (enhanced debugging and logging, complex force filtering and cutoff, graphical output, combining more devices) the capabilities and behaviour of already supported devices without actual modifying the original exposing plugin.
5.8 Building Devices

Each device-component must encapsulate the code and data to use an actual device, and expose itself to users (either application or other plugins) through interfaces (at least IHaptikDevice). The simplest way to build plugins is to use the object-oriented approach implementing each device as a class, and the plugin component only concerned with device enumeration and management as in the following sample code:

```cpp
class MyHapticDevice : public IHaptikDevice
{
    private:
    // device specific data...

    public:
    // all methods from the IHaptikDeviceInterface

    private:
    // device specific functions and utilities...
};
```

It is essential that device-code sees its plugin-singleton in some way, in order to issue increments/decrements of the reference count when interfaces are obtained/released.
5.9 Invoking Callbacks

When the device should invoke the callback for the application, plugin MUST use the provided function from the HaptikPlugin.hpp header:

```c
void InvokeCallback(PVOID methodOrProc, PVOID object, PVOID argument1, PVOID argument2);
```

argument1 should be a pointer to a HaptikData structure and argument2 a pointer to a HaptikUserData structure as in the following:

```c
if (eventDriven) InvokeCallback(method, object, &data, &userdata);
```

5.10 Coordinate Systems

In order to being easily usable with other libraries it's mandatory that exposed devices support both coordinate systems (left handed used by DirectX, right handed used by OpenGL) based on a per-application configuration setting, converting when needed vectors and matrices from the device native API coordinate system.
Part III

HAPTIK LIBRARY REFERENCE
Chapter 6

C++ Application Developer’s Reference

The following topics provide reference information about using the Haptik Library API with the C/C++ programming language:

- Classes
- Functions
- Constants
- Interfaces
- Structures
6.1 Classes

The Haptik Library C++ API provides the following classes:

- Haptik
6.1.1 Haptik

The Haptik class is an inline wrapper which provides fields and methods to enumerate and access installed devices.

**Fields**

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UINT32</td>
<td>numberOfDevices</td>
<td>Number of available devices.</td>
</tr>
<tr>
<td>IHaptikLibraryInterface</td>
<td>library</td>
<td>An interface to the library.</td>
</tr>
<tr>
<td>HaptikDevicePtr</td>
<td>device</td>
<td>An array of HaptikDeviceInfo structs containing info about installed devices.</td>
</tr>
</tbody>
</table>

**Methods**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetDeviceInterface</td>
<td>Retrieve an interface to a device.</td>
</tr>
<tr>
<td>RefreshDeviceList</td>
<td>Updates the device list.</td>
</tr>
<tr>
<td>ResultToString</td>
<td>Convert Return Values to Human-Readable ANSI Strings.</td>
</tr>
</tbody>
</table>
6.1.1.1 GetDeviceInterface

The GetDeviceInterface method retrieves an interface to a haptic device.

```cpp
PVOID GetDeviceInterface(
    UINT32 deviceId = HAPTIK_DEFAULT_DEVICE,
    UINT32 interfaceId = HAPTIK_IHAPTIKDEVICE
);
```

**Parameters**

**deviceId**

[in] A device identifier or an ordinal. The Id can be taken from a HaptikDeviceInfo struct or a default value (see 6.3.1). The ordinal is relative to the device array.

**interfaceId**

[in] an interface identifier.

**Return Values**

If the function succeeds, the return value is an interface of the requested type. If the function fails, the return value is NULL. This function typically fails for one of the following reasons:

- an invalid deviceId
- an ordinal deviceId greater or equal to numberOfDevices
- the device is not ready
- the device is already in use

**Remarks**

Invocations of Haptik::GetDeviceInterface are routed to IHaptikLibrary::GetDeviceInterface. This method internally updates the device list.
6.1.1 Haptik

6.1.1.2 RefreshDeviceList

The **RefreshDeviceList** method updates device list.

```c
UINT32 RefreshDeviceList();
```

**Parameters**

*None*

**Return Values**

On success return value is **HAPTIK_SUCCESS**. If interface to the library has not been correctly obtained return value is **HAPTIK_UNSUPPORTED**.

**Remarks**

This method is useful for updating the readiness of a device in the list. It should be called when some device interfaces have been released. Internally values are always updated, so there is no danger of getting an interface to a busy device when requesting a default.
6.2 Functions

**Haptik.Library.dll** exposes the following functions:

- GetHaptikLibraryInterface
6.2.1 GetHaptikLibraryInterface

The **GetHaptikLibraryInterface** function must be used to obtain an interface to the Haptik Library. If the Haptik wrapper class is used then there is no need to call this function.

```c
extern "C" PVOID GetHaptikLibraryInterface(
    UINT32 interfaceId=HAPTIK_IHAPTIKLIBRARY
);
```

**Parameters**

`interfaceId`

[in] a 32 bit constant identifying the desired interface. Currently the only defined id is HAPTIK_IHAPTIKLIBRARY.

**Return Values**

If the library supports the requested interface it returns an interface of that kind. Otherwise it returns NULL.

**Remarks**

This function always succeeded and is used for future compatibility. Currently the only interface defined is the IHaptikLibrary. Future releases of Haptic Library could support other interfaces for extended interaction. See also Haptik class (6.1.1).
6.3 Constants

The Haptik Library defines the following constants.

6.3.1 Default Devices

These constants can be used as arguments to `GetDeviceInterface(...)` to request the first available device of a certain class:

- `HAPTIK_DEFAULT_DEVICE`
  Indicates any available device.

- `HAPTIK_MOUSE_SPECTRE`
  Indicates the Mouse Spectre software device

- `HAPTIK_BETAMAX_PLAYER`
  Indicates the Betamax Player software device

- `HAPTIK_BETAMAX_RECORDER`
  Indicates the Betamax Recorder software device using the default device

- `HAPTIK_PHANTOM DESKTOP`
  Indicates an available PHANToM Desktop device.

- `HAPTIK_PHANTOM PREMIUM`
  Indicates an available PHANToM Premium device.

- `HAPTIK_PHANTOM OMNI`
  Indicates an available PHANToM Omni device.

- `HAPTIK_DELTA`
  Indicates an available Delta device.

- `HAPTIK_OMEGA`
  Indicates an available Omega device.

- `HAPTIK_FREEDOM6S`
  Indicates an available Freedom6S device.

- `HAPTIK_CUBIC`
  Indicates an available Cubic device.
6.3.2 Return Values

All return values across library and plugins are 32 bits unsigned integers (UINT32). Bit 31 (most significant bit) indicates severity error (thus value below 0x80000000 indicates success, while values above indicates failure). You can use Windows already defined macros as SUCCEEDED(result) and FAILED(result), which are redefined also in the Haptik included headers, to check for severity errors.

HAPTIK_SUCCESS
  Operation successfully completed.

HAPTIK_ALREADY
  Not performed because not needed (such as stopping a stopped device).

HAPTIK_ERROR
  Generic error.

HAPTIK_SDK_ERROR
  A device specific SDK error happened.

HAPTIK_MEMORY_ERROR
  A memory-related generic error.

HAPTIK_UNSUPPORTED
  Unsupported operation, device, interface.

HAPTIK_OUT_OF_RANGE
  Ordinal deviceId out of range.

HAPTIK_NO_SUCH_DEVICE
  There is no device matching the requesting id.
6.4 Interfaces

Haptik Library defines the following interfaces:

- IHaptikLibrary
- IHaptikDevice
6.4.1 IHaptikLibrary

The IHaptikLibrary interface exposes functionality of current generation of the Haptik Library. The Haptik class (6.1.1) is a convenient inline wrapper which hides the usage of this interface, therefore if the Haptik class is used no IHaptikLibrary interface needs to be directly used.

Methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Init</td>
<td>Initialize library’s internal structures.</td>
</tr>
<tr>
<td>Release</td>
<td>Release the interface.</td>
</tr>
<tr>
<td>GetNumberOfDevices</td>
<td>Returns the number of all available devices.</td>
</tr>
<tr>
<td>GetDeviceInfo</td>
<td>Retrieves info about a device.</td>
</tr>
<tr>
<td>GetDeviceInterface</td>
<td>Retrieves an interface to a device.</td>
</tr>
<tr>
<td>GetNumberOfPlugins</td>
<td>Retrieves the number of all loadable devices.</td>
</tr>
<tr>
<td>GetPluginInfo</td>
<td>Retrieves info about a plugin.</td>
</tr>
<tr>
<td>GetPluginInterface</td>
<td>Retrieves an interface to a plugin.</td>
</tr>
<tr>
<td>GetLibraryInfo</td>
<td>Retrieves info about the library.</td>
</tr>
</tbody>
</table>
6.4.1.1 Init

The Init method initializes the library.

```c
UINT32 Init(
    PVOID reserved = NULL
);
```

Parameters

reserved

[in] reserved for future use to pass arguments to library.

Return Values

If the function succeeds, the return value is HAPTIK_SUCCESS. Otherwise is HAPTIK_ERROR.

Remarks

An application must call Init for every interface to library it uses. Internal reference count prevents multiple initializations of internal data structures. The interface to the library must be always released even if failed to initialize as for any other interface.
6.4.1.2 Release

The Release method decrements internal reference count and eventually frees up all used memory resources.

\[ \text{UINT32 Release( } \) \]

Parameters
None

Return Values
This method always return HAPTIK_SUCCESS.

Remarks
The interfaces to library must be always released even if failed to initialize.
6.4.1.3 GetNumberOfDevices

The **GetNumberOfDevices** method retrieves the total number of devices.

```c
UINT32 GetNumberOfDevices(
);
```

**Parameters**

*None*

**Return Values**

This function always succeeded. The return value is the number of devices exposed by all loadable plugins.
6.4.1.4 GetDeviceInfo

The GetDeviceInfo method retrieves information about a device.

```c
UINT32 GetDeviceInfo(
    UINT32 deviceId,
    HaptikDeviceInfoPtr pInfo
);
```

**Parameters**

- `deviceId`
  
  [in] a device id or an ordinal.

- `pInfo`
  
  [out] a pointer to a HaptikDeviceInfo that will be filled with info about device.

**Return Values**

If the function succeeds, the return value is HAPTIK_SUCCESS. If deviceId is an ordinal return value can be HAPTIK_OUT_OF_RANGE. If no device has the requested id return value is HAPTIK_NO_SUCH_DEVICE.
6.4.1.5 GetDeviceInterface

The GetDeviceInterface method retrieves an interface to a device.

```c
PVOID GetDeviceInterface(
    UINT32 deviceId = HAPTIK_DEFAULTDEVICE,
    UINT32 interfaceId = HAPTIK_IHAPTIKDEVICE
);
```

**Parameters**

- `deviceId`
  
  [in] a device id or an ordinal.

- `interfaceId`
  
  [in] an interface id.

**Return Values**

If the function succeeds, the return value is an interface of the request type to the requested device. If the function fails the return value is NULL.
6.4.1.6 GetNumberOfPlugins

The GetNumberOfPlugins method retrieves the number of plugins.

```c
UINT32 GetNumberOfPlugins()
```

**Parameters**

*None*

**Return Values**

This function always succeeded. The return value is the number of loadable plugins.

**Remarks**

GetNumberOfPlugins is reserved for future.
6.4.1.7 GetPluginInfo

The **GetPluginInfo** method retrieves information about a plugin.

```c
UINT32 GetPluginInfo(
    UINT32 pluginId,
    HaptikPluginInfoPtr pInfo
);
```

**Parameters**

- **pluginId**
  
  [in] an ordinal plugin id.

- **pInfo**
  
  [out] a pointer to a HaptikPluginInfo that will be filled with info about the plugin.

**Return Values**

If the function succeeds, the return value is HAPTIK_SUCCESS. Otherwise HAPTIK_OUT_OF_RANGE can be returned.

**Remarks**

This method is reserved for future use.
6.4.1.8 GetPluginInterface

The **GetPluginInterface** method retrieves an interface to a plugin.

```c
PVVOID GetPluginInterface(
    UINT32 pluginId,
    UINT32 interfaceId = HAPTIK_IHAPTIKPLUGIN
);
```

**Parameters**

`pluginId`

[in] an ordinal plugin id.

`interfaceId`

[in] an interface id.

**Return Values**

If the function succeeds, the return value is an interface of the requested type to the requested plugin. NULL is returned if `pluginId` is out of range, or the requested interface in not supported.

**Remarks**

*GetPluginInterface* is reserved for future use and currently unimplemented. It always returns NULL.
6.4.1.9 GetLibraryInfo

The GetLibraryInfo method retrieves information about the library.

```c
UINT32 GetLibraryInfo(
    HaptikLibraryInfoPtr pInfo
);
```

**Parameters**

*pInfo*

[out] the address of a HaptikLibraryInfo that will be filled with info about the Library.

**Return Values**

If the function succeeds, the return value is HAPTIK_SUCCESS. Otherwise HAPTIK_ERROR can be returned.
6.4.2 IHaptikDevice

The *IHaptikDevice* interface represents a generic 6DoF device. It is exposed by all devices and can be used to uniformly access existing and future devices.

**Methods**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Init</td>
<td>Initialize device and callback system</td>
</tr>
<tr>
<td>Release</td>
<td>Release the interface</td>
</tr>
<tr>
<td>Start</td>
<td>Start the device</td>
</tr>
<tr>
<td>Stop</td>
<td>Stop the device</td>
</tr>
<tr>
<td>Read</td>
<td>Read data from the device (polling)</td>
</tr>
<tr>
<td>Write</td>
<td>Send data to the device (polling)</td>
</tr>
<tr>
<td>SetRate</td>
<td>Set the rate of the device</td>
</tr>
<tr>
<td>GetRate</td>
<td>Get the rate of the device</td>
</tr>
<tr>
<td>SetUserData</td>
<td>Set user data to be passed to callback</td>
</tr>
<tr>
<td>GetUserData</td>
<td>Retrieve currently set user data</td>
</tr>
<tr>
<td>SetCallback</td>
<td>Set the new callback</td>
</tr>
<tr>
<td>GetCallback</td>
<td>Get the currently set callback</td>
</tr>
<tr>
<td>GetLastError</td>
<td>Get the last asynchronous error</td>
</tr>
<tr>
<td>GetInfo</td>
<td>Retrieves info about device</td>
</tr>
<tr>
<td>GetCalibrationStatus</td>
<td>Get calibration status</td>
</tr>
<tr>
<td>Recalibrate</td>
<td>Recalibrate the device</td>
</tr>
</tbody>
</table>
6.4.2.1 Init

The `Init` method initializes the device.

```c
UINT32 Init(
    PVOID methodOrProc = NULL,
    PVOID object = NULL
);
```

**Parameters**

`methodOrProc`

[in] the address of a procedure, or a method. If object is NULL, it’s interpreted as a procedure, else as a method. If NULL is passed the application will use polling to access device.

`object`

[in] a pointer to the object on which the method will be invoked. Default to NULL if procedure based callback is to be used.

**Return Values**

If the function succeeds, the return value is HAPTIK_SUCCESS. Otherwise return value can be HAPTIK_SDK_ERROR if the error has been reported by specific device SDK, or HAPTIK_MEMORY_ERROR for a memory-related generic error.

