
BASIS

Release v1.3.0

Andreas Schuh

June 09, 2012

Contents

1	Download	ii
2	Installation	ii
2.1	Prerequisites	ii
2.2	This Software	ii
3	Documentation	ii
3.1	Software Manual	iii
	The Standard	iii
	How-to Guides	xxi
3.2	API Documentation	xlii
4	Tutorials	xlii
4.1	Quick Start Guides	xlii
4.2	Tutorials	xlii

The Build system And Software Implementation Standard (BASIS) project was started early in 2011 in order to improve and standardize the software packages developed at [SBIA](#). Based on the decision to use [CMake](#) and its accompanying tools for testing and packaging software, the standard for building software from source code was based on this popular, easy to use, and yet powerful cross-platform, open-source build system. The previously used CMake Project Template was entirely reworked and became a major component of BASIS. In fact, the BASIS project evolved from this initial project template and greatly improved it. See the [Software Project Template](#) of the [Build System Standard](#) for a details on the template.

See the [Implementation Standard](#) for details on the standard for implementing software.

[How-to Guides](#) which help to understand the standard and how to comply with it can be found here as well.

Projects following the standard include the [BASIS Modules](#) and make use of the [BASIS Utilities](#). They are hence dependent on the BASIS package, similarly to a software implemented in C++ depends on third-party libraries used by this implementation, for example. Therefore, in order to be able to build a BASIS project, the BASIS package has to be installed. Note, however, that BASIS is not necessarily required during the runtime, i.e., for executing the software. This depends on which utilities are used by the software.

The BASIS [Introduction](#) slides further explain its purpose and the parts it is made of ([ref](#)).

1 Download

The BASIS software is freely available under a BSD-style open source license that is compatible with the Open Source Definition by [The Open Source Initiative](#) and contains no restrictions on use of the software.

The full [license](#) text is included with the distribution package and available online.

Please fill [this online form](#) with the appropriate information to receive an email with the download links.

2 Installation

2.1 Prerequisites

See the *Prerequisites* listed in the *Build and Installation* guide for a list of software tools required to build BASIS. These are further the tools required to build any software which is based on BASIS. Moreover, BASIS makes use of the following software packages:

Package	Version	Description
ITK	3.14	The standalone <code>basistest-driver</code> executable currently makes use of the ITK, either version 3 or 4 and above, for the comparison of a test image to one or more baseline images. If no installation of this library is found, this executable is excluded from the build and installation. Note that many packages developed at SBIA make use of this executable in their tests. If BASIS has been built without the <code>basistest-driver</code> , these packages have to be build with the <code>BUILD_TESTING</code> option set to OFF (the default).
MATLAB	R2009b	The MATLAB tools are used by BASIS to build MEX-Files from C++ sources. If <code>BUILD_TESTING</code> is set to ON and the MEX script is found, the tests for the build of MEX files are run. Otherwise, these are excluded from the test.
MATLAB Compiler	R2009b	The MATLAB Compiler (MCC) is used by BASIS to build standalone executables from MATLAB source files. If <code>BUILD_TESTING</code> is set to ON and MCC is found, the tests for the build of such binaries are run. Otherwise, these are excluded from the test.

For instructions on how to build or install these prerequisites, please refer to the documentation of the respective software package.

2.2 This Software

The build, test, and installation steps which are common to any BASIS based software, including BASIS itself, are given in the *Build and Installation* guide.

If you have problems to build, test, or install the software on your system and this guide does not help you to resolve the issue, please contact us at `sbia-software` at `uphs.upenn.edu`.

3 Documentation

A PDF `Version` of this documentation is available online and included with the distribution package.

3.1 Software Manual

The Standard

Build System Standard

The build system standard, i.e., the B in BASIS, is defined in the following.

Filesystem Hierarchy Standard This document describes the filesystem hierarchy of projects following the Build system And Software Implementation Standard (BASIS). This standard is in particular based on the [Filesystem Hierarchy Standard of Linux](#). The configuration of the installation is further designed such that it is possible to account for the differences in both Unix and Windows systems as well as the installation of multiple versions of each package. Furthermore, the directory structure by default integrates software developed at SBIA not only seamlessly into the target system but also the various separately managed SBIA software packages with each other. The latter can in particular be employed by a superproject concept based on a super-build.

In this document, names of CMake variables defined by the BASIS modules are given. These are used within the CMake modules to refer to the particular directories of a software project. These variables should be used either directly or indirectly where possible such that a change of the actual directory names does not require a modification of the software itself.

The BASIS Filesystem Hierarchy Standard is in particular implemented by the *Software Project Template*, which is the template for any software project which follows BASIS. See the *Create/Modify a Project* guide for details on how to create a new project which complies with this Filesystem Hierarchy Standard.

Legend In the reminder, `<project>` is a placeholder for the project name in lowercase letters only and `<Project>` is the case-sensitive project name.

Further, `<version>` is a placeholder for the project version string `<major>.<minor>.<patch>`, where `<major>` is the major version number, `<minor>` the minor version number, and `<patch>` the patch number.

The root directory of a particular project source tree is denoted by `<source>`, while the root directory of the project binary tree is referred to as `<build>`. Note that each software project consists of more than one project components (not identical to, but related with software package components). Each component can be downloaded, configured, build, and installed separately. See design-components file and section Source Tree for details.

The root directory of a development branch such as in particular the trunk (see *Repository Organization*), is considered relative to the base URL of the project repository and denoted by `<tag>`, while the base URL is referred to as `<url>`.

Repository Organization Each [Subversion](#) (SVN) repository contains the top-level directories `trunk/`, `branches/`, and `tags/`. No other directories may be located next to these three top-level directories.

Repository Path	Description
trunk/	The current development version of the project. Most development is done in this master branch.
branches/<name>/	Separate branches named <name> are developed in sub-directories under the branches directory. One reason for branching is, for example, to develop new features separate from the main development branch, i.e., the trunk, and merging the desired changes back to the trunk once the new feature is implemented and tested.
branches/<project>-<major>.<minor>/	This particular branch is used prior to releasing a new version of the project. This branch is commonly referred to as release candidate of version <major>.<minor> of the project. It is used to adjust the project files prior to tagging a particular release. For example, to set the correct version number in the project files. This branch is further be used to apply bug fixes to a previous release of this version, in which case the patch number has to be increased before tagging a new release of this software version. See the Branch and Release guide for further details.
tags/<project>-<version>/	Tagged release version of the project. The reason for including the project name in the name of the tagged branch is, that SVN uses the last URL part as name for the directory to which the URL's content is checked out or exported to if no name for this directory is specified.

See the [Branch and Release](#) guide for details on how to create new branches and the process of releasing a new version of a software project.

Underneath the trunk and the release branches, a version of the entire source tree has to be found. Other branches underneath the `branches/` directory may only contain a subset of the trunk such as, for example, only the source code of the software but not the example or tests.

Source Tree The directory structure of the source tree, i.e., the directories and files which are managed in a revision controlled repository, is summarized in what follows. Other than the build and installation tree, which are created and populated from the source tree, is the source tree even because of this the beating heart of a software project. The directories and files in the source tree can be classified into the following categories: software, build configuration, documentation, and testing. The example which is part of a software project is considered to be part of both documentation and testing. Furthermore, any files essential to the execution of the software are considered to be part of the software. Examples are a pre-computed lookup table and a medical image atlas.

The testing at SBIA can further be divided into two subcategories: system testing and unit testing. It is important to note the difference of system tests and unit tests. Most often, only system tests will be performed due to the research character of the projects. These tests are usually implemented in a scripting language such as Python, Perl, or BASH. System tests simply run the built executables with different test input data and compare the output to the expected results. Therefore, system tests can also be performed on a target system using the installed software where both the software and system tests are distributed as separate binary distribution packages. Unit tests, on the other side, only test a single software module such as a C++ class or Python module, for example. The size of the additional data required for unit tests shall be reasonably small. Entire medical image data sets should only be required for system tests. The unit tests are compiled into separate executable files called test drivers. These executable files are not essential for the functioning of the software and are solely build for the purpose of testing.

As the testing as well as the example in the field of medical imaging often requires a huge amount of image data, these datasets are stored and managed outside the source tree. Please refer to the [Managing Test Data](#) guide for details on this topic.

Filesystem Hierarchy Below, the filesystem hierarchy of the source tree of a software project is delineated. On the left side the names of the CMake variables defined by BASIS are given, while on the right side the actual names of the directories are listed:

```

- PROJECT_SOURCE_DIR          - <source>/
  + PROJECT_CODE_DIR          + src/
  + PROJECT_CONFIG_DIR        + config/
  + PROJECT_DATA_DIR          + data/
  + PROJECT_DOC_DIR           + doc/
  + PROJECT_EXAMPLE_DIR       + example/
  + PROJECT_MODULES_DIR       + modules/
  + PROJECT_TESTING_DIR       + test/

```

Following a description of the directories, where the names of the CMake variables defined by BASIS are used instead of the actual directory names:

Directory Variable	Description
PROJECT_SOURCE_DIR	Root directory of source tree.
PROJECT_CODE_DIR	All source code files.
PROJECT_CONFIG_DIR	BASIS configuration files.
PROJECT_DATA_DIR	Software configuration files including auxiliary data such as medical atlases.
PROJECT_DOC_DIR	Software documentation.
PROJECT_EXAMPLE_DIR	Example application of software.
PROJECT_MODULES_DIR	<i>Project Modules</i> , each residing in its own subdirectory.
PROJECT_TESTING_DIR	Implementation of tests and test data.

Build Tree Even though CMake supports in-source tree builds, BASIS permits this and requires that the build tree is outside the source tree. Only the files in the source tree are considered of importance.

In the following, only the directories which do not reflect the source tree are considered as these directories are created and populated by CMake itself.

Filesystem Hierarchy

```

- PROJECT_BINARY_DIR          - <build>/
  + RUNTIME_OUTPUT_DIRECTORY  + bin/
  + LIBRARY_OUTPUT_DIRECTORY  + lib/
  + ARCHIVE_OUTPUT_DIRECTORY  + lib/
  + TESTING_RUNTIME_DIR       + Testing/bin/
  + TESTING_LIBRARY_DIR       + Testing/lib/
  + TESTING_OUTPUT_DIR        + Testing/Temporary/

```

Following a description of the directories, where the names of the CMake variables defined by BASIS are used instead of the actual directory names:

Directory Variable	Description
RUNTIME_OUTPUT_DIRECTORY	All executables and shared libraries (Windows).
LIBRARY_OUTPUT_DIRECTORY	Shared libraries (Unix).
ARCHIVE_OUTPUT_DIRECTORY	Static libraries and import libraries (Windows).
TESTING_RUNTIME_DIR	Directory of test executables.
TESTING_LIBRARY_DIR	Directory of libraries only used for testing.
TESTING_OUTPUT_DIR	Directory used for test results.

Installation Tree When installing the software package by building either the install target, extracting a binary distribution package, or running an installer of a binary distribution package, the following directory structure is used.

Option	Description
INSTALL_PREFIX	Installation directories prefix (<prefix>). Defaults to /usr/local on Unix-like systems and C:\Program Files\SBIA on Windows. Note that this variable is initialized by the value of CMAKE_INSTALL_PREFIX, the default variable used by CMake. Once it is initialized, the value of CMake's CMAKE_INSTALL_PREFIX variable is forced to always reflect the value of this variable.
INSTALL_SINFIX	Installation directories suffix or infix, respectively (<sinfix>). Defaults to @PROJECT_NAME_LOWER@.

In order to install different versions of a software, choose an installation prefix that includes the package name and software version, for example, /usr/local/@PROJECT_NAME_LOWER@-@PROJECT_VERSION@. In this case, INSTALL_SINFIX should be set to an empty string.

Filesystem Hierarchy Based on above options, the installation directories are set as follows:

```

- INSTALL_PREFIX                - <prefix>/
  + INSTALL_CONFIG_DIR          + lib/cmake/<sinfix>/ (Unix) | cmake/ (Windows)
  + INSTALL_RUNTIME_DIR         + bin/<sinfix>/
  + INSTALL_LIBEXEC_DIR         + lib/<sinfix>/ | bin/<sinfix>/ (Windows)
  + INSTALL_LIBRARY_DIR         + lib/<sinfix>/
  + INSTALL_ARCHIVE_DIR         + lib/<sinfix>/
  + INSTALL_INCLUDE_DIR         + include/sbia/<project>/
  + INSTALL_SHARE_DIR           + share/
    + INSTALL_DOC_DIR           + <sinfix>/doc/
    + INSTALL_EXAMPLE_DIR       + <sinfix>/example/
    + INSTALL_MAN_DIR           + <sinfix>/man/

```

Note that the include directory by intention always ends in sbia/<project>, such that header files of a project have to be included as follows:

```
#include <sbia/<project>/header.h>
```

Hence, the include directory which is added to the search path has to be set to <prefix>/include/.

