BASIS Software Manual

Release v3.3 (fb18c98)

Andreas Schuh

February 02, 2017

Contents

The **CMake Build system And Software Implementation Standard (BASIS)** makes it easy to create sharable software and libraries that work together. This is accomplished by combining and documenting some of the best practices, utilities, and open source projects available. More importantly, BASIS supplies a fully integrated suite of functionality to make the whole process seamless!

1 Features

Project Creation

- *Quick project setup* with mad-libs style text substitution
- *Customizable project templates*

Standards

- *Filesystem layout standards*
- Basic software implementation standards
- *Command-line parsing standards*
- *Coding Style Guidelines*

Build system utilities

- New *CMake Module APIs*
- Version Control Integration
- Automatic Packaging

Documentation

- Documentation generation tools
- Manuals
- PDF and HTML output of each
- Integrated with CMake APIs

Testing

- Unit testing
- Continuous integration
- Executable testing frameworks

Program Execution

- Parsing library
- Command execution library
- Unix philosophy and tool chains

Supported Languages:

- C++, BASH, Python, Perl, MATLAB

Supported Packages:

- CMake, CPack, CTest/CDash, Doxygen, Sphinx, Git, Subversion, reStructuredText, gtest, gflags, Boost, and many more, including custom packages.

2 Quick Start

2.1 First Steps

The following steps will show you how to

- download and install BASIS on your system.
- use the so-called "basisproject" command line tool to create a new empty project.
- add some example source files and edit the build configuration files to build the executable and library files.
- build and test the example project.

You need to have a Unix-like operating system such as Linux or Mac OS X installed on your machine in order to follow these steps. At the moment, there is no separate tutorial available for Windows users, but you can install CygWin as an alternative. Note, however, that BASIS can also be installed and used on Windows. Only the tools for *automated software tests* will not be available then. These tools are for advanced users who want to set up an automated software build and test on dedicated test machines. The testing tools are not needed for what follows.

Install BASIS

Get a copy of the source code

Clone the Git repository from GitHub as follows:

```
mkdir -p ~/local/src
cd ~/local/src
git clone --depth=1 https://github.com/cmake-basis/BASIS.git basis
cd basis
```

or download a pre-packaged `.tar.gz` of the latest BASIS release and unpack it using the following command:

```
mkdir -p ~/local/src
cd ~/local/src
tar xzf /path/to/downloaded/cmake-basis-$version.tar.gz
cd cmake-basis-$version
```

Configure the build

Configure the build system using CMake 2.8.4 or a more recent version:

```
mkdir build && cd build
ccmake ..
```

- Press `c` to configure the project.
- Change `CMAKE_INSTALL_PREFIX` to `~/local`.
- Set options `BUILD_APPLICATIONS` and `BUILD_EXAMPLE` to `ON`.
- Press `g` to generate the Makefiles and exit `ccmake`.

Build and install BASIS

CMake has generated Makefiles for GNU Make. The build is thus triggered by the make command:

```
make
```

To install BASIS after the successful build, run the following command:

```
make install
```

As a result, CMake copies the built files into the installation tree as specified by the `CMAKE_INSTALL_PREFIX` variable.

Set up the environment

For the following tutorial steps, set up your environment as follows. In general, however, only the change of the `PATH` environment variable is recommended. The other environment variables are only needed for the tutorial sessions.

Using the C or TC shell (csh/tcsh):

```
setenv PATH "${HOME}/local/bin:${PATH}"
setenv BASIS_EXAMPLE_DIR "${HOME}/local/share/basis/example"
setenv HELLOBASIS_RSC_DIR "${BASIS_EXAMPLE_DIR}/hellobasis"
```

Using the Bourne Again SHell (bash):

```
export PATH="${HOME}/local/bin:${PATH} "
export BASIS_EXAMPLE_DIR="${HOME}/local/share/basis/example"
export HELLOBASIS_RSC_DIR="${BASIS_EXAMPLE_DIR}/hellobasis"
```

Create an Example Project

Create a new and empty project as follows:

```
basisproject create --name HelloBasis --description "This is a BASIS project." \
            --root ~/local/src/hellobasis
```

The next command demonstrates that you can modify a previously created project by using the project tool again, this time with the *update* command.

```
basisproject update --root ~/local/src/hellobasis --noexample --config-settings
```

Here we removed the `example/` subdirectory and added some configuration file used by BASIS. These options could also have been given to the initial command above instead.

See also:

The guide on how to *Create/Modify a Project*, *BasisProject.cmake*, and basis_project().

Install Your Project

The build and installation of the just created empty example project is identical to the build and installation of BASIS itself:

```
mkdir ~/local/src/hellobasis/build
cd ~/local/src/hellobasis/build
cmake -D CMAKE_INSTALL_PREFIX=~/local ..
make
```

See also:

The guide on how to *Install any Software*.

Add an Executable

Copy the source file from the example to `src/`:

```
cd ~/local/src/hellobasis
cp ${HELLOBASIS_RSC_DIR}/helloc++.cxx src/
```

Add the following line to `src/CMakeLists.txt` under the section "executable target(s)":

```
basis_add_executable(helloc++.cxx)
```

Alternatively, you can use the implementation of this example executable in Python, Perl, BASH or MATLAB. In case of MATLAB, add also a dependency to MATLAB:

```
basisproject update --root ~/local/src/hellobasis --use MATLAB
```

Note: The `basis_add_executable` command, if given only a single (existing) source code file or directory as argument, uses the name of this source file without extension or the name of the directory containing all source files of the executable, respectively, as the build target name.

Change target properties

- The name of the output file is given by the `OUTPUT_NAME` property.
- To change this property, add the following line to the `src/CMakeLists.txt` file (**after** `basis_add_executable`):

```
basis_set_target_properties(helloc++ PROPERTIES OUTPUT_NAME "hellobasis")
```

If you used another source file, you need to replace "helloc++" by its name (excl. the extension).

Test the Executable

Now build the executable from the previously added source code. As the build system has been configured before using CMake, only GNU `make` has to be invoked. It will recognize the change of the `CMakeLists.txt` file and therefore reconfigure the build system before re-building the software.

```
cd ~/local/src/hellobasis/build
make
bin/hellobasis
How is it going?
```

Install the executable and test it:

```
make install
hellobasis
How is it going?
```

Note that the `hellobasis` executable was installed into the `~/local/bin/` directory as we set the installation root directory to `~/local` using the `CMAKE_INSTALL_PREFIX` CMake variable. This directory should be listed in your *PATH* environment variable when you followed the *environment set up* steps at the begin of this tutorial.

Add Libraries

Next, you will add three kinds of libraries, i.e., collections of binary or script code, to your example project. We distinguish here between private, public, and script libraries. A private library is a library without public interface which is only used by other libraries and in particular executables of the project itself. A public library provides a public interface for users of your software. Therefore, the declarations of the interface given by `.h` files in case of C/C++ are copied to the installation directory along with the binary library file upon installation. Another kind of library is one written in a scripting language such as Python, Perl, or BASH. Such library is more commonly referred to as *module*.

Add a private library

Copy the files from the example to `src/`:

```
cd ~/local/src/hellobasis
cp ${HELLOBASIS_RSC_DIR}/foo.* src/
```

Add the following line to `src/CMakeLists.txt` under the section "library target(s)":

```
basis_add_library(foo foo.cxx)
```

Add a public library

Create the subdirectory tree for the public header files declaring the public interface:

```
cd ~/local/src/hellobasis
basisproject update --root . --include
mkdir include/hellobasis
```

Copy the files from the example. The public interface is given by `bar.h`.

```
cp ${HELLOBASIS_RSC_DIR}/bar.cxx src/
cp ${HELLOBASIS_RSC_DIR}/bar.h include/hellobasis/
```

Add the following line to `src/CMakeLists.txt` under the section "library target(s)":

```
basis_add_library(bar bar.cxx)
```

Add a scripted module

Copy the example Perl module to `src/`:

```
cd ~/local/src/hellobasis
cp ${HELLOBASIS_RSC_DIR}/FooBar.pm.in src/
```

Add the following line to `src/CMakeLists.txt` under the section "library target(s)":

```
basis_add_library(FooBar.pm)
```

Note: Unlike C++ libraries, which are commonly build from multiple source files, libraries written in a scripting language are separate script module files. Therefore, `basis_add_library` can be called with only a single argument, the name of the library source file. The name of this source file will be used as build target name including the file name extension, with `.` replaced by _. This is to avoid name conflicts between library modules written in different languages which have the same name such as, for example, the BASIS Utilities for Python (`basis.py`), Perl (`basis.pm`), and Bash (`basis.sh`).

The .in suffix

- Note that some of these files have a `.in` file name suffix.
- This suffix can be omitted in the `basis_add_library` statement. It has however an impact on how this function treats this file.
- The .in suffix indicates that the file is not usable as is, but contains patterns such as `@PROJECT_NAME@` which BASIS should replace during the build of the module.
- The substitution of these `@*@` patterns is what we refer to as "building" script files.

Install the libraries

Now build the libraries and install them:

```
cd ~/local/src/hellobasis/build
make && make install
```

Create a Modularized Repository

BASIS is designed to integrate multiple BASIS libraries as part of a modular build system where components can be added and removed with ease. A top-level repository contains one or more modules or sub-projects, then builds those modules based on their dependencies.

See also:

See *Modularize a Project* for usage instructions, *Project Template* for a reference implementation, and *Project Modularization* for the design.

Create a Top Level Project

```
export TOPLEVEL_DIR="${HOME}/local/src/collection"
basisproject create --name Collection --description "This is a BASIS TopLevel project. It demonstrate
```

Create a Sub-project Containing a Library

Create a sub-project module similarly to how helloBasis was created earlier.

```
export MODA_DIR="${HOME}/local/src/collection/modules/moda"
basisproject create --name moda --description "Subproject library to be used elsewhere" --root ${MODA
cp ${HELLOBASIS_RSC_DIR}/moda.cxx ${MODA_DIR}/src/
mkdir ${MODA_DIR}/include/moda
cp ${HELLOBASIS_RSC_DIR}/moda.h ${MODA_DIR}/include/moda/
```

Add the following line to `${MODA_DIR}/src/CMakeLists.txt` under the section "library target(s)":

```
basis_add_library(moda SHARED moda.cxx)
```

Create a Sub-project that uses the Library

Create a sub-project module similarly to how helloBasis was created earlier.

```
export MODB_DIR="${TOPLEVEL_DIR}/modules/modb"
basisproject create --name modb --description "User example subproject executable utility repository
cp ${HELLOBASIS_RSC_DIR}/userprog.cxx ${MODB_DIR}/src/
```

Add the following line to `${MODB_DIR}/src/CMakeLists.txt` under the section "executable target(s)":

```
basis_add_executable(userprog.cxx)
basis_target_link_libraries(userprog moda)
```

Install the Projects

```
mkdir ${TOPLEVEL_DIR}/build
cd ${TOPLEVEL_DIR}/build
cmake -D CMAKE_INSTALL_PREFIX=~/local -D MODULE_moda=ON -D MODULE_modb=ON  ..

make install
```

Next Steps

Congratulations! You just finished your first BASIS tutorial.

So far you have already learned how to install BASIS on your system and set up your own software project. You have also seen how you can add your own source files to your newly created project and build the respective executables and libraries. The essentials of any software package! Thanks to BASIS, only few lines of CMake code are needed to accomplish this.

Now check out the various *How-to Guides* which will introduce you to even more BASIS concepts and best practices.

3 How-to Guides

The how-to guides describe BASIS concepts and best practices which help to conform with the *Standards* defined by BASIS, and explain common tasks such as creating a new project or its installation.

3.1 Create/Modify a Project

This how-to guide introduces the `basisproject` command-line tool which is installed as part of BASIS. This 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 *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 `basisproject` with the name of the new project and a brief project description as arguments:

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

Note: Use the *–full* option to create a project from all template features. Most projects, however, will only need the default set of features.

