SimTK Coding Guidelines

DRAFT, Version 0.7

SimTK Staff

October 10, 2007

Abstract

This article presents the organization of the SimTK Core software, some important design patterns we use, and our preferred coding style. This is primarily oriented to those contributing to the SimTK Core effort, but can also be useful for users of the SimTK Core and as a guide for organizing your software so that it works smoothly with other SimTK software. [This document is still being written.]

1 Purpose of this document

This document provides guidance to programmers who would like to produce software which is compatible in functionality and style with SimTK Core software modules.

By encouraging adoption of the methods described, we hope (1) to support an open source, collaborative development environment with a shared "culture" while inviting individual style and contribution, (2) to achieve a critical set of technical goals described below, and (3) ensure long-term robustness of SimTK code through a rigorous and transparent build/test/release/install/update process.

2 Background

"SimTK Core" is the name we give to a collection of software tool libraries (APIs) designed for use by application programmers to build applications which make effective use of physics-based simulation of biological structures. These tools are logically organized in a six-layer hierarchy as shown in Figure 1 with higher-level tools dependent on lower ones but not vice versa. Depending on needs and level of sophistication, application programmers can access the tools at any level, ignoring those above.

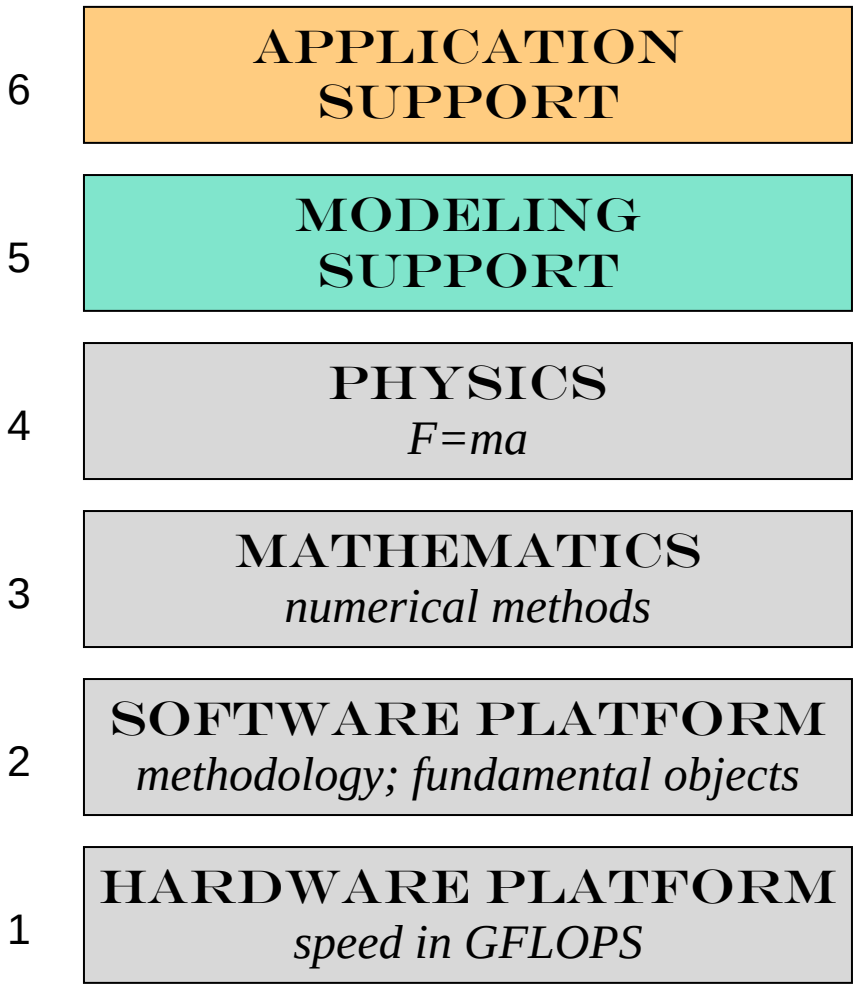


Figure 1: Logical structure of the SimTK Core software stack. Upper level tools can be dependent on everything below, but lower levels never depend on upper ones.

2.1 Physical structure of libraries and projects

Physically, the SimTK Core toolkit is organized as a set of libraries and corresponding header files. The libraries are generally confined to a single logical layer, but there may be several independent libraries in any given layer. Some of the libraries are third-party tools; that is, they are written and distributed by others, are unchanged by us, but a particular version and build has been made available on SimTK.org and qualified to work well with the SimTK Core. Others are original software produced by the SimTK Core team, or third party open source software that has been adapted by the SimTK Core team.

The current set of libraries is shown here:

Library name	Level	SimTK.org project name	Description
`SimTKsimbody`	4	`simbody`	Internal coordinate rigid-body mechanics toolset for use in "multibody biology" applications from human gait simulation to coarse-grained molecular simulations.
`SimTKcpodes`	3	`cpodes`	SimTK adaptation of the CPODES integrator developed in collaboration with Radu Serban at LLNL. This is derived from the DOE Sundials suite of numerical software tools.
`SimTKmath`	3	`simmath`	Numerical mathematics algorithms including optimization, numerical integration, root finding, linear and nonlinear algebra.
`SimTKcommon`	2	`simtkcommon`	Classes supporting the basic SimTK architecture. Also includes the Simmatrix vector- and matrix-handling toolset.
`SimTKlapack`	1	`lapack`	Machine-dependent high performance linear algebra library via LAPACK and BLAS.

Currently we also intend to support the following third-party libraries as binaries installed in SimTK.org projects.

Library name	Level	SimTK.org project name	Description
`VTK` (including `KWwidgets`)	6	`vtkbin`	Binaries built from the open source "visualization toolkit" project from Kitware at http://www.vtk.org.
`Xerces`	5	`xerces-c`	Binaries built from the open source distribution http://xml.apache.org/xerces-c. This is a toolkit for parsing XML files.

2.2 Guidelines for SimTK Core contributors

Anyone who would like to contribute to the SimTK Core effort should first read through this document. The latest version can be found on SimTK.org in the "resources" project, that is, https://simtk.org/home/resources. Follow the "Documents" link there.

All new code written for the SimTK Core should be written in C++. Code which is derived from earlier third party efforts can be in other languages. Currently C code is in use in various places and at level 1 (LAPACK) there is FORTRAN and even some assembler.

Contributions to SimTK Core must be non-viral, unrestricted open source code. We want to encourage as many people as possible to use the SimTK Core software, and we think that biological knowledge and medical treatment will gain from such use regardless of whether the users are academic, government, or commercial. Consequently, for the SimTK Core we insist on "BSD-like" licensing (we actually use the MIT license). Please note that the "GPL" license does not qualify because of its viral nature. If you can't stand the thought of other people using your code for their own gain, please don't contribute it! We do encourage contributors to put their names and affiliations in the code, and insist that users propagate those acknowledgments when they use the code. Fame is thus a possibility for our contributors, but fortune must be sought elsewhere.

3 Style vs. Substance

Many differences in programming technique fall into the realm of personal aesthetics (style) where one approach is not inherently better than another. It is our intent to be as accommodating as possible in this regard so that individuals may express themselves within the SimTK framework. That said, however, we do not think it is a good idea to mix incompatible styles within the same program module, because it makes the software increasingly hard to read and understand over time. So we ask that modifications to existing software be made in the original style of that software as much as possible. An alternative (for substantial changes only) is to convert the module *completely* over to your style. Please do leave an acknowledgment to the work of the original author in any case.

We include among the stylistic issues the choice of programming language. While new SimTK Core modules are primarily written in C++, we expect to include (for example) a great deal of numerical code in Fortran and C and user interface code in Java, Python, Tcl, etc. and are comfortable with programmers choosing their favorite language for any purpose provided that the results are compatible with the overall goals of SimTK.

There are other stylistic issues that, while arbitrary, must be consistent for functional reasons. These include naming conventions (e.g. capitalization) because mixing them (at least at interfaces) not only creates potential confusion but can introduce irresolvable name conflicts and difficulty linking together modules written with different conventions. One approach to mixing styles and languages that we like is to enforce uniformity only at well-defined interfaces. Almost any software can be wrapped to permit access in a uniform style, while internal conventions are much less restrictive. For SimTK, we are much more concerned with the style of interfaces than that of the hidden implementation.