Following a description of the directories, where the names of the CMake variables defined by BASIS are used instead of the actual directory names:

Directory Variable	Description
INSTALL_CONFIG_DIR	CMake package configuration files.
INSTALL_RUNTIME_DIR	Main executables and shared libraries on Windows.
INSTALL_LIBEXEC_DIR	Utility executables which are called by other executables only.
INSTALL_LIBRARY_DIR	Shared libraries on Unix and module libraries.
INSTALL_ARCHIVE_DIR	Static and import libraries on Windows.
INSTALL_INCLUDE_DIR	Public header files of libraries.
INSTALL_DOC_DIR	Documentation files including the software manual in particular.
INSTALL_EXAMPLE_DIR	All data required to follow example as described in manuals.
INSTALL_MAN_DIR	Man pages are installed to this directory.
INSTALL_MAN_DIR/man1/	Man pages of main executables.
INSTALL_MAN_DIR/man3/	Man pages of libraries.
INSTALL_SHARE_DIR	Shared package files including required auxiliary data files.

Links On Unix, the following symbolic links are created when the option INSTALL_LINKS is set to ON. Note that the link creation will fail if a file or directory with the links' name already exists. This is desired and will simply be reported to the user. If a symbolic name of the same name already exists, it is replaced however.

Link	Target
<prefix>/bin/<exec>	INSTALL_RUNTIME_DIR/<exec>
<prefix>/share/doc/<sinfix>/	INSTALL_DOC_DIR
<prefix>/share/man/man.?.<name>.	INSTALL_MAN_DIR/man.?.<name>.

Software Project Template The starting point for every project is to setup a directory structure which helps to organize the project files. Moreover, in order to configure the build system using **CMake**, `CMakeLists.txt` files have to be added to the project with proper CMake code to add build targets with the corresponding build flags to the build system. In order to unify the structure of projects following common practices and further to simply take off the burden from the developer on setting up a basic project, BASIS provides a so-called (Software) Project Template. Please refer to the *Filesystem Hierarchy Standard* for a definition of the common project directory structure. This directory structure is part of the standard and thus required by BASIS. The project template therefore follows this standard regarding the directory structure. The further purpose of the software project template is to provide template files concerned with the configuration of the build, testing, installation, and packaging. In the following, the purpose of each template file is summarized.

Required Project Files The following files have to be part of any project which follows the *Filesystem Hierarchy Standard*. These are thus the minimal set of project files which need to be selected when instantiating a new software project. Besides these files, a project will have either a `src/` directory or a `modules/` directory, or both of them. See below for a description of these directories.

README.txt This is the main (root) documentation file. Every user is supposed to first read this file, which in turn will refer them to the more extensive documentation. This file in particular introduces the software package shortly, including a summary of the package files. Moreover, it refers to the `INSTALL.txt` and `COPYING.txt` files for details on the build and installation and software license, respectively. Furthermore, references to scientific articles related to the software package shall be included in this file.

AUTHORS.txt Names the authors of the software package. Moreover, people who notably contributed to the software directly shall be named here as well, even if they did not actually edit any project file. Others, who mostly contributed indirectly should be named in the `README.txt` file instead.

COPYING.txt Contains copyright and license information. If some files of the project were copied from other sources, the copyright and license of these files shall be included in this file as well. It is important to clearly state which copyright and license text corresponds to which project file.

INSTALL.txt Contains build and installation instructions. As the build of all projects which follow BASIS is very similar, this file shall only describe additional steps/CMake variables which are not described in the *Build and Installation* guide.

BasisProject.cmake This file contains the meta-data of the project such as the project name, its brief description which is used for the packaging, and the dependencies. Note that additional dependencies may be given by the CMake code in the `config/Depends.cmake` file, if such file is present. This file mainly consists of a call to the `basis_project()` command. If the project is a module of another project, this file is read by the top-level project to be able to identify its modules and the dependencies among them.

CMakeLists.txt The root CMake configuration file. **Do not edit this file.**

Common Project Files

CTestConfig.cmake The **CTest** configuration file. This file in particular specifies the URL of the **CDash** dashboard of the project where test results should be submitted to.

config/Settings.cmake This is the main CMake script file used to configure the build system, and BASIS in particular. Any CMake code required to configure the build system, such as adding common compiler flags, or adding common definitions which have not yet been added by the generic code used by BASIS to utilize a found dependency should go into this file.

config/ScriptConfig.cmake.in See the documentation on the *build of script targets* for details on how this *script configuration* is used.

data/CMakeLists.txt This CMake configuration file contains code to simply install every file and directory from the source tree into the `INSTALL_DATA_DIR` directory of the installation tree.

doc/CMakeLists.txt This CMake configuration file adds rules to build the documentation form, for example, the in-source comments using *Doxygen*. Moreover, for every documentation file, such as the user manual, the `basis_add_doc()` command has to be added to this file.

example/CMakeLists.txt This CMake configuration file contains by default code to install every file and directory from the source tree into the `INSTALL_EXAMPLE_DIR` directory of the installation tree. It may be modified to configure and/or build certain files of the example if applicable or required.

src/CMakeLists.txt The definition of all software build targets shall be added to this file, using the commands `basis_add_library()` to add a shared, static, or module library, which can also be a module written in a scripting language, and `basis_add_executable()` to add an executable target, which can be either a binary or a script file. If appropriate, the source code files may be further organized in subdirectories of the `src/` directory, in which case either separate `CMakeLists.txt` files can be used for each subdirectory, or yet all targets are added to the `src/CMakeLists.txt` file using relative paths which include the subdirectory in which the source files are found. In general, if the number of source code files is low, i.e., close to or below 20, no subdirectory structure is required.

test/CMakeLists.txt Tests are added to this build configuration file using the `basis_add_test()` command. The test input files are usually put in a subdirectory named `test/input/`, while the baseline data of the expected test output is stored inside a subdirectory named `test/baseline/`. Generally, however, the *Filesystem Hierarchy Standard* of BASIS does not dictate how the test sources, input, and baseline data have to be organized inside the `test/` directory.

modules/ If the project files are organized into conceptual cohesive groups, similar to the modularization goal of the ITK 4, this directory contains these conceptual modules of the project. The files of each module reside in a subdirectory named after the module. Note that each module itself is a project derived from this project template.

Advanced Project Files The customization of the following files is usually not required, and hence, in most cases, most of these files need not to be part of a project.

config/Components.cmake Contains CMake code to configure the components used by component-based installers. Currently, component-based installers are not very well supported by BASIS, and hence this file is mostly unused and is yet subject to change.

config/Config.cmake.in This is the template of the package configuration file. When the project is configured/installed using CMake, a configured version of this file is copied to the build or installation tree, respectively, where the information about the package configuration is substituted as appropriate for the actual build/installation of the package. For example, the configured file contains the absolute path to the installed public header files such that other packages can easily add this path to their include search path. The `find_package()` command of CMake, in particular, will look for this file and automatically import the CMake settings when this software package was found. For many projects, the default package configuration file of BASIS which is used if this file is missing in the project's `config/` directory, is sufficient and thus this file is often not required.

config/ConfigSettings.cmake This file sets CMake variables for use in the `config/Config.cmake.in` file. As the package configuration for the final installation differs from the one of the build tree, this file has to contain CMake code to set the variables used in the `config/Config.cmake.in` file differently depending on whether the variables are being set for use within the build tree or the installation tree. This file only needs to be present if the project uses a custom `config/Config.cmake.in` file, which in turn contains CMake variables whose value differs between build tree and installation.

config/ConfigUse.cmake.in This file is provided for convenience of the user of the software package. It contains CMake code which uses the variables set by the package configuration file (i.e., the file generated from the file `config/Config.cmake.in`) in order to configure the build system of packages which use this software packages properly such that they can make use of this software. For example, the package configuration sets a variable `<Pkg>_INCLUDE_DIRS` to a list of include directories which have to be added to the include search path. Then, this file would contain CMake instructions such as the line `include_directories(${<Pkg>_INCLUDE_DIRS})` to actually add these directories to the search path for header files.

config/ConfigVersion.cmake.in This file accompanies the package configuration file generated from the `config/Config.cmake.in` file. It is used by CMake's `find_package()` command to identify versions of this software package which are compatible with the version requested by the dependent project. This file needs almost never be customized by a project and thus should not be included in a project's source tree.

config/Depends.cmake If the generic code used by BASIS is not sufficient to resolve the dependencies on external packages properly, add this file to your project. It can contain CMake code to find and make use of external software packages.

config/Package.cmake Configures `CPack`, the package generator of CMake. The packaging of software using `CPack` is currently not completely supported by BASIS. This template file is yet subject to change.

CTestCustom.cmake.in This file defines `CTest` variables which `customize CTest`.

Project Modularization This article details the design and implementation of the project modularization, where a project, also referred to as top-level project (note that BASIS only allows one level of submodularization), can have other projects as subprojects (referred to as modules in this context to distinguish them from the superproject/subproject relationship in case of the superbuild approach, see below). The project modules are conceptual cohesive groups of the project files.

A top-level project can also be described as a modularized project, where each module of this project may depend on other modules of the same project or external projects (also referred to as packages). Note that an external project can also be another top-level project which utilizes the same project modularization as discussed herein. The idea, and partly the CMake implementation, has been borrowed from the [ITK 4](#) project. See the Wiki for details on the [ITK 4 Modularization](#).

Implementation The modularization is mainly implemented by the `ProjectTools.cmake` module, in particular, the `basis_project_modules()` function which is called at the very top of the `basis_project_impl()` macro, which is the foundation for the root CMake configuration file (`CMakeLists.txt`) of every BASIS project. The `basis_project_modules()` function searches the subdirectories in the `modules/` directory for the presence of the `BasisProject.cmake` file. It then loads this file, to retrieve the meta-data of each module, i.e., its name and dependencies. It then adds for each module a `MODULE_<module>` option to the build configuration. When this option is set to `OFF`, the module is excluded from both the project build and any package generated by `CPack`. Otherwise, if it is set to `ON`, the module is build as part of the top-level project. These options are superseded by the `BUILD_ALL_MODULES` option if this option is set to `ON`, however.

Besides adding these options, the `basis_project_modules()` function ensures that the modules are configured and build in the right order, such that a module which is needed by another module is build prior to this dependent module. Moreover, it helps the `basis_find_package()` function to find the other modules, in particular their configured package configuration files, which are either generated from the default `Config.cmake.in` file or a corresponding file found in the `config/` directory of each module.

The other BASIS CMake functions may further change their actual behaviour depending on the `PROJECT_IS_MODULE` variable which specifies whether the project that is currently being configured is a module of another project (i.e., `PROJECT_IS_MODULE` is `TRUE`) or a top-level project (i.e., `PROJECT_IS_MODULE` is `FALSE`).

As modules are more or less independent subprojects, which might even be maintained by different persons, it is desirable that naming conflicts are handled automatically by the top-level project. Therefore, if the CMake variable `BASIS_USE_MODULE_NAMESPACES` is set to `ON`, BASIS will take actions to avoid naming conflicts. For example, the CMake build target names are prepended by the name of the module in lowercase followed by a dot (.) as namespace separator. By default, the `BASIS_USE_MODULE_NAMESPACES` option is, however, set to `OFF`. Even though modules shall be developed as independent from each other or the top-level project as applicable, are they yet considered to be closely related and their sources kept in the same repository. If a more loose coupling of projects is desired, these should be separate projects with separate repositories. These may then be distributed, configured, and build as part of a common superproject.

Build of Script Targets Unlike source files written in non-scripting languages such as C++ or Java, source files written in scripting languages such as Python, Perl, or BASH do not need to be compiled before their execution. They are interpreted directly and hence do not need to be build (in case of Python, however, they are as well compiled by the interpreter itself to improve speed). On the other side, CMake provides a mechanism to replace CMake variables in a source file by their respective values which are set in the `CMakeLists.txt` files (or an included CMake script file). As it is often useful to introduce build specific information in a script file such as the relative location of auxiliary executables or data files, the `basis_add_executable()` and `basis_add_library()` commands also provide a means of building script files. How these functions process scripts during the build of the software is discussed next. Afterwards it is described how the build of scripts can be configured.

Prerequisites and Build Steps During the build of a script, the CMake variables as given by `@VARIABLE_NAME@` patterns are replaced by the value of the corresponding CMake variable if defined, or by an empty string otherwise. Similar to the configuration of source files written in C++ or MATLAB, the names of the script files which shall be configured by BASIS during the build step have to end with the `.in` suffix. Otherwise, the script file is not modified by the BASIS build commands and simply copied to the build tree or installation tree, respectively. Opposed to configuring the source files already during the configure step of CMake, as is the case for C++ and MATLAB source files, script files are configured during the build step to allow for the used CMake variables to be set differently depending on whether the script is intended for use inside the build tree or the installation tree. Moreover, certain properties of the script target can still be modified after the `basis_add_executable()` or `basis_add_library()` command, respectively, using the `basis_set_target_properties()` or `basis_set_property()` command. Hence, the final values of these variables are not known before the configuration of the build system has been completed. Therefore, all CMake variables which are defined when the `basis_add_executable()` or `basis_add_library()` command is called, are dumped to a CMake script file to preserve their value at this moment and the dump of the variables is written to a file in the build tree. This file is loaded again during the build step by the custom build command which eventually configures the script file using CMake's `configure_file()` command with the `@ONLY` option. This build command configures the script file twice. The first “built” script is intended for use within the build tree while the second “built” script will be copied upon installation to the installation tree.