**Remarks**

An application using the callback system will use `Init()` to configure the callback. Otherwise if polling is used than `Init()` must be invoked with no arguments, and the `Read()` and `Write()` methods should be used for accessing the device instead of the callback. Callback system can be reconfigured at any time using the `SetCallback()` method.

Using direct method invocations the Library can perform object/method callbacks without requiring the use of inheritance. Perhaps, in this way, language type checking is completely bypassed and so it requires developer’s attention. The passed procedure must be stdcall, with the caller cleaning the stack. The passed method, must be thiscall, with the callee cleaning the stack, the this pointer passed in the ecx register, and other arguments loaded on the stack. Fortunately these are the default conventions for procedures and methods compiled with Microsoft Visual C++ 6.0/.NET. Please check your project/compiler setting.

A device must be successfully initialized before any other method (obviously except releasing the interface) can be called, otherwise results are unpredictable. The interface to the device must be always released even if failed to initialize.
6.4.2.2 Release

The Release method frees all the allocated resources and make device available again.

```c
UINT32 Release();
```

**Parameters**

None

**Return Values**

This method always succeeded.

**Remarks**

The interface to device must be always released even if failed to initialize. If the interface is not released, internal reference count is not decrement causing device to remain busy and the library unable to unload the exposing plugin.
6.4.2.3 Start

The Start method activates the device.

```
UINT32 Start();
```

**Parameters**

None

**Return Values**

This function always succeeded. If device is already running the return value is `HAPTIK_ALREADY`, otherwise `HAPTIK_SUCCESS`.

**Remarks**

If using the callback system, calling Start() makes the callback specified with Init() invoked periodically at the configured rate. If using polling, calling Start() allows Read() and Write() methods to be successfully invoked. Start and Stop do not need to be called in pair. A Start invocation on an already running device simply does nothing.
6.4.2.4 Stop

The **Stop** method suspends the device.

```c
UINT32 Stop();
```

**Parameters**

*None*

**Return Values**

This function always succeeded. If device is already running the return value is HAPTIK_ALREADY, otherwise HAPTIK_SUCCESS.

**Remarks**

If using the callback system, calling **Stop** suspend invocation of the callback. **Start** and **Stop** do not need to be called in pair. A **Stop** invocation on an already stopped device simply does nothing.
6.4.2.5 Read

The **Read** method gets data from the device.

```c
UINT32 Read(
    HaptikData& data
);
```

**Parameters**

- **data**
  - [out] a HaptikData struct that receives data from the device.

**Return Values**

If the function succeeds, the return value is HAPTIK_SUCCESS.

**Remarks**

Only the read-related part of the HaptikData struct is updated. Write-only part is left unchanged. If device is stopped **Read** may fail depending on device and plugin implementation.
6.4.2.6 Write

The **Write** method sends data to the device.

```c
UINT32 Write(
    HaptikData& data
);
```

**Parameters**

*data*

[in] a HaptikData struct that contains data to be sent to the device.

**Return Values**

If the function succeeds, the return value is HAPTIK_SUCCESS.

**Remarks**

Only the write-related part of the HaptikData struct is used. Read-only part is ignored. If device is stopped **Write** may fail depending on device and plugin implementation.
6.4.2.7  SetRate

The SetRate method set the rate of the device.

```cpp
UINT32 SetRate(
    FLT32 rateInHz
);
```

Parameters

rate

[in] the desired rate in Hz.

Return Values

If the function succeeds, the return value is HAPTIK_SUCCESS.

Remarks

This function sets the rate at which the device internal state is update and, if used, callback is invoked. Not every rate can be set and not all devices’ SDK can modify this value. Depending on implementation the rate can be simulated even on devices/SDKs that do not support it. Usually rate granularity is in milliseconds, that is only rates for which the period is an integral number of milliseconds can be set, and therefore the rateInHz parameter is usually clamped down to the highest valid rate. This behaviour is plugin specific. For these reasons an application should use GetInfo before calling Rate to get the Max/Min rates possible, and GetRate after to retrieve the new applied rate.
6.4.2.8 GetRate

The GetRate method gets the current rate of the device.

```c
FLT32 Rate()
```

**Parameters**

*None*

**Return Values**

The function returns the current of the device. This function will always succeed.
6.4.2.9  SetUserData

The SetUserData method sets a 32 bit value to be passed to the callback.

```
UINT32 SetUserData(
    DWORD userData
);`

**Parameters**

*userData*

[in] a 32 bit value to be passed to the callback.

**Return Values**

If the function succeeds, the return value is HAPTIK_SUCCESS.

**Remarks**

Applications can use the user data to pass a pointer to a device related data structure. In this way a single procedural callback can be used to handle different devices with different behaviors.
6.4.2.10 GetUserData

The GetUserData method retrieves the currently set user data.

```
DWORD GetUserData();
```

**Parameters**

*None*

**Return Values**

The currently set user data.
6.4.2.11 SetCallback

The SetCallback method reconfigures the callback system.

```c
UINT32 SetCallback(
PVOID methodOrProc = NULL,
PVOID object = NULL,
DWORD reserved = NULL,
);
```

**Parameters**

*methodOrProc*

[in] the address of a procedure, or a method. If object is NULL, it’s interpreted as a procedure, else as a method. If NULL is passed the application will use polling to access device.

*object*

[in] a pointer to the object on which the method will be invoked. Default to NULL if procedure-based callback is to be used.

*reserved*

[in] reserved for future use. Must be NULL.

**Return Values**

If the function succeeds, the return value is HAPTIK_SUCCESS.

**Remarks**

The reserved field will be used in future minor release to allow for additional asynchronous communications (error reporting, user messages, etc).
6.4.2 IHaptikDevice

6.4.2.12 GetCallback

The GetCallback method retrieves current configuration of the callback system.

```c
UINT32 GetCallback(
PVOID& methodOrProc,
PVOID& object,
DWORD reserved,
);
```

Parameters

`methodOrProc`

[out] the address of variable that receives current set callback.

`object`

[out] optional. Address of a variable that receive the current object pointer. Pass a NULL_REFERENCE if not required.

`reserved`

[in] reserved.

Return Values

If the function succeeds, the return value is HAPTIK_SUCCESS.

If the callback is object/method based but the object parameter is a NULL_REFERENCE than HAPTIK_WARNING is returned.

Remarks

The reserved field will be used in future minor release to allow for additional asynchronous communications (error reporting, user messages, etc).
6.4.2.13 GetLastError

The GetLastError method retrieve the last asynchronous error and then clears it.

```c
UINT32 GetLastError();
```

**Parameters**

*None*

**Return Values**

Last asynchronous error or HAPTIK_SUCCESS if none has occurred.

**Remarks**

The GetLastError method can be used with polling-based applications to check for asynchronous errors.
6.4.2.14 GetCalibrationStatus

The GetCalibrationStatus method gets the current calibration status of the device.

```c
UINT32 GetCalibrationStatus()
```

**Parameters**

*None*

**Return Values**

HAPTIK_CALIBRATED or HAPTIK_NOTCALIBRATED.
### 6.4.2.15 Recalibrate

The Recalibrate method tries to recalibrate the device.

```c
UINT32 Recalibrate(
    UINT32 reserved = NULL);
```

**Parameters**

- `reserved`
  
  [in] reserved for future use.

**Return Values**

If the function succeeds, the return value is HAPTIK_SUCCESS.

**Remarks**

The Calibration procedure is specific to each device. Usually a MessageBox will used to ask the user to perform needed actions (such as holding stylus in a particular position).
6.4.2.16 GetInfo

The GetInfo function retrieves info on the device.