Then there are truly substantive issues, and we will try to focus most on these here. Examples include programming techniques which might preclude achieving SimTK goals such as correct results, high performance, thread safety, binary compatibility, execution on Windows, Macintosh, and Linux 64-bit multiprocessor machines, and access to core capabilities from multiple languages.

4 Some preliminaries

This section includes general policies which broadly affect almost all code.

4.1 SimTK.org vs. SimTK Core

There are at least two distinct kinds of SimTK software: (1) a federation of loosely-related biomedical computation software efforts organized on SimTK.org as largely-independent *projects*, and (2) the SimTK *Core* software, which is a collection of high-quality software components specifically engineered to be used together in constructing biomedical computation software. Some SimTK.org projects contribute to the SimTK Core but most do not. There is a "project" named `SimTKcore` on SimTK.org (that is, https://www.simtk.org/home/simtkcore) which is used to group the Core software together for release and

distribution purposes, but it is actually comprised of software from several different separately-managed projects, each of which has been designated as part of the SimTK Core. The source code and related content are kept with the individual projects; only the binaries and released documentation are packaged in the `SimTKcore` project.

The rules for software which is to be part of the Core are of necessity much stricter than those for general software contributions. Most of the guidelines in this document therefore apply to SimTK Core software, and in most cases adhering to those rules is a requirement for certification as a SimTK Core component. For other SimTK.org-resident projects, following any or all of these guidelines is entirely under the discretion of the project administrators.

There are a few places where the guidelines apply only to one category or the other and those will be pointed out; otherwise you should assume that the rules are mandatory or highly recommended for Core software and optional otherwise.

4.2 Use of English

Although SimTK supports a wide variety of programming languages, it supports only a single natural language, English. While we hope to attract international contributors and users, there is much to be gained by choosing a single language, and English is the obvious choice for us. Any code to be an official part of SimTK, and particularly part of the SimTK Core, must be in English, with English comments and documentation. We will not be fussy about British vs. American spelling, as either is easily understood by people familiar with the other. Additional documentation in other languages is welcome, but we will not provide internationalized indexing and search facilities.

As a practical consequence, this means SimTK code can use `char` rather than `wchar_t` (wide characters) and embedded English text to be displayed at run time is acceptable and need not be sequestered in separate files to facilitate translation. Contributors of SimTK code thus do not need to be familiar with techniques for international programming.

4.3 Systems of units

Above all, SimTK code must be absolutely precise and clear about the units it supports. Keeping in mind the ill fate of the Mars Observer, it is hard to overemphasize the importance of clear units.

We will support both metric and common English units, with the default units chosen as appropriate for individual application domains. For example, molecular mechanics conventionally uses a strange set of mixed metric and English units, such as the Ångström and kilocalorie. We do not intend to attempt to retrain scientists; rather, we think it more practical to accommodate their preferences as we find them, through dedicated, special-purpose applications. When there is no reason to choose otherwise, we prefer metric units.

It is worth noting that many software components in physics-based simulation require only *consistent* sets of units; they are otherwise completely agnostic. A typical consistency requirement would be that time, mass, length, and force units are chosen so that F=ma. Physical constants must of course also be consistent with the chosen set of units.

4.4 Dates and times

The need for date and time stamps arises frequently enough, and causes enough trouble, that we wanted to state some general preferences here, although not specific requirements for any particular situation.

Maybe this goes without saying, but just go ahead and use four digits for the year! Let’s not go through that again. Compact date stamps such as those appearing in file names and source comments should have the form `yyyymmdd`, e.g. `20060322` which has the distinct advantage of being sortable, with the most significant part first. Applications which format friendly dates for user consumption should avoid ambiguous formats like

7/5/2005 (July 5 in the U.S. and May 7 in Europe). Instead, use July 5, 2005 or 5 July 2005 or 2005-May-07, for example.

For binary time stamps generated programmatically, please give careful thought to time zone independence. All the platforms we support have comprehensive yet simple facilities for manipulating time and date in a zone-independent fashion while converting to the local zone (and maybe daylight savings time) for display to users if you want. The basic idea is just to store all time stamps in GMT (a.k.a. "coordinated universal time") so that they have the same meaning anywhere.

5 Conventions for installed SimTK software

Here we will describe first the way a SimTK project should look when installed on an end user's machine. (The topic of how a project should look on the project *developers* machine is a different discussion and is deferred until section 6.) By "end user" here we mean a person who is not involved with development of the project in question; that person may be a developer in other contexts but is simply a consumer in this one.

We make a strict distinction here between SimTK Core software, which must be installed together and follow strict guidelines to avoid name clashes, and general SimTK projects which can be managed in any way their project administrators choose. In the latter case we have some suggested installation guidelines that, if followed, guarantee a certain degree of compatibility with other SimTK projects that follow the same guidelines.

5.1 Installation of non-Core Software

If your software is intended to become part of the SimTK Core eventually, please be sure to follow the instructions for Core software below. For non-Core software residing on SimTK, there is nothing to prevent you from handling installation on end user machines however you see fit. This may in fact be advisable for "heavyweight" applications with high end user visibility. But for most projects, we suggest you follow the instructions here which will allow you to peacefully coexist on end-user machines with software installed from other SimTK projects.

When such software is installed, we recommend the following as the default location:

Windows: `C:\Program Files\SimTK\`*yourProjectName*`\` …

Linux and Mac: `/usr/local/SimTK/`*yourProjectName*`/` …

On Windows, any drive or drives may be used—there is no requirement to install on `C:`, and yes, there is a space between "`Program`" and "`Files`" (don't blame us). Be sure to use exactly the capitalization shown here for "`SimTK`" on every platform. All SimTK software that has dependencies on other SimTK software should check these locations first to resolve those dependencies if no other "sandbox" location has been designated.

To accommodate users without administrative (root) access, or those desiring a local sandbox installation, you should allow specification of another location, say `/tmp`, and then install below a `SimTK` subdirectory there, e.g., `/tmp/SimTK/`*yourProjectName*`/`… . Your installation procedure should create the `SimTK` subdirectory if it isn't already there.

The remaining structure ('…' and below) should be identical on any platform. (Substitute your favorite "slash" and top level directory below.) Organize the level just below `SimTK/`*yourProjectName*`/` like this:

```
include/     doc/
bin/         examples/
lib/         lib64/
etc/
```

This is an almost-standard organization for Unix-like systems but may be less familiar to Windows users. Nevertheless since there really is no standard Windows approach, you might as well do it the same way on all platforms and save yourself some headaches. Below is a quick description of what goes where:

`include/` `include/internal/`	This is for header files which are available for users to `#include` in their programs to provide the declarations and definitions they need in order to use your software. *Only* end-user targeted headers should be placed in `include`; if your headers require internal headers put them in a subdirectory named `internal` (this is common for template definitions, for example). Do not mix internal and user-includable files.
`bin/` `bin/internal/`	Executable program files, shell scripts, etc. go in `bin` ("binaries"). Again, do not mix executables required for internal operation with ones for users to execute directly; use the `bin/internal` subdirectory if necessary. For Windows installations, private dynamically linked libraries (DLLs) typically also go in the `bin` directory because Windows uses the same `PATH` search for executables and the DLLs they depend on. By "private" we mean DLLs that are needed only by a particular executable in bin and not intended to be accessible to others. On other systems you may be able to place dynamically linked libraries in `bin/internal` if users do not need to access them directly.
`lib/`	This directory is for linkable libraries that are intended for users to use. This should be used even on Windows for DLLs that are intended for users. There is no need to treat them as though they were executables since the user is not going to be placing his or her own executables in the same directory. Note that on Windows this implies that this `lib` directory will have to be on the `PATH` environment variable.
`lib64/`	When it is necessary to have separate 32- and 64-bit libraries for the same platform, put the 32-bit libraries in `lib` and the 64-bit ones (with identical names) in `lib64`.
`etc/`	Miscellaneous data files needed for runtime operation of your project go here. This can include databases, image files,
`doc/`	End user documentation, including Doxygen output if appropriate, goes here. Release notes should be placed in here as well.
`examples/`	Demo programs, programming examples, and so on go here. This is usually supplementary material to documentation found in the `doc` directory.