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, respectivley:

```
basisproject create --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 `basisproject` 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

`basisproject` 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 `basisproject` 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 `basisproject` 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 `basisproject`. 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 update --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 update --config-script
```

Removing Features

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

```
basisproject update --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. `basisproject` 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 update --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 update --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 update --use MATLAB
```

Removing Dependencies

`basisproject` does not currently 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.

Modularize a Project

Project Modularization is a technique that aims to maximize code reusability, allowing components to be split up as independent modules that can be shared with other projects while only building and packaging the components that are really needed. Modularized projects consist of a Top Level Project and one or more Project Modules.

Create the Top Level Project

First create the top-level project as follows (or simply add a `modules/` directory to an existing project):

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

Create the Modules

To add modules to your Top Level project, which has a `modules/` subdirectory, change to the modules/ subdirectory of the top-level project, and run the command:

```
cd MyToolkit/modules
basisproject create --name MyModule --description "A module in MyToolkit." --module
```

More than one module can be in the same folder:

```
basisproject create --name OtherModule --description "Another module in MyToolkit." --module
```

You may also add an existing BASIS project module to the `/modules` folder, but not another Top Level project.

Configure the build

Configure the build system using CMake 2.8.4 or a more recent version:

```
cd ../..
mkdir build && cd build
ccmake ../MyToolkit
```

- Press `c` to configure the project.
- Change `CMAKE_INSTALL_PREFIX` to `~/local`.
- Set option `BUILD_ALL_MODULES` to `ON`.
- Press `g` to generate the Makefiles and exit `ccmake`.

See also:

Module CMake Variables

Build the Top Level Project and its Modules

CMake has generated Makefiles for GNU Make. The build and installation are then thus triggered with the make command:

```
make
make install
```

As a result, CMake copies the built files into the installation tree as specified by the `CMAKE_INSTALL_PREFIX` variable.

Upgrade a Project

Occasionally, the project template of BASIS may be modified as the development of BASIS progresses, you may want or need to upgrade the files from a previous version to the current version of the template. `basisproject` provides the ability to upgrade by using 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. However, `basisproject` will never discard your changes. There will always be a backup of your current project file before the automatic file merge is performed.

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

```
basisproject upgrade
```

If the project template has not been changed since the last upgrade, 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, `basisproject` 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 occured 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 `basisproject` 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 upgrade --cleanup
```

3.2 Using and Customizing Templates

The BASIS Project Templates define how the `basisproject` utility performs quick project setup with mad-libs style text substitution. In other words, the template defines what substitution options are available when you run the `basisproject` command, and what files are created in a new or updated project.

Available Templates

Name	Version	Description
basis	1.1	This is the default template provided by BASIS and the one we recommend. It is easy to get started with and follows all of the BASIS *Standards*. To use it simply follow the *Quick Start*.
sbia	1.8	The original template for the Section of Biomedical Image Analysis (SBIA) of the University of Pennsylvania. This template will only be useful as an example for those that are not a member of this group.
custom	n/a	You can create your own custom template. For instructions see the *Create a Custom Template* section below.

You can find the actual templates provided by BASIS in the `data/templates` directory.

Use a Template

To use a template provided by BASIS or one that you have created, specify the name of the template including the version as subdirectory as part of the `basisproject` command as follows:

```
basisproject create --name MyProject --template basis/1.1
```

If you want to use your own custom template, simply specify the full path to the respective template directory which contains the template configuration file named `_config.py`. A relative file path must be relative to the current working directory. Other than that you can use your custom template in the same manner as described in *The How-To on Creating and Modifying a Project*.

Change the Default Template

During the installation of BASIS, it is possible to specify a custom template as the default used by `basisproject` when called without the `--template` argument. See the BasisInstallationOptions for details.

Create a Custom Template

The template includes the files that are generated and the parameters that are available to the `basisproject` utility. If you plan to create new projects frequently and have some special requirements or files that you need it may be worthwhile to create a custom template. That way everything can be instantly set up in exactly the way you need.

See also:

The *Project Template Standard* explains the layout of templates, versioning, and how custom substitutions work.

In addition to creating new projects from an existing project template, the `basisproject` command-line tool can also be used to generate a new *Project Template* customized for your needs.

The fastest way to create a new template is to call `basisproject` with the name of the new template and the option `--new-template`. Use :

```
basisproject create --name MyTemplate --new-template [--optional-command-options]
```

This will create a subdirectory called `MyTemplate/1.0` under the current working directory and populate it with the current default project template structure and BASIS configuration. To copy an entire existing template, use the `--full` option and possibly `--template` to specify the location or name and version of the existing template.

For a detailed description and overview of the available command options, please refer to the output of the `basisproject help create` command. The template options of the existing template can be used to specify which features to copy when creating the new template.

With this you can modify the the default substitutions and file contents for your needs. You can also create new versions so that users can update their source tree automatically as you improve and update your customized template.

3.3 CMake Options

The following BASIS specific options are available when building packages. For the full set of options and descriptions use the ccmake tool. For CMake specific options see the documentation for your CMake installation.

The following standard CMake options/variables can be configured, see the documentation of CMake itself for more details:

Standard CMake

-DCMAKE_BUILD_TYPE:`STRING`
: Specify the build configuration to build. If not set, the `Release` configuration will be build. Common values are `Release` or `Debug`.

-DCMAKE_INSTALL_PREFIX:`PATH`
: Prefix used for package *installation*. See also the CMake reference.

-DUSE_`<Package>`**:**`BOOL`
: If the software you are building has declared optional dependencies, i.e., software packages which it makes use of only if available, for each such optional package a `USE_<Package>` 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.

BASIS Options

There are a number of CMake options that are specific to BASIS listed throughout the following documents:

- *Filesystem Layout*
- *Module CMake Variables*

Frequently Used

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

-DBUILD_DOCUMENTATION:`BOOL`
: 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 can be used to build the documentation. You may still need to run make doc, make manual, make site, etc. by hand, this option enables those settings.

Note: Though surprising at first glance, the build of the documentation may often be preceeded by the build of the software itself. The reason is that the documentation can in general only be generated after script files have been configured. Thus, do not be surprised if `make doc` will actually first build the software if not up to date before generating the API documentation.

-DBUILD_EXAMPLE:`BOOL`
: Whether the examples should be built (if required) and/or installed.

-DBUILD_TESTING:BOOL
: 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).

Advanced

Advanced users may further be interested in the settings of the following options which in most cases are automatically derived from the non-advanced CMake options summarized above. To view these options in the CMake GUI, press the `t` key in `ccmake` (Unix) or check the `Show Advanced Values` box (Windows).

-DBASIS_ALL_DOC:BOOL
: Request the build of all documentation targets as part of the `ALL` target if `BUILD_DOCUMENTATION` is `ON`.

-DBASIS_COMPILE_SCRIPTS:BOOL
: Enable compilation of Python modules. If this option is enabled, only the compiled `.pyc` files are installed.

-DBASIS_COMPILE_MATLAB:BOOL
: Whether to compile MATLAB sources using the MATLAB Compiler (mcc) if available. If set to `OFF`, the MATLAB source files are copied as part of the installation and a Bash script for the execution of `matlab` with the `-c` option is generated on Unix or a Windows NT Command script on Windows, respectively. This allows the convenient execution of the executable implemented in MATLAB even without having a license for the MATLAB Compiler. Each instance of the built executable will take up one MATLAB license, however. Moreover, the startup of the executable is longer every time, not only the first time it is launched as is the case for mcc compiled executables. It is therefore recommended to enable this option and to obtain a MATLAB Compiler license if possible. By default, this option is `ON`.

-DBASIS_DEBUG:BOOL
: Enable debugging messages during build configuration.

-DBASIS_INSTALL_APIDOC_DIR:PATH
: Installation directory of the API documentation relative to the installation prefix.

-DBASIS_INSTALL_RPATH:BOOL
: Whether to have BASIS set the appropriate INSTALL_RPATH property of executables and shared libraries instead of CMake. This option is `ON` by default which complies with the *BASIS standard*. Note that this option may be overridden by the project developer or on the command-line by setting the variable *CMAKE_SKIP_RPATH* to *FALSE*. This is typcially done in the *config/Settings.cmake*.

-DBASIS_INSTALL_SCHEME:STRING
: Installation scheme, i.e., filesystem hierarchy, to use for the installation of the software files relative to the installation prefix specified by the *-DCMAKE_INSTALL_PREFIX*. Valid values are `default`, `usr`, `opt`, or `win`. See *Installation Tree* as defined by the *Filesystem Layout* of BASIS for more details.

-DBASIS_INSTALL_SITE_DIR:PATH
: Installation directory of the web site relative to the installation prefix.

-DBASIS_INSTALL_SITE_PACKAGES:BOOL
: Whether to install public module libraries written in a scripting language such as Python or Perl in the system-wide default locations for site packages. This option is disabled by default as write permission to these directories are required otherwise.

-DBASIS_MCC_FLAGS:STRING
: Additional flags for MATLAB Compiler separated by spaces.

-DBASIS_MCC_MATLAB_MODE:BOOL
: 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).

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

-DBASIS_MCC_RETRY_DELAY:INT
Delay in number of seconds between retries to build MATLAB Compiler targets after a license checkout error has occurred.

-DBASIS_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.

-DBASIS_MEX_FLAGS:STRING
Additional flags for the MEX script separated by spaces.

-DBASIS_MEX_TIMEOUT:INT
Timeout in seconds for the build of MEX-Files.

-DBASIS_REGISTER:BOOL
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.

-DBASIS_SUPERBUILD_MODULES:BOOL
Experimental Enable the superbuild of project modules. For projects with a large number of modules, this can dramatically reduce the build system configuration time, because the configuration of each module is deferred until the build step. The superbuild of modules is disabled by default. See *Superbuild of Modules* for more information.

-DBASIS_VERBOSE:BOOL
Enable verbose messages during build configuration.

-DBUILD_BASIS_UTILITIES_FOR_<LANG>**:**BOOL
By default, the BASIS Utilities for a given programming language are only build if any of the project's executable or library targets build from source code in the respective language makes use of these utilities. Use these options to force the build of the BASIS Utilities for the respective language. Even if not used by the project itself, the generated utility functions and header or scripted module files can be used by another project to access the project meta-data such as its name and version by including the respective project-specific BASIS Utilities.

-DBUILD_CHANGELOG:BOOL
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 the Software*).

-DBUILD_MODULES_BY_DEFAULT::BOOL
Whether to enable project modules (i.e., subprojects) by default or not. This option has only effect when given directly on the command-line when calling cmake or ccmake, respectively. Otherwise the default value of this option will be used for the first build system configuration run which adds the MODULE_* options already and sets them to the respective default (TRUE). This default value cannot be overriden by consecutive configuration runs unless the MODULE_* options themselves are changed.

-DITK_DIR:PATH
Path to the directory of your ITK installation, if applicable.

-DMATLAB_DIR:PATH
Path to the directory of your MATLAB installation, if applicable.

-DSPHINX_DIR:PATH
: Path to the directory of your Sphinx installation, if applicable.

3.4 Configure a Project

This guide demonstrates some of the more advanced details, tricks, and tools to modify and configure your project.

See also:

The guide on how to *Create/Modify a Project* and the *Project Template* which defines the typical project layout.

Build Configuration

BasisProject.cmake

The key file for any project is the BasisProject.cmake file which can be found in the root directory of each project or module if the project is a subproject of another. It sets basic information about a project such as its name, version, and dependencies. Therefore it calls the basis_project() command which provides several parameters for setting these project attributes.

Dependencies Dependencies specified as arguments of the basis_project() command also support more advanced selection of specific version and package components. If some components of an external package are optional while others are required, multiple dependency declarations to the same package can be used which will only differ in the list of package components.

The syntax for specifying dependencies is:

```
basis_project(
  # [...]
  DEPENDS
    <package_name>[-<version>][{<componen1>,<component2>,...}]
  # [...]
)
```

Note: The components can be separated by whitespace characters such as spaces, tabs, and newlines. In this case, the dependency declaration has to be enclosed in double quotes such that it is treated by CMake as a single argument of the `basis_project` command.

In the example below, `ITK-4{IOKernel}` therefore is a dependency on the external `ITK` package, in particular version 4 or above and only the `IOKernel` component is required. You can also be more specific regarding the version using a dependency declaration such as `ITK-4.2` or `ITK-3.18.0`. Whether or not an external dependency meets the version requirements is determined by CMake's find_package command. See the CMake documentation of this command for more details, where the `VERSION` and `COMPONENTS` options directly relate to the respective parts of the BASIS dependency declaration.