```c
UINT32 GetInfo(
    HaptikDeviceInfo& info
);
```

Parameters

`info`

[out] a structure that will receive info.

Return Values

If the function succeeds, the return value is HAPTIK_SUCCESS.
6.5 Structures

Haptik Library defines the following structures:

- HaptikDeviceInfo
- HaptikData
- HaptikUserData
- HaptikPluginInfo
- HaptikLibraryInfo
6.5.1 HaptikDeviceInfo

The HaptikDeviceInfo struct contains information about a device.

Fields

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UINT32</td>
<td>id</td>
<td>The device unique identifier.</td>
</tr>
<tr>
<td>PCHAR8</td>
<td>name</td>
<td>A plugin-allocated string with the name of the device.</td>
</tr>
<tr>
<td>PCHAR8</td>
<td>model</td>
<td>A plugin-allocated string with the model of the device.</td>
</tr>
<tr>
<td>PCHAR8</td>
<td>manufacturer</td>
<td>A plugin-allocated string with the oem of the device.</td>
</tr>
<tr>
<td>BOOL32</td>
<td>ready</td>
<td>A Boolean value describing if the device can be used.</td>
</tr>
<tr>
<td>FLT32</td>
<td>currentRate</td>
<td>The current update/callback rate of the device.</td>
</tr>
<tr>
<td>FLT32</td>
<td>defaultRate</td>
<td>The default update rate of device.</td>
</tr>
<tr>
<td>FLT32</td>
<td>maxRate</td>
<td>The maximum rate of the device.</td>
</tr>
<tr>
<td>FLT32</td>
<td>minRate</td>
<td>The minimum rate supported by the device.</td>
</tr>
<tr>
<td>FLT32</td>
<td>maxX</td>
<td>Workspace limit in the positive x axis direction.</td>
</tr>
<tr>
<td>FLT32</td>
<td>maxY</td>
<td>Workspace limit in the position y axis direction.</td>
</tr>
<tr>
<td>FLT32</td>
<td>maxZ</td>
<td>Workspace limit in the positive z axis direction.</td>
</tr>
<tr>
<td>FLT32</td>
<td>minX</td>
<td>Workspace limit in the negative x axis direction.</td>
</tr>
<tr>
<td>FLT32</td>
<td>minY</td>
<td>Workspace limit in the negative y axis direction.</td>
</tr>
<tr>
<td>FLT32</td>
<td>minZ</td>
<td>Workspace limit in the negative z axis direction.</td>
</tr>
<tr>
<td>FLT32</td>
<td>maxForceIntensity</td>
<td>The maximum force intensity the device can apply to user.</td>
</tr>
<tr>
<td>FLT32</td>
<td>maxTorqueIntensity</td>
<td>The maximum torque intensity on every axis the device can apply to user.</td>
</tr>
<tr>
<td>UINT32</td>
<td>numberOfButtons</td>
<td>Number of buttons on the device.</td>
</tr>
<tr>
<td>UINT32</td>
<td>capabilities</td>
<td>A bit mask of capabilities supported in hardware.</td>
</tr>
</tbody>
</table>

6.5.1.1 Capabilities

The capabilities field is a bit mask containing flags describing the features that the device supports in hardware. All missing capabilities are always implemented in software:

HAPTIK_CAPABILITY_VELOCITY

Device computes velocity in hardware.

HAPTIK_CAPABILITY_ORIENTATION

Device stylus can be rotated in space.

HAPTIK_CAPABILITY_FORCEFEEDBACK

Device can apply force to user.

HAPTIK_CAPABILITY_TORQUEFEEDBACK

Device can apply torque feedback on the stylus.
**HAPTIK\_CAPABILITY\_6DOF**

Device is a 6 degree of freedom.

**HAPTIK\_CAPABILITY\_VARIABLE\_RATE**

Device supports variable rate in hardware.

**Remarks**

The maxRate and minRate are plugin dependent. These values are typically fixed for a specific device but some implementation can support in software different rates.
6.5.2 HaptikData

The HaptikData struct contains data to be passed back and forth to the device.

Fields

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix4x4</td>
<td>matrix</td>
<td>[in] a 4x4 matrix of 32 bits floating points describing orientation (upper 3x3) and position (last row) of the stylus switch relative to its default position.</td>
</tr>
<tr>
<td>Vector3</td>
<td>position</td>
<td>[in] the position of stylus switch relative to its default position. It's the same as the fourth row of matrix.</td>
</tr>
<tr>
<td>Vector3</td>
<td>velocity</td>
<td>[in] instant velocity of the stylus switch.</td>
</tr>
<tr>
<td>UINT32</td>
<td>buttonMask</td>
<td>[in] a bitmask indicating what buttons (if any) are pressed. Use the helper method Button(n) for easy access.</td>
</tr>
<tr>
<td>DWORD</td>
<td>Reserved0[1]</td>
<td>Padding space reserved for future use.</td>
</tr>
<tr>
<td>Vector3</td>
<td>forceFeedback</td>
<td>[out] a vector of force to be applied to the device. The actual force sent to the device depends on the configuration (e.g. ceiling).</td>
</tr>
<tr>
<td>Vector3</td>
<td>torqueFeedback</td>
<td>[out] a vector of torques to be applied to the device.</td>
</tr>
</tbody>
</table>

Methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL32 Button(UINT32 n=0)</td>
<td>Returns the state on the n-th button of the device (n-th bit of buttonMask).</td>
</tr>
</tbody>
</table>

Remarks

The use of the HaptikData struct is not mandatory. HaptikData uses some matrix and vector classes which are part of the RSLib framework. You can define a custom version of HaptikData, using your own preferred matrix/vector classes, and pass that as the argument to callbacks and methods. The memory layout should be exactly the same as in the following sample:

class MyHaptikData
{
    public:
    float [4][4] matrix;
    float [3] position;
    float [3] velocity;
    UINT32 buttonMask;
    DWORD reserved0[1];
    float [3] torqueFeedback;
    DWORD reserved1[2];
}
operator HaptikData&(){return *this;}
VOID Callback(MyHaptikData& data);

Otherwise you can use HaptikData unchanged and use in your code the SEEAS(...) macro or SeeAs<type> template function from <RSLib/Types.hpp> as in the following samples:

SEEAS(MyVector,data.forceFeedback) = MyVector(0,0,1);
MyMatrix R = SeeAs<MyMatrix>(data.matrix) * MyMatrix(IDENTITY);

They’re defined in the following way:

#define SEEAS(type,object) (*(type*)(void*)&(object))
template<class T,class O> T& SeeAs(O& o){return *(T*)&o;}

And allows to use a variable as it was of another type. **Care should be taken when using this macro/function to ensure that the original and final types corresponds at the binary level both in size and layout, because no type cast occurs.**
6.5.3 HaptikPlugInfo

The HaptikPluginInfo struct contains information about a loadable plugin.

Remarks
This structure is reserved for future use.
6.5.4 HaptikLibraryInfo

The HaptikLibraryInfo struct contains information about the library runtime.

Remarks
This structure is reserved for future use.
Chapter 7

C++ Plugin Developer’s Reference

The following topics provide reference information about using the Haptik Library Plugin API with the C/C++ programming language:

- Functions
- Constants
- Interfaces
7.1 Function

A Haptik Library Plugin DLL must export the following function:

- GetInterface
7.1 Function

7.1.0.1 GetInterface

The GetInterface function is called by the Haptik Library to obtain an interface to the Plugin component.