Any of the above directories can contain an "`internal/`" subdirectory for items that must be installed on the end user's system but are not intended for direct use by the end user.

Note that SimTK.org guarantees that project names are unique, but cannot make a similar guarantee for subprojects and other names which are under the sole discretion of the project administrator. The structure above guarantees that installations of any SimTK software can peacefully coexist on the same machine with any other, while given project administrators full discretion in managing anything beneath the project name.

5.2 Library naming conventions

There are many issues that need to be addressed for installed libraries. These include dynamically linked vs. static libraries, thread safety, operating system and build system quirks, dependencies on particular kinds of

hardware, 32 vs. 64 bit, number of CPUs, versioning of the library, and the versions of compilers and other libraries that a library is compatible with.

To be absolutely sure what library was linked with a running program requires a callable "about" routine to be built into the library itself. Any program built with a library can call this routine and then unambiguously determine exactly what is there. [see spec for SimTK Core "about" routine]. However, certain variants of libraries must coexist on the same machine and therefore must have different names or reside in different locations. Rather than doing this in an ad hoc way, we recommend that you follow the conventions set out below. All SimTK Core libraries use these conventions.

A few assumptions: we assume that most libraries will be dynamically linked and therefore want the "nicest" names to be allocated to the `.dll`, `.so`, and `.dylib` files (Windows, Linux, Mac, resp.). We are addressing only the issue of libraries installed for local use, meaning that we are not addressing "server" issues in which many unrelated libraries might have to coexist. We consider SimTK.org to be the server in that case, and installation means obtaining the best library for a single end-user machine. We are addressing installation only of libraries intended for use by others; if a library is merely a part of an application, you should put it somewhere that makes it easy for the application to find.

We are assuming the following directory structure, as described above:

Windows: `C:\Program Files\SimTK\`*yourProjectName*`\lib`
`\lib64`

Linux and Mac: `/usr/local/SimTK/`*yourProjectName*`/lib`
`/lib64`

Within these directories, we can accommodate up to four versions of each library—static and dynamic libraries available in both optimized ("release") and debuggable form. Note that the default form, i.e. the one that gets the nicest name, is the optimized, dynamic library. We add "`_static`" to the library name to indicate that it is a static library and to allow it to coexist with the dynamic library on Windows systems where the suffix is the same for either style of library. In practice, we like this unambiguous naming convention, so we use it consistently across all our supported platforms, even on Unix-derived systems where the "`.a`" suffix unambiguously means "static library." Each platform has a unique way of identifying dynamically linked libraries (also called "shared libraries"), and no platform has a uniform way to identify debuggable libraries. For the latter we append "`_d`" to the file name on all platforms. We try to adhere to all other platform conventions. Table 1 below spells out the naming scheme we recommend, and which we follow strictly for SimTK Core libraries:

		Windows	Linux	Mac
Dynamically linked library	Optimized	*name*`.lib` *name*`.dll`	`lib`*name*`.so`	`lib`*name*`.dylib`
	Debug	*name*`_d.lib` *name*`_d.dll`	`lib`*name*`_d.so`	`lib`*name*`_d.dylib`
Static library	Optimized	*name*`_static.lib`	`lib`*name*`_static.a`	`lib`*name*`_static.a`
	Debug	*name*`_static_d.lib`	`lib`*name*`_static_d.a`	`lib`*name*`_static_d.a`

Table 1: Recommended names for installed libraries, for all combinations of static/dynamic and optimized/debug. Note that our "`.so`" and "`.dylib`" files are not symbolic links as is a common convention; they are actual files.

For a variety of reasons it can be difficult to make dynamically linked libraries. You should try to provide them if you can because they have numerous advantages, the most substantial being that they can be updated individually and applications that use them can pick up bug fixes without having to be rebuilt or relinked. However, if you only have a static library available, you should follow the following naming conventions (assuming your library's base name is *name*):

Windows:

1. Install *name*`_static.lib` and *name*`_static_d.lib` as in the table above.
2. Install another copy of the same libraries under the names *name*`.lib` and *name*`_d.lib`. This allows a user to specify the default name, which will yield your static library for now but later will be the dynamic library when you have it. Only users who actually *want* static libraries should have to put "`_static`" in the name.

Linux and Mac:

1. Install `lib`*name*`_static.a` and `lib`*name*`_static_d.a` as in the table above.
3. Install another copy of the same libraries under the names `lib`*name*`.a` and `lib`*name*`_d.a`. This allows a user to specify the default name, which will yield your static library for now but later will pick up the `.so` or `.dylib` when you have it. Only users who actually *want* static libraries should have to put "`_static`" in the name.

Making optimized libraries can also be problematic occasionally. If you only have debug libraries, you should again install two copies, one under the default name and one with the proper "`_d`" suffix. That way a user who wasn't trying to get the debug version will still be able to link using the default name, and will eventually pick up the optimized version under the same name when it is released.

If you have a dynamic library but no static library, no problem—just install the dynamic library as in the table and leave out the static library altogether. Similarly if you have an optimized library but no debug available, just don't install any "`_d`" libraries. The general idea is to make sure that there is *always* a library available which has the default library name.

5.3 Thread safety

Some conventions allow both thread-safe (multithreaded) libraries and non-thread-safe libraries to coexist on the same platform. We don't. Here is our convention: *all libraries should be thread-safe*! We follow this policy strictly for SimTK Core modules, because to violate it in one library would mean that users would end up with dangerous thread-unsafe code, even if all other libraries were thread-safe. We feel that one of the biggest initial advantages to using SimTK over home-grown systems is its ability to effortlessly exploit tightly-coupled multi-CPU nodes to squeeze out more performance from affordable hardware. We also believe the current trend towards low-cost multi-core CPUs is likely to continue and expand, so thread safety will only become more important in the future. The past justification for non-thread-safe libraries has been to avoid the (usually tiny) overhead incurred in ensuring thread safety in certain contexts. But as Bryan Keller pointed out, if you are happy running on a single CPU then you must not have much of a performance problem. And in that case, why object to a small amount of overhead?

In any case, we won't accept a non-thread-safe library as part of the SimTK Core, and we suggest that you don't accept them in your project either! [See "how to write thread-safe code"]

5.4 Versioning & hardware-sensitive builds

Every library should have a version number. We recommend the following nearly-universal three-level version numbering scheme: *major.minor.revision*, in which each of *major*, *minor*, and *revision* is an integer and these are ordered from most significant to least. The major version number changes only when significant new functionality has been added, involving substantial changes to the API, possibly involving incompatibilities with lesser major version numbers (documented of course). The minor version number is used for bug fixes and lesser API changes which add to the API but maintain backwards compatibility. The revision number is used to reflect releases containing bug fixes to code or documentation, and performance improvements to the *major.minor* functionality. Upgrading to a newer revision should never require a change to the user documentation for a library, except to correct documentation errors or revise performance-related material.

It is important to note that major, minor, and revision numbers are under human control and are intended to convey some user-meaningful information. There will always be release notes associated with a unique version number, if only to document which bugs are fixed in the revision. Internally, there may also be a *build* number which is typically a serial number counting the number of Subversion (source code control system) check-ins made during development. This is a machine-generated integer, typically very large, and is not considered part of the version number since it does not have a user-meaningful interpretation. In some contexts the build number will be appended to the version number as *major.minor.revision.build.* For a more detailed discussion of software versioning, see Wikipedia: http://en.wikipedia.org/wiki/Software_versioning.

Some conventions attempt to handle version numbers through library naming. On Unix-based systems this is done with a multilevel system of symbolic links, so for example a library whose default name is `libx.so` will actually be a symbolic link to a particular *major.minor* version number, say `libx.so`→`libx.so.2.3`, and then in turn the version numbered library is a symbolic link to a particular revision, say `libx.so.2.3`→`libx.so.2.3.11`. This is a nice scheme, but it doesn't work on Windows. And since we needed a way to deal with Windows anyway, we use the same method on all our supported platforms. Our convention is: *we do not encode version numbering in file names*. However, we do understand that in some cases multiple versions of a library must be installed on the same computer. Here is our approach:

- In addition to individual library version numbers, we associate a *major.minor* version number (no *revision*) with a *project* release, which may include many libraries. You can think of this as a version number for a "package" (e.g. a `tar` file) which will in general consist of many components (with their own version numbers, including revision numbers) that have been tested together. The `SimTKcore` project groups several independently-developed libraries together this way.