Before each configuration of the (template) script file (the `.in` source file in the source tree), the file with the dumped CMake variable values and the various script configuration files are included in the following order:

1. Dump file of CMake variables defined when the script target was added.
2. Default script configuration file of BASIS (`BasisScriptConfig.cmake`).
3. Default script configuration file of individual project (`ScriptConfig.cmake`, optional).
4. Script configuration file specified using the `CONFIG_FILE` argument of the `basis_add_executable()` or `basis_add_library()` command.
5. Script configuration code specified using the `CONFIG` argument of the `basis_add_executable()` or `basis_add_library()` command.

Script Configuration The so-called script configuration is CMake code which defines CMake variables for use within script files. This code is either saved in a CMake script file with the `.cmake` file name extension or specified

directly as argument of the `CONFIG` option of the `basis_add_executable()` or `basis_add_library()` command used to add a script target to the build system. The variables defined by the script configuration are substituted by their respective values during the build of the script target. Note that the CMake code of the script configuration is evaluated during the build of the script target, not during the configuration of the build system. During the configuration of the build systems, the script configuration is, however, configured in order to replace `@VARIABLE_NAME@` patterns in the configuration by their respective values as defined by the build configuration (`CMakeLists.txt` files). Therefore, the variables defined in the script configuration can be set differently for each of the two builds of the script files. If the script configuration is evaluated before the configuration of the script file for use inside the build tree, the CMake variable `BUILD_INSTALL_SCRIPT` is set to `FALSE`. Otherwise, if the script configuration is evaluated during the build of the script for use in the installation tree, this variable is set to `TRUE` instead. It can therefore be used to set the variables in the script configuration depending on whether or not the script is build for use in the build tree or the installation tree.

For example, the project structure differs for the build tree and the installation tree. Hence, relative file paths to the different directories of data files, for instance, have to be set differently depending on the value of `BUILD_INSTALL_SCRIPT`, i.e.,

```
if (BUILD_INSTALL_SCRIPT)
  set (DATA_DIR "@INSTALL_PREFIX@/@INSTALL_DATA_DIR@")
else ()
  set (DATA_DIR "@PROJECT_DATA_DIR@")
endif ()
```

Avoid the use of absolute paths, however! Instead, use the `DIR` variable which is set in the build script to the directory of the output script file to make these paths relative to this directory which contains the configured script file. These relative paths which are defined by the script configuration are then used in the script file as follows:

```
# /usr/bin/env bash
@BASIS_BASH_UTILITIES@
get_executable_directory EXEC_DIR && readonly EXEC_DIR
[ $? -eq 0 ] || { echo 'Failed to determine directory of this executable!'; exit 1; }
readonly DATA_DIR="${EXEC_DIR}/@DATA_DIR@"
```

where `DATA_DIR` is the relative path to the required data files as determined during the evaluation of the script configuration. See documentation of the `basis_set_script_path()` function for a convenience function which can be used therefore. Note that this function is defined in the custom build script generated by BASIS for the build of each script target and hence can only be used within a script configuration. For example, use this function as follows in the `PROJECT_CONFIG_DIR/ScriptConfig.cmake.in` script configuration file of your project:

```
basis_set_script_path(DATA_DIR "@PROJECT_DATA_DIR@" "@INSTALL_DATA_DIR@" )
```

Note that most of the more common variables which are useful for the development of scripts are already defined by the default script configuration file of BASIS. Refer to the documentation of the `BasisScriptConfig.cmake` file for a list of available variables.

Implementation Standard

The software implementation standard, i.e., the SI in BASIS, is defined in the following.

Command-line Parsing Most of the software developed in a research environment is based on the command-line, as command-line tools are easier and thus faster implemented than tools with graphical user interface. To help the developer, who wants to focus on the actual image processing algorithm rather than the parsing of the command-line arguments, BASIS intends to provide a command-line parsing library for each of the commonly used programming languages. The usage of these libraries is documented in the following:

Note: Yet, there exist only libraries for C++ and BASH, but solutions for Java, Python, and Perl will be provided in future BASIS releases as well.

Parsing the Command-line Arguments in C++ For the parsing of command-line arguments in C++, BASIS includes a slightly extended version of the Templated C++ Command Line Parser ([TCLAP](#)) Library. For details and usage of this library, please refer to the TCLAP documentation. It is in particular recommended to read the [TCLAP manual](#). Further, the *TCLAP API documentation* is a good reference on the available command-line argument classes. The API documentation of the TCLAP classes can also be found as part of this documentation.

Note: BASIS provides its own subclass of the `TCLAP::CmdLine` class which is also named `CmdLine`, but in the `sbia::basis` namespace, i.e., `sbia::basis::CmdLine`. Most of the argument implementations are, however, simply typedefs of the commonly used `TCLAP::Arg` subclasses. See the [API documentation](#) for a list of command-line arguments which are made available as part of the `sbia::basis` namespace.

The usage of the command-line parsing library shall be demonstrated in the following on the implementation of an example command-line program. It should be noted that the try-catch block in the `main()` function will only help to track errors in the command-line specification, but once the `cmd` instance is initialized properly, all runtime exceptions related to the parsing of the command-line are handled by BASIS.

```
/**
 * @file smoothimage.cxx
 * @brief Smooth image using Gaussian or anisotropic diffusion filtering.
 *
 * Copyright (c) 2011 University of Pennsylvania. All rights reserved.<br />
 * See http://www.rad.upenn.edu/sbia/software/license.html or COPYING file.
 *
 * Contact: SBIA Group <sbia-software at uphs.upenn.edu>
 */

#include "basis.h" // include BASIS C++ utilities

// acceptable in .cxx file
using namespace std;
using namespace sbia::basis;

// =====
// smoothing filters
// =====

// -----
int gaussianfilter(const string& imagefile,
                  const vector<unsigned int>& r,
                  double std)
{
    // [...]
    return 0;
}

// -----
int anisotropicfilter(const string& imagefile)
{
    // [...]
    return 0;
}
```

```

}

// =====
// main
// =====

// -----
int main(int argc, char* argv[])
{
    // -----
    // define command-line arguments

    SwitchArg gaussian(                                // option switch
        "g", "gaussian",                                // short and long option name
        "Smooth image using a Gaussian filter.", // argument help
        false);                                         // default value

    SwitchArg anisotropic(                              // option switch
        "a", "anisotropic",                            // short and long option name
        "Smooth image using anisotropic diffusion filter.", // argument help
        false);                                         // default value

    MultiUIntArg gaussian_radius(                      // unsigned integer values
        "r", "radius",                                // short and long option name
        "Radius of Gaussian kernel in each dimension.", // argument help
        false,                                         // required?
        "<rx> <ry> <rz>",                             // value type description
        3,                                             // number of values per argument
        true);                                         // accept argument only once

    DoubleArg gaussian_std(                            // floating-point argument value
        "", "std",                                    // only long option name
        "Standard deviation of Gaussian in voxel units.", // argument help
        false,                                         // required?
        2.0,                                           // default value
        "<float>");                                     // value type description

    // [...]

    PositionalArg imagefile(                           // positional, i.e., unlabeled
        "image",                                       // only long option name
        "Image to be smoothed.",                     // argument help
        true,                                         // required?
        "",                                           // default value
        "<image>");                                     // value type description

    // -----
    // parse command-line
    try {
        vector<string> examples;

        examples.push_back(
            "EXECNAME --gaussian --std 3.5 --radius 5 5 3 brain.nii\n"
            "Smooths the image brain.nii using a Gaussian with standard"
            " deviation 3.5 voxel units and 5 voxels in-slice radius and"
            " 3 voxels radius across slices.");

        examples.push_back(
            "EXECNAME --anisotropic brain.nii\n"

```

```

        "Smooths the image brain.nii using an anisotropic diffusion filter.");

CmdLine cmd(
    // program identification
    "smoothimage", cProjectName,
    // program description
    "This program smooths an input image using either a Gaussian "
    "filter or an anisotropic diffusion filter.",
    // example usage
    examples,
    // version information
    cVersionAndRevision,
    "Copyright (c) 2011 University of Pennsylvania."
    " All rights reserved.");

// The constructor of the CmdLine class has already added the standard
// arguments --help, --helpshort, --helpxml, --helpman, and --version.

cmd.xorAdd(gaussian, anisotropic);
cmd.add(gaussian_std);
cmd.add(gaussian_radius);
cmd.add(imagefile);

cmd.parse(argc, argv);
} catch (CmdLineException& e) {
    // invalid command-line specification
    cerr << e.error() << endl;
    exit(1);
}

// -----
// smooth image - access parsed argument value using Arg::getValue()
unsigned int r[3];

if (gaussian.getValue()) {
    return gaussianfilter(imagefile.getValue(),
                           gaussian_radius.getValue(),
                           gaussian_std.getValue());
} else {
    return anisotropicfilter(imagefile.getValue());
}
}

```

Running the above program with the `--help` option will give the output:

SYNOPSIS

```

smoothimage [--std <float>] [--radius <rx> <ry> <rz>] [--verbose|-v]
             [--gaussian|--anisotropic] <image>
smoothimage [--help|-h|--helpshort|--helpxml|--helpman|--version]

```

DESCRIPTION

This program smooths an input image using either a Gaussian filter or an anisotropic diffusion filter.

OPTIONS

Required arguments:

- g or --gaussian
Smooth image using a Gaussian filter.
- or -a or --anisotropic

Smooth image using anisotropic diffusion filter.

<image>

Image to be smoothed.

Optional arguments:

-s or --std <float>

Standard deviation of Gaussian in voxel units.

-r or --radius <rx> <ry> <rz>

Radius of Gaussian kernel in each dimension.

Standard arguments:

-- or --ignore_rest

Ignores the rest of the labeled arguments following this flag.

-v or --verbose

Increase verbosity of output messages.

-h or --help

Display help and exit.

--helpshort

Display short help and exit.

--helpxml

Display help in XML format and exit.

--helpman

Display help as man page and exit.

--version

Display version information and exit.

EXAMPLE

```
smoothimage --gaussian --std 3.5 --radius 5 5 3 brain.nii
```

Smooths the image brain.nii using a Gaussian with standard deviation 3.5 voxel units and 5 voxels in-slice radius and 3 voxels radius across slices.

```
smoothimage --anisotropic brain.nii
```

Smooths the image brain.nii using an anisotropic diffusion filter.

CONTACT

SBIA Group <sbia-software at uphs.upenn.edu>

The --helpshort output contains the synopsis of the full help only:

```
smoothimage [--std <float>] [--radius <rx> <ry> <rz>] [--verbose|-v]
             {--gaussian|--anisotropic} <image>
smoothimage [--help|-h|--helpshort|--helpxml|--helpman|--version]
```

Parsing the Command-line Arguments in BASH See [shflags.sh](#) as a reference until this guide is completed, keeping in mind, though, that this module will have to be revised.

Calling Conventions This document discusses and describes the conventions for calling other executables from within an executable. The calling conventions address the question whether to use relative or absolute file paths when calling other executables and introduce a name mapping from build target names to actual executable file names. These calling conventions are, however, hidden to the developer through automatically generated utility functions for each supported programming language. See *Implementation* for details on the specific implementations in the different languages.

Relative vs. Absolute Paths Relative paths such as only the executable file name require a proper setting of the PATH environment variable. If more than one version of a particular software package should be installed or in case of name conflicts with other packages, this is not trivial and it may not be guaranteed that the correct executable is executed. Absolute executable file paths, on the other side, restrict the relocatability and thus distribution of pre-build binary packages. Therefore, BASIS proposes and implements the following convention on how absolute paths of (auxiliary) executables are determined at runtime by taking the absolute path of the directory of the calling executable into consideration.

Main executables in <prefix>/bin/<sinfix>/ call utility executables relative to their own directory. For example, a BASH script called main that executes a utility script util in <prefix>/lib/<sinfix>/ would do so as demonstrated in the following example code (for details on the @VAR@ patterns, please refer to the *Build of Script Targets* page):

```
# among others, defines the get_executable_directory() function
@BASIS_BASH_UTILITIES@

# get absolute directory path of auxiliary executable
get_executable_directory _EXEC_DIR && readonly _EXEC_DIR
_LIBEXEC_DIR=${_EXEC_DIR}/@LIBEXEC_DIR@

# call utility executable in libexec directory
${_LIBEXEC_DIR}/util
```

where LIBEXEC_DIR is set in the BasisScriptConfig.cmake configuration file to either the output directory of auxiliary executables in the build tree relative to the directory of the script built for the build tree or to the path of the installed auxiliary executables relative to the location of the installed script. Note that in case of script files, two versions are built by BASIS, one that is working directly inside the build tree and one which is copied to the installation tree. In case of compiled executables, such as in particular programs built from C++ source code files, a different but similar approach is used to avoid the build of two different binary executable files. Here, the executable determines at runtime whether it is executed from within the build tree or not and uses the appropriate path depending on this.

If an executable in one directory wants to execute another executable in the same directory, it can simply do so as follows:

```
# call other main executable
${_EXEC_DIR}/othermain
```

File vs. Target Name In order to be independent of the actual names of the executable files—which may vary depending on the operating system (e.g., with or without file name extension in case of script files) and the context in which a project was built—executables should not be called by their respective file name, but their build target name.