```c
extern "C" PVOID GetInterface(
    UINT32 interfaceId = HAPTIK_IHAPTIKPLUGIN
);
```

**Parameters**

`interfaceId`

[in] an interface id identifying the desired interface.

**Return Values**

If the plugin supports the requested interface it should return an interface of that kind. Otherwise it should return NULL.

**Remarks**

Currently the only interface defined is the IHaptikPlugin. Future releases of Haptik Library could support other interfaces for extended interaction. Interfaces will be requested from the newest to the oldest, so future compatibility will be guaranteed.
7.2 Constants

The Haptik Library defines the following category of constants for use by Plugins:

- Devices
- Interfaces
7.2.1 Device Constants

The following constants and macros are used to build and interpret device ids.

7.2.1.1 DeviceIDs

The deviceID is a 32 bit value structured as following:

*Bit 0-15 hardware id*

A 16-bit value univocally identifying a particular device. Most significant bits are used for the manufacturer (SensAble, ForceDimension, etc), least significant bits for device type (Premium, Desktop, Omega).

*Bit 16-23 plugin id*

An 8-bit value identifying the plugin. Assigned at runtime by the library to each plugin.

*Bit 24-28 number*

A 5-bit number used by the plugin to distinguish between exposed devices with the same hardware id.

*Bit 29-31 flags*

A 3-bit mask. The highest bit is always set for a device id. Clear if the id is to be interpreted as an ordinal. The other bits are reserved for future use.

Defined Values

Currently defined values for hardware id are:

- 0xFFFF Default Device
- 0x8001 Mouse Spectre
- 0x4001 Phantom Premium
- 0x4002 Phantom Desktop
- 0x4003 Phantom Omni
- 0xC001 Delta
- 0xC002 Omega
- 0x2001 Freedom
- 0x2002 Cubic

Currently defined value for plugin id is:

- 0xFF Any Plugin
Note: Plugins receive their assigned plugin id at runtime from the library and should use it to build device ids.

**Macros**

Some macros are defined to help build and manage device ids. `HARDWAREID(id)`

Extract hardware id from device id.

`PLUGINID(id)`

Extract plugin id from device id.

`NUMBER(id)`

Extract number from device id.

`FLAGS(id)`

Extract flags from device id.

`IS_ORDINAL(id)`

Check the Highest Flag Bit to see if id is a simple ordinal value.

`MATCH_DEVICEID(requestedDeviceld, deviceld)`

TRUE if requestedDeviceld is for any device or equals deviceld. For comparison pluginid and flags are ignored and only number and hardware id are taken into account.

`MATCH_PLUGINID(requestedDeviceld, pluginId)`

TRUE if requestedDeviceld is for all plugins or equals pluginId. For comparison only the plugin id is taken into account.

`UNMATCH_DEVICEID(requestedDeviceld, deviceld)`

FALSE if requestedDeviceld is for any device or equals deviceld. For comparison pluginid and flags are ignored and only number and hardware id are taken into account.

`UNMATCH_PLUGINID(requestedDeviceld, pluginId)`

FALSE if requestedDeviceld is for all plugins or equals pluginId. For comparison only the plugin id is taken into account.

`FLAGGED_DEVICEID(flags, number, pluginid, hardwareid)`

Build a device id with flags.

`DEVICEID(number, pluginid, hardwareid)`

Build a device id with no flags (highest bit is set).
### 7.2.2 Interface Constants

The following constants identifies interfaces:

- **HAPTIK_IHAPTIKLIBRARY**
  - IHaptikLibrary interface. Exposed by the library to upper level (application).

- **HAPTIK_IHAPTIKLIBRARYLOWER**
  - IHaptikLibraryLower interface. Exposed by the library to lower level (plugins).

- **HAPTIK_IHAPTIKPLUGIN**
  - IHaptikPlugin interface. Exposed by plugins to library (and to application in future).

- **HAPTIK_IHAPTIKDEVICE**
  - IHaptikDevice interface. Exposed by devices to application.
7.3 Interfaces

Haptik Library defines the following interfaces:

- IHaptikPlugin
- IHaptikLibraryLower
7.3.1 IHaptikPlugin

The IHaptikPlugin interface is a generic interface exposed by the plugins that is used by the Haptik Library to uniformly access plugins.

Methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Init</td>
<td>Initialize plugin.</td>
</tr>
<tr>
<td>Release</td>
<td>Release the interface.</td>
</tr>
<tr>
<td>GetNumberOfDevices</td>
<td>Gets the number of devices exposed.</td>
</tr>
<tr>
<td>GetDeviceInfo</td>
<td>Gets info about a device.</td>
</tr>
<tr>
<td>GetDeviceInterface</td>
<td>Gets an interface.</td>
</tr>
<tr>
<td>GetPluginInfo</td>
<td>Gets info about a plugin.</td>
</tr>
</tbody>
</table>
7.3.1.1 Init

The Init method initializes the plugin.

```c
UINT32 Init(
    IHaptikLibraryLowerInterface iHaptikLibrary,
    UINT32 pluginId
);
```

**Parameters**

- **iHaptikLibrary**
  - [in] an IHaptikLibraryLower interface to the library component. The plugin should store this interface.

- **deviceId**
  - [in] a runtime assigned plugin id that should be used to build device ids.

**Return Values**

If the plugin initialize correctly it should return HAPTIK_SUCCESS. Otherwise it should return an error value.

**Remarks**

The Init method is called soon after the plugin dll is loaded, and should be used to perform one time initialization. If Init returns an error the library unloads the plugin dll. Release is invoked even if Init fails.
7.3.1.2 Release

The Release method frees all resources allocated by the plugin.

UINT32 Release();

Return Values
This function should always return HAPTIK_SUCCESS.

Remarks
The Release method is called before unloading the plugin dll and should be used to perform clean up and resource deallocation. Release is guaranteed to be called only if no interfaces are currently active on this plugin. A Release called on a plugin with opened interfaces indicates an incorrect usage of the Lock/Unlock protocol by the plugin. Release is invoked even if Init fails.
7.3.1.3 GetNumberOfDevices

The GetNumberOfDevices method returns the number of devices exposed by the plugin.

```cpp
UINT32 GetNumberOfDevices();
```

**Return Values**
The function must return the number of devices exposed.

**Remarks**
This method is invoked by Library to allocate needed structure before requesting info for each exposed device. This method should never fail; internal plugin device enumeration should be performed in the Init function.
### 7.3.1.4 GetDeviceInfo

The GetDeviceInfo method is called by the library to get information about an exposed device.

```c
UINT32 GetDeviceInfo(
    UINT32 deviceId,
    HaptikDeviceInfoPtr pInfo
);
```

**Parameters**

- `deviceId`
  
  [in] a device identifier or an ordinal.

- `pInfo`
  
  [in] a pointer to a HaptikDeviceInfo struct that should be filled with info about the device.

**Return Values**

If the function succeeds, the return value is HAPTIK_SUCCESS.

**Remarks**

The HaptikDeviceInfo struct must be filled by the plugin with pointers to persistent data. Device related string must be allocated, if necessary, in the Init function and freed in the Release. This constrains eventually will be relaxed in next library release (maybe data persistent until next invocation of GetDeviceInfo). Requests with valid device ids (or ordinals) should never fail.
7.3.1.5 GetDeviceInterface

The GetDeviceInterface method requests an interface to a device.

```c
PVOID GetDeviceInterface(
    UINT32 deviceld,
    UINT32 interfaceld
);
```

**Parameters**

*deviceld*

[in] a device identifier or an ordinal.

*interfaceld*

[in] an interface identifier.

**Return Values**

If the function succeeds, the return value is the requested interface to the requested device. Otherwise NULL must be returned.
7.3.1.6 GetPluginInfo

The GetPluginInfo method is called by the library to get information about the plugin.