- Everything installed under `/usr/local/SimTK/`*yourProjectName*`/` is expected to be the most-recently-installed (current) version of the package representing project *yourProjectName*. A file `version.txt` contained directly in the *yourProjectName*`/` directory identifies the package version number. Its contents are exactly *major.minor* on one line; nothing more or less. That way it can be used easily in scripts.
- When a new version of *yourProjectName* is to be installed, the contents of the old one are first moved down one level, to a subdirectory whose name is the old version number. As an example, if we are upgrading a project `foo` from `foo` 3.2 to `foo` 3.3, we follow something like this procedure:
  1. `cd /usr/local/SimTK/foo`
  2. `mkdir \`cat version.txt\``
  3. `mv * \`cat version.txt\``
  4. install version 3.3 into `/usr/local/SimTK/foo`
- After that referencing `/usr/local/SimTK/foo` yields the new current version, while `/usr/local/SimTK/foo/3.2` refers to the old version (assuming `version.txt` contained the single line "`3.2`"). So if you want header files from the old version, you would put `-I/usr/local/SimTK/foo/3.2/include` on the compile line. Old libraries can be referenced similarly.
- We define two standard entry points (function names) in every library which can be called to obtain complete "version" and "about" information, including specific hardware dependencies. It is only these binary functions that can provide complete confidence about exactly which libraries are being used in a given application. All SimTK applications should provide a way to access this information from their user interfaces. A good place for this is in the "about" screen if there is one. The SimTK Core specification includes a definition for the two routines; we use it in all Core libraries and recommend that you use it for yours as well.
- The information that is available from the about function is also present in the binary in string form and can be obtained using the Linux or Cygwin '`strings`' command. The strings always have the form "`SimTK` *keyword=value*". A command that usually yields just these strings is:
  `strings` *libfilename* `| grep "SimTK.*="`

5.5 Installation for SimTK Core software

First, for SimTK Core software the Mitiguy Dictum applies: the letters `s-i-m-t-k` are always written like this: `SimTK`. Yes, that includes directory names. Yes, function names. Yes, it includes namespaces. Yes it means the `ALL_CAPS` rule for preprocessor macros is slightly modified: `SimTK_EVERYTHING_ELSE_CAPS`. Etc. There is one exception: project names on SimTK.org are always all lowercase, even if they contain the string "simtk."

SimTK Core software is intended to work behind the scenes, generally in combination with other SimTK software, and here the primary goal is to ensure a smooth and predictable installation which plays well with other SimTK Core software. Multiple projects on SimTK.org are combined for installation purposes into a single "project" named `core`, with a single project version number including a full complement of Core modules.

This software should be installed as follows.

Windows: `C:\Program Files\SimTK\core\` …

Linux and Mac: `/usr/local/SimTK/core/` …

On Windows, any drive or drives may be used—there is no requirement to install on `C:`, and again note the space between "`Program`" and "`Files`". Be sure to use exactly the capitalization shown here for "`SimTK`" on every platform. All SimTK software that has dependencies SimTK Core software should check these locations first to resolve those dependencies if no other "sandbox" location has been designated.

The remaining structure (… and below) is identical on any platform. (Substitute your favorite "slash" and top level directory below.) The level just under `SimTK/core/` looks like this:

```
include/     doc/
bin/         examples/
lib/         lib64/
etc/
```

Below this level (e.g. `/usr/local/SimTK/core/include/`…) we have to be very careful to avoid collisions among the many SimTK Core modules we hope to have some day. Many of those will have include files named "`common.h`", "`types.h`" and so on. The solution is to require the unique SimTK project name to appear next. The idea is that for a SimTK Core project whose SimTK.org "Unix name" is *`myCoreProject`*, the SimTK standard installation structure reserves all of the following for that project:

`myCoreProject``/`
`myCoreProject``.`*anysuffix*
`myCoreProject``_`*anything*

Note that this means that SimTK Core "Unix names" should not contain underscores or dots.

This organization is used consistently in all the subdirectories (include,bin,lib,…) so that project administrators do not always have to use project subdirectories as long as they limit their top level names to the ones above. For example, each project gets to have a single header file like `/usr/local/SimTK/core/include/myCoreProject.h` if that is useful, with everything else going in `/usr/local/SimTK/core/include/myCoreProject/...` . A project will not necessarily have a presence in all the subdirectories, just those that make sense for that project. Note that for SimTK Core software, the order of the include, lib, etc. directories is inverted as compared to non-Core SimTK projects as described above. That is because "`core`" itself is considered to be the project in this case, with the individual project names being relegated to subproject status here.

More specialized files which will be used by some users should be placed immediately below the (sub)project name, e.g. `core/include/myCoreProject/AdvancedUsers.h`. Installed files which must be available but are *not* intended for users to access directly (e.g. low level header files or template expansions which must be included by user-visible friendly headers) go in a further subdirectory "`internal`", e.g.

```
/usr/local/SimTK/core/include/myCoreProject/internal/templateExpansionDefs.h.
```

This can also be used for the odd little utility needed at run time, which can be installed in `bin/internal/`, and `internal/` may appear in other subdirectories also if needed. The idea is to hide the ugly stuff so that a look in the upper subdirectory shows only files which have some meaning to end users. Keep your dirty laundry in the closet, not in the middle of the living room.

Note that a SimTK Core project which intends to use this installation structure should organize its source code tree (Subversion repository) so that header file names in the development tree can match the installed ones. That is, we want includes to look like:

```
#include "myCoreProject.h"
#include "myCoreProject/blah.h"
```

```
#include "myCoreProject/internal/blahMessyStuff.h"
```

or whatever. This implies that wherever there is an "`include`" directory in your subprojects, you'll need to repeat the project name below it and put the actual headers in that project directory if they are going to end up that way when installed. Note that this does not apply to anything completely internal to your project, just things that are going to get installed on an end-users system. (Reminder: the "`include`" directory is not a dumping ground for all headers! It is only for headers which are to be copied into the installation directory. Private headers should be in the `src` subdirectory along with the rest of the code.)

6 Structure of a SimTK Development Project

A SimTK *project* is hosted on SimTK.org with its source code managed in a single Subversion repository which is used solely for that project. A single project can be a very substantial entity involving many people and in general will consist of many *subprojects* (or *modules*; the terms are interchangeable), mailing lists, user communities and forums, bug tracking, build and release management, downloads, project governance, etc. Note that while "project" is an entity understood and actively supported by the SimTK.org infrastructure, the meaning of "subproject" or "module" is entirely up to a project's administrator. Most subprojects will contain software, but that is by no means required.

In deciding whether to create a new project rather than a subproject, we recommend this distinction: *projects* should be organized around groups of people with related interests; *subprojects* organize content rather than people. Projects are largely self-contained while subprojects can be closely interdependent, even at the source level.

Here is an example of the recommended organization for the top-level, Subversion-managed area of a project. Note that each subproject can be a separately branchable Subversion entity, however everything below a Subversion `trunk` is tagged and branched together; don't try to nest `branches, tags, trunk` directories.

```
SimTKProjectName/
    SubprojectOne/          SubprojectOne/      ASoftwareModule/    AnotherSubproject/
    ASoftwareModule/            branches/           branches/           SubSubProject/
    AnotherSubproject/          tags/               tags/                 branches/
    . . .                       trunk/              trunk/                tags/
                                                                          trunk/
                                                                      SecondSubSub/
                                                                      . . .


                    ASoftwareModule/trunk/
                        doc/            src/        Doxyfile
                        etc/            examples/   CMakeList.txt
                        include/
```

Software subprojects should be organized in the manner shown for `ASoftwareModule`. This is a de facto standard for Unix- or Linux-based programming and works fine in other environments as well. Note that this is the organization for software *development*; the organization that users see when the software is installed is described in section 5.2. This organization by no means precludes development with Visual Studio on Windows, or Eclipse, or Emacs, or practically any other popular way to develop code (we use *CMake* to generate the appropriate makefiles or project files when needed).