It is in the responsibility of the BASIS auxiliary functions to properly map this project specific and (presumably) constant build target name to the absolute file path of the built (and installed) executable file. This gives BASIS the ability to modify the executable name during the configuration step of the project, for example, to prepend them with a unique project-specific prefix, in order to ensure uniqueness of the executable file name. Moreover, if an executable should be renamed, this can be done simply through the build configuration and does not require a modification of the source code files which make use of this executable.

Search Paths All considered operating systems—or more specifically the used shell and dynamic loader—provide certain ways to configure the search paths for executable files and shared libraries which are dynamically loaded on demand. The details on how these search paths can be configured are summarized next including the pros and cons of each method to manipulate these search paths. Following these considerations, the solution aimed at by BASIS is detailed.

Unix On Unix-based systems (including in particular all variants of Linux and Mac OS) executables are searched in directories specified by the `PATH` environment variable. Shared libraries, on the other side, are first searched in the directories specified by the `LD_LIBRARY_PATH` environment variable, then in the directories given by the `RPATH` which is set within the binary files at compile time, and last the directories specified in the `/etc/ld.so.conf` system configuration file.

The most flexible method which can also easily be applied by a user is setting the `LD_LIBRARY_PATH` environment variable. It is, however, not always trivial or possible to set this search path in a way such that all used and installed software works correctly. There are many discussions on why this method of setting the search path is considered evil among the Unix community (see for example [here](#)). The second option of setting the `RPATH` seems to be the most secure way to set the search path at compile time. This, however, only for shared libraries which are distributed and installed with the software because only in this case can we make use of the `$ORIGIN` variable in the search path to make it relative to the location of the binary file. Otherwise, it is either required that the software is being compiled directly on the target system or the paths to the used shared libraries on the target system must match the paths of the system on which the executable was built. Hence, using the `RPATH` can complicate or restrict the relocatability of a software. Furthermore, unfortunately is the `LD_LIBRARY_PATH` considered before the `RPATH` and hence any user setting of the `LD_LIBRARY_PATH` can still lead to the loading of the wrong shared library. The system configuration `/etc/ld.so.conf` is not an option for setting the search paths for each individual software. This search path should only be set to a limited number of standard system search paths as changes affect all users. Furthermore, directories on network drives may not be included in this configuration file as they will not be available during the first moments of the systems start-up. Finally, only an administrator can modify this configuration file.

The anticipated method to ensure that the correct executables and shared libraries are found by the system for Unix-based systems is as follows. As described in the previous sections, executables which are part of the same software package are called by the full absolute path and hence no search path needs to be considered. To guarantee that shared libraries installed as part of the software package are considered first, the directory to which these libraries were installed is prepended to the `LD_LIBRARY_PATH` prior to the execution of any other executable. Furthermore, the `RPATH` of binary executable files is set using the `$ORIGIN` variable to the installation directory of the package's shared libraries. This ensures that also for the execution of the main executable, the package's own shared libraries are considered first. To not restrict the administrator of the target system on where other external packages need to be installed, no precaution is taken to ensure that executables and shared libraries of these packages are found and loaded properly. This is in the responsibility of the administrator of the target system. However, by including most external packages into the distributed binary package, these become part of the software package and thus above methods apply.

Note: The inclusion of the runtime requirements should be done during the packaging of the software and thus these packages should still not be integrated into the project's source tree.

Mac OS bundles differ from the default Unix-like way of installing software. Here, an information property list file (Info.plist) is used to specify for each bundle separately the specific properties including the location of frameworks, i.e., private shared libraries (shared libraries distributed with the bundle). Most shared libraries required by the software will be included in the bundle.

Windows On Windows systems, executable files are first searched in the current working directory. Then, the directories specified by the `PATH` environment variable are considered as search path for executable files where the extensions `.exe`, `.com`, `.bat`, and `.cmd` are considered by default and need not be included in the name of the executable that is to be executed. Shared libraries, on the other side, are first searched in the directory where the using module

is located, then in the current working directory, the Windows system directory (e.g., `C:\WINDOWS\system32\`), and then the Windows installation directory (e.g., `C:\WINDOWS`). Finally, the directories specified by the `PATH` environment variable are searched for the shared libraries.

As described in the previous sections, executables which are part of the software package are called by the full absolute path and hence no search path is considered. Further, shared runtime libraries belonging to the software package are installed in the same directory as the executables and hence will be considered by the operating system before any other shared libraries.

Implementation In the following the implementation of the calling conventions in each supported programming language is summarized.

Note that the **BASIS Utilities** provide an `execute_process()` function for each of these languages which accepts either an executable file path or a build target name as first argument of the command-line to execute. This function makes use of the so-called `ExecutableTargetInfo` module of the particular programming language whose implementation is summarized here.

C++ For C++ programs, the BASIS C++ utilities provide the function `get_executable_path()` which maps a build target name to the absolute path of the executable file built by this target. This function makes use of the static instance of the class `sbia::basis::ExecutableTargetInfo` whose constructor is automatically generated during the configuration of a project. This constructor initializes the data structures required for the mapping of target names to absolute file paths. Note that BASIS generates different implementations of this module for different projects.

The project implementations will, however, mainly make use of the `execute_process()` function which accepts either an actual executable file path or a build target name as first argument of the command-line to execute. This function shall be used in C++ code as a substitution for the commonly used `system()` function on Unix. The advantage of `execute_process()` is further, that it is implemented for all operating systems which are supported by BASIS, i.e., Linux, Mac OS, and Windows. The declaration of the `execute_process()` function can be found in the `stdaux.h` header file, which is included by the main `basis.h` header file. Note that these files are as well unique to each BASIS project build.

Java The Java programming language is not yet supported by BASIS.

Python A Python module named `executabletargetinfo.py` stores the location of the executables relative to its own path in a dictionary where the UIDs of the corresponding build targets are used as keys. The functions `get_executable_name()`, `get_executable_directory()`, and `get_executable_path()` can be used to get the name, directory, or path, respectively, of the executable file built by the specified target. If no target is specified, the name, directory, or path of the calling executable itself is returned.

Perl The `ExecutableTargetInfo.pm` Perl module uses a hash reference to store the locations of the executable files relative to the module itself. The functions `get_executable_name()`, `get_executable_directory()`, and `get_executable_path()` can be used to get the name, directory, or path, respectively, of the executable file built by the specified target. If no target is specified, the name, directory, or path of the calling executable itself is returned.

BASH For **BASH**, the module `executabletargetinfo.sh` implements associative arrays to store the location of the built executable files relative to this module. The functions `get_executable_name()`, `get_executable_directory()`, and `get_executable_path()` can be used to get the name, directory, or path, respectively, of the executable file built by the specified target. If no target is specified, the name, directory, or path of the calling executable itself is returned.

Additionally, the `executabletargetinfo.sh` module can setup aliases named after the UID of the build targets for the absolute file path of the corresponding executables. The target names can then be simply used as aliases for the

actual executables. The initialization of the aliases is, however, at the moment expensive and delays the load time of the executable which sources the `executabletargetinfo.sh` module. Note further that this approach requires the option `expand_aliases` to be set via `shopt -s expand_aliases` which is done by the `executabletargetinfo.sh` module if aliases were enabled. A `shopt -u expand_aliases` disables the expansion of aliases and hence should not be used in BASH scripts which execute other executables using these aliases.

Unsupported Languages In the following, languages for which the calling conventions are not implemented are listed. Reasons for not supporting these languages regarding the execution of other executables are given for each such programming language. Support for all other programming languages which are not supported yet and not listed here may be added in future releases of BASIS.

MATLAB Visit [this MathWorks page](#) for a documentation of external interfaces [MathWorks](#) provides for the development of applications in **MATLAB**. None of these interfaces includes the direct execution of other executables using something like a system call. There are better ways to execute code written in C++ or Java from MATLAB code. If you would need to execute a BASH or Python script from MATLAB, you need to rethink the design of your software.

Style Guides

Style Guide for Plain Text Files The following guide, which itself is styled according to the plain text format it describes, for details on how plain text documentation files of a software project shall be formatted.

Section of Biomedical Image Analysis
Department of Radiology
University of Pennsylvania
3600 Market Street, Suite 380
Philadelphia, PA 19104

Web: <http://www.rad.upenn.edu/sbia/>
Email: sbia-software at uphs.upenn.edu

Copyright (c) 2011 University of Pennsylvania. All rights reserved.
See <http://www.rad.upenn.edu/sbia/software/license.html> or COPYING file.

INTRODUCTION =====

This document defines guidelines on how to style plain text documentation files such as the `readme` file and the build and installation instructions. A common style makes it easier for users of software developed at SBIA to navigate through the documents and recognize what is of importance to them. Moreover, it serves as a branding of the software. Each individual software project shall finally be integrated with all the other software projects to form one unique software package. Here a common documentation style is desired such that the separate subprojects nicely integrate with each other.

HEADER =====

Each plain text document has to start with the following header with one blank line before and each line indented by two space characters.

Section of Biomedical Image Analysis
Department of Radiology
University of Pennsylvania
3600 Market Street, Suite 380
Philadelphia, PA 19104

Web: <http://www.rad.upenn.edu/sbia/>
Email: sbia-software at uphs.upenn.edu

Copyright (c) <year> University of Pennsylvania. All rights reserved.
See <http://www.rad.upenn.edu/sbia/software/license.html> or COPYING file.

HEADINGS

=====

Headings of level 1 are capitalized as in this document. All other headings are spelled case-sensitive where the first letter of words with four or more characters are started with an uppercase letter. Headings of level 1 are not intended, while all other headings are intended by two space characters.

For headings of level 1, a line of = characters as long as the heading is used to underline it. For headings of level 2, - characters are used instead. All other headings are not underlined.

Before each heading of level 1, three blank lines are inserted.
Before each heading of level 2, two blank lines are inserted.
Before any other heading, one blank line is inserted.

A heading and its text block are indented by $(\text{level} - 1) * 2$ space characters. Hence, headings of level 1 are not indented, while headings of level 2 are indented by two space characters.

TEXT BLOCKS

=====

The number of columns in a text block is limited to about 80 characters. Each text block is indented equally to the indentation of its heading, where at least two space characters are used to intend a text block. Hence, even though headings of level 1 are not indented, so are the corresponding text blocks.

There are no space characters on blank lines.

ENUMERATIONS

=====

Use -, +, and * characters as bullet points.

How-to Guides

Create/Modify a Project

This how-to guide introduces the `basisproject` command-line tool which is installed as part of BASIS. This tool, also simply referred to as “project tool”, is used to create a new project based on BASIS or to modify an existing BASIS project. The creation of a new project based on BASIS is occasionally also referred to as instantiating the *Software Project Template*.

For a detailed description and overview of the available command options, please refer to the output of the following command:

```
basisproject --help
```

Create a New Project The fastest way to create a new project is to call the project tool with the name of the new project and a brief project description as arguments:

```
basisproject --name MyProject \  
--description "This is a brief description of the project."
```

This will create a subdirectory called `MyProject` under the current working directory and populate it with the standard project directory structure and BASIS configuration. No CMake commands to resolve dependencies to other software packages will be added. These can be added later either manually or as described *below*. However, if you know already that your project will depend, for example, on `ITK` and optionally make use of `VTK` if available, you can specify these dependencies when creating the project using the `--use` or `--useopt` option, respectively:

```
basisproject --name MyProject \  
--description "This is a brief description of the project." \  
--use ITK --useopt VTK
```

The `basisproject` tool will in turn modify the `BasisProject.cmake` file to add the named packages to the corresponding lists of dependencies.

Note: In order for the project tool to be able to find the correct place where to insert the new dependencies, the `<dependency>` et al. placeholders have to be present. See the `BasisProject.cmake` template file.

Modify an Existing Project The project tool allows a detailed selection of the features included in the project template for a particular BASIS project. Which of these features are needed will often not be known during the creation of the project, but change during the work on the project. Therefore, an existing BASIS project which was created as described *above* can be modified using the project tool to add or remove certain project features and to conveniently add CMake commands to resolve further dependencies on other software packages. How this is done is described in the following.

General Notes The two project attributes which cannot be modified using the project tool are the project name and its description. These attributes need to be modified manually by editing the project files. Be aware that changing the project name may require the modification of several project files including source files. Furthermore, the project name is used to identify the project within the lab and possibly even externally. Therefore, it should be fixed as early as possible. In order to change the project description, simply edit the `BasisProject.cmake` file which you can find in the top directory of the source tree. Specifically, the argument for the `DESCRIPTION` option of the `basis_project()` function.

Hence, in order to modify an existing project, the `--name` and `--description` options cannot be used. Instead, use the `--root` option to specify the root directory of the source tree of the project you want to modify or run the command without either of these options with the root directory as current working directory.

Adding Features By features, we refer here to the set of directories and contained CMake/BASIS configuration files for which template files exist in the BASIS project template. For a list of available project features, please have a look at the help output of the project tool. You can either select a pre-configured project template consisting of a certain set of directories and configuration files and optionally modify these sets by removing features from them and/or adding other features, or you can simply remove and/or add selected features only from/to the current set of directories and configuration files which already exist in the project's source tree.