```c
UINT32 GetPluginInfo(
    HaptikPluginInfoPtr pInfo
);
```

**Parameters**

`pInfo`

[in] a pointer to a HaptikPluginInfo struct that should be filled with info about the device.

**Return Values**

If the function succeeds, the return value is HAPTIK_SUCCESS.

**Remarks**

The HaptikPluginInfo struct must be filled by the plugin with pointers to persistent data. Device related string must be allocated, if necessary, in the Init function and freed in the Release. This constrains eventually will be relaxed in next library release (maybe data persistent until next invocation of GetPluginInfo). Requests with valid device ids (or ordinals) should never fail.
7.3.2 IHaptikLibraryLower

The IHaptikLibraryLower interface is exposed by the Haptik Library to the plugins. It can be used by plugin code to access the centralized configuration tree/file, to perform locking/unlocking operations, to access other plugins for internal recursion.

Methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetVariable</td>
<td>Get a variable from configuration tree.</td>
</tr>
<tr>
<td>SetVariable</td>
<td>Save a variable to configuration tree.</td>
</tr>
<tr>
<td>Lock</td>
<td>Prevents Library from unloading the plugin.</td>
</tr>
<tr>
<td>Unlock</td>
<td>Allows Library to unload this plugin.</td>
</tr>
<tr>
<td>GetNumberOfDevices</td>
<td>Gets the number of devices exposed by other plugins.</td>
</tr>
<tr>
<td>GetDeviceInfo</td>
<td>Gets info about a device.</td>
</tr>
<tr>
<td>GetDeviceInterface</td>
<td>Gets an interface to a device exposed by another plugin.</td>
</tr>
</tbody>
</table>
7.3.2.1 GetVariable

The GetVariable method reads a variable from the configuration tree.

```c
PCHAR8 GetVariable(
    PCHAR8 name
);
```

**Parameters**

`name`

[in] the name of the variable to retrieve.

**Return Values**

If the variable exists, a pointer to a ZERO terminated ASCII string containing the value is returned. Otherwise NULL is returned.

**Remarks**

Variable names must be in the form of "PluginName.Set1.Set2.SetN.Variable".

At this time the configuration namespace is global for all plugins, so it’s mandatory that each plugin adopts a unique textual description to distinguish its data. In some special sequences are reserved for special usage, they are:

- "Registry" which maps to system-wide persistent storage where plugins could store/retrieve persistent data.
- "Volatile" which maps to system-wide session-only storage, where plugins could store/retrieve volatile data such as calibration data.
7.3.2.2 SetVariable

The SetVariable method saves a variable to the configuration tree.

```c
UINT32 SetVariable(
    PCHAR8 name,
    PCHAR8 value
);
```

**Parameters**

- **name**
  
  [in] the name of the variable to save.

- **value**
  
  [in] a string to save.

**Return Values**

If the function succeeds, the return value is HAPTIK_SUCCESS.

**Remarks**

Variable names must be in the form of "PluginName.Set1.Set2.SetN.Variable".
At this time the configuration namespace is global for all plugins so it’s mandatory that each plugin adopts a unique textual description to distinguish its data. In Some special sequences are reserved for special usage, they are:
- "Registry" which maps to system-wide persistent storage where plugins could store/retrieve persistent data.
- "Volatile" which maps to system-wide session-only storage, where plugins could store/retrieve volatile data such as calibration data.
7.3.2.3 Lock

The Lock method activates the library lock on the plugin.

```c
VOID Lock(
);
```

Remarks

Locking a plugin prevents the library from unloading the plugin dll. A plugin should keep internal reference count and use the Lock/Unlock methods to prevent/allow unloading.
7.3.2.4 Unlock

The Lock method removes the library lock on the plugin.

UINT32 Unlock(
);

Return Values
This method always returns HAPTIK_SUCCESS.

Remarks
Unlocking a plugin allows the library to eventually unload the plugin dll. A plugin should keep internal reference count and use the Lock/Unlock methods to prevent/allow unloading. Once a plugin unlocks itself it could eventually be unloaded during the Unlock call. Thus if the call to Unlock was not performed with a jmp instruction, but with a ret instruction, after unloading the execution would return to memory that has been deallocated. The Library is able to detect such situation (using the GetModuleHandleEx) and will delay the unloading without crashing. The correct "jumping" behavior can be achieved by compiling code with optimization enabled, using a return iHaptikLibrary->Unlock() call as last code of line, and not using local classes (put them in a scope brace if you really need them) with destructors. Methods that internally go to Unlock (typically a Release on a device which is then routed to a decrement reference count of the plugin and then eventually to the Unlock) should "return through": Take a look at the example plugin code for a way to correctly handle this issue. On Windows2000 systems where the GetModuleHandleEx is not available plugin unloading is always delayed.
### 7.3.2.5 GetNumberOfDevices

The `GetNumberOfDevices` method returns the number of devices exposed by all other plugins.

```c
UINT32 GetNumberOfDevices(  );
```

**Return Values**

The method returns the number of devices exposed by all plugins except the caller.

**Remarks**

This method can be used by a plugin to gain access to devices exposed by other plugins.
7.3.2.6 GetDeviceInfo

The GetDeviceInfo method retrieves information about a device exposed by another plugin.

```c
UINT32 GetDeviceInfo(
    UINT32 deviceId,
    HaptikDeviceInfoPtr pInfo
);
```

**Parameters**

- `deviceId`  
  [in] a device id or an ordinal.

- `pInfo`  
  [out] the address of a HaptikDeviceInfo that will be filled with info about device.

**Return Values**

If the function succeeds, the return value is HAPTIK_SUCCESS.

If `deviceId` is an ordinal return value can be HAPTIK_OUT_OF_RANGE. If no device has the requested id return value is HAPTIK_NO_SUCH_DEVICE.

**Remarks**

Ordinals seen by a plugin are different from ordinals seen by application because plugin exposed devices are skipped. This method can be used by a plugin to gain access to devices exposed by other plugins.
7.3.2.7 GetDeviceInterface

The GetDeviceInterface requests an interface to a device from another plugin.