Note that by `include` here we mean "header files which comprise the *external* interface to the module." This is not intended as a dumping ground for all header files; any private headers belong in the `src` subtree. During installation it should be reasonable to copy the entire contents of the `include` directory as part of the install; there should be no mysterious internals exposed there. Note that to support the SimTK standard installation structure as described in section 5.2, you should always repeat the SimTK project name as a subdirectory of `include`, and then put your actual include files in the that subdirectory. This allows the include

statements in your development source code to be identical to the ones in user code, and it is well worth the annoyance of the extra level of directory. For ugly internals that must nevertheless end up in the installation, use an additional subdirectory `internal/`. See section 5.2 for details.

`Doxyfile` is a text file which provides instructions to the *doxygen* (www.doxygen.org) program for generating beautified source documentation. If you want cross-referenced documentation for several related modules, put the `Doxyfile` instead in the parent directory of those modules. `CMakeList.txt` is a text file which provides instructions to *CMake* (www.cmake.org) regarding how to build the software module. A `CMakeList.txt` file appears in every source subdirectory of a given module. Note: you do not *have* to use CMake or Doxygen! But if you choose to do so we recommend you organize their files as above.

[TODO: Does this structure need to be modified for development in Java or other interpreted languages?]

6.1 File suffixes

The following table gives our preferred suffixes for various kinds of files.

C++ code	`.cpp`
C++ header	`.h`
C++ template source	`.h`
Java source	`.java`
Python source	`.py`
Perl source	`.pl`
Tcl source	`.tcl`
C code	`.c`
C header	`.h`
Fortran code	`.f`
HTML document	`.html`
man page	`.[1-9][a-z]`
plain text	`.txt`
Microsoft Word document	`.doc`
Adobe portable document format	`.pdf`

7 Interfaces

An interface (or API) must satisfy very restrictive criteria compared to general programming. Stability is the primary one—an interface should be extremely stable once defined because many programs will depend on it for their correct functioning.

SimTK sets the bar higher by promising binary compatibility for the SimTK Core. That means that software that depends on the SimTK Core interface can take advantage of new releases without being recompiled or

relinked in the case of dynamically linked library upgrades, where the major version number has not changed. This level of stability requires that objects which appear in the interface must have opaque implementations. Typically this means that interface classes have only a single data member, an opaque pointer (a.k.a. a "handle") to the classes hidden implementation.

Although the implementations are exposed, we do permit use of the C++ standard template library containers like `std::vector` and `std::string` in interfaces because we have determined from discussions with C++ compiler writers that they are painfully aware of the binary compatibility issues and actively seek stability for those classes, to the degree that they are likely to be as stable as other compiler-induced binary compatibility difficulties such as changes to the calling sequence. This has the advantage that end users are already very familiar with these classes and know how to use them effectively, and it substantially reduces the size our API would be if it also had to provide the STL functionality.

8 SimTK Conventions for C, C++, Java

These conventions apply to C-like languages. Conventions for other languages will be added when we have some.

8.1 Capitalization and spelling

We do not believe it is helpful to attempt to encode too much information into symbol names; consequently we have fewer conventions than many systems. Much of the need for such conventions has passed with the wide availability of interactive language-sensitive code browsing and debugging, such as that provided by Visual Studio or Eclipse. Thus we do not use code letters to provide information that can easily be obtained while browsing code or debugging. We trust programmers to add appropriate conventions in their own code when those conventions are necessary for clarity or convenience, and to explain them in nearby comments.

We reserve the ugly `ALL_CAPS_CONVENTION` for preprocessor macros both to discourage their use and to signal the out-of-language loophole being employed. In particular, we discourage the use of preprocessor macros for constants and use a different, less violent convention for in-language constants.

We prefer consistency with existing precedent to our own conventions whenever appropriate. For example, C++ containers like `std::vector` define standard names like `const_iterator` so if you are building a container intended to be compatible with one of those you should follow the existing precedent rather than use the SimTK convention which would have been `ConstIterator`.

Types: classes, typedefs, structs, enumeration types	Names should be nouns with initial cap and cap for each word	`System`
		`StateClient`
		`Vector`
		`Real`

Functions and methods	Names should begin with a verb, start with lower case, initial cap for each subsequent word	`getBodyVelocity()`

Variables (includes local, global, members, etc.)	Generally descriptive noun phrases, start with lower case, caps for each word. Spell things out unless there is a compelling reason to abbreviate. Follow conventional exceptions for mathematics and indexing.	`fileName`
		`nextAvailableSlot`
		`i,j,k`

Constants (defined within the language, using `enum` or `const`).	Initial cap, cap for each subsequent word (same convention as for classes).	`enum Color {`
		`  Red,`
		`  Blue,`
		`  LightPink`
		`};`
		`static const Color AttentionColor = Red;`

Preprocessor macro (includes both constants and macros with arguments)	All caps, words separated by underscores. When these appear in interfaces they must be prefixed with a distinctive prefix to keep them from colliding with other symbols. Use an all-caps version of the associated name space when possible. The names of all macros from SimTK Core software are prefixed with "`SimTK_`".	`SimTK_DEBUG`
		`MYMODULE_UGLY_MACRO`
		`MYMODULE_UGLIER_ONE`
		See section 8.2 below for conventions which apply specifically to preprocessor names used as header guards.

Namespaces	Short, cryptic, low probability of having the same name as someone else's namespace. We reserve namespaces containing '`SimTK`' (in any combination of upper/lowercase) for SimTK core modules.	`std::`
		`SimTK::`
		`xsimtk::`
		`simtk_impl::`

8.2 Header guards

It is conventional in C and C++ programming to surround the contents of every header file with "header guards" to permit header files to be included multiple times in a single computation unit with no ill effects. A typical SimTK header guard looks like this:

```
#ifndef SimTK_SIMBODY_COMMON_H_
#define SimTK_SIMBODY_COMMON_H_

    … the header contents …

#endif // SimTK_SIMBODY_COMMON_H_
```

There are a few important things to note about our header guard conventions:

- The names follow the preprocessor convention mentioned above: all caps preceded by "`SimTK_`"
- There is no leading underscore at the beginning of the name (despite common use, that convention is not allowed by the ISO C++ standard since it reserves names beginning with an underscore followed by a capital letter for the implementation).
- Most important: the header guard name *must be unique* across all SimTK Core components! Different components are likely to have headers of the same name (`common.h` for example being, well, common), yet they will all end up included together in the SimTK Core user's compilation units.

To ensure that this last issue doesn't cause problems, we use the following convention for header guards in SimTK Core headers:

`SimTK_`*<project_name>*`_`*<header_file_name>*`_H_`

The *project name* is an all-caps spelling of the unique SimTK project name (e.g. `SIMBODY`, `SIMMATH`, `SimTKCOMMON`), with the usual exception that "`SimTK`" itself has the capitalization shown. The *header file name* is an all-caps version of the name of the file for which this is the header guard, with words separated by underscores. The other characters (in bold) should appear as written. The trailing underscore (permitted by the standard) is just there to make the name less likely to collide with anything but probably is overkill.

8.3 "const" correctness

This probably belongs under section 8.5, but it is so important it gets its own section here. One of the best features of C++ (and to some degree ANSI/ISO C), unfortunately lost in Java, is the ability to write a method signature so that the compiler can guarantee that an argument or data member will not be modified. This is specified using the "`const`" keyword. A program which uses `const` wherever it is appropriate, and propagates constness throughout, is called "const correct." It is very difficult to take a non-const correct program and make it const correct later; that should be designed in from the start.

In addition to catching many otherwise difficult-to-find or worse, unnoticed, bugs const correctness can have a direct impact on performance. A large data structure which must not be modified can be passed *by reference* (i.e., by address) safely to a black-box routine that declares that argument const. Conscientious programmers who would otherwise copy the data to ensure its integrity do not need to do so in that case, providing a large savings in memory use and often in run time performance.

All SimTK Core software which is written in C or C++, or provides a C or C++ interface, must be const correct. We highly recommend this strategy for all programmers. It works.