For example, if you created a project using the standard project template (i.e., by supplying no particular option or the option `--standard` during the project creation), but your software requires auxiliary data such as a pre-computed lookup table or a medical image atlas, you can add the `data/` directory in which these auxiliary files should be stored in the source tree using the command:

```
basisproject --data
```

As another example, if you want to extend the default *script configuration file* which is used to configure the build of scripts written in Python, Perl, BASH, or any other scripting language (even if not currently supported by BASIS will it likely still be able to “build” these), use the command:

```
basisproject --config-script
```

Removing Features For example, in order to remove the `conf/Settings.cmake` file and the `example/` directory tree, run the command:

```
basisproject --noconfig-settings --noexample
```

If any of the project files which were initially added during the project creation differ from the original project file, the removal of such files will fail with an error message. If you are certain that the changes are not important and still want to remove those files from the project, use the `--force` option. Moreover, if a directory is not empty, it will only be removed if the `--force` option is given. Note that a directory is also considered empty if it only contains hidden subdirectories which are used by the revision control software to manage the revisions of the files inside this directory, i.e., the `.svn/` subdirectory in case of Subversion or the `.git/` subdirectory in case of Git. Before using the `--force` option, you should be certain which directories would be removed and if their content is no longer needed. Thus, run any command first without the `--force` option, and only if it failed consider to add the `--force` option.

Adding Dependencies A dependency is either a program required by your software at runtime or an external software package such as the `nifticlib` or `ITK`. The project tool can be used to add the names of packages your project depends on to the lists of dependencies which are given as arguments to the `basis_project()` command. For each named package in this list, the `basis_find_package()` command is called to look for a corresponding package installation. In order to understand how CMake searches for external software packages, please read the documentation of CMake's `find_package()` command.

The BASIS package provides so-called *Find modules* (e.g., `FindMATLAB.cmake` or `FindNiftiCLib.cmake`) for external software packages which are commonly used at SBIA and not (yet) part of CMake or improve upon the standard modules. If you have problems resolving the dependency on an external software package required by your software due to a missing corresponding Find module, please contact the maintainer of the BASIS project and state your interest in a support by BASIS for this particular software package. Alternatively, you can write such Find module yourself and save it in the `PROJECT_CONFIG_DIR` of your project.

As an example on how to add another dependency to an existing BASIS project, consider the following scenario. We created a project without any dependency and now notice that we would like to make use of ITK in our implementation.

Thus, in order to add CMake code to the build configuration to resolve the dependency on ITK, which also includes the so-called Use file of ITK (named `UseITK.cmake`) to import its build configuration, run the command:

```
basisproject --use ITK
```

If your project can optionally make use of the features of a certain external software package, but will also built and run without this package being installed, you can use the `--useopt` option to exploit CMake code which tries to find the software package, but will not cause CMake to fail if the package was not found. In this case, you will need to consider the `<Pkg>_FOUND` variable in order to decide whether to make use of the software package or not. Note that the package name is case sensitive and that the case must match the one of the first argument of `basis_find_package()`.

For example, let's assume your software can optionally make use of CUDA. Therefore, as CMake includes already a `FindCUDA.cmake` module, we can run the following command in order to have CMake look for an installation of the CUDA libraries:

```
basisproject --useopt CUDA
```

If this search was successful, the CMake variable `CUDA_FOUND` will be `TRUE`, and `FALSE` otherwise.

Another example of a dependency on an external package is the compilation of MATLAB source files using the [MATLAB Compiler](#) (MCC). In this case, you need to add a dependency on the MATLAB package. Please note that it is important to capitalize the package name and not to use `Matlab` as this would refer to the `FindMatlab.cmake` module included with CMake. The `FindMATLAB.cmake` module which we are using is included with BASIS. It improves the way CMake looks for a MATLAB installation and furthermore looks for executables required by BASIS, such as in particular `matlab`, `mcc`, and `mex`. Use the following command to add a dependency on MATLAB:

```
basisproject --use MATLAB
```

Removing Dependencies The project tool does at the moment not support the removal of previously added dependencies. Therefore, please edit the `BasisProject.cmake` file manually and simply remove all CMake code referring to the particular package you do no longer require or use.

Add Modules BASIS supports the *modularization* of a project similar to the [ITK 4 Modularization](#), where each module is itself a BASIS project which may depends on other modules of the top-level project or other external packages. As each module itself is a project, modules are created just the same way as projects are created. The only difference might be that modules may include different sets of features (directories and files) than the top-level project. A project which uses such modularization in turn often does not include source files by its own, but is a collection of the projects (i.e., subprojects) which are its modules.

Therefore, the top-level project often excludes the `src/` subdirectory, but includes the `modules/` directory instead, in which the project's modules reside. First create the top-level project as follows (or simply add a `modules/` directory to an existing project):

```
basisproject --name MyToolkit --description "A modularized project." --toplevel
```

To now add modules to your modularized project, i.e., one which has a `modules/` subdirectory, change to the `modules/` subdirectory of the top-level project, and run the command:

```
basisproject --name MyModule --description "A module of MyToolkit." --module
```

Update a Project Occasionally, the project template of BASIS may be modified as the development of BASIS progresses. In such case, you may want or need to update the files of a project which have been created from a previous version of the project template. In order to help updating a project to a newer project template version, the project tool uses a three-way file comparison similar to Subversion to merge changes in the template files with those changes you have made to the corresponding files of your project. If such merge fails because both the template as

well as the project file have been changed at the same lines, a merge conflict occurs which has to be resolved manually. In no case, however, the project tool will discard your changes. There will always be a backup of your current project file, before the automatic file merge is performed.

To update the project files, run the following command in the root directory of your project's source tree:

```
basisproject --update
```

If the project template has not been changed since the last update, no files will be modified by this command.

Resolving Merge Conflicts When the same lines of the template file as well as the project file have been modified since the creation or last update of the project, you will get a merge conflict. A merge conflict results in a merged project file which contains the changes of both the template and your current project file. Markers such as the following are used to highlight the lines of the merged file which are in conflict with each other.

Marker	Description
<<<<<<<	Marks the start of conflicting lines. This marker is followed by your changes from the corresponding lines of your project file.
	Marks the start of the corresponding lines from the original template file which was used to create the project or which the project has been updated to last.
=====	Marks the start of the corresponding lines from the current template file, i.e., the one the project file should be updated to.
>>>>>>>	Marks the end of the conflicting lines.

In order to resolve the conflicts in one file, you have to edit the merged project file manually. For reference, the project tool writes the new template file to a file named like the project file in conflict with this project file, using .template as file name suffix. It further keeps a backup of your current project file before the update. The file name suffix for this backup file is .mine. For example, if conflicts occurred when updating the README.txt file, the following files are written to your project's directory.

File Name	Description
README.txt.mine	A copy of the project file before the update.
README.txt.template	A copy of the current template file which differs from the template file used to create the project or corresponds to the version of the template file of the last update.
README.txt	The file containing changes from both the README.txt.template and README.txt.mine file, where conflicts have been highlighted using above markers.

After you edited the project files which contain conflicts, possibly using merge tools installed on your system, you need to remove the .template and .mine files to let the project tool know that the conflicts are resolved. Otherwise, when you run the update command again, it will fail with an error message indicating that there are unresolved merge conflicts. You can delete those files either manually or using the following command in the root directory of your project's source tree.

```
basisproject --cleanup
```

Managing Test Data

This document describes how example and test data can be stored outside the source tree.

See Also:

<http://www.cmake.org/Wiki/ITK/Git/Develop/Data#ExternalData>

See Also:

http://vtk.org/Wiki/ITK_Release_4/Testing_Data

Documenting Software

ChangeLog Generated from revision history.

Software Manual Introduces users to software tools and guides them through example application.

Developer's Guide Describes implementation details.

API Documentation Documentation generated from source code, e.g., using Doxygen, Epydoc, Sphinx,...

Software Web Site A web site can be created using the documentation generation tool [Sphinx](#). The main input to this tool are text files written in the lightweight markup language [reStructuredText](#). A default theme for use at SBIA has been created which is part of BASIS. This theme together with the text files that define the content and structure of the site, the HTML pages of the software web site can be generated by `sphinx-build`. The CMake function `basis_add_doc()` provides an easy way to add such web site target to the build configuration. For example, the template `doc/CMakeLists.txt` file contains the following section:

```
# -----  
# web site (optional)  
if (EXISTS "${CMAKE_CURRENT_SOURCE_DIR}/site/index.rst")  
  basis_add_doc (  
    site  
    GENERATOR      Sphinx  
    BUILDER        html dirhtml pdf man  
    MAN_SECTION    7  
    SHOW_RELEASE   false  
    RELLINKS       news download examples help publications  
    COPYRIGHT      "<year> University of Pennsylvania"  
    AUTHOR         "<author>"  
  )  
endif ()
```

where `<year>` and `<author>` should be replaced by the proper values. This is usually done by the *[basisproject](#)* command-line tool upon creation of a new project.

This CMake code adds a build target named `site` which invokes `sphinx-build` with the proper default configuration to generate a web site from the `reStructuredText` source files with file name extension `.rst` found in the `site/` subdirectory. The source file of the main page, the so-called master document, of the web site must be named `index.rst`. The main pages which are linked in the top navigation bar are named using the `RELLINKS` option of `basis_add_sphinx_doc()`, the CMake function which implements the addition of a Sphinx documentation target. The corresponding source files must be named after these links. For example, given above CMake code, the `reStructuredText` source of the page with the download instructions has to be saved in the file `site/download.rst`.

See the *[corresponding section](#)* of the *[Build and Installation](#)* guide for details on how to generate the HTML pages from the `reStructuredText` source files given the specification of a Sphinx documentation build target such as the `site` target defined by above template CMake code.

Branch and Release

This guide defines the process of creating a new development branch other than the trunk and the creation of a release version of a software. Before reading this document, you should be familiar with the basic structure of any revision controlled software project as described in the *[Filesystem Hierarchy Standard](#)*.

Branching and Merging Please read the corresponding [SVN Book](#) article.

Releasing Software Whenever the software of a project is to be used by another project or user, the following steps have to be performed in order to create a new release version of the software.

1. If the development was carried out in a branch other than the trunk, the changes which shall be part of the release version have to be merged back to the trunk. Therefore, use the `svn merge` command as described in the [SVN Book](#).
2. Then the trunk is copied to a branch which is used to apply release specific adjustments such as setting the version number or to apply bug fixes to this particular release version. Therefore, name this branch “<project>-<major>.<minor>” (note that the patch number is excluded!) to indicate that this branch represents the “<major>.<minor>” series of software releases.

See [Branching and Merging](#) for details on how to create a new branch.

3. Edit the `BasisProject.cmake` file of the new release branch and change the `VERSION` argument to the proper version as described below.

The version number consists of three components: the major version number, the minor version number, and the patch number. The format of the version number is “<major>.<minor>.<patch>”, where the minor version number and patch number default to 0 if not given. Only digits are allowed except of the two separating dots.

For release candidates which are made available for review, on the other side, instead of the patch number, prepend “rc<N>” to the release version, where N is the number of the release candidate. For example, the first release candidate of the first stable release will have the version number “1.0.0rc1”, the second release candidate which is tagged after bug fixes have been applied, will have the version “1.0.0rc2”, etc. Once the 1.0 version was reviewed and is ready for final release, change the version to “1.0.0”. From now on, the patch number will be increased by one for each consecutive maintenance release of the 1.0 version.

- Beta releases have the major version number 0. The first stable release the major version number 1, the second major stable release the number 2, etc.
 - A change of the major version number indicates changes of the software [API](#) (and often [ABI](#)) and/or its behavior and/or the change or addition of major features.
 - A change of the minor version number indicates changes that are not only bug fixes and no major changes. Hence, changes of the [API](#), but not [ABI](#).
 - A change of the patch number indicates changes only related to bug fixes which did not change the software [API](#) nor [ABI](#). It is the least important component of the version number.
4. After setting the version number, tag the release branch as “<project>-<version>”, i.e., copy the branch “branches/<project>-<major>.<minor>” to “tags/<project>-<version>”.
 5. Now select the reviewers and ask them to retrieve a copy of the tagged release candidate. According to the reviewers feedback, the release branch is bug fixed and a new release candidate is tagged (after increasing the N in “<major>.<minor>rc<N>”) and made available for the next review iteration.
 6. The previous step is iterated until the release candidate passed all reviews. Once this is the case, set the version to “<major>.<minor>.0” and create a corresponding tag.
 7. Optionally, binary and source distribution packages are generated from the tagged release branch and uploaded to the public domain. See the [Packaging Software](#) guide for details on how to create such distribution packages.
 8. Inform the users that a new release is available and update any internal and external documentation related to the software package.
 9. Finally, make sure that all bug fixes which were applied to the release branch are merged back to the trunk where the development continues. Do not implement new features in the created release branch. This branch will only be used for maintenance of the “<major>.<minor>” series of the software.

Note: The trunk is not associated with a version other than the revision number as it is always in development. Therefore, the trunk always uses the invalid version 0.0.0.