```
basis_project(
  # [...]
  DEPENDS
    ITK-4{IOKernel}
  OPTIONAL_DEPENDS
    PythonInterp
    JythonInterp
    Perl
```

```
    MATLAB{matlab}
    BASH
    Doxygen
    Sphinx{build}
    #<optional-dependency>
  TEST_DEPENDS
    #<test-dependency>
  OPTIONAL_TEST_DEPENDS
    MATLAB{mex}
    MATLAB{mcc}
    #<optional-test-dependency>
  # [...]
)
```

Note that in case of a modularized project, the top-level project cannot have one of its own modules as dependency. The modules themselves, however, can and usually will depend on other modules of the same top-level project. If the module should also be able to exist as a standalone project or as part of other top-level projects, the dependency declaration should refer to another module as `PackageName{OtherModule}` instead of just `OtherModule`, where `PackageName` is the name of the top-level project which provides the other module, i.e., which defines the root namespace that the modules belong to.

Settings.cmake

Besides the `BasisProject.cmake` file, the `config/Settings.cmake` file contains the second most important project build configuration settings of a BASIS project. It is not required, but will be present in many projects. It is included by the root `CMakeLists.txt` of a typical BASIS project after the project meta-data is defined, information about the project modules has been collected, and the default BASIS settings were set. It is used by projects to override these default settings and to add additional project specific CMake options to the cache, e.g., using CMake's option command. Another use case of this file is to set global project build settings such as common include directories or library paths which have not automatically been set by BASIS. In particular if an external dependency's CMake configuration or `FindPackage.cmake` module set some non-standard CMake variables, a project can make use of these in the `config/Settings.cmake` file. An example of such settings are compiler and linker flags. If you want to add certain compiler flags or override the defaults, then do so in the `config/Settings.cmake` file. It should be noted that some BASIS settings cannot be overridden using this file if the BASIS standard does not allow so. But most settings can be overridden using this file.

For example if you want to enable all compiler warnings for your project and consider them moreover as errors, you would add the following to the `config/Settings.cmake` file:

```
if(CMAKE_COMPILER_IS_GNU_CXX)
  add_definitions(-Wall -Werror)
endif()
```

Depends.cmake

This build configuration file is for advanced use only in cases where the generic resolution of external dependencies used by BASIS fails due to an incompatible external package. In other words, if you need to call basis_find_package() or even CMake's find_package directly to find a particular external dependency, add the needed commands to the Depends.cmake file. One use case would be if a package or the corresponding `FindPackage.cmake` module, respectively, requires certain CMake variables to be set prior to the `find_package` call. In such case, set these variables in `config/Depends.cmake` and specify the dependency as usual in the `BasisProject.cmake` file. If this approach is still not feasible for the particular package, add any code needed to find the dependency to `config/Depends.cmake` and remove the dependency declaration from `BasisProject.cmake` such that BASIS is not itself attempting to resolve the dependency automatically. This should only be needed and used in rare

cases where the external dependency is not following the usual CMake guidelines. Often such situation is better resolved by providing a suitable `FindPackage.cmake` module for the external dependency. This module can then be added to BASIS, or put in the `config/` directory of the project. If you have a CMake module to contribute to BASIS, we encourage you to open an issue with a patch attached or to send a pull request on GitHub.

Config.cmake.in

The `Config.cmake.in` file is the template for the so-called CMake package configuration file which is generated by BASIS at the end of the build system configuration. The generated file will be named `PackageConfig.cmake`, where `Package` is the name of the top-level project, and contain information about the installation, the exported library targets, and possibly compiler options that were used to build the project. CMake's find_package command searches for this file when looking for the package named `Package` and includes it to import the build and installation settings. Besides the typical attributes of the build and installation which are written automatically by BASIS to the `PackageConfig.cmake` file, additional custom project settings can be added using the `config/Config.cmake.in` file, along with a file named `config/ConfigSettings.cmake` which sets the CMake variables that are used in the `Config.cmake.in` template.

Version.cmake.in

This file is the template for the `PackageConfigVersion.cmake` file which is examined by CMake's find_package command in order to determine whether the found package with the package configuration in `PackageConfig.cmake` meets the requested version requirements. The default file written by BASIS contains the following CMake code which is suitable for most projects. Otherwise, add a custom `config/Version.cmake.in` template file to your project and it will be used instead.

```
# Package version as specified in BasisProject.cmake file
set (PACKAGE_VERSION "@PROJECT_VERSION@")

# Perform compatibility check here using the input CMake variables.
# See example in http://www.cmake.org/Wiki/CMake_2.6_Notes.
set (PACKAGE_VERSION_COMPATIBLE TRUE)
set (PACKAGE_VERSION_UNSUITABLE FALSE)

if ("${PACKAGE_FIND_VERSION_MAJOR}" EQUAL "@PROJECT_VERSION_MAJOR@")
  if ("${PACKAGE_FIND_VERSION_MINOR}" EQUAL "@PROJECT_VERSION_MINOR@")
    set (PACKAGE_VERSION_EXACT TRUE)
  endif ()
endif ()
```

ScriptConfig.cmake.in

The so-called script configuration file sets CMake variables which can be used in scripted executables or libraries (i.e., modules). The respective build targets are added via `basis_add_executable` or `basis_add_library`. See the *Build of Script Targets* standard for details, in particular the section about the *Script Configuration*.

Package.cmake and Components.cmake

The configuration of CPack for the generation of installers or other distribution packages, such as source code or binary packages, is done by the BasisPack.cmake module. This module includes the `config/Package.cmake` file after the CPack variables have been set to the BASIS defaults if it exists.

The default configuration is derived from the project information as specified in the `BasisProject.cmake` file. As the `config/Package.cmake` file is included before the CPack module, it can be used to override the default CPack configuration. For example, additional exclude patterns can be added to `CPACK_SOURCE_IGNORE_FILES` to exclude additional files from the source code distribution package. Another example would be to change the type of installers that should be generated by CPack by selecting the preferred CPack generators. The default generator chosen by BASIS is the TGZ generator.

To define any package components for installers which support the installation of selected components, you can use the basis_add_component(), basis_add_component_group(), basis_add_install_type(), and basis_configure_downloads() commands. The respective CPack commands used by these `basis_` counterparts are defined by the `CPack.cmake` module which is included, however, after the `config/Package.cmake` file as required by CPack. Therefore, the BasisPack module considers another project configuration file named `config/Components.cmake`. This optional file should contain any custom installation component definitions using aforementioned `basis_add_` commands.

See also:

cpack_add_component, cpack_add_component_group, cpack_add_install_type, cpack_configure_downloads

Header Files

Public Interface

Header files are considered part of the public interface of a project, if they are placed in any of the directories specified using the `INCLUDE_DIRS` parameter of the basis_project() command, which by default is the `include` directory of the project source tree. Using the recommended project layout, public header files have to be put in

- *Top Level Project*: `include/<package>/`
- *Project Module* : `<module>/include/<package>/`
- *Subproject* : `<subproject>/include/<package>/<subproject>/`

Notice the subdirectories inside the include directory that help prevent the collision of header file names across packages and subprojects. Here, `<package>` is usually the name of the top-level project which in case of a module or subproject is the argument of the `PACKAGE_NAME` (or short `PACKAGE`) parameter of basis_project().

Note: In most cases, the package name of the module is identical to the project/package name of the top-level project. Such module is considered an *internal* module of the top-level project.

In cases where the module is imported from another package, using for example a submodule feature of the used version control system, the module is considered *external* to the importing top-level project, unless the package name of the module corresponds to the (package) name of the top-level project. Even though the source tree of the top-level project includes the module source tree directly, external modules should still be considered part of an external package, i.e., the one named by the `PACKAGE_NAME` of the respective module.

Note that a top-level project whose name is specified as `PACKAGE_NAME` of a module does not have to exist. The package name serves rather as *namespace* for the module. All symbols of a software project belong to this (package) namespace. It should be emphasized that the concept of a *namespace* can be extended to all aspects of a software project, not only symbols of programming languages which have it built in such as C++. Therefore, the *symbols* which belong to the package namespace include project modules, target names, C++ classes and functions, as well as scripted libraries.

Private Interface

Header files which are located in a source code directory can be included in a source file without the need for a subdirectory structure such as the one used for public header files. These files are not automatically installed as they are assumed to be only used by `.cpp` modules which are eventually linked to an executable binary.

Private header files are generally located next to the `.cpp` files that include them. They can be included using paths relative to the location of the `.cpp` module using the `#include "header.h"` preprocessor directive. Alternatively, private header files can be included relative to a directory which is listed in the search path for header files using the syntax `#include <header.h>` which is also used for public header files.

Note: Header files which are included by other public header files or contain public definitions of object classes that are linked to a library for use by other projects, are by definition part of the public interface and therefore must be located in one of the include directories.

Search Path

All directories which are given as arguments of either the `INCLUDE_DIRS` or the `CODE_DIRS` parameter of basis_project() are automatically added to the include search path using the `BEFORE` option of CMake's `include_directories` command to ensure that the header files of the current project are preferred by the preprocessor.

Additional include paths can be added using the basis_include_directories() command. This can be done either in the `CMakeLists.txt` of the respective source code subtree or in the `config/Settings.cmake` file (recommended).

Custom Layout

Note: Using a custom project layout is not recommended.

The *BASIS layout* has been battle tested and is based on standards. It is both reusable and cross-platform with a design that prevents subtle incompatibilities and assumptions that we have encountered with other layouts. Through experience and standardization we settled on the recommended layout which we believe should be effective for most use cases.

Nonetheless, we understand that requirements and existing code cannot always accomodate the standard layout, so it is possible to customize the layout. Therefore, the basis_project() command provides several options to change the default directories and add additional custom include and source code directories to be considered by BASIS during the build system configuration.

For example, a project may contain source code of a common static library in the `Common` subdirectory, image processing related library code in `ImageProcessing`, and implementations of executables in `Tools`, while the documentation is located in the subdirectory named `Documentation` and any CMake BASIS configuration files in `Configuration`. The `BasisProject.cmake` file of this project could contain the following `basis_project()` call:

```
basis_project(
  NAME        CustomLayoutProject
  DESCRIPTION "A project which demonstrates the use of a custom source tree layout."
  CONFIG_DIR   Configuration
  DOC_DIR      Documentation
  INCLUDE_DIRS Common ImageProcessing
```

```
  CODE_DIRS    Common ImageProcessing Tools
)
```

Another example for customization is given below for a top-level project which contains different subprojects named `ModulaA`, `ModuleB`, and `ModuleC`. By default, BASIS would look for these modules in the `modules` directory. This can be changed using either of the following `basis_project` commands, where in the first case it is assumed that all modules are located in a common subdirectory named `Components`:

```
basis_project(
  NAME        TopLevelProjectWithCustomModulesDirectory
  DESCRIPTION "A project which demonstrates the use of a custom modules directory."
  MODULES_DIR Components
)
```