8.4 Code layout and indentation

The goal here is to balance code clarity with a strong desire to pack a lot of code into a reasonably small space to allow a reader to see as much as possible simultaneously. This requires some basic rules, subject to numerous exceptions since no set of rules has yet been discovered which is optimal in all circumstances. We trust programmers with the responsibility of making use of these guidelines intelligently. Your code should be easy for others to read and understand; ugly hard-to-understand code is bad, even if it follows the rules. Do what you have to do to make it clear.

8.4.1 Order of public and private class members

Public ones should come first. Don't make people look at your dirty laundry in order to use your classes. Start with the basic constructors (and copy assignment in C++). Then put important likely-to-be-used methods first, relegating obscure bookkeeping stuff to the end.

Avoid public data members; use inline accessors instead. Even protected data members should be viewed suspiciously, especially if you expect people other than yourself to be deriving classes from yours. Occasionally this seems silly, especially for simple "plain old data" (POD) classes as described in the C++ standard. In that case you should at least put your public data members at the beginning of your class declaration so that they appear as part of the public interface rather than buried with the private stuff at the end. Use good judgment!

In external interfaces, even *private* data members can be problematic. See the section on achieving binary compatibility—in an interface it is often best to have a single private member, essentially a void* pointing to the opaque implementation.

8.4.2 Indenting

Regarding indentation, the most important rule is: *no tabs*. However much you indent, it must be done with *spaces* rather than tabs so that alignment will be preserved when someone else looks at it, even if their tab settings don't match the ones you originally used (and they probably don't). While it is easy for an automated program like `indent` to alter general tab settings, it is an AI project to capture a user's original intent in alignment within comments, or between comments and code, and in many cases in code itself. Note: this doesn't mean you can't use tabs when you enter your code, just be sure to set your editor so that it inserts spaces for tabs rather than actual tab characters.

SimTK uses a standard indentation unit of *four* spaces. Although we understand that many people like to use two spaces to maximize the amount of available horizontal real estate, we feel this is not enough to make the

indentation structure obvious at a glance. Also, code which is so deeply nested that it cannot accommodate a four-space indent probably ought to be restructured anyway. Many other programmers use eight spaces since that is the default in some editors. We find that excessive.

8.4.3 Curlies

We do not like to see a lot of content-free lines using up vertical space in code and consequently prefer the style sometimes called "the one true brace" over conventions which attempt to align all paired braces. Here are some examples:

```
if (a <= b) {
    // some code
} else {
    // some more
}

int myFunction() {
    // function body begins here
}

class MyClass {
public:
    // public members
};
```

When there is only a single statement within a control structure, there is no need for braces and we prefer that they not be used since that saves space. The primary means for conveying the scope of control to human readers is indentation. It matters a lot more that the indentation is right than where the braces are.

8.4.4 Use of spaces

Add spaces where they improve clarity, otherwise leave them out. In particular, parentheses do a fine job of surrounding if and for conditions and do not require further setting off with spaces. On the other hand, operators within those conditions are sometimes hard to spot and worth setting apart. For example, we prefer

```
if (nextItem <= minItemSoFar)
for (int i=0; i < length; ++i)
```

to

```
if ( nextItem <= minItemSoFar )
for ( int i=0; i < length; ++i )
```

TODO: need more here.

8.4.5 Placement of '*' and '&' in declarations

One might wish that declarations in C were roughly of the form "*type var1, var2;*" and for the most part they are. However, declarations of variables of pointer type in C unfortunately do not work this way and C++ had to inherit this behavior for backwards compatibility, both for pointers and references. A typical C declaration looks like this:

```
char *p, *q, ch1, ch2;  // ← don't do this
```

Although this is legal in C++, most C++ programmers (including us) prefer to follow a convention which permits one to think of the '*' as being associated with the type rather than the variable. This would require the above to be written:

```
char* p;
char* q;
char  ch1, ch2;
```

An often nicer alternative for dealing with pointers is to use a `typedef` to create a pointer type, for example:

```
typedef char* PtrToChar;
```

which permits all declarations to use the same syntax:

```
PtrToChar p, q;
char      ch1, ch2;
```

(OK, maybe we wouldn't use this for simple char pointers, but it is the only decent way to deal with function pointers and other complicated pointer types.) In any case, we follow the convention that the '*' and '&' type modifiers are placed adjacent to the types they modify and if necessary forego the convenience of multiple variable declarations in a single statement.

8.5 Use of specific language constructs and facilities

Beyond stylistic issues, there are certain features of languages (especially C++) which when used properly can substantially clarify code readability and/or efficiency. Here are a few. Please send more if you have them.

8.5.1 Pre- or post-increment?

This one is easy: always use pre-increment if you have a choice. That is, prefer `++i` over `i++` when the difference in their behavior is not significant. The reason is that the post-increment operator must make a copy of the previous value, while pre-increment just returns a reference to the new value. The copy is a waste which can be significant when these operators are overloaded. Compilers are not always free to optimize away the copy because semantics may require invocation of the copy constructor, which can of course have significant side effects.

Typically, experienced C programmers have a long standing habit of using post-increment, and C's successor is even called C++ rather than ++C. But, you can get over it—if you haven't already retrained yourself, this is the time!

8.5.2 Pointers vs. references

In C++ it is much less necessary than in C to use pointers, because of the addition of the reference-to-object type modifier "&" to C's pointer-to-object type modifier "*". We highly recommend using references whenever possible, especially in interfaces. The resulting code is much easier to use and to read, and the compile-time type checking is more stringent and informative. Run time execution is as efficient as pointers, but the code reads as nicely as if arguments had been copied rather than referenced.

8.5.3 Use of iostream vs cstdio

Yes, iostream is clever and extensible. In many cases it is far superior to stdio, especially when providing I/O operators for new classes. However, many programmers find it inconvenient to use, especially for formatted output. We remain fans of the original C stdio system, especially `printf()`, `scanf()` and their relatives and prefer that over iostream in many, but certainly not all cases. Conveniently, the C++ Standard (ISO 2003, §27.3, 27.4.2.4) specifies that the standard I/O stream objects (`cout`, `cin`, etc.) must be the same as the corresponding stdio objects (`stdout`, `stdin`, etc.), unless the programmer explicitly requests otherwise, and can be interspersed on a character-by-character basis. So both kinds of I/O can be freely intermixed in a conforming C++ program. However, be sure to use the C++ header `<cstdio>` rather than the C header `<stdio.h>` so that the stdio routines are properly sequestered in the `std::` namespace.

9 Commenting source code

Every source and header file should begin with a comment containing the SimTK copyright and licensing terms. Boiler plate for this can be grabbed out of any existing file, and is also provided in

Appendix A – SimTK license below.

We encourage the use of Doxygen but with a few caveats:

- Doxygen comments and directives should be used in a minimally intrusive way so that the code remains directly readable.
- Doxygen alone is not sufficient as a way of documenting source code. It is a great way to cover the details, but falls far short as a way to convey the big picture. That requires written text.
- Interfaces generally deserve different treatment from internal code. Be sure that your Doxygen-generated documentation can be presented to users without drowning them in internal details.

TODO: what does that mean in practice?

10 Statelessness

In order to permit arbitrary composition of modules, it is critical that those modules are stateless. This is also helpful for ensuring thread safety. This means that the actual state should be external to the object that needs it; it is passed in from outside rather than stored internally in the object.

The SimTK Core provides a general-purpose class State which implements this concept and is used extensively with the SimTK Core.

11 Binary compatibility issues

At a minimum, every substantial SimTK Core module should be built as a separate library. However, we prefer dynamically linked libraries whenever possible and provide tools and guidance to support the creation of such libraries.

Either way, we expect that later releases and builds of such a module are *binary compatible* with earlier releases, except on rare occasions accompanied by a change in the module's major version number. That means that an existing application (client) which used an earlier version of the module can run with the new one without recompiling and, in the case of dynamically linked libraries, without relinking, even after a change to the minor version number. This permits bug fixes to be incorporated into already-existing clients without any need to rebuild those clients.

Besides the obvious requirements for binary compatibility (e.g., don't change anything in the interface) there are several much less obvious requirements. The most important is that this rule must be followed: any knowledge of an object's memory layout must be restricted to one side of the interface or the other. That is, if an object is allocated on the client side, it must be manipulated entirely on the client side, and similarly for objects allocated on the library side. An exception to this is any object whose layout can reasonably be expected never to change (such as a 3-vector represented as three consecutive doubles). In practice, one can have this confidence only for very simple objects; anything else is likely to change over time.