```c
PVOID GetDeviceInterface(
    UINT32 deviceId,
    UINT32 interfaceId
);
```

**Parameters**

- `deviceId`
  - [in] a device identifier or an ordinal.

- `interfaceId`
  - [in] an interface identifier.

**Return Values**

If the function succeeds, the return value is the requested interface to the requested device. Otherwise NULL must be returned.

**Remarks**

This method can be used by a plugin to gain access to devices exposed by other plugins. Ordinals seen by a plugin are different from ordinals seen by application because plugin exposed devices are skipped. A plugin will never receive a self-request (which could lead to infinite recursion).
Part IV

LANGUAGE BINDINGS
Chapter 8
Java

Haptik can be used from Java applications and applets through a simple class implemented using Java Native Interface (JNI). The JNI interface (haptik.java.dll) is located together with the other library binaries in the /RSLib/Bin directory. The directory /Java contains the source of the java classes (HaptikDevice.java, HaptikDeviceInfo.java HaptikData.java) as well as some examples. Please Note that this feature is experimental!!

8.1 Getting Started

Adding the binaries directory to the path is usually enough to start using Haptik in Java applications, while additional steps are needed to run Haptik enabled applets. Java Applets have more restrictions than standard Java applications, and loading external dlls is denied by default and will cause an exception:

java.security.AccessControlException:
access denied (java.lang.RuntimePermission loadLibrary.haptik_java)

You can instruct your Java Runtime Environment (JRE) to grant this loading by adding the following line:

permission java.lang.RuntimePermission "loadLibrary.haptik_java";

in the java.policy file which is usually located on windows systems in
C:\PROGRAM FILES\Java\jreX.X.X\lib\security\java.policy

(search your disk for java.policy if you can’t find it).
Moreover it should be noted that when running a Java Applet the actual application using Haptik is your browser, therefore dlls loading rules (i.e. a needed dll should be in the exe directory or in the system path) apply to the browser executable and not to the applet page. You must ensure that the haptik.java.dll (as well as all the Haptik binaries) are in a system path directory or in the browser directory. Usually putting all Haptik binaries (including the JNI interface) in the same system-path-included directory is the best solution. Same consideration apply to configuration file: The browser is using Haptik and config file loading rules (i.e. application current directory or directory containing Haptik.Library.dll)
apply to the browser. You must ensure a configuration file is either in the browser current directory (which is different from browser executables directory!) or in the Haptik binaries directory. If you’re experiencing strange behaviours check out the Haptik log file to see if the configuration file getting loaded is the one you want (also make sure the log file you’re reading is not an old one). If you still have problems in using Haptik from your Java programs please check out the FAQs.

Following is a screen shot of the Java3D example provided running both as an application and as an applet.
Chapter 9

Matlab & Simulink

9.1 Matlab

Haptik can be used from matlab command line and from .m files through a mex interface (haptik_mlab.dll). A simple matlab class has been written on top of this to make easy to use.

- `haptikdevice_list`
  print a list of current devices

- `h = haptikdevice`
  open for polling the default device

- `h = haptikdevice(id)`
  open for polling a specific device

- `h = haptikdevice(@mycallback,rate)`
  open default device and use the mycallback(.m) function at the specified rate in Hz

- `h = haptikdevice(id,@mycallback,rate)`
  open a specific device and use the mycallback(.m) function at the specified rate in Hz

- `[matrix,button] = read(h)`
  get orientation from device (matrix(4,1) matrix(4,2) and matrix(4,3) contains position)

- `position = read_position(h)`
  get position

- `button = read_button(h)`
  get button status

- `write(h,ff)`
  send forces and torques to device. FF can be 2x3, 3x2, 1x3, 3x1, 1x6, 6x1.

- `close(h)`
  close the device

Unfortunately, because matlab is not synchronized when invoked from a mex, callbacks are implemented with matlab timers, that are based on java timers and so they are pretty unreliable. Therefore currently using callbacks result in a poor user interaction. Best results can be achieved by simply reading and writing in a cycle.
9.2 Simulink

Haptik can be used in a simulink model through a S-function (haptik_simulink.dll). Simply add an user defined block to your model and use "haptic_simulink" as the s-function name. The output is a 3 vector containing position, while input is a 3 vector used as forcefeedback. Support for compiling the model with Real Time Workshop will be added in a future release.
APPENDICES
Appendix A

Frequently Asked Questions

A.1 General FAQs

Q) What is The Haptik Library?
A) Haptik is a small component-based library to access haptic devices in a uniform way. Using Haptik you write (and compile) once and then your application will run on every supported devices without caring about driver versions, installed SDKs, missing DLLs. Your applications will run even on systems without a haptic device!

Q) How can I render graphically with Haptik?
A) You can’t. Haptik is only for haptic device access. Anyway Haptik easily integrates with both OpenGL and DirectX by supporting both their coordinate systems and exposing matrices in the format used by both. Moreover it can also be used within higher level frameworks.

Q) What devices are implemented in Haptik?
A) Support for each particular device is actually implemented in plugins and not in the library itself. Currently bundled with Haptiks you get plugins for PHANToMs and Delta/Omega. Support for other devices is easily added through new plugins, therefore once you got the plugin for a class of devices those devices can be directly used in any of your existing Haptik applications.

Q) What does it means that Haptik ensures binary compatibility?
A) It means that an application built to use Haptik will always run with never version of libraries and plugins without recompiling even if major changes have made to the underlying interfaces and protocols. Therefore old applications executables can use new devices without being recompiled.

Q) Is Haptik C++ only?
A) While Haptik is written in C++ (and a bit of x86 assembly) and C/C++ application is the most common usage scenario, it can be used from many languages/environments. Current release provides binding for Matlab and Simulink as well as for Java, which allows to use haptik, and thus haptic devices, also through a web browser. Future release will likely include support for C# and Lua. Only Perl will never be supported for religious issues.
Q) I have read all the FAQs but Haptik still doesn’t work as expected! What can I do?
A) Haptik is developed and tested on some Windows XP workstations and on a Kubuntu Linux system, and it should work also on different systems but sometimes a simple flag in a system API call could cause problems. The Java Runtime Environment (JRE) version is 1.5.0, while the Matlab version is 6.5 R13. If you’re not able to run Haptik on very different systems please mail us and we will solve the compatibility problem. If you’re successfully running Haptik on very different systems please let us know.

A.2 User FAQs

Q) My device gets not listed by Haptik!
A) Make sure you have a plugin for your device type and that this plugin is listed in the configuration file for your application.

Q) I’ve added the plugin to the configuration file but my device still doesn’t get listed!
A) Make sure your plugin could be loaded. Each plugin has usually dependencies from device related dlls. For example the Phantom31 plugin depends from files that gets installed with Phantom Drivers 3.1 and could not be loaded by the os if they have not been installed. The Library will report in the Debug Output window of your debugger and in the log file what plugins gets loaded and what failed.

Q) I have a working application but when using a PHANToM Desktop I get no forces!
A) Try disabling the autorecalibration feature setting Phantom.AutoRecalibratePeriod = 0. If set to a value different than 0 the Phantom plugin automatically tries every N milliseconds to recalibrate the device if needed. During recalibration forces gets disabled for safety, therefore it can happen that 1) your device easily loses calibration. 2) plugin start autorecalibrating and suspends forces 3) you did not move the device around the workspace so it didn’t recalibrate and forces stay suspended forever!

A.3 Developer FAQs

Q) I’m using OpenGL. The Z axis looks inverted!
A) Make sure you’ve set in the configuration file LeftHanded = FALSE for every device. This is usually the default if not specified but depends on plugin implementation.

Q) I’m using DirectX. The Z axis looks inverted!
A) Make sure you’ve set in the configuration file LeftHanded = TRUE for every device. By default they’re usually set to FALSE.
Q) I’ve my fullscreen app which gets blocked when Haptik outputs message boxes!
A) There is an entry in the configuration file, HaptikLibrary::allowMessageBoxes, which usually prevents message boxes to be displayed. Unfortunately no one can prevent plugins from using message boxes functions (which they do for example when recalibrating). This will surely addressed in a future subversion release by including the ability for plugins to route messages through the application and not directly to the user. By the way if your application can’t survive the focus loss caused by message boxes as well as unpredictable events like firewalls messages, users alt-tabbing, resolution changes, and all those kind of Unavoidable & Unpredictable Events (TM) than you should reconsider taking a few hours to make it more robust ;)

Q) I’ve built a new device. How can I expose through Haptik?
A) You have to build a related plugin. Among the examples there is a template project for Visual Studio .NET that you can use. Support for other compilers will be provided in a future minor release. By the way you can simply reuse sources for other compilers. If you have problems feel free to contact us, will be glad to help you.