```
basis_project(
  NAME        TopLevelProjectWithCustomModuleDirectories
  DESCRIPTION "A project which demonstrates the use of custom module directories."
  MODULE_DIRS ModuleA ModuleB ModuleC
)
```

Superbuild

CMake's ExternalProject module is sometimes used to create a superbuild, where external components are compiled separately.

This has already been done with several projects. A superbuild can also take care of building BASIS itself if it is not installed on the system, as well as any other external library that is specified as dependency of the project.

The default project template of BASIS implements a superbuild of BASIS itself. This process is referred to as *Bootstrapping BASIS* and detailed below. A superbuild of other dependencies requires a custom superbuild script. A possible implementation of such superbuild is summarized below as well, including a working example.

Be aware, however, that there are also a number of details that become more difficult when making sure your superbuild is cross platform between operating systems and supports all of the generators and IDEs supported by CMake, such as Eclipse, Xcode, and Visual Studio, because the commands you select may only account for the platform you are using with the side effect of breaking others.

Bootstrapping BASIS

The bootstrapping of BASIS is implemented by the default `basis` template since version 1.1, which is included in BASIS since version 3.1. It is the recommended superbuild approach to automate the build of BASIS. Because BASIS is downloaded and build right away during the build system configuration, no separate ExternalProject target is required for BASIS.

The *basis project template* includes a BasisBootstrapping.cmake module which is included by the root CMakeLists.txt file. This module contains the definition of the basis_bootstrap() function which downloads, configures, and builds BASIS during the configuration of the project. It is called by the default root CMake configuration only if no BASIS installation was found on the system.

The basis_bootstrap() function accepts arguments which define the configuration for the bootstrapped BASIS build. This BASIS configuration should be such that all features of BASIS that are required to build the software project are enabled (incl. any required documentation generation support). Unused BASIS features should be disabled to not waste time for the configuration and build of these features. The resulting BASIS build will be tailored towards the needs of the project and should further only be used by this project. Users who wish a single BASIS installation for multiple packages should download and install BASIS manually.

Note: The basis_bootstrap() function will only build BASIS in the build tree of the project and use this build directly without installation. An installation of BASIS is required, however, if any of the project's executable or library targets make use of the BASIS Utilities. In this case, the BASIS_INSTALL_PREFIX must be set by the user to specify an installation prefix for the bootstrapped BASIS installation. This installation prefix should be either set to the CMAKE_INSTALL_PREFIX or a subdirectory within it as this installation should only be used by the software it was built for.

The following excerpt from the root CMakeLists.txt of the *basis project template* demonstrates the use of basis_bootstrap():

```
# look for an existing CMake BASIS installation and use it if found
find_package (BASIS QUIET)

if (NOT BASIS_FOUND)

  # otherwise download and build BASIS in build tree of project
  basis_bootstrap(
    VERSION 3.1.0          # CMake BASIS version to download
    USE_MATLAB       FALSE # Enable/disable Matlab support
    USE_PythonInterp FALSE # Enable/disable Python support
    USE_JythonInterp FALSE # Enable/disable Jython support
    USE_Perl         FALSE # Enable/disalbe Perl   support
    USE_BASH         FALSE # Enable/disable Bash   support
    USE_Doxygen      TRUE  # Enable/disable documentation generation using Doxygen
    USE_Sphinx       TRUE  # Enable/disable documentation generation using Sphinx
    USE_ITK          FALSE # Enable/disable image processing regression testing
    INFORM_USER            # Inform user during first configure step
                           # that BASIS needs to be bootstrapped or installed manually
  )

  # look for local installation
  find_package (BASIS QUIET)
  if (NOT BASIS_FOUND)
    message (FATAL_ERROR "Automatic CMake BASIS setup failed! Please install BASIS manually.")
  endif ()
endif ()
```

The INFORM_USER option causes basis_bootstrap() to display an error message during the very first configure step of CMake to inform the user that the CMake BASIS package is required to configure and build the software. It further gives users a chance to edit the BASIS_DIR path in the CMake GUI to use an existing BASIS installation.

Attention: Do not set the BASIS_INSTALL_PREFIX automatically in the root CMakeLists.txt of your project, unless the INFORM_USER option of basis_bootstrap is used. Any change of the BASIS_INSTALL_PREFIX will install BASIS in the new location during the next configure run. The user would then possibly end up with (multiple) obsolete BASIS installations. The INFORM_USER option gives users at least a chance to edit the BASIS_INSTALL_PREFIX. They must do so, however, before another configure run to avoid multiple installations.

Superbuild of other Dependencies

After the bootstrapping of BASIS, other dependencies can be build using separate external projects for each of the dependencies and one final external project which builds the software itself. This last external project will depend on all the other external projects.

Please see the *nested superbuild script of DRAMMS* for reference on how to use the ExternalProject module of CMake to implement a superbuild. As BASIS will be bootstrapped and available already when the external projects of the dependencies are added, no nested superbuild is required in this case. Thus, skip the first section of the example superbuild script (the one which adds the external project `basis`) and set `BUNDLE_EXTERNAL_PROJECTS` to `OFF`. In fact, we suggest to only copy those lines from the nested superbuild example script, which are relevant for the non-nested superbuild. The CMake code required for this will be less complex and contain considerably fewer lines of code.

Note: One goal of future BASIS releases will be to automate this proecess such that most common dependencies declared in the `BasisProject.cmake` file are automatically downloaded and build if no existing installation was found and the superbuild is enabled for this dependency. Additonal custom superbuild scripts for individual external packages would enable the superbuild of non-standard packages which are not yet supported by BASIS out-of-the-box as well.

Nested Superbuild of BASIS and other Dependencies

The second alternative uses CMake's ExternalProject module and a nested super-build approach. This approach has been applied first for the superbuild of the DRAMMS software package with an older version of BASIS. If no BASIS installation is found, an external project for BASIS is added, which downloads and installs BASIS. A second external project, named `bundle` is used to build all the other dependencies, including the software project itself. This second external project recursively uses the same CMake configuration file, but this time with a valid `BASIS_DIR`. It adds for each package to be build after BASIS an external project. Note that these external projects are build targets of the `bundle` target which itself is an external project. Therefore this approach is referred to as *nested* superbuild. All build configurations of the various packages which are build by the superbuild have to be specified in the CMakeLists.txt which implements this superbuild. Any options and variables which a user should be able to modify must be passed to the respective `ExternalProject_Add` command in this script.

See also:

Copy of the nested superbuild script of DRAMMS.

Test Configuration

CDash

BASIS supports the tools CTest/CDash which are related to CMake and provide continuous integration testing.

See also:

CDash Integration for more detailed information.

Code Coverage

The test results such as the summary files generated by gcov are uploaded by CTest to a CDash server which can visualize them. The analysis of the gcov (or Bullseye) output and its conversion to the XML format used by CDash is done by the ctest_coverage CTest command. The information needed by CTest for the upload is read from a configuration file named `CTestConfig.cmake` which must be located in the top-level directory of the project. To get a visual report without a CDash server, the command-line tool lcov can be used to transform the gcov output into an HTML page.

The relevant compiler options when using the GNU Compiler Collection (GCC) are added by the `basistest.ctest` script when the coverage option is passed in, i.e.,

```
ctest -S basistest.ctest,coverage
```

See also:

- Introduction to CTest
- How to use gcov and lcov

Installation

Prefix

The CMAKE_INSTALL_PREFIX is initialized by BASIS based on the platform which the build is configured on and the package vendor ID, i.e., the argument of the PACKAGE_VENDOR (short VENDOR) parameter of basis_project(). This package vendor ID is usually set to a combination of package provider and division or an acronym which the respective division is known by.

This default installation prefix can be overriden by the project in the config/Settings.cmake file. It can also be modified at any time from the command line, i.e.,

```
cmake -DCMAKE_INSTALL_PREFIX:PATH=/path/to/installation /path/to/code
```

RPATH

By default, BASIS sets the INSTALL_RPATH property of executables and shared libraries based on the dependencies of the target. For each shared library which the binary is linked to and belongs to the same project (or package bundle), a path relative to the location of the binary is added to the RPATH of the installed binary. To figure out all the dependencies of a build target, BASIS has to perform a depth search on the dependency graph which is rather costly. Therefore, this feature can be disabled if desired either for performance reasons or because it is preferred that CMake sets the RPATH. There are two CMake variables which decide whether the RPATH is set by BASIS. The first is the advanced option *-DBASIS_INSTALL_RPATH* which can be set during the configuration of the build system to OFF (or better before, i.e., on the command-line to avoid the unnecessarily longer configuration time). If the feature should always be disabled, add the following line to the config/Settings.cmake file of the project.

```
set (CMAKE_SKIP_RPATH TRUE)
```

Redistributable Files

In general, try to keep redistributable sources and binaries as small as possible.

3.5 Managing Test Data

Note: This how-to guide has to be written yet.

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

3.6 Documenting Software

Note: This how-to guide is yet not complete.

BASIS supports two well-known and established documentation generation tools: Doxygen and Sphinx.

Documentation Quick Start

When you use the `basisproject` tool to generate a project as described in *Create/Modify a Project*, you will have a tree with a `/doc` directory preconfigured to generate a starter documentation website and PDF just like the BASIS website.

Here is how to create a new project that supports documentation:

```
basisproject --name docProject --description "This is a BASIS project." --full
```

We will assume that you ran this command in your `~/` directory for simplicity in the steps below.

Writing Documentation

Now you can simply open the `~/docProject/doc/*.rst` files and start editing the existing reStructuredText files to create your Sphinx documentation.

You can also update your doxygen mainpage by opening `~/docProject/doc/apidoc/apidoc.dox`.

We also suggest taking a look at the `/doc` folder of the BASIS source code itself for more examples of how to write documentation.

Generating Documentation

Once you have the project ready the docs can be generated.

```
mkdir ~/docProject-build
cd ~/docProject-build
cmake ../docProject -DBUILD_DOCUMENTATION=ON -DCMAKE_INSTALL_PREFIX=~/docProject-install
make doc
make install
```

The web documentation will be in `~/docProject-install/doc/html/index.html`, and the PDF docs will be in `~/docProject-install/doc/docProject_Software_Manual.pdf`.

Serving Website Locally

Note that simply opening the documentation will not render all pages correctly due to the use of the iframe HTML tag to embed the Doxygen generated API docs and the security settings built into modern browsers. Instead, display your docs via a server, for example, using Python by running the following command in the root directory of the (installed) documentation.

Python 2:

```
python -m SimpleHTTPServer
```

Python 3:

```
python -m http.server
```

Then go to localhost:8000 to view the pages.

Doxygen Documentation

Language Support

Since version 1.8.0, Doxygen can natively generate documentation from

- C/C++
- Java
- Python
- Tcl
- Fortran.

The markup language used to format documentation comments was originally a set of commands inherited from Javadoc. Recently Doxygen also adopted Markdown and elements from Markdown Extra.

Doxygen Filters

To extend the reportoire of programming languages processed by Doxygen, so-called custom Doxygen filters can be provided which transform any source code into the syntax of one of the languages well understood by Doxygen. The target language used is commonly C/C++ as this is the language best understood by Doxygen.

BASIS includes Doxygen filters for:

- CMake
- Bash
- Perl
- MATLAB
- Python

Generating Doxygen

The basis_add_doxygen_doc() CMake command can be used to create your own custom doxygen documentation.

Sphinx Documentation

BASIS makes use of Sphinx for the alternative documentation generation from Python source code and corresponding doc strings. The markup language used by Sphinx is reStructuredText (reST).

Sphinx Documentation has the advantages of being able to be produced in many different formats, and it can be used inline in Python code, and producing documentation in a much more usable layout. However, it cannot generate documentaiton from inline code for C++ in the way that doxygen can.

Output Formats

Sphinx and restructured text allow documentation to be generated in a wide number of useful formats, including:

- HTML
- LaTeX
- man pages
- Docutils

These can be used to produce:

- software manual
- developer's guide
- tutorial slides,
- project web site

This is accomplished by providing text files marked up using reST which are then processed by Sphinx to generate documentation in the desired output format.

BASIS includes two Sphinx extensions breathe and doxylink which are included with BASIS can be used to include, respectively, link to the the documentation generated by Doxygen from the documentation generated by Sphinx. The latter only for the HTML output, which, however, is the most commonly used and preferred output format. Given that the project web site and manuals are generated by Sphinx and only the more advanced reference documentation is generated by Doxygen, this one directional linking of documentation pages is sufficient for most use cases. Currently BASIS uses doxylink because it is able to work with more complete and better organized output than breathe can handle as of the time of writing.

Themes

A number of Sphinx themes are provided with BASIS, and the recommended default theme is readable-wide that is used by the BASIS website.

- readable-wide
- readable
- agogo
- default
- haiku
- pyramid
- sphinxdoc
- basic
- epub
- nature
- readable
- scrolls
- traditional

You can also use your own theme from the web or include it yourself by simply providing a path to the theme using the HTML_THEME parameter of basis_add_doc() and basis_add_sphinx_doc().

Markdown

Markdown, GitHub flavored Markdown and Markdown Extra can be used for the root package documentation files such as the AUTHORS.md, README.md, INSTALL.md, and COPYING.md files. Many online hosting platforms for the distribution of open source software such as SourceForge and GitHub render markdown on the project page with the marked up formatting.

Note: Not all of these documentation tools are supported for all languages.

Creating Documentation

The best example for creating documenation is the BASIS documentation itself, which can be found in the `doc/apidoc` folder. The most important function for generating documentation is basis_add_doc(), which can handle the parameters of the related basis_add_sphinx_doc() and basis_add_doxygen_doc() commands.

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 and in-source comments, integrated with default template.

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
    HTML_THEME    readable-wide
    HTML_SIDEBARS globaltoc
```

```
    RELLINKS      installation documentation publications people
    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 reST 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 ../install guide for details on how to generate the HTML pages from the reST source files given the specification of a Sphinx documentation build target such as the `site` target defined by above template CMake code.

3.7 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 Layout*.

Branching and Merging

See the *Filesystem Layout* for details.

For SVN please also 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 prvieous 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 corresonding 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.

3.8 Packaging Software

This document describes the packaging of BASIS projects.

Distribution of Sources

A source package for distribution which only includes basic tests and selected modules can be 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 enabled modules 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
```

3.9 Install any Software

The following contains general build and installation instructions which apply to any project which is developed using CMake BASIS.

Build Steps Overview

See *Prerequisites* below for information on dependencies.

Build Steps

The common steps to build, test, and install software from source code based on CMake are as follows:

1. Extract source files.
2. Create build directory and change to it.
3. Run CMake to configure the build tree.
4. Build the software using selected build tool.
5. Test the built software.
6. Install the built files.

On Unix-like systems with GNU Make as build tool, these build steps can be summarized by the following sequence of commands executed in a shell, where $package and $version are shell variables which represent the name of this package and the obtained version of the software.

```
$ tar xzf $package-$version-source.tar.gz
$ mkdir $package-$version-build
$ cd $package-$version-build
$ ccmake ../$package-$version-source
```

- Press 'c' to configure the build system and 'e' to ignore warnings.
- Set CMAKE_INSTALL_PREFIX and other CMake variables and options.
- Continue pressing 'c' until the option 'g' is available.
- Then press 'g' to generate the configuration files for GNU Make.

```
$ make
$ make test    (optional)
$ make install (optional)
```

An exhaustive list of minimum build dependencies, including the build tools along detailed step-by-step build, test, and installation instructions can be found in the corresponding "Building from Sources" section of the BASIS how-to guide on software installation [2].

Please refer to the rest of this guide first if you are uncertain about above steps or have problems to build, test, or install the software on your system. If this guide does not help you resolve the issue, please contact the provider of the respective software package. In case of failing tests, please attach the output of the following command to your email:

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

Prerequisites

The following software packages are prerequisites for any software that is based on BASIS. Note that the stated package versions are usually the minimum versions for which it is known that the software is working with. Newer versions will usually be fine as well if not otherwise stated by the particular software documentation, but less certainly older versions.

See the installation instructions of the specific software package for details on what is required and which optional software is being used if available. For instructions on how to build or install any of the following software packages, please refer to the documentation of the respective package.

Required Packages

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 CMake Build system And Software Implementation Standard (BASIS) among other features defines the project directory structure and provides CMake implementations to ease and standardize the packaging, build, testing, and installation. Refer to the `INSTALL` document of the software package you want to build for information on which particular BASIS version is required by this package.
GNU Make, ninja, etc.		All build tools supported by the CMake generator
GNU Compiler Collection, Clang, etc.		A C++ compiler is required to compile the BASIS source code.

Optional Packages

Package	Version	Description
Doxygen	1.8.0	This tools is required for the generation of the API documentation from in-source comments in C++, CMake, Bash, Python, and Perl. Note that only since version 1.8.0, Python and the use of Markdown (Extra) are support by Doxygen.
Python	2.7	Python is used by the basisproject tool that generates template projects. Python is also generally supported for the implementation of tools and libraries following the BASIS standard.
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).
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 (`.m` files), you do not need the MATLAB Compiler to build it.

Build and Installation

These are the build, test, and installation steps common to any BASIS based software, including BASIS itself. See BasisInstallationSteps for installation instructions specific to the CMake BASIS package itself.

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.

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.

Package Names

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 has limited support as an operating system. The most tested platform is the Linux platform CentOS, in particular, 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:

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

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

```
tar -xzf $package-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 ~/$package-build
cd ~/$package-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 ~/$package-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 `<Package>_DIR`. These variables usually have to be set to the directory which contains a file named `<Package>Config.cmake` or `<package>-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 `<Package>_DIR` variables that may have to be set if the packages were not found automatically by CMake.

See the documentation of the available *CMake Options* for more options that can be used to configure the build of any project developed with BASIS. Please refer also to the package specific build instructions given in the `INSTALL` file or software manual of the corresponding package for information on available additional project specific configuration options.

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

Build the Software 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 *-DBUILD_DOCUMENTATION* option has to be set to `ON`. If not set before, this option can be enabled using the command:

```
cmake -D BUILD_DOCUMENTATION:BOOL=ON ~/$package-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 *-DBASIS_INSTALL_APIDOC_DIR* configuration option can be set to an absolute path or a path relative to the *-DCMAKE_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 *-DBASIS_INSTALL_SITE_DIR* configuration option can be set to an absolute path or a path relative to the *-DCMAKE_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 the Software 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, run CTest directly with the `-V` option instead:

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

and attach the file `$package-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 the Software First, make sure that the CMake configuration options *-DCMAKE_INSTALL_PREFIX*, *-DBASIS_INSTALL_SCHEME*, and *-DBASIS_INSTALL_SITE_PACKAGES* are set properly, where for normal use cases only *-DCMAKE_INSTALL_PREFIX* may be modified. These variables can be set as follows:

```
cmake -D "CMAKE_INSTALL_PREFIX:PATH=<prefix>" ~/$package-build
```

or:

```
cmake -D "CMAKE_INSTALL_PREFIX:PATH=<prefix>" \
      -D "BASIS_INSTALL_SCHEME:STRING=default|usr|opt|win" \
      -D "BASIS_INSTALL_SITE:BOOL=ON|OFF" \
      ~/$package-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, *-DCMAKE_INSTALL_PREFIX* is by default set to `/opt/<provider>/<package>[-<version>]` and on Windows to `C:/Program Files/<Provider>/<Package>[-<version>]`.

The advanced *-DBASIS_INSTALL_SCHEME* option specifies how to install the files relative to this installation prefix. If it is set to `default` (the default), BASIS will decide the appropriate directory structure based on the set installation prefix. On Unix, if the installation prefix contains the package name, the `opt` installation scheme is selected which skips the addition of subdirectories named after the package within the different installation subdirectories. This corresponds to the suggested Linux Filesystem Hierarchy for Add-on Packages , where the installation prefix is set to `/opt/<package>` or `/opt/<provider>/<package>`. Otherwise, the `usr` installation scheme is chosen which will append the package name to each installation directory to avoid conflicts between software packages installed in the same location. This installation scheme follows the Linux Filesystem Hierarchy Standard for /usr. Given the installation prefix `/usr/local`, for example, the package library files will be installed into `/usr/local/lib/<package>`. On Windows, the `win` scheme is used which does not add any package specific subdirectories to the installation path similar to the `opt` scheme. Furthermore, the directory names are more Windows-like and start with a capital letter. For example, the default installation directory for package library files on Windows given the installation prefix `C:\Program Files\<Provider>\<Package>` is `C:\Program Files\<Provider>\<Package>\Lib`.

If the *-DBASIS_INSTALL_SITE_PACKAGES* option is `ON`, module libraries written in a scripting language such as Python or Perl are installed to the system-wide default directories for site packages of these languages. As this requires write permission to these directories, this option is disabled by default.

Note: The binary executables which are intended to be called by the user are copied to the `bin/` directory, where no package subdirectory is created regardless of the installation scheme. It is in the responsibility of the package provider to choose names of the executables that are unique enough to avoid conflicts with other available software packages. Auxiliary executables, on the other side, i.e., executables which are called by the executables in the `bin/` directory, are installed in the directory for library files.

The executables and auxiliary files can be installed using 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.

If more than one version of a software package shall be installed, include the package version in the installation prefix by setting *-DCMAKE_INSTALL_PREFIX* to `/opt/[<provider>/]/<package>[-<version>]`, for example (the default). Otherwise, you may choose to install the package in `/usr/local`, which will by default make the executables in the `bin/` directory and the header files available to other packages without the need to change any environment settings.

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 *-DBASIS_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.

Set up the Environment

PATH

In order to ease the execution of the main executable files, we suggest to add the path `<prefix>/bin/` to the search path for executable files, i.e., the `PATH` environment variable. This is, however, generally not required. It only eases the execution of the command-line tools provided by the software package.

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

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

PYTHONPATH

To be able to use any provided Python modules of the software package in your own Python scripts, you need to add the path `<prefix>/lib/[<package>/]python<version>/` to the search path for Python modules if such path exists after installation:

```
export PYTHONPATH=${PYTHONPATH}:/opt/<provider>/<package>-<version>/lib/python2.7
```

or, alternatively, insert the following code at the top of your Python scripts:

```
#! /usr/bin/env python
import sys
sys.path.append('/opt/<provider>/<package>-<version>/lib/python2.7')
from package import module
```

PERL5LIB

To be able to use the provided Perl modules of the software package in your own Perl scripts, you need to add the path `<prefix>/perl5/` to the search path for Perl modules if such path exists after installation:

```
export PERL5LIB=${PERL5LIB}:/opt/<provider>/<package>-<version>/lib/perl5
```

or, alternatively, insert the following code at the top of your Perl scripts:

```
use lib '/opt/<provider>/<package>-<version>/lib/perl5';
use Package::Module;
```

Deinstallation

Makefile-based Uninstall

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 ~/$package-build
make uninstall
```

> **Warning:** This command will only delete all 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/<package>/`.

The uninstaller is located in `<prefix>/bin/` and named `uninstall-<package>`. 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-$package
```

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.

3.10 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.

Note: This how-to guide details the automated software testing at SBIA and is therefore specific to the lab's computing environment.

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:

```
<m>         Interval in minutes between consecutive test runs.
            Defaults to "0" if "*" is given.
<h>         Interval in hours between consecutive test runs.
            Defaults to "0" if "*" is given.
<d>         Interval in days (i.e., multiples of 24 hours) between consecutive
            test runs. Defaults to "0" if "*" is given.
<project>   Name of the BASIS project.
<branch>    Branch within the project's SVN repository, e.g., "tags/1.0.0".
            Defaults to "trunk" if a "*" is given.
<model>     Dashboard model, i.e., either one of "Nightly", "Continuous",
            and "Experimental". Defaults to "Nightly".
<options>   Additional options to the CTest script.
            The "basisctest" 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 sbiaCDash 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.