Do not forget to commit all changes to the release branch, not the trunk. In particular the adjustment of the version number shall not be applied to the trunk as it will always keep the invalid version 0.0.0.

Packaging Software

This document describes the packaging of BASIS projects.

Source Distribution Package A source package for distribution, which only includes basic tests and selected modules, is generated using [CPack](#). In particular, the build target `package_source` is used to generate a `.tar.gz` file with the source files of the distribution package. This package will include all source files except those which match one of the patterns in the `CPACK_SOURCE_IGNORE_FILES` CMake list which is set to common default patterns in the [BasisPack.cmake](#) module. Additional exclude patterns for a particular package shall be added to the `Settings.cmake` file of the project. Moreover, if the project contains different *modules*, only the modules which were enabled are included. For general steps on how to configure a build tree, see the [common build instructions](#). Given a configured build tree with a generated `Makefile`, run the following command to generate the source distribution package:

```
make package_source
```

Build and Installation

This document describes the common procedure to build and install any software developed at and distributed by SBIA. In particular, software built on top of BASIS, a meta-project which defines the Build system And Software Implementation Standard established in the fall of 2011. See the [Overview](#) page for details on this meta-project.

If you obtained a binary distribution package for a supported platform, please follow the installation instructions corresponding to your operating system. The build step can be omitted in this case.

In case of problems to build, install, or use the software, please contact the SBIA Group at the University of Pennsylvania, PA, at `sbia-software` at `uphs.upenn.edu`.

Note: The commands given in this guide have to be entered in a terminal, in particular, the Bourne Again Shell ([BASH](#)). If you are not using the BASH, see the documentation of your particular shell for information on how to perform these actions using this shell instead.

Get a Copy Visit the [SBIA homepage](#) for an overview of publicly available software distribution packages. For each of these packages, download links of the available distribution packages can be requested by submitting [this form](#) with a valid email address. An email with the respective download links will be sent to you automatically. If you do not receive an email within 24 hours, please contact the SBIA Group at `sbia-software` at `uphs.upenn.edu`.

The file names of the distribution packages follow the convention `<package>-<version>-<arch><ext>`, where `<package>` is the name of the package in lowercase letters, and `<version>` is the package version in the format `<major>.<minor>.<patch>`. The `<arch>` file name part specifies the operating system and hardware architecture, i.e.,

<arch>	Description
linux-x86	Linux, 32-bit
linux-x86_64	Linux, 64-bit
darwin-i386	Darwin x86 Intel
darwin-ppc	Darwin Power PC
win32	Windows, 32-bit
win64	Windows, 64-bit
source	Source files

The file name extension <ext> is `.tar.gz` for a compressed tarball, `.deb` for a Debian package, and `.rpm` for a RPM package.

Binary Distribution Package

Debian Package This package can be installed on [Debian](#) and its derivatives such as [Ubuntu](#) using the Advanced Package Tool ([APT](#)):

```
sudo apt-get install <package>-<version>-<arch>.deb
```

RPM Package This package can be installed on [Red Hat Enterprise Linux](#) and its derivatives such as [CentOS](#) and [openSUSE](#) using the Yellowdog Updater, Modified ([YUM](#)):

```
sudo yum install <package>-<version>-<arch>.rpm
```

Mac OS Bundles for [Mac OS](#) might be available for some software packages, but this is not supported by default. Please refer to the `INSTALL` file which is located in the top directory of the respective software package.

Windows Currently, [Microsoft Windows](#) is not supported as operating system. The software development at SBIA is based on Linux, in particular [CentOS](#), and most software packages are therefore dependent on a Unix-based operating system. Thus, building and executing SBIA software under Windows will most likely require an installation of [Cygwin](#) and the build of the software from sources as described below. Some packages, on the other side, can be build on Windows as well, using, for example, [Microsoft Visual Studio](#) as build tool. The Visual Studio project files have to be generated using CMake (see [Building From Sources](#)).

As an alternative, consider the use of a Live Linux Distribution, a dual boot installation of Linux or an installation of a Linux operating system in a virtual machine using virtualization tools such as [VirtualBox](#) or proprietary virtualization solutions available for your host operating system.

Building From Sources In the following, we assume you obtained a copy of the source package as compressed tarball (`.tar.gz`). The name and version part of the package file is referred to as [BASH](#) variable:

```
pkg=<package>-<version>
```

Prerequisites The following software packages are prerequisites for any software that is based on BASIS. When you are building BASIS itself, the dependency on BASIS is obviously already fulfilled. Furthermore, the stated package versions are the minimum versions for which it is known that the software is working with. Newer version will usually be fine as well if not otherwise stated, but less certainly older versions.

Required Packages This section summarizes software packages which have to be installed on your system before this software can be build from its sources.

Package	Version	Description
CMake	2.8.4	A cross-platform, open-source build tool used to generate platform specific build configurations. It configures the system for the various build tools which perform the actual build of the software. If your operating system such as certain Linux distribution does not include a pre-build binary package of the required version yet, download a more recent CMake version from the CMake download page and build and install it from sources. Often this is easiest accomplished by using the CMake version provided by the Linux distribution in order to configure the build system for the more recent CMake version. To avoid conflict with native CMake installation, it is recommended to install your own build of CMake in a different directory.
BASIS		The Build system And Software Implementation Standard (BASIS) among other features defines the project directory structure and provides CMake implementations to ease and standardize the packing, build, testing, and installation. Refer to the <code>INSTALL</code> document of the software package you want to build for information on which particular BASIS version is required by this package.
GNU Make		The standard CMake generator used on Unix-like systems.
GNU Compiler Collection		The standard compiler collection used on Unix-like systems.

Optional Packages The packages named in the following table are used only if installed on your system, but their presence is general no requirement and you will likely be able to use the basic components of the software without these. See the `INSTALL` file of the software package for details and which packages are indeed made use of or required by this software.

Package	Version	Description
MATLAB	R2009b	The MATLAB tools such as, in particular, the MEX script are used to build MEX-Files from C++ source code. A MEX-File is a loadable module for MATLAB which implements a single function. If the software package you are building does not define any MEX build target, MATLAB might not be required.
MATLAB Compiler	R2009b	The MATLAB Compiler (MCC) is required for the build of standalone executables and shared libraries from MATLAB source files. If the software package you are building does not include any MATLAB sources (<code>.m</code> files), you do not need the MATLAB Compiler to build it.
Doxygen	1.5.9	This tools is required for the generation of the API documentation from in-source comments in C++, CMake, BASH, Python, and Perl.
Sphinx	1.1.3	This tool can be used for the generation of the documentation from in-source Python comments and in particular from reStructuredText .
LaTeX		The LaTeX tools may be required for the generation of the software manuals. Usually these are, however, already included in PDF in which case a LaTeX installation is only needed if you want to regenerate these from the LaTeX sources (if available after all).

Extract sources At first, extract the downloaded source package, e.g.:

```
tar xzf $pkg-source.tar.gz ~
```

This will extract the sources to a new diretory in your home directory named “<package>-<version>-source”.

Configure Create a directory for the build tree of the package and change to it, e.g.:

```
mkdir ~/ $pkg-build
cd ~/ $pkg-build
```

Note: An in-source build, i.e., building the software within the source tree is not supported to force a clear separation of source and build tree.

To configure the build tree, run CMake’s graphical tool `ccmake`:

```
ccmake ~/ $pkg-source
```

Press `c` to trigger the configuration step of CMake. Warnings can be ignored by pressing `e`. Once all CMake variables are configured properly, which might require the repeated execution of CMake’s configure step, press `g`. This will generate the configuration files for the selected build tool (i.e., GNU Make Makefiles in our case) and exit CMake.

Variables which specify the location of other required or optionally used packages if available are named `<Pkg>_DIR`. These variables usually have to be set to the directory which contains a file named `<Pkg>Config.cmake` or `<pkg>-config.cmake`. Alternatively, or if the package does not provide such CMake package configuration file, the installation prefix, i.e., root directory should be specified. See the build instructions of the particular software package you are building for more details on the particular `<Pkg>_DIR` variables that may have to be set if the packages were not found automatically by CMake.

Common configuration options are:

BASIS_DIR <dir>

Directory where the `BASISConfig.cmake` file is located. Alternatively, the installation prefix used to install BASIS can be specified instead.

BUILD_DOCUMENTATION ON|OFF

Whether build and installation instructions for the documentation should be added. If OFF, the build configuration of the `doc/` directory is skipped. Otherwise, the “doc” target is added which is used to build the documentation.

BUILD_EXAMPLE ON|OFF

Whether the examples should be built (if required) and/or installed.

BUILD_TESTING ON|OFF

Whether the testing tree should be built and system tests, i.e., tests that execute the installed programs and compare the outputs to the expected results should be installed (if done so by the software package).

CMAKE_BUILD_TYPE Debug|MinSizeRel|RelWithDebInfo|Release

Specify the build configuration to build. If not set, the “Release” configuration will be build.

INSTALL_PREFIX <dir>

Prefix used for package *installation*.

Note: The `INSTALL_PREFIX` option is initialized by the value of `CMAKE_INSTALL_PREFIX`, the default used by CMake to specify the installation prefix. Then, the value of `CMAKE_INSTALL_PREFIX` is forced to be identical to `INSTALL_PREFIX`, effectively renaming `CMAKE_INSTALL_PREFIX` to `INSTALL_PREFIX`.

INSTALL_SUFFIX ON|OFF

Whether to use suffix/infix for *installation*.

USE_<Pkg> ON|OFF

If the software you are building has declared optional dependencies, i.e., software packages which it makes use of if available, for each such optional package a `USE_<Pkg>` option is added by BASIS if this package

was found on your system. It can be set to OFF in order to disable the use of this optional dependency by this software.

The advanced configuration options are:

BASIS_ALL_DOC ON|OFF

Request the build of all documentation targets as part of the ALL target if BUILD_DOCUMENTATION is ON.

BASIS_COMPILE_SCRIPTS ON|OFF

Enable compilation of Python modules. If this option is enabled, only the compiled .pyc files are installed.

BASIS_DEBUG ON|OFF

Enable debugging messages during build configuration.

BASIS_INSTALL_SINFIX <sinfix>

The sinfix to use for the installation if INSTALL_SINFIX is set to ON. Otherwise, this values is ignored.

BASIS_MCC_FLAGS <flags separated by space>

Additional flags for MATLAB Compiler.

BASIS_MCC_MATLAB_MODE ON|OFF

Whether to call the MATLAB Compiler in MATLAB mode. If ON, the MATLAB Compiler is called from within a MATLAB interpreter session, which results in the immediate release of the MATLAB Compiler license once the compilation is done. Otherwise, the license is reserved for a fixed amount of time (e.g. 30 min).

BASIS_MCC_RETRY_ATTEMPTS <int>

Number of times the compilation of MATLAB Compiler target is repeated in case of a license checkout error.

BASIS_MCC_RETRY_DELAY <int>

Delay in seconds between retries to build MATLAB Compiler targets after a license checkout error has occurred.

BASIS_MCC_TIMEOUT <int>

Timeout in seconds for the build of a MATLAB Compiler target. If the build of the target could not be finished within the specified time, the build is interrupted.

BASIS_MEX_FLAGS <flags separated by space>

Additional flags for the MEX script.

BASIS_MEX_TIMEOUT <int>

Timeout in seconds for the build of MEX-files.

BASIS_REGISTER ON|OFF

Whether to register installed package in CMake's [package registry](#). This option is enabled by default such that packages are found by CMake when required by other packages based on this build tool.

BASIS_VERBOSE ON|OFF

Enable verbose messages during build configuration.

BUILD_CHANGELOG ON|OFF

Request build of ChangeLog as part of the ALL target. Note that the ChangeLog is generated either from the [Subversion](#) history if the source tree is a SVN working copy, or from the Git history if it is a [Git](#) repository. Otherwise, the ChangeLog cannot be generated and this option is disabled again by BASIS. In case of Subversion, be aware that the generation of the ChangeLog takes several minutes and may require the input of user credentials for access to the Subversion repository. It is recommended to leave this option disabled and to build the "changelog" target separate from the rest of the software package instead (see [Build](#)).

INSTALL_APIDOC_DIR <dir>

Installation directory of the API documentation relative to the INSTALL_PREFIX.

INSTALL_SITE_DIR <dir>

Installation directory of the web site relative to the INSTALL_PREFIX.

INSTALL_LINKS ON|OFF

Whether (symbolic) links should be created (see step 5).

Please refer also to the package specific build instructions given in the `INSTALL` file of the corresponding package which is located in the top directory of the source tree. In this document, additional project specific configuration options are document if existent.

Note: The `ccmake` tool also provides a brief description to each variable in the status bar.

Build To build the executables and libraries, run GNU Make in the root directory of the configured build tree:

```
make
```

In order to build the documentation, the `BUILD_DOCUMENTATION` option has to be set to `ON`. If not done before, this option can be enabled using the command:

```
cmake -D BUILD_DOCUMENTATION:BOOL=ON ~/ $pkg-build
```

Note that the build of the documentation may require the build of the software beforehand. If the software was not build before, the build of the documentation will also trigger the build of the software.