Q) I would add extended behaviours to a particular device, but I can’t modify the plugin. What can I do?
A) Haptik supports a feature we call Internal Recursion, that is a plugin can use devices exposed by other plugins to implement its own, which allows to easily extend device behaviours by chaining plugins. Among the examples there is a completely useless add-white-noise plugin which shows that.

Q) I have an old application compiled to use Haptik prior of version 1.0. Can I use with version 1.0?
A) No, you can’t. Interfaces and structures in 0.X releases where not final, as clearly stated in the documentation, and so they’ve been changed in 1.0. We have to choose between saving backward binary compatibility between 0.X and 1.0 or between 1.0 and 1.X...and we choosed the latter. By the way, interfaces and structures in 1.0 are final, that is they will never change, but runtime mechanisms have been implemented in the library and headers have been modified (the reserved fields and parameters...) in order to allow minor modifications to be included in 1.X versions without breaking code and binary compatibility. Major modifications will be released with new set of structures and interfaces (i.e. IHaptikDevice2) which will coexist side by side with older ones. Recompiling applications using 0.X versions is usually enough. The only breaking mod is the Rate(...) method being renamed SetRate(...).

Q) I have an old plugin compiled for Haptik version 0.X. Can I use it with version 1.0?
A) No, you can’t. As for the interfaces and structures used by applications also those used by plugins have been changed to allow for minor mods preserving backward binary compatibility. Recompiling plugins is usually enough. Only check all your calls to IHaptikLibraryLower::SetVariable because a change in the return type has inverted semantic.
A.4 Java FAQs

Q) Can I use Haptik from Java?
A) Yes you can. Haptik can be used through a set of JNI (Java Native Interface) based classes which reflects the C++ API.

Q) Can I use Haptik in an Java Applet?
A) Yes you can. but you must instruct your Java Runtime Environment (JRE) to allow the loading of the JNI interface. read the next faq...

Q) I am launching a Java application using Haptik but I get a message from the library which states that Haptik.Config.txt can’t be loaded!
A) Make sure a configuration file is in the current directory or in the Haptik.Library.dll directory.

Q) I have a working Java application using Haptik but when I run it as an applet from the browser I get the following exception "java.security.AccessControlException: access denied (java.lang.RuntimePermission loadLibrary.haptik_java)"
A) Java Applets have more restriction than standard Java applications. The exception is caused by the runtime refusing to load the JNI interface to Haptik (haptik.java.dll). You can instruct your JRE to grant this loading by adding the following line:
permission java.lang.RuntimePermission "loadLibrary.haptik_java";
in the java.policy file which is usually located on windows systems in
C:\PROGRAM FILES\Java\jreX.X.X\lib\security\java.policy
(search your disk for java.policy if you can’t find it).

Q) I have a working Java application using Haptik but when I run it from the browser I get the following exception "java.lang.UnsatisfiedLinkError: no haptik_java in java.library.path"
A) When running a Java Applet the actual application using Haptik is your browser, therefore dlls loading rules (i.e. a needed dll should be in the exe directory or in the system path) apply to the browser executable and not to the applet page. You must ensure that the haptik.java.dll (as well as all the Haptik binaries) are in a system path directory or in the browser directory. Usually putting all Haptik binaries (including the JNI interface) in the same system-path-included directory is the best solution.

Q) I have a working Java application using Haptik but when I run it from the browser I get a message from the library which states that Haptik.Config.txt can’t be loaded!
A) Same as before. The browser is using Haptik and config file loading rules (i.e. application current directory or directory containing Haptik.Library.dll) apply to the browser. You must ensure a configuration file is either in the browser current directory (which is different from browser executables directory!) or in the Haptik binaries directory. If you’re experiencing strange behaviours check out the Haptik log file to see if the configuration file getting loaded is the one you want (also make sure the log file you’re reading is not an old one).

Q) I am using a Java3D application. The Z axis looks inverted!
A) Java3D uses a right handed reference frame (just like OpenGL). Make sure your devices are configured as right handed in the used configuration file.
Q) I am using a Haptik Applet but when I make changes to code nothing actually changes!
A) This a more a Java/Browser related issue. Current Java/Browsers implementations are very fragile and not well standardized as they should, therefore loading/unloading/reloading/caching of applets by browser is almost unpredictable and differs from vendor to vendor and sometimes amongst different versions. The only way to ensure that your newly compiled applet get used is to close all browser’s windows and then reopen a new one.

A.5 Matlab FAQs

Q) Can I use Haptik from Matlab?
A) Yes you can. Haptik has a mex interface (haptik_matlab.dll) and a bunch of m-files which implements a class that you can use on the command line or in your m-files.

Q) Can I use Callbacks in Matlab?
A) Yes you can, but you shouldn’t. Unfortunately big parts of latest Matlab versions are written in Java so current releases are indecently slooow. Moreover it seems impossible to make invocations to the matlab engine from more than one thread without crashing the JVM and so callbacks are actually implemented using matlab timers, which are based on java timers, which are unreliable, resulting in a poor user experience.

Q) Can I use Haptik in a Simulink model?
A) Yes you can. Haptik has a simulink sfunction block which can be used in any model. Remember to add a realtime-delayer block to ensure your simulation doesn’t runs too fast. Support fro real time workshop will be soon added in a future minor releases.

A.6 Miscellanea FAQs

Q) I have noted that Haptik is behind a namespace named RSLib. What is the RSLib?
A) The RSLib is a collection of software modules which can be used independendly from each other (but designed to fit well together) to build portable realtime visio-haptic 3D applications. The RSLib will include The HaptikLibrary, the RS3DFX Engine, a next-generation completely-shader-based 3D rendering engine, a template based collision detection library specifically tailored for haptic interactions, a wrapper layer to external physics engines, a scripting engine. Currently most modules of the RSLib are still in an early prototyping stage. First releases are scheduled for 2006.
Appendix B

A small Introduction to the Component Model

Following is a brief introduction of the Component Model. More in-depth descriptions and introductions can be found on COM and CORBA related sources of info. A simple search on ”COM” in the MSDN is a very good starting point.

B.1 Components and Interfaces

A component can be thought as an object, as those from object-oriented languages like C++, that additionally enforces a stricter encapsulation and some well-defined behavior. Just like objects, components provide a well-specified set of publicly available services, but differently from objects, their methods cannot be directly invoked. In fact a component’s public methods are grouped into one or more interfaces. Besides being groups of semantically related functions, interfaces are concrete entities that provide access to components, defining a sort of contract that allows interactions with and between components. Each component implements one or more interfaces. Once a client obtains an appropriate interface it can request services. Any methods that are not part of an interface are not accessible. Interfaces are separated from components and different components can expose the same interface, each providing a different implementation, but a common interface-associated behavior. The way in which components are created and interfaces are requested/obtained is peculiar to each implementation/definition of the Component Model, which typically adopts only a subset of the features of the whole model.

B.2 Component Model Extras

In what follows we recall some other relevant ideas of component model that are related to the Haptik Library. Component Model introduces many benefits both at compile and run time. Components are truly opaque: there is no need for something like a C++ class definition, therefore changes in the component layout do not requires recompilation. Invocation of methods through interfaces can be specified as a binary standard with mappings for many languages [omg2004idl] (even for non object-oriented languages like C) thus
guaranteeing cross-language compatibility. Moreover interface invocation is commonly defined as a runtime binding. In this way the same executable can run against different versions of a component, or even components from different vendors, thus guaranteeing binary compatibility and implementation transparency. Last but not the least, the indirection introduced by the use of interfaces brings an unprecedented level of location indifference: a component can live in the same process of the client, in a different process on the same machine, or perhaps on a different machine with a different hardware architecture somewhere over the network, and all these possible configurations are transparent to its users.

There are some common mistakes about the interface concept that can be quite confusing to non-component developers: namespaces are separated and two methods of a component, even with the same name but belonging to different interfaces, may (or may not) have separate implementations. On the contrary even the same method on two different instances of the same interface could execute different code. This namespace behavior is quite different, for example, from C++ and Java, where each object has only one namespace, and cannot have two different methods with the same decorated name (i.e. name plus arguments).
Bibliography


The Haptik Library - Developer’s Manual
compiled despite \LaTeX\ on Wednesday 30\textsuperscript{th} November, 2005