The practical result of this requirement is that any substantial classes appearing in the interface should be implemented using an opaque pointer (essentially a void*) as its sole data element. This is called a "handle" class. The real class remains entirely hidden on one side of the interface, most commonly on the library side. SimTK provides a small set of templatized base classes which automate the management of handles and their implementations.

Abstract classes must be handled very carefully. If you can, avoid them in interfaces. When necessary, keep in mind that although they may appear to have no data members, they actually contain a hidden set of data members called the "virtual function table." This table's layout must be known on both sides of the interface, a violation of the above requirement for binary compatibility. In practice, the table is filled in precisely in the order virtual functions appear in the class declaration, and correspondence is done by position. Thus an insertion or rearrangement of the class declaration on the library side will cause it to call the wrong virtual func-

tions for an object which was allocated on the client side and then passed across the interface. Needless to say, that is very, very bad!

Another difficulty is the use of enumerated types (`enum`) in interface methods. There are two problems: one is that if you don't specify the numerical values for the enumerated constants they are assigned by the compiler and will change whenever new constants are added, deleted or rearranged in subsequent releases. This problem is easily addressed by specifying the constant values and treating them as sacrosanct in later releases. A more difficult problem with `enum` is that the C++ standard permits the compiler to choose as a physical representation for an `enum` the smallest size integer that can hold all of its values. For most enumerated types, that means the compiler can choose a char (8 bit) integer, but if any enumeration value is greater than 255 then a larger integer must be used. Obviously this can cause a problem if a later release adds enough new values to exceed the size limit, or adds a particular enumeration which has a large numerical value. Less obviously, different versions of a given compiler are free to change the size integer they use, so the `int` passed through the interface may chance size just because a later version of the compiler is more frugal with space than an earlier one was, even if no code has changed.

TODO: An extensive discussion and code examples will serve to clarify how to ensure that your code will be binary compatible over many releases.

11.1 Private Implementation

The single most important tool for achieving binary compatibility is to make sure that the *implementations* of SimTK classes are separated from their *declarations*. This is a widely used C++ design pattern called "private implementation" or "PIMPL." Every PIMPL object is defined by two classes instead of one – the first, called the "handle" is declared in an API-visible header file and defines the interface seen by the API user. The second, called the "implementation" is referenced by the handle but not declared anywhere visible.

Handle declarations are made public in SimTK header files which are to be included in API-user's programs, and objects allocated in user code from those declarations are constructed by the user's compiler in accordance with the visible declarations. If the implementation were part of that declaration then user code would become dependent on the specifics of the implementation, preventing binary compatible implementation changes. So handle class declarations instead specify no data other than a single pointer to an undeclared class (essentially a void*), representing the private implementation. The handle class is guaranteed to consist of nothing but a single pointer forever, that is, through any number of future SimTK Core releases, while the implementation can change in any manner as long as it still supports the original interface.

> **SimTK CONVENTION:** A PIMPL Handle is just a pointer
>
> Every SimTK handle class is a C++ POD ("plain old data") class consisting of exactly one pointer *and nothing else*! That means all handle subclasses derived from a handle base class, whether built in or user defined, must be *dataless*.

In the internal implementation, it is very useful to be able to find the handle given the implementation. This is done by having a "back pointer" from every implementation object to its corresponding handle object. The referenced handle is called the "owner handle;" it is possible for other handle objects to reference the same implementation but those are not owners. Owner handles are responsible for managing the heap space associated with their implementation objects. That is, a handle class destructor checks to see if it is the owner of the implementation it references and if so deletes the implementation.

SimTK CONVENTION: A PIMPL Implementation always points back to its owner Handle

The SimTK Core uses the PIMPL design pattern in three ways, depending on the desired behavior of the object being defined. The following table describes the three types of PIMPL object.

Simple PIMPL	Handle class constructor allocates an implementation object, which points back to the handle.
Polymorphic PIMPL	Here the object is polymorphic and the Handle class is conceptually an abstract base class from which concrete Handles are derived. No implementation object is allocated by the Handle base class, and the opaque type it refers to is an actual abstract class in the C++ sense. The concrete Handle constructors allocate the appropriate concrete implementation object.
Extendable PIMPL	Provide the ability for a user to implement a subclass of a PIMPL class hierarchy. In this case the user derives two classes: (1) a dataless handle derived from the appropriate handle superclass, and (2) a concrete implementation class derived from the associated virtual implementation class. The user's handle class's constructor allocates a concrete implementation and passes it to the superclass handle to be registered with the library side implementation. At run time the library side implementation will call out to the user's concrete class, using a virtual function map to ensure binary compatibility.

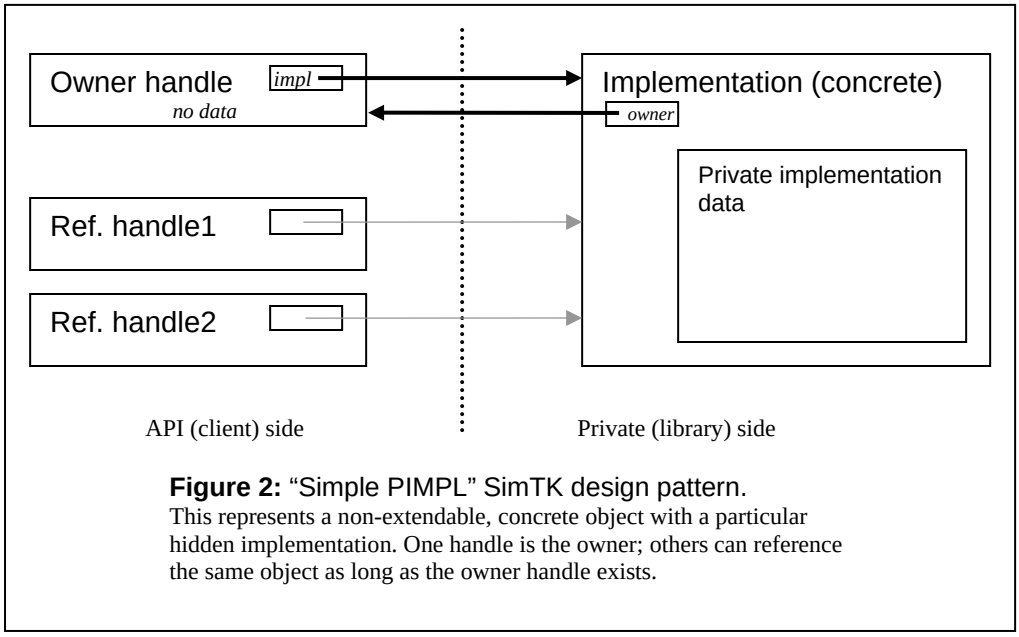
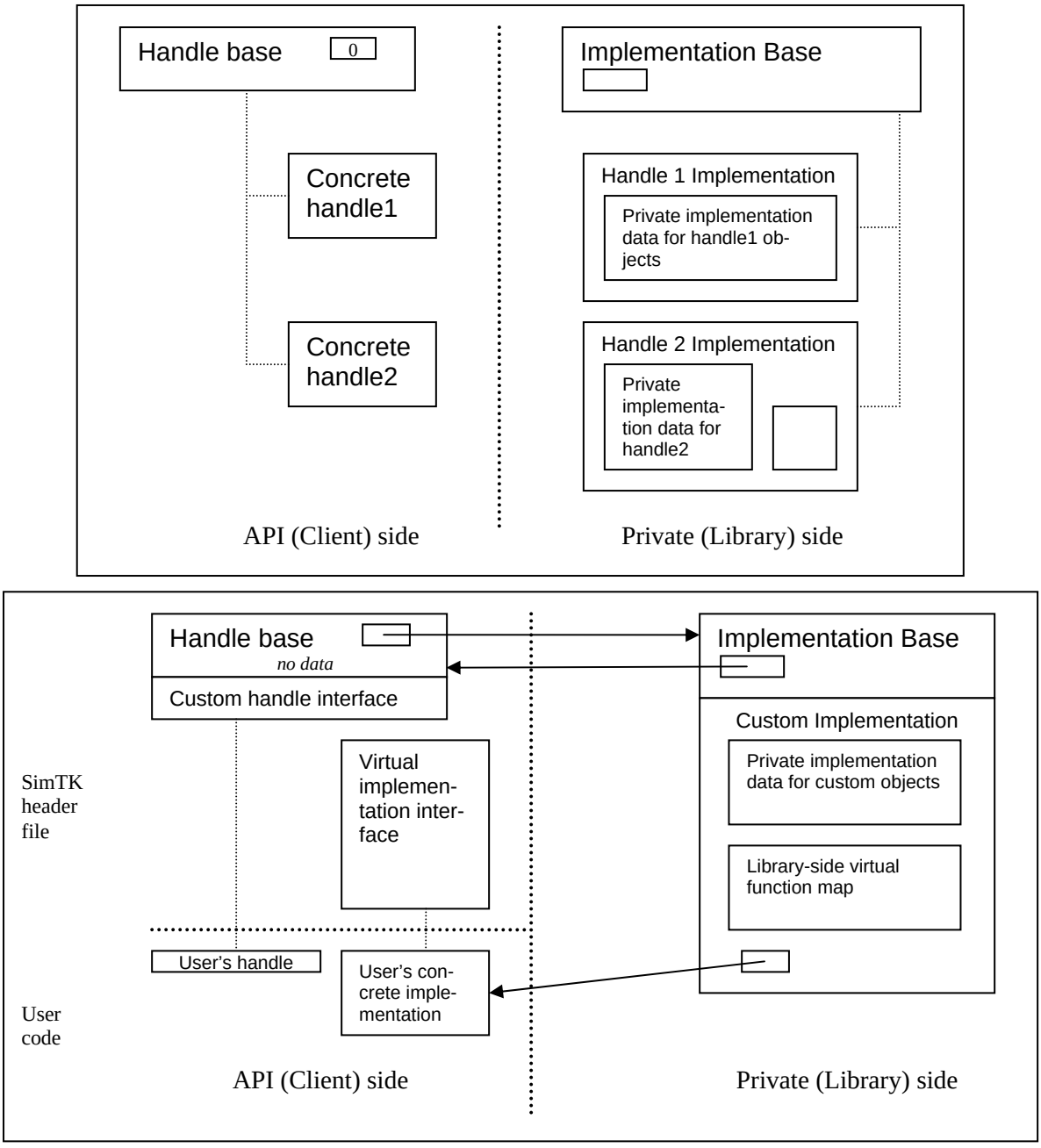