CDash Integration

The first step for CDash integration is to set up a CDash server by following the instructions provided in the CDash documentation.

Then you need to create a project on the CDash site of your server through the Admin interface.

Finally, you can configure CTest through the CTestConfig.cmake file which must be in a project's top-level directory to specify the URL of the CDash server as well as the project to submit test results to.

Running tests via `ctest` (**not** `make test`) will then try to submit the results to the CDash server.

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
ccmake -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.

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.

4 Standards

The following sections detail the Build system (i.e., the **B** in BASIS) and Software Implementation (i.e., the **SI** in BASIS) Standard.

4.1 Filesystem Layout

This document describes the filesystem hierarchy of BASIS projects, which is based on the Filesystem Hierarchy Standard of Linux. It has a goal of supporting:

- Unix and Windows
- Installation of multiple versions of each package on a single system
- Seamless integration of BASIS software packages
- A superproject, or super-build, concept based on a bundle build

Please note that the variable names used below are defined by BASIS using CMake, and will often refer to particular directories of a software project. These variables should be used where possible, so that directories can be renamed without breaking the build system.

The *Project Template* provides a reference implementation of this standard. See the *Create/Modify a Project* How-to Guide for details on how to make use of this template to create a new project which conforms with the filesystem hierarchy standard detailed in this section.

Legend

- <project> (<package>) is a placeholder for the lowercase project (or package) name
- <Project> is the case-sensitive project name.
- <major> is the major release number
- <minor> is the minor update number
- <patch> is the patch number
- <version> is a placeholder for the project version string <major>.<minor>.<patch>
- <source> is the root directory of a particular project source tree
- <build> is the root directory of the project's build or binary tree

Source Code Repository

Git

BASIS recommends that Git distributed version control users follow the nvie git-flow branching model. The Atlassian Gitflow Workflow Tutorial is another excellent source for this information.

Mercurial

BASIS recommends that Mercurial (hg) distributed version control users follow the hg-flow branching model. This is identical to the git-flow branching model explained in *Source Code Repository*, but uses mercurial as the version control system. The hg-flow extension is useful for assisting with development, but not required.

Subversion

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.

The root directory of a development branch, typically the trunk (see *Subversion*), is denoted by <tag> and considered relative to the base URL of the project repository. The base URL is referred to as <url>.

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 subdirectories 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.

Below the trunk and the release branches a version of the entire source tree should be present. Other branches below the `branches/` directory may contain a subset of the trunk such as the source code of the software without the examples and tests.

Source Code Tree

The Soruce Code Tree refers to the filesystem directory structure of all source code that is managed by version control. The build and intallation trees are separate entities created and populated from the source tree, so the source tree is essentially the "beating heart" of a software project.

Source Categories Source files can fall under the categories of software, build, configuration, documentation, or testing. Any files essential to the execution of the software are also considered to be part of the software source. Examples of essential files include a pre-computed lookup table and a medical image atlas.

Documentation Examples within a software project are considered to be part of both documentation and testing.

Testing The testing category can be divided into system testing and unit testing. It is important to note the difference of system tests and unit tests. As testing can often require a huge amount of image data, these datasets may be stored and managed outside the source tree. Please refer to the *Managing Test Data* guide for details on this topic.

- **System Tests** System 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. Large data sets, such as medical image data sets in their entirety, should only be required for system tests and downsampled to a very low resolution for practical reasons whenever possible.
- **Unit Tests** Unit tests, provide a specialized test of a single software module such as a C++ class or Python module. Generally, the size and amount of additional data required for unit tests is kept reasonably small. 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.

Source Code Filesystem Heirarchy The filesystem hierarchy of a software project's source tree is defined below. The names of the CMake variables defined by BASIS are on the left, while the actual names of the directories are listed on the right:

```
- 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/
    + PROJECT_SUBDIRS                   + <multiple additonal subdirs>
```

Here are CMake variables defined in place of the default name for each of the following directories:

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	*Project Modules*, each residing in its own subdirectory.
PROJECT_TESTING_DIR	Implementation of tests and test data.
PROJECT_SUBDIRS	List of additional directories for source code files.

Build Tree

CMake supports but recommends against in-source builds. Therefore, BASIS requires that the build tree be outside the source tree. Only the files in the source tree are considered to be important.

Directories in the build tree are separate from the source tree, and they are created and populated when CMake configuration and the build step are run.

```
- 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/
```

Here are CMake variables defined in place of the default name for each of the following directories:

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

Installation Schemes

An installation scheme is a specific installation tree layout that is utilized based on contextual information.

BASIS automatically switches the installation scheme if you change `CMAKE_INSTALL_PREFIX` from `/opt/...` to `/usr/...` and vice versa.

The following directory structure is used when installing the software package, either by building the install target with "make install", extracting a binary distribution package, or running an installer.

Different installation hierarchies are defined in order to account for different installation schemes depending on the location and target system on which the software is being installed.

The directory structures including the installation is defined in DirectoriesSettings.cmake. The default "/opt" prefix is hard coded for Unix. On Windows it is "C:/Program Files" or, if the registry value can be read, the corresponding directory in the installation language of the OS, e.g., "C:/Programme" in German.

BASIS knows about a few "installation schemes". These distinguish between common filesystem hierarchy standards such as the one for "/opt" or "/usr" on Unix. The difference is that under "/opt", packages are installed in their own respective subdirectories which contain then subdirectories such as "include", "lib", "doc", etc. Under the "/usr" directory, however, the hierarchy is first divided by "include", "lib", "bin", "doc", and then by package name.

Possible Schemes

The first installation scheme is referred to as the `usr` scheme which is in compliance with the Linux Filesystem Hierarchy Standard for /usr:

```
- CMAKE_INSTALL_PREFIX           - <prefix>/
    + INSTALL_CONFIG_DIR            + lib/cmake/<package>/
    + INSTALL_RUNTIME_DIR           + bin/
    + INSTALL_LIBEXEC_DIR           + lib/<package>/
    + INSTALL_LIBRARY_DIR           + lib/<package>/
    + INSTALL_ARCHIVE_DIR           + lib/<package>/
    + INSTALL_INCLUDE_DIR           + include/<package>/
    + INSTALL_SHARE_DIR             + share/
        + INSTALL_DATA_DIR              + <package>/data/
        + INSTALL_DOC_DIR               + doc/<package>/
        + INSTALL_EXAMPLE_DIR           + <package>/example/
        + INSTALL_MAN_DIR               + man/
        + INSTALL_INFO_DIR              + info/
```

Another common installation scheme, here referred to as the `opt` scheme and the default used by BASIS packages, follows the Linux Filesystem Hierarchy Standard for Add-on Packages:

```
- CMAKE_INSTALL_PREFIX           - <prefix>/
    + INSTALL_CONFIG_DIR            + lib/cmake/<package>/
    + INSTALL_RUNTIME_DIR           + bin/
    + INSTALL_LIBEXEC_DIR           + lib/
    + INSTALL_LIBRARY_DIR           + lib/
    + INSTALL_ARCHIVE_DIR           + lib/
    + INSTALL_INCLUDE_DIR           + include/<package>/
    + INSTALL_SHARE_DIR             + share/
        + INSTALL_DATA_DIR              + data/
        + INSTALL_DOC_DIR               + doc/
        + INSTALL_EXAMPLE_DIR           + example/
```

```
        + INSTALL_MAN_DIR                + man/
        + INSTALL_INFO_DIR               + info/
```

The installation scheme for Windows is:

```
- CMAKE_INSTALL_PREFIX           - <prefix>/
    + INSTALL_CONFIG_DIR            + CMake/
    + INSTALL_RUNTIME_DIR           + Bin/
    + INSTALL_LIBEXEC_DIR           + Lib/
    + INSTALL_LIBRARY_DIR           + Lib/
    + INSTALL_ARCHIVE_DIR           + Lib/
    + INSTALL_INCLUDE_DIR           + Include/<package>/
    + INSTALL_SHARE_DIR             + Share/
    + INSTALL_DATA_DIR              + Data/
    + INSTALL_DOC_DIR               + Doc/
    + INSTALL_EXAMPLE_DIR           + Example/
```

In order to install different versions of a software, choose an installation prefix that includes the package name and software version, for example, `/opt/<package>-<version>` (Unix) or `C:/Program Files/<Package>-<version>` (Windows).

Note that the directory for CMake package configuration files is chosen such that CMake finds these files automatically given that the `<prefix>` is a system default location or the `INSTALL_RUNTIME_DIR` is in the `PATH` environment.

It is important to note that the include directory always contains the package name. This way, project header files must use an include path that avoids conflicts with other packages that use identical header names. Here is a usage example:

```
#include <package/header.h>
```

Thus, the include directory that is added to the search path must be set to the `include/` directory, but not the `<package>` subdirectory.

Here are CMake variables defined in place of the default name for each of the following directories:

Directory Variable	Description
`CMAKE_INSTALL_PREFIX`	Common prefix (`<prefix>`) of installation directories. Defaults to `/opt/<provider>/<package>-<version>` on Unix and `C:/Program Files/<Provider>/<Package>-<version>` on Windows. All other directories are specified relative to this prefix.
`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_DATA_DIR`	Auxiliary data files required for the execution of the software.
`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.
`INSTALL_MAN_DIR/man1/`	Man pages of the executables in `INSTALL_RUNTIME_DIR`.
`INSTALL_MAN_DIR/man3/`	Man pages of libraries.
`INSTALL_SHARE_DIR`	Shared package files including required auxiliary data files.

Forcing Schemes

Schemes can be selected using the CMake *`-DBASIS_INSTALL_SCHEME`* variable.

You can force BASIS to use one specific scheme using `BASIS_INSTALL_SCHEME`. For example, if you want to install the software in `/usr/<package>` using the same hierarchy typically used under "/opt".

4.2 Project Template

While you can create or use any custom template you like, it is highly recommended that templates follow the BASIS *Standards*. In addition to the other standards, for BASIS compliance templates must meet the requirements outlined below.

See also:

The *Using and Customizing Templates* How-to explains how to make use of templates.

Benefits

Anyone familar with the standard will be able to quickly navigate the source tree and easily integrate your project into their own because the setup is designed for consistency and interoperability. The idea is to make projects easier for developers to create, share, and use.

BASIS Standardized Templates provide and automate the following steps:

- Configuration of the build, testing, installation, and packaging.
- Common directory structure which can be found at *Filesystem Layout*.
- CMake's `CMakeLists.txt` file setup.
- Basic build flags that are required.

Standard Project Files

File Formats

Standard project files utilize the following formats:

`.txt`	A utf8 plain text file.
`.md`	Markdown
`.rst`	reStructuredText
`CMakeLists.txt`	CMake listfile format.
`.cmake`	CMake listfile format.

Required Project Files

The following files have to be part of any project which follows the *Filesystem Layout*. This is the minimal set of project files provided when instantiating a new software project. Besides these files, a project will have either a `src/` directory or a `modules/` directory, or even both of them. See below for a description of these directories.

README.md

This is the main (root) documentation file.

- Every user is assumed to first read this file, which in turn will refer them to the more extensive documentation.
- Briefly introduces the software package, including a summary of the package files.
- Refer to the *INSTALL.txt* and *COPYING.txt* files for details on the build and installation and software license, respectively.

- Include references to scientific articles related to the software package in this file.