Each software package provides different documentation. In general, however, each software has a manual, which by default is being build by the `manual` target if the software manual is not already included as PDF document. In the latter case, the manual does not have to be build. Instead, the PDF file will simply be copied (and renamed) during the installation. Otherwise, in order to build the manual from source files such as `reStructuredText` or `LaTeX`, run the command:

```
make manual
```

If the software provides a software library for use in your own code, the API documentation may be useful which can be build using the `apidoc` target:

```
make apidoc
```

The advanced `INSTALL_APIDOC_DIR` configuration option can be set to an absolute path or a path relative to the `INSTALL_PREFIX` directory in order to modify the installation directory for the API documentation which is generated from the in-source comments using tools such as `Doxygen` and `Sphinx`. This can be useful, for example, to install the documentation in the document directory of a web server.

Some software packages further generate a project web site from text files marked up using a lightweight markup language such as `reStructuredText`. This web site can be build using the `site` target:

```
make site
```

This will generate the HTML pages and corresponding static files of the web site in `doc/site/html/`. If you prefer a single directory per document which results in prettier URLs without the `.html` extension, run the following command instead:

```
make site_dirhtml
```

The resulting web site can then be found in `doc/site/dirhtml/`. Optionally, the advanced `INSTALL_SITE_DIR` configuration option can be set to an absolute path or a path relative to the `INSTALL_PREFIX` directory in order to modify the installation directory for the generated web site. This can be useful, for example, to install the web site in the document directory of a web server.

For maintainers of the software, a developer's guide may be provided which would then be build by the `guide` target if not included as PDF document:

```
make guide
```

If the source tree is a [Subversion](#) working copy and you have access to the Subversion repository of the project or if the project source tree is a [Git](#) repository, a ChangeLog file can be generated from the commit history by building the `changelog` target:

```
make changelog
```

In case of Subversion, be aware that the generation of the ChangeLog takes several minutes and may require the input of your user credentials for access to the Subversion repository. Moreover, if the command `svn2cl` is installed on your system, it will be used to format the ChangeLog prettier. Otherwise, the plain output of the `svn log` command is written to the ChangeLog file.

Note: Not all of the above build targets are provided by each software package. You can see a list of available build targets by running `make help`. All available documentation targets, except the ChangeLog, can be build by executing the command `make doc`.

Test In order to run the software tests, execute the command:

```
make test
```

For more verbose test output, which in particularly is of importance when submitting an issue report to <sbia-software@uphs.upenn.edu>, run [CTest](#) directly with the `-V` option instead:

```
ctest -V >& $pkg-test.log
```

and attach the file `$pkg-test.log` to the issue report.

Note: If the software package does not include tests, follow the steps in the software manual to test the software manually with the provided example dataset.

Install First, make sure that the CMake configuration option `INSTALL_PREFIX` and `INSTALL_SINFIX` are set properly by running CMake:

```
cmake -D "INSTALL_PREFIX:PATH=<prefix>" -D "INSTALL_SINFIX:BOOL=ON|OFF" ~/ $pkg-build
```

This can be omitted if these variables were set already during the configuration of the build tree or if the default values should be used. On Linux, `INSTALL_PREFIX` is by default set to `/usr/local`. Note that the following strings can be used in the specification of this variable which will be substituted by the corresponding package specific values.

Pattern	Description
@PROJECT_NAME@	The case-sensitive name of the software package.
@PROJECT_NAME_UPPER@	The name of the package in uppercase only.
@PROJECT_NAME_LOWER@	The name of the package in lowercase only.
@PROJECT_VERSION@	The package version.
@PROJECT_VERSION_MAJOR@	The major number of the package version.
@PROJECT_VERSION_MINOR@	The minor number of the package version.
@PROJECT_VERSION_PATCH@	The patch number of the package version.

After the package was configured successfully, the executables and auxiliary files can be installed using the either the command:

```
make install
```

or:

```
make install/strip
```

in the top directory of the build tree. The available install targets copy the files intended for installation to the directories specified during the configuration step. The `install/strip` target additionally strips installed binary executable and shared object files, which can save disk space.

The package files are installed in the following locations on Unix:

Directory	Installed Files
<prefix>/bin/<sinfix>/	Main executable files.
<prefix>/etc/<sinfix>/	Package configuration files.
<prefix>/include/sbia/<package>/	Include files, where <prefix>/include/ needs to be in the includes search path.
<prefix>/lib/<sinfix>/	Libraries and auxiliary executables.
<prefix>/lib/cmake/<package>/	CMake package configuration files.
<prefix>/lib/python/sbia/<package>/	Python modules.
<prefix>/lib/perl5/SBIA/<Package>/	Perl modules.
<prefix>/share/<sinfix>/doc/	Package documentation files.
<prefix>/share/<sinfix>/example/	Example files used by software manual.
<prefix>/share/<sinfix>/man/man.1/	Man pages of main executables.
<prefix>/share/<sinfix>/man/man.3/	Man pages of library files.

where <prefix> is the value of `INSTALL_PREFIX` and <sinfix> is an empty string if `INSTALL_SINFIX` is OFF and the package name in lowercase otherwise preceded by the common infix `sbia/` (the default).

Additionally, if both `INSTALL_SINFIX` and `INSTALL_LINKS` are ON (the default), the following (symbolic) links are created:

Link	Target
<prefix>/bin/<exec>	<prefix>/bin/<sinfix>/<exec>
<prefix>/share/man/man.1/<exec>.1	<prefix>/share/<sinfix>/man/man.1/<exec>.1
<prefix>/share/man/man.3/<func>.3	<prefix>/share/<sinfix>/man/man.3/<func>.3

On Windows, the installation directories are named as follows instead:

Directory	Installed Files
<prefix>/Bin/<sinfix>/	Main executable files.
<prefix>/CMake/	CMake package configuration files.
<prefix>/Config/<sinfix>/	Package configuration files.
<prefix>/Include/sbia/<package>/	Include files, where <prefix>/include/ needs to be in the includes search path.
<prefix>/Library/<sinfix>/	Libraries and auxiliary executables.
<prefix>/Library/Python/sbia/<package>/	Python modules.
<prefix>/Library/Perl5/SBIA/<Package>/	Perl modules.
<prefix>/Doc/<sinfix>/	Package documentation files.
<prefix>/Example/<sinfix>/	Example files used by software manual.
<prefix>/Share/<sinfix>/	Shared files of this software package.

where <sinfix> is an empty string if `INSTALL_SINFIX` is OFF and the package name otherwise (the default).

If more than one version of a software package shall be installed, include the package version in the <prefix> by setting `INSTALL_PREFIX` to `/usr/local/@PROJECT_NAME_LOWER@-@PROJECT_VERSION@`, for example, and disable the use of the <sinfix> by setting `INSTALL_SINFIX` to OFF.

Besides the installation of the built files of the software package to the named locations, the directory where the CMake configuration file of the package was installed is added to CMake's [package registry](#) if the advanced option `BASIS_REGISTER` is set to `ON` (the default). This helps CMake to find the installed package when used by another software package based on CMake.

After the successful installation, the build tree can be deleted. It should be verified before, however, that the installation indeed was successful.

Environment In order to ease the execution of the main executable files, we suggest to add either the path `~/ $pkg-build/bin/` or `<prefix>/bin/` to the search path for executable files, i.e., the `PATH` environment variable. This is, however, generally not requirement for the correct functioning of the software.

For example, if you use `BASH` add the following line to the `~/ .bashrc` file:

```
export PATH="<prefix>/bin/:<prefix>/bin/<sinfix>/:${PATH}"
```

To be able to use the provided Python modules of the software package if any in your own Python scripts, you need to add the path `<prefix>/python/` to the search path for Python modules, e.g.:

```
export PYTHONPATH="${PYTHONPATH}:<prefix>/lib/python/"
```

or in your Python script:

```
#!/usr/bin/env python
import sys
sys.path.append(' <prefix>/lib/python/ ')
from sbia.package import module
```

To be able to use the provided Perl modules of the software package if any in your own Perl scripts, you need to add the path `<prefix>/perl5/` to the search path for Perl modules, e.g.:

```
export PERL5LIB="${PERL5LIB}:<prefix>/lib/perl5/"
```

or in your Perl script:

```
use lib ' <prefix>/lib/perl5/ ';
use SBIA::Package::Module;
```

Deinstallation

Makefile-based Deinstallation In order to undo the installation of the package files built from the sources, run the following command in the root directory of the build tree which was used to install the package:

```
cd ~/ $pkg-build
make uninstall
```

Warning: With the current implementation, this command will simply delete all the files which were installed during the **last** build of the `install` target (`make install`).

Uninstaller Script During the installation, a manifest of all installed files and a CMake script which reads in this list in order to remove these files again is generated and installed in `<prefix>/lib/cmake/$pkg/`.

If `INSTALL_SINFIX` was set to `ON` during the installation, a shell script named `uninstall` was written to the `<prefix>/bin/ <sinfix>/` directory on Unix and a corresponding Batch file on Windows. Additionally, if `INSTALL_LINKS` was set to `ON`, a symbolic link named `uninstall-$pkg` was created in `<prefix>/bin/`.

Otherwise, if `INSTALL_SINFIX` was set to `OFF`, the uninstaller is located in `<prefix>/bin/` and named `uninstall-$pkg`.

Hence, in order to remove all files installed by this package as well as the empty directories left behind inside the installation root directory given by `<prefix>`, run the command:

```
uninstall-$pkg
```

assuming that you added `<prefix>/bin/` to your `PATH` environment variable.

Note: The advantage of the uninstaller is, that the build tree is no longer required in order to uninstall the software package. Thus, you do not need to keep a copy of the build tree once you installed the software only to be able to uninstall the package again.

Automated Testing

This how-to guide describes the implementation and configuration of automated tests of software implemented on top of BASIS. Note that this guide is mainly of interest for software maintainers who have permissions to change the configuration of the software testing process and system administrators. Other lab members and software developers generally do not need to bother with these details. Note, however, that the automated tests can generally also be setup on any machine outside the lab. But in order for `CTest` to be able to submit test results to the CDash server, a VPN connection to the University of Pennsylvania Health System (UPHS) network is required.

The basistest family of scripts The BASIS package comes with a family of scripts whose name starts with the prefix `basistest`. All these scripts respond to the usual command-line options such as `--help` and `--version` to provide detailed information regarding usage and version. Further, a wrapper script named `basistest` is available which understands the subcommands `cron`, `master`, `slave` (the default), and `svn`.

- ***basistest-cron***: The command executed by the scheduled cron job.
- ***basistest-master***: The master script which runs the scheduled tests.
- ***basistest-slave***: The test execution command which is executed by the master script for each test job.
- ***basistest-svn***: The wrapper for the `svn` command which can be run non-interactively.

basistest-cron This command is run by a cron job. The configuration of the test execution command is coded into this script, optionally including the submission command used to submit test jobs to the batch-queuing system such as the [Oracle Grid Engine](#), formerly known as Sun Grid Engine (SGE), in particular. Moreover, the location of the test configuration file and test schedule file, both used by the `basistest-master` script, are specified here. Another reason for implementing this script is the setup of the environment for the execution of the master script because cron jobs are run with a minimal configuration of environment variables. Therefore, the `basistest-cron` script sources the `~swtest/.bashrc` file of the `swtest` user which is used at our lab for the automated software testing in order to, for example, add the `~swtest/bin/` directory where all the `basistest` scripts are installed to the `PATH` environment variable.

basistest-master This so-called master script is executed by the `basistest-cron` command. On each run, it reads in the configuration file given by the `--config` option line-by-line. Each line in the configuration file specifies one test job to be executed. The format of the configuration file is detailed here. Comments within the configuration file start with a pound (`#`) character at the beginning of each line.

For each test of a specific branch of a project, the configuration file contains a line following the format:

`<m> <h> <d> <project> <branch> <model> <options>`

where:

<code><m></code>	Interval in minutes between consecutive test runs. Defaults to "0" if "*" is given.
<code><h></code>	Interval in hours between consecutive test runs. Defaults to "0" if "*" is given.
<code><d></code>	Interval in days (i.e., multiples of 24 hours) between consecutive test runs. Defaults to "0" if "*" is given.
<code><project></code>	Name of the BASIS project.
<code><branch></code>	Branch within the project's SVN repository, e.g., "tags/1.0.0". Defaults to "trunk" if a "*" is given.
<code><model></code>	Dashboard model, i.e., either one of "Nightly", "Continuous", and "Experimental". Defaults to "Nightly".
<code><options></code>	Additional options to the CTest script. The "basistest" script of BASIS is used by default. Run "ctest -S <path>/basistest.ctest,help" to get a list of available options. By default, the default options of the CTest script are used. Note that this option can in particular be used to define CMake variables for the build configuration.

Note that either `<m>`, `<h>`, or `<d>` needs to be a positive number such that the interval is valid. Otherwise, the master script will report a configuration error and skip the test.

Note: Neither of these entries may contain any whitespace character!