Figure 2: "Simple PIMPL" SimTK design pattern.
This represents a non-extendable, concrete object with a particular hidden implementation. One handle is the owner; others can reference the same object as long as the owner handle exists.



SimTK provides a set of templatized base classes which automate the above relationships among handles, their implementations, and user extensions. See the MobilizedBody and Constraint classes for examples.

12 Multi-language issues

The primary issue here is the combination of modules written in different languages to form a single application. It occurs primarily in two ways:

- High level applications written in interpretive languages would like to access low-level SimTK Core libraries which are typically written in C++.

- Low-level numerical algorithms written in Fortran, C or assembler should be callable from higher-level SimTK code in C++ or directly from application code written in Java or other interpreted languages.

We address these by using ISO C++ (including its standardized subset ANSI/ISO C) as the lingua franca. Low level numerical algorithms must be wrapped to present a C++ compatible interface (often just ANSI C). Provision of a Fortran 77 interface is encouraged but not required although we encourage support for the native-language interface as well as the C++ wrapped interface.

At the other end, SimTK Core C++ interfaces will be wrapped, typically using the open-source interface generator SWIG (www.swig.org). SWIG generates wrappers for specific interpreted languages, including Java, Perl, Python, and Tcl/Tk and Ruby.

The need to add C++ wrappers to low level numerical routines does not generally impose any particular requirements on those routines. SWIG, however, does have limits on the C++ constructs it can successfully wrap. TODO: this will be discussed in a separate document.

13 Supported hardware platforms

We define a small set of popular hardware/OS configurations on which all SimTK Core software runs. We intend to provide binaries for all these platforms. We expect the list to change over time. Here is the current list:

- 32 & 64-bit Windows on Intel & AMD single- and multiprocessors.
- 32 & 64-bit Apple Macintosh systems running OS X (version TBD). We will support only Intel-based Macs, including multiprocessors.
- 32 & 64-bit Linux platforms. We initially support only Red Hat Enterprise Linux, including shared-memory multiprocessor systems on Intel and AMD x86 hardware. Cluster support will come later.

SimTK Core code should be written so that it can be deployed on these platforms. SimTK.org provides extensive infrastructure to support multiplatform deployment.

14 Parallel processing issues

Initially we will provide direct support only for threaded parallelism on affordable, tightly-coupled shared memory multiprocessors. As a next step, we will add support for clusters (possibly restricted to Linux clusters) using MPI for parallel computation. SimTK software may assume that any hardware it sees is completely available to it; any resource-sharing issues must have been handled at a higher level and we consider those issues outside the scope of SimTK development.

14.1 Thread safety

The SimTK Core is conceived as a system which can routinely exploit shared-memory multiprocessors when they are available. The most common techniques for this sort of parallelism involve several cooperating threads which run on different processors but share a single address space. Each thread will expect to have independent access to basic computational algorithms and library routines (matrix multiply and sort, for example). This is impossible if these algorithms maintain internal state using the shared address space (which means the algorithms must not employ writable global or static variables or common blocks).[†] Any persistent memory must instead be passed in from outside the computational routine, which must then restrict its addi-

[†] Persistent state can be allowed in some cases if the module is written with specialized control structures which ensure correct behavior during simultaneous access by multiple threads. System service libraries and drivers are typically written in this manner but it is a very specialized skill which is unlikely to be possessed by most SimTK programmers.

tional memory use to stack variables and heap data tracked by stack pointers. Code with this property is called “thread safe” and is required for any SimTK code which is intended for general reuse.

TODO: What are other thread safety issues?

Appendix A – SimTK license

The following comment should be placed verbatim at the beginning of every source file (right after the header guard for header files). This is a maximally permissive, non-viral license, often called generically a "BSD-type license" although this one is actually derived from the MIT license originally developed for the X-windows system. It allows SimTK code to be used by anyone for any purpose, commercial or non-profit, with minimal burden. Use of this license is required for code that is part of the SimTK Core. It is recommended for other SimTK software, but project administrators are free to choose the restrictions they want.

```
/* ---------------------------------------------------------------------- *
 *                        SimTK Core: module (e.g. Simmatrix)             *
 * ---------------------------------------------------------------------- *
 * This is part of the SimTK Core biosimulation toolkit originating from  *
 * Simbios, the NIH National Center for Physics-Based Simulation of       *
 * Biological Structures at Stanford, funded under the NIH Roadmap for    *
 * Medical Research, grant U54 GM072970. See https://simtk.org.           *
 *                                                                        *
 * Portions copyright (c) 20yy-yy Stanford University and the Authors.    *
 * Authors: primary authors                                               *
 * Contributors: others whose contributions should be acknowledged        *
 *                                                                        *
 * Permission is hereby granted, free of charge, to any person obtaining a    *
 * copy of this software and associated documentation files (the "Software"), *
 * to deal in the Software without restriction, including without limitation  *
 * the rights to use, copy, modify, merge, publish, distribute, sublicense,   *
 * and/or sell copies of the Software, and to permit persons to whom the      *
 * Software is furnished to do so, subject to the following conditions:       *
 *                                                                            *
 * The above copyright notice and this permission notice shall be included in *
 * all copies or substantial portions of the Software.                        *
 *                                                                            *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR *
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,   *
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL    *
 * THE AUTHORS, CONTRIBUTORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM,    *
 * DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR      *
 * OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE  *
 * USE OR OTHER DEALINGS IN THE SOFTWARE.                                     *
 * ---------------------------------------------------------------------- */
```

Acknowledgments

This work was funded by the National Institutes of Health through the NIH Roadmap for Medical Research, Grant U54 GM072970. Information on the National Centers for Biomedical Computing can be obtained from http://nihroadmap.nih.gov/bioinformatics.

Many thanks to David Paik for his insightful comments on an early draft of this document. His contributions, both in details and in overall organization improved it substantially.

References

[TBD]