AUTHORS.md Names the authors of the software package and people who dirctly made notable contributions to the software, even if they did not actually edit any project files. Others who mostly contributed indirectly should be named in the *README.txt* file instead. It is not necessary to list author names in each source file, as these are generally edited by multiple people and updating the authors information within each source file is tedious.

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

INSTALL.md 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 *Install any Software* guide.

BasisProject.cmake Sets basic information about a BASIS Project and calls the basis_project() command.

The basic project information, also known as metadata, will typically include:

- the project name and release version
- a brief description which is used for the packaging
- dependencies

Note that additional dependencies may optionally be specified using by the CMake code in the *config/Depends.cmake* file. 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.

BasisProject.cmake explains using this file to configure your project.

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

Common Project Files

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

config/:apidoc:‘Settings.cmake‘ This is the main CMake script file used to configure the build system, and BASIS. Put CMake code required to configure the build system in this file.

You may want to:

- Add common compiler flags
- Add new variable definitions or modifying existing CMake BASIS variables
- Write specialized code required to utilize dependencies
- Make CMake `configure_file()` calls

Examples:

- Setting the project directory variables. The line `set(PROJECT_SUBDIRS random)` will cause BASIS to call basis_add_subdirectory() on `<source>/random` at the appropriate time during the execution of BASIS.
- See basis/config/Settings.cmake for more examples.

modules/ This directory contains independent project 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.

CMakeLists.txt Build Files

build/CMakeLists.txt CMake configuration file for performing super-build of external library components and requirements by utilizing the CMake `ExternalProject_Add()` call.

The source packages of the prerequisites are either:

- downloaded during the bundle build
- included with the distribution package.

In the latter case, these source packages should be placed in the `build/` directory next to this CMake configuration file.

data/CMakeLists.txt This CMake configuration file can contains code to acquire or simply install every data 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. For example, the in-source comments using Doxygen or reStructuredText sources using Sphinx. Moreover, for every documentation file, such as the software manual, the basis_add_doc() command has to be added to this file.

example/CMakeLists.txt This CMake configuration file contains 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 example programs if applicable or required.

src/CMakeLists.txt

This is the CMake file where your primary software packages are built.

- Use the command basis_add_library() to add a shared, static, or module library, which can also be a module written in a scripting language.
- Use the command basis_add_executable() to add an executable target, which can be either a binary or a script file.
- All targets can added to the `src/CMakeLists.txt` file using relative paths.
- If necessary, source code files may be organized in subdirectories of the `src/` directory.
- Typically subdirectories aren't necessary for less than 20 files.
- Separate `CMakeLists.txt` files can be used for each subdirectory.

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 Layout* of BASIS does not dictate how the test sources, input, and baseline data have to be organized inside the `test/` directory.

test/internal/CMakeLists.txt Tests for internal use only that require data specific to your work organization. These files are expected to be excluded from the public source distribution package are configured using this CMake configuration file.

Reasons for excluding tests from a public distribution include:

- some tests may depend on the internal software environment
- may require a particular machine architecture.
- The size of the downloadable distribution packages my otherwise be excessively large.

Documenation Files

doc/manual.rst The main page of the software manual.

doc/index.rst The main page of the project web site.

doc/intro.rst Introductory description of the project, will appear at top of website front page and at the beginning of the manual.

doc/features.rst Page listing project features that will appear after the intro on website front page and at the beginning of the manual.

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

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 currently 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 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 An optional convenience file for CMake code which uses the variables set by the standard CMake packageConfig.cmake file. BASIS generates a standard packageConfig.cmake file from *config/Config.cmake.in*, which is used by other packages to set all the CMake variables they need to utilize your package.

Example:

- The package configuration sets a variable to a list of include directories have to be added to the include search path. ConfigUse.cmake.in would then contain CMake instructions to actually add these directories to the path.

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 generally not be included in a project's source tree.

config/Depends.cmake If the generic code used by BASIS to resolve the dependencies on external packages is not sufficient, add this file to your project. CMake code required to find and make use of external software packages properly shall be added to this file. In order to only make use of the variables set by the package configuration

of the found dependency, consider to add a dependency entry to the *BasisProject.cmake* file instead and code to use these variables to *config/Settings.cmake*.

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

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

Template Layout

```
- template_name/
    - 1.0/
        + _config.py
        + src/
        + config/
        + data/
        + doc/
        + example/
        + modules/
        + test/
    - 1.1/
    - 2.0/
    - 2.1/
    - .../
```

Note: Only the files which were modified or added have to be present in the new template. The `basisproject` tool will look in older template directories for any missing files.

Template Versions

The template system is designed to help automate updates of existing libraries to new template versions. Whenever a template file is modified or removed, the previous project template has to be copied to a new directory with an updated template version! Otherwise, the three-way diff merge used by the `basisproject` tool to update existing projects to this newer template will fail.

Custom Substitutions

The template configuration file named `_config.py` and located in the top directory of each project template defines not only which files constitute a project, but also the available substitution parameters and defaults used by `basisproject`. The template configuration file requires a basic understanding of Python syntax, but is fairly easy to understand even without much experience. To get an idea of the syntax, take a look at the example below. A complete example can be found in the BASIS source package in `data/templates/basis/1.0/_config.py`.

```
# project template configuration script for basisproject tool

# ============================================================================
# required project files
required = [
  'AUTHORS.txt',
  'README.txt',
  'INSTALL.txt',
  'COPYING.txt',
  'CMakeLists.txt',
```

```
  'BasisProject.cmake'
]

# ----------------------------------------------------------------------------
# optional project files
options = {
  'config-settings' : {
    'desc' : 'Include/exclude custom Settings.cmake file.',
    'path' : [ 'config/Settings.cmake' ]
  },
  'config' : {
    'desc' : 'Include/exclude all custom configuration files.',
    'deps' : [
               'config-settings'
             ]
  },
  'data' : {
    'desc' : 'Add/remove directory for auxiliary data files.',
    'path' : [ 'data/CMakeLists.txt' ]
  }
}

# ----------------------------------------------------------------------------
# preset template options
presets = {
  'minimal' : {
    'desc' : 'Choose minimal project template.',
    'args' : [ 'src' ]
  },
  'default' : {
    'desc' : 'Choose default project template.',
    'args' : [ 'doc', 'doc-rst', 'example', 'include', 'src', 'test' ]
  },
  'toplevel' : {
    'desc' : 'Create toplevel project.',
    'args' : [ 'doc', 'doc-rst', 'example', 'modules' ]
  },
  'module' : {
    'desc' : 'Create module of toplevel project.',
    'args' : [ 'include',   'src',   'test' ]
  }
}

# ----------------------------------------------------------------------------
# additional substitutions besides <project>, <template>,...
from datetime import datetime as date

substitutions = {
  # fixed computed substitutions
  'date'  : date.today().strftime('%x'),
  'day'   : date.today().day,
  'month' : date.today().month,
  'year'  : date.today().year,
  # substitutions which can be overridden using a command option
  'vendor' : {
    'help'    : "Package vendor ID (e.g., acronym of provider and/or division).",
    'default' : "SBIA"
  },
```

```
  'copyright' : {
    'help'    : "Copyrigth statement optionally including years.",
    'default' : str(date.today().year) + " University of Pennsylvania"
  },
  'license' : {
    'help'    : "Software license statement, e.g., \"Simplified BSD\".",
    'default' : "See http://www.cbica.upenn.edu/sbia/software/license.html or COPYING file."
  },
  'contact' : {
    'help'    : "Package contact information.",
    'default' : "<vendor> <<vendor>-software at uphs.upenn.edu>"
  }
}
```

Note: The substitutions are applied recursively up to a depth of 3. Hence, if the value of a substitution is another substitution tag, it will be replaced by the value of that respective substitution. See the `contact` substitution above for an example.

Binary Template Files In general, template files are assumed to be binary and thus no substitution is performed, unless the template file is known to be a text file. Whether or not a template file is considered to be a text file for which subsitution takes place depends on its MIME type . The `basisproject` tool uses the Python MIME types module in order to determine the type of each template file. In addition to the default types known by this module, the file name extensions .cmake, .md, .mdown, .markdown, .rst, .dox, and .in are treated as text files.

4.3 Project Modularization

Project modularization is a technique that aims to maximize code reusability, allowing components to be split up as independent modules that can be shared with other projects, while only building and packaging the components that are really needed. **Top Level Project**

A top level project is a project that is split into separate independent subprojects, and each of those subprojects are referred to as modules. A top level project will often have no source files of its own, simply serving as a lightweight container for its modules. **Project Module**

A (project) module is a completely independent BASIS project with its own dependencies that resides in the `modules/` directory of a top-level project. Each module will often reside in a separate repository that is designed to be shared with other projects.

Because modules are usually developed by the same development team, name conflicts are uncommon and can be avoided by appropriate naming conventions. Therefore, all modules share a common *namespace*, namely the one of the top-level project.

For example, if `BASIS_USE_TARGET_UIDS` is enabled in `config/Settings.cmake` of the top-level project, the actual build target names of the top-level project and its modules are of the form `<toplevel>.<target>`, where `<toplevel>` is the package name of the top-level project which usually is the same as the name of the top-level project, and `<target>` is the target name argument of basis_add_executable() or basis_add_library(). Note that if `BASIS_USE_FULLY_QUALIFIED_TARGET_UIDS` is disabled (the default), the `<toplevel>` part is only used for the export of the target.

The basis_project() call of a module must use the `NAME` parameter to set the name of the module (instead of `SUBPROJECT`). **Subproject**

A subproject is very similar to a project module with a few important differences. While project modules are lightweight subprojects which are tightly integrated into the top-level project, subprojects are more self-sustained

and should be treated as separate smaller projects. The top-level project serves as meta-project to group multiple subprojects. A use case would be to bundle several more or less independent software projects in a single package. The top-level project can be thus be seen as collection of related software packages, which may or may not depend on each other.

Because subprojects are usually developed by different development teams, name conflicts are more likely to occur. Therefore, each subproject has its own (nested) *namespace* inside the namespace of the package it belongs to, whereas the symbols of modules have no own namespace, but are directly defined within the namespace of the top-level project.

For example, if `BASIS_USE_TARGET_UIDS` is enabled in `config/Settings.cmake` of the top-level project, the actual build target names are of the form `<package>.<subproject>.<target>`, where `<package>` is the package name of the subproject which corresponds to the package name of the top-level project if not specified, and `<target>` is the target name argument of basis_add_executable() or basis_add_library(). Note that if `BASIS_USE_FULLY_QUALIFIED_TARGET_UIDS` is disabled (the default), the `<package>` part is only used for the export of the target.

Other differences are that BASIS will install separate uninstaller scripts for each subproject and also register each subproject installation if *-DBASIS_REGISTER* is enabled. Therefore, a subproject which is installed by one package can be used directly by other packages as if the subproject was installed separate from the other subprojects and modules of the top-level project.

The basis_project() call of a subproject must use the `SUBPROJECT` parameter to set the name of the subproject (instead of `NAME`). Additionally, as subprojects are likely shared by multiple top-level projects, it is recommended to set the `PACKAGE_NAME` (short `PACKAGE`) to the name of the package which this subproject belongs to primarily. Note that this package need not actually exist. By providing this package name, the namespace of the subproject will always be the same no matter what the name of the top-level project is.

Note: It should be noted that the concept of a *namespace* can be extended to all aspects of a software project, not only symbols of programming languages which have it built in such as C++. Therefore, the *symbols* which belong to the package namespace include project modules, target names, C++ classes and functions, as well as scripted libraries.

See also:

See *Modularize a Project* for usage instructions and *Project Template* for a reference implementation.

Filesystem Layout

By default each module is placed in its own `modules/<module_name>` subdirectory, but this can be configured in `config/Settings.cmake` by modifying the `PROJECT_MODULES_DIR` variable. More details can be found in the *Filesystem Layout*.