For example, nightly tests of the main development branch (trunk) of the project BASIS itself which are run once every day including coverage analysis are scheduled by:

```
* * 1 BASIS trunk Nightly coverage,memcheck
```

Besides the configuration file, which has to be edited manually, a test schedule file is maintained by the testing master. For each configured test job, the master consults the current schedule to see whether the test is already due for execution given the testing interval specified in the configuration file and the last time the test was executed. If the test is due for execution, the testing command, i.e., by default the *basistest-slave*, is executed and the test schedule updated by the testing master. Otherwise, the execution of the test is skipped.

basistest-slave This script wraps the execution of the CTest script used for the automated testing of BASIS projects including the submission of the test results to the CDash server. It mainly converts the command-line arguments to the correct command-line for the invocation of the CTest script.

The *basistest.ctest* script performs the actual testing of a BASIS project, i.e., the

- initial check out of the sources from the Subversion controlled repository,
- update of an existing working copy,
- build of the test executables,
- execution of the tests,
- optional coverage analysis,
- optional memory checks,
- submission of test results to the CDash server.

Run the following command in a shell to have the CTest script print its help to screen and exit. However, the *basistest-slave* script should be used instead of executing this CTest script directly. The help displayed by

this command can be used in order to determine which additional options are available (such as `coverage` and `memcheck`).

```
ctest -S basistest.ctest,help
```

basistest-svn This script simply wraps the execution of the `svn` command as the `svnuser` user as this allows for non-interactive check outs and updates of working copies without the need to provide a user name and password. The code of the script is at the moment the single line:

```
exec sudo -u svnuser /bin/sh /sbia/home/svn/bin/svnwrap "$@"
```

Note: There is another wrapper script named `svnwrap` owned by the `svnuser` involved which does the actual invocation of the `svn` command.

Administration of Software Testing The following describes the setup and configuration of the automated software tests at SBIA. Hence, these instructions are only of interest for the administrators of the automated software testing at our lab. Other users do not have the permission to become the `swtest` user. To become the `swtest` user execute:

```
sudo -u swtest sudosh
```

Note: If you want to start with a clean setup, keep in mind that the directories `~swtest/etc/` and `~swtest/var/` contain files which are not part of the BASIS project. These need to be preserved and backed up separately.

Initial BASIS Installation The testing scripts described above are part of the BASIS project. As long as this project is not installed system-wide, it has to be installed locally for use by the `swtest` user. Executing the following commands as this testing user will install BASIS locally in its home directory.

1. Check-out the BASIS sources into the directory `~swtest/src/`:

```
cd
svn --username <your own username> co "https://sbia-svn/projects/BASIS/trunk" src
```

2. Create a directory for the build tree and configure it such that BASIS will be installed in the home directory of the `swtest` user:

```
mkdir build
cd build
cmake -DINSTALL_PREFIX:PATH=~ -DINSTALL_SINFIX:BOOL=OFF \
      -DINSTALL_LINKS:BOOL=OFF \
      -DBUILD_DOCUMENTATION:BOOL=OFF \
      -DBUILD_EXAMPLE:BOOL=OFF \
      -DBUILD_TESTING:BOOL=OFF \
      ../src
```

3. Build and install BASIS with `~swtest` as installation prefix:

```
make install
```

The testing scripts described above are then installed in the directory `~swtest/bin/` and the CTest script is located in `~swtest/share/cmake/`.

Updating the BASIS Installation In order to update the testing scripts, run the following commands as the `swtest` user on `olympus` (this is important because the cron job which executes the tests will run on `olympus`).

```
cd
svn up src
cmake build
make -C build install
make clean
```

This updates the working copy of the BASIS sources in `~swtest/src/` and builds the project in the build tree `~swtest/build/`. Finally, the updated BASIS project is installed. Note that the explicit execution of CMake might be redundant. However, some modifications may not re-trigger a configuration even though it is required. Thus, it is better to run CMake manually before the make. The final `make clean` is optional. It is done in order to remove the temporary object and binary files from the build tree and thus reduce the disk space occupied.

Configuring Test Jobs

Setting up the Test Environment All tests are executed by the `swtest` user. Therefore, the common test environment can be set up in the `~swtest/.bashrc` file. Here, the [environment modules](#) which are required by all tests should be loaded. Moreover, a particular project can depend on another project and should always be build using the most recent version of that other project. Every BASIS project, in particular, depends on BASIS. Thus, after each successful test of a project which is required by other projects, the files of this project are installed locally in the home directory of the `swtest` user. By setting the `<Pkg>_DIR` environment variable, CMake will use this reference installation if available. Otherwise, it will keep looking in the default system locations.

For an example on how the test environment can be set up, have a look at the following example lines of the `~swtest/.bashrc`:

```
# BASIS is required by all tested projects
module load basis
# ITK 3.* is required by BASIS (for the test driver), HardiTk, GLISTR
module unload itk
module load itk/3.20
# Boost (>= 1.45) is required by HardiTk
module load boost
# TRILINOS is required by HardiTk
module load trilinos

# root directory for installation of project files after successful test execution
#
# Note: When logged in on olympus, we usually want to configure
#       the setup of the test environment such as updating the BASIS
#       installation used by the automated testing infrastructure itself.
#       In this case, we actually want to install the files in ~swtest/
#       and not in the DESTDIR set here.
if ! [[ 'hostname' =~ "olympus" ]]; then
    export DESTDIR="${HOME}/comp_space/destdir"
fi

# Set <Project>_DIR environment variables such that the most recent
# installations in DESTDIR are used. If a particular installation is
# not available yet, the default installation as loaded by the module
# commands above will be used instead.
export BASIS_DIR="${DESTDIR}/usr/local/lib/cmake/basis"
```

Note: The environment set up this way is common for the build of all tested projects. Hence, all projects which use ITK will use ITK version 3.20 in this example. If certain projects would require a different ITK version, the environment for these test jobs would need to be adjusted before the execution of `ctest`. This is currently not further supported by BASIS, but is an open feature to be implemented.

Available Build Configurations Every BASIS projects can be build using one of the following build configurations:

Debug Used for debugging programs.

Coverage Enables coverage analysis.

MemCheck Enables memory checks.

Release Used to build programs for deployment. This is the default build configuration.

When the `coverage` option is given, the `CTest` script `basistest.ctest` will choose the `Coverage` configuration. If the `memcheck` option is given instead, the `MemCheck` configuration is used. If both options are given, the `Coverage` configuration is chosen because it generally provides already the required options also for the memory checks. The main difference may be that the `Coverage` configuration does not necessarily include debug information (e.g., using the `-g` option for GCC) while for more precise output messages by the memory checker (e.g., `valgrind`), debug information is desirable. Another difference can be static vs. dynamic linking. Thus, generally consider to run separate test jobs for coverage analysis and memory checking.

Note: At least one test job should use the **Release** build configuration.

Adding Test Job to basistest Configuration The automated tests of BASIS projects are configured in the test configuration file of the `basistest-master` script. The format of this configuration file is detailed [here](#). Where this file is located and how it is named is configured in the `basistest-cron` script. By default, the `basistest-master` script looks for the file `/etc/basistest.conf`, but the current installation is setup such that the configuration is located in `~swtest/etc/`. The current test schedule file which is maintained and updated by the `basistest-master` script is at the moment saved as `~swtest/var/run/basistest.schedule`. The log files of the test executions are saved in the directory `~swtest/var/log/`. Note that these paths are configured in the `basistest-cron` script. Old log files are deleted by the `basistest-cron` script after each execution of the test master.

An example test jobs configuration file is given below:

```
# MM HH DD   Project Name      Branch   Dashboard   Arguments
#                                     (e.g., build configuration)
# -----
# Note: The destination directory for installations is specified by the DESTDIR
#       environment variable as set in the ~swtest/.bashrc file as well as the
#       default CMAKE_INSTALL_PREFIX.
# -----
0  1  0   BASIS                trunk    Continuous
0  0  1   BASIS                trunk    Nightly    doxygen,coverage,memcheck,install
# -----
0  6  0   DRAMMS               trunk    Continuous
0  0  1   DRAMMS               trunk    Nightly    doxygen,coverage,memcheck,install
# -----
0  0  1   GLISTR               trunk    Continuous include=sbia
0  0  7   GLISTR               trunk    Nightly    doxygen,memcheck,coverage,install
0  0  61  GLISTR               trunk    Nightly    exclude=sbia # non-parallel
# -----
0  1  0   HardiTk              trunk    Continuous BUILD_ALL_MODULES=ON
0  0  1   HardiTk              trunk    Nightly    install,BUILD_ALL_MODULES=ON
```

```
# -----
0 0 1 MICO trunk Continuous
0 0 7 MICO trunk Nightly doxygen,memcheck,coverage,install
```

Adjustment of Test Schedule The current implementation of the *basistest-master* script does not allow to specify specific times at which a test job is to be executed. It only allows for the specification of the interval between test executions. Hence, if the test master script is executed the first time with a job that should be executed every day, the job will be executed immediately and then every 24 hours later. For nightly tests, it is however often desired to actually run these tests after midnight (more specifically after the nightly start time configured in CDash such that the test results are submitted to the dashboard of the current day). To adjust the time when a test job is executed, one has to edit the test schedule file (i.e., `~swtest/var/run/basistest.schedule`) manually. This file lists in the first two columns the date and time after when the next execution of the test job corresponding to the particular row should be run. Note that the actual execution time depends on when the *basistest-cron* script is executed. So for the example of nightly test jobs, the time in the second column for this test job should be changed to “3:30:00” for example. Choosing a time after midnight will show the nightly test results on the dashboard page of CDash for the “following” work day. The nightly test of BASIS itself which is used by the other projects should be executed first such that the updated BASIS installation is already used by the other tests.

Note: As the test schedule file is generated by the *basistest-master* script, run either this script or the *basistest-cron* script with the `--dry` option if this file is missing or was not generated yet. This will skip the immediate execution of all tests, but only create the test schedule file which then can be edited manually to adjust the times.

The following is an example of such test schedule file:

```
2012-01-11 13:55:04 BASIS trunk Continuous
2012-01-11 13:55:05 HardiTk trunk Continuous BUILD_ALL_MODULES=ON
2012-01-11 18:55:04 DRAMMS trunk Continuous
2012-01-12 03:00:00 BASIS trunk Nightly doxygen,coverage,memcheck,install
2012-01-12 02:00:00 DRAMMS trunk Nightly doxygen,coverage,memcheck,install
2012-01-12 12:55:04 GLISTR trunk Continuous include=sbia
2012-01-12 02:00:00 HardiTk trunk Nightly install,BUILD_ALL_MODULES=ON
2012-01-12 12:55:05 MICO trunk Continuous
2012-01-18 03:30:00 GLISTR trunk Nightly doxygen,memcheck,coverage,install
2012-01-18 03:30:00 MICO trunk Nightly doxygen,memcheck,coverage,install
2012-03-12 03:30:00 GLISTR trunk Nightly exclude=sbia
```

Remember that the test schedule is processed by the *basistest-master* script on every script invocation. It will output the scheduled tests in chronic order of their next due date. If a test has been removed from the test configuration file, it will also no longer show up in the test schedule.

Setting up a Cron Job for Automated Testing Before you schedule a cron job for the automated software testing, open the *basistest-cron* script located in the `~swtest/bin/` directory and ensure that the settings are correct.

Then run `crontab -e` as `swtest` user on `olympus` and add an entry such as:

```
*/5 * * * * /sbia/home/swtest/bin/basistest cron
```

This will run the *basistest-cron* script and hence the testing master script every 5 minutes on `olympus`. Note that the actual interval for executing the test jobs in particular depends on the test configuration. Hence, even when the cron job is executed every 5 minutes, the actual tests may only be run once a night, a week, a month,... depending on the *configuration file* which is provided for the *basistest-master* script, no matter if any files were modified or not.

3.2 API Documentation

The [API documentation](#) is generated from in-source comments using [Doxygen](#) and the filters provided by BASIS for languages other than C++.

- [Tools v1.3 v1.2 v1.1 v1.0](#)
- [Modules v1.3 v1.2 v1.1 v1.0](#)
 - [Public CMake Functions v1.3 v1.2 v1.1 v1.0](#)
 - [Find Package Modules v1.3 v1.2 v1.1 v1.0](#)
 - [Settings v1.3 v1.2 v1.1 v1.0](#)
 - [Package Configuration v1.3 v1.2 v1.1 v1.0](#)
 - [Project Directories v1.3 v1.2 v1.1 v1.0](#)
 - [Script Configuration v1.3 v1.2 v1.1 v1.0](#)
- [Utilities v1.3 v1.2 v1.1 v1.0](#)
 - [C++ Utilities v1.3 v1.2 v1.1 v1.0](#)
 - [Python Utilities v1.3 v1.2 v1.1 v1.0](#)
 - [Perl Utilities v1.3 v1.2 v1.1 v1.0](#)
 - [BASH Utilities v1.3 v1.2 v1.1 v1.0](#)

4 Tutorials

The [BASIS Tutorials](#) and [Quick Start Guides](#) are a good start to get familiar with BASIS and its use. While the tutorials give more in-depth information and references to [The Standard](#), the quick start guides focus on getting you started as quickly as possible without all the details. Therefore, it is recommended to first have a look at the quick start guides and, if questions remain, to follow the corresponding tutorial.

4.1 Quick Start Guides

1. [Getting Started \(ref\)](#)

4.2 Tutorials

1. [Getting Started \(ref\)](#)