The Top Level project often excludes the `src/` subdirectory, and instead includes the `modules/` directory where the project's modules reside.

Dependency Requirements

There are several features and limitations when one top level or subproject uses code from another.

- Modules may depend on each other.
- Each module of a top level project may depend on other modules of the same project, or external projects and packages.
- Only one level of submodules are allowed in a top level project
- An external project can also be another top-level project with its own modules.

Module CMake Variables

CMake variables available to any project utilizing BASIS. These options can be modified with the `ccmake` command. *CMake Options* describes other important CMake options.

CMake Variable	Description
`MODULE_<module>`	Builds the module named `<module>` when set to `ON` and excludes it when `OFF`. It is automatically set to `ON` if it is required by another module that is `ON`.
`BUILD_MODULES_BY_DEFAULT`	Sets the default state of each `MODULE_<module>` switch. `ON` by default.
`BUILD_ALL_MODULES`	Global switch enabling the build of all modules. Overrides all `MODULE_<module>` variables.
`PROJECT_IS_MODULE`	Specifies if the current project is a module of another project.

It is recommended that customized defaults for these variables be set in *config/Settings.cmake*.

Implementation

The modularization is mainly implemented with the following hierarchy presented in the same manner as a stack trace with the top function being the last function called:

- ProjectTools.cmake - basis_project_modules()
- ProjectTools.cmake - basis_project_begin()
- BasisProject.cmake - script file that is executed directly
- `CMakeLists.txt` - root file of any CMake project

The script then takes the following steps:

1. The basis_project_modules() function searches the subdirectories in the `modules/` directory for the presence of the BasisProject.cmake file.
2. BasisProject.cmake is then loaded to retrieve the meta-data of each module such as its name and dependencies.
3. A `MODULE_<module>` option is added to the build configuration for each module and module dependencies are defined that correspond to the settings in BasisProject.cmake. This enables the eventual execution of the build step to be in the correct topological order. The `MODULE_<module>` settings obey the following constraints:
    - When `OFF` the module is excluded from both the project build and any package generated by CPack.
    - When `ON` the module builds as part of the top-level project.
    - If one module requires another, the required module will automatically be set to `ON`.
    - All `MODULE_<module>` options are superceded by the `BUILD_ALL_MODULES` when it is set to `ON`.

Besides adding these options, the basis_project_modules() function ensures that the modules are configured with the right dependencies so that the generated build files will compile them correctly.

It also helps the basis_find_package() function find the other modules' 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 also 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`).

Origin

The modularization concepts and part of the CMake implementation are from the ITK 4 project. See the Wiki of this project for details on the ITK 4 Modularization.

Reuse

Modules can be built standalone without a Top Level Project.

This is why the BasisProject.cmake meta-data requires an explicit `PACKAGE_NAME`. When you configure the build system of a project module directly, i.e., by using the module's subdirectory as root of the source tree, it will still build as if it was part of a Top Level Project with name equal to the `PACKAGE_NAME` of the project.

The explicit package name is also important for the executable (target) referencing that is used for subprocess invocations covered in *Calling Conventions*. A developer can use the target name (e.g., basis.basisproject) in the BASIS utility functions for executing a subprocess, and the path to the actually installed binary is resolved by BASIS. This allows the developer of the respective module to change the location/name of a binary file through the CMake configuration and other code which uses this module's executable can still call it by its unchanged build target name. As the target name includes the package name of a project to avoid name conflicts among packages, the package name which a module belongs to must be known even if the module is build independently without any Top Level Project.

Superbuild

Note: The superbuild of project modules is yet experimental and not fully documented!

CMake's ExternalProject module is sometimes used to create a superbuild, where components of a software or its external dependencies are compiled separately. This has already been done with several projects.

An experimental superbuild of project modules is implemented by the basis_add_module() function. It is disabled by default, i.e. each module is configured right away using `add_subdirectory`. The *-DBASIS_SUPERBUILD_MODULES* option can be used to enable the superbuild of modules. This can dramatically speed up the build system configuration for projects which contain a large number of modules, because the configuration of each module is deferred until the build step. Moreover, only modules which were modified since the last build will be reconfigured when the top-level project is re-build. Without the superbuild approach, the entire build system of the top-level project needs to be reconfigured in such case.

If the superbuild of modules should always be enabled, add the following CMake code to `config/Settings.cmake`:

```
if (NOT BASIS_SUPERBUILD_MODULES)
  set (
    BASIS_SUPERBUILD_MODULES ON CACHE BOOLEAN
      "This project always builds the modules using a superbuild approach."
    FORCE
  )
  message (WARNING "Option BASIS_SUPERBUILD_MODULES set to ON as this project"
                   " always builds its modules using a superbuild approach."
                   " The BASIS_SUPERBUILD_MODULES option cannot be changed.")
endif ()
```

Alternatively, the following line would be sufficient as well without feedback for the user:

```
set (BASIS_SUPERBUILD_MODULES OFF)
```

See also:

A superbuild can also take care of building BASIS itself if it is not installed on the system, as well as any other external library that is specified as dependency of the project. See the *Superbuild of BASIS and other dependencies*.

4.4 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 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 "@CMAKE_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} || { echo "Failed to import BASIS utilities!" 1>&2; exit 1; }
exedir EXEDIR && readonly EXEDIR
[ $? -eq 0 ] || { echo 'Failed to determine directory of this executable!'; exit 1; }
readonly DATA_DIR="${EXEDIR}/@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.

4.5 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 then 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 following sections document the usage of these libraries for each respective programming language:

Parsing the Command-line Arguments in C++

For the parsing of command-line arguments in C++, BASIS includes a slightly extended version of the Templatized 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 `basis` namespace, i.e., `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 `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
 * @breif Smooth image using Gaussian or anisotropic diffusion filtering.
 */

#include <package/basis.h> // include BASIS C++ utilities


// acceptable in .cxx file
using namespace std;
using namespace 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(
                "EXENAME --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(
                "EXENAME  --anisotropic brain.nii\n"
                "Smooths the image brain.nii using an anisotropic diffusion filter.");

        CmdLine cmd(
                // program identification
                "smoothimage", PROJECT,
                // program description
                "This program smooths an input image using either a Gaussian "
                "filter or an anisotropic diffusion filter.",
                // example usage
                examples,
                // version information
                RELEASE, "2011 University of Pennsylvania");
```

```
        // 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

Note: This how-to guide has to be written yet. See the shflags.sh module as a reference until this guide is completed, keeping in mind, though, that this module will have to be revised.

Note: Yet there exist only libraries for C++ and BASH, but solutions for Java, Python, and Perl will be part of future releases.

4.6 Calling Conventions

This document discusses and describes the conventions for calling other executables from a program. The calling conventions address problems stemming from the use of relative or absolute file paths when calling executables. It also

introduce a name mapping from build target names to actual executable file paths. These calling conventions are handled through automatically generated utility functions for each supported programming language. See *Implementation* for details on the specific implementations in each language.

Purpose

One nice feature about using the target name instead of the actual executable file allows a developer of project B to call executables of project A using the ("full qualified") target names, e.g.,

```
execute("projecta.utility");
```

This target has been imported from the export file during CMake configuration and the BASIS execute function will map this target name to the installed executable of project A. The developer of project A can rename the executable or change the installation location as they wish. They only need to keep the internal target name.

The file name of executable scripts, for example, will be different on Unix and Windows. On Unix, we don't use file name extensions and instead rely on the hashbang/shebang #! directive such that script executables look and are used just like binary executables. On Windows, any executable script (i.e., only Python or Perl at the moment) is wrapped into a Windows Command file with the .cmd file name extension. This file contains a few lines additional Windows Command code to invoke the script interpreter with the very same file. The Windows Command code is just a comment to the Python/Perl interpreter which will ignore it.

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 the `bin/` directory call utility executables relative to their own location. For example, a Bash script called `main` that executes a utility script `util` in the `lib/` directory 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
. ${BASIS_Bash_UTILITIES} || { echo "Failed to import BASIS utilities!" 1>&2; exit 1; }
# get absolute directory path of auxiliary executable
exedir _EXEC_DIR && readonly _EXEC_DIR
_LIBEXEC_DIR=${_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 build 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 prepand 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 where 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()` 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.

C++

For C++ programs, the BASIS C++ utilities provide the function exepath() which maps a build target name to the absolute path of the executable file built by this target. This function makes use of an implementation of the basis::util::IExecutableTargetInfo interface 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 whose documentation is linked here is the one generated for BASIS itself.

The project implementations will, however, mainly make use of the execute() 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()` 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()` function can be found in the `basis.h` header file. Note that this file is unique to each BASIS project.

Java

The Java programming language is not yet supported by BASIS.

Python

A Python module named basis.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 exename(), exedir(), and exepath() 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 Basis.pm Perl module uses a hash reference to store the locations of the executable files relative to the module itself. The functions exename(), exedir(), and exepath() 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

The module basis.sh imitates associative arrays to store the location of the built executable files relative to this module. The functions exename(), exedir(), and exepath() 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 basis.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 basis.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 `basis.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 the aliases defined by basis.sh.

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. An implementation of the `execute()` function in MATLAB is yet not provided by BASIS.

5 Guidelines

The following sections define common guidelines for the formatting of documents such as in particular program code as well as other recommended coding guidelines. Each organization employing BASIS, however, may define their own guidelines, possibly using the following guidelines as reference.

5.1 Plain Text Format

Note: This guideline is out-dated and a new set of rules must be defined which is compatible with either Markdown or reStructuredText, nowadays the preferred lightweight markup languages for plain text files.

The following guideline, which itself is styled according to the plain text format it describes, details how plain text documentation files of a software project should be formatted.

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

Web:   http://www.cbica.upenn.edu/sbia/
Email: sbia-software at uphs.upenn.edu

Copyright (c) 2011 University of Pennsylvania. All rights reserved.
See http://www.cbica.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.cbica.upenn.edu/sbia/
  Email: sbia-software at uphs.upenn.edu

  Copyright (c) <year> University of Pennsylvania. All rights reserved.
  See http://www.cbica.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 (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.
```

6 Reference

6.1 Basic Tools

In order to ease certain tasks, the BASIS package also includes the following command-line tools:

basisproject	Creates a new project or modifies an existing one in order to add or remove certain components of the template or to upgrade to a newer template.
basistest	Implements automated software tests.
doxyfilter	Doxygen filter for all supported languages.

6.2 CMake Modules

The CMake modules and corresponding auxiliary files are used by any BASIS project for the configuration of the CMake-based build system, so that many setup steps can be automated. These commands often replace the standard CMake commands. For example, the CMake function basis_add_executable() replaces CMake's add_executable() command.

The main CMake modules are:

BasisProject.cmake	File in every BASIS project defining basic project information.
BasisTools.cmake	Defines CMake functions, macros, and variables.
BasisTest.cmake	Replacement for the CTest.cmake module of CMake.
BasisPack.cmake	Replacement for the CPack.cmake module of CMake.

6.3 Utilities

For each supported programming language, BASIS provides a library of utility functions. Some of these utilities are project independent and thus built and installed as part of the CMake BASIS package itself. Other utility implementations are project dependent. Therefore, the BASIS installation contains only template files which are customized and built during the configuration and build, respectively, of the particular BASIS project. This customization is done by the functions implemented by the UtilitiesTools.cmake module which is included and utilized by the main BasisTools.cmake module.

The BASIS utilities address the following aspects of the software implementation standard:

- *Command-line Parsing*
- *Calling Conventions*
- *Software Testing* (TODO)

6.4 Project Layout

A brief summary of the common project layout required by all projects that follow BASIS is given below. Project templates are supplied by the BASIS package to make it easy for projects to follow this *BASIS Project Directory Layout* and standard *Project Template*. How to create and use such template is explained in the *Using and Customizing Templates* guide. The *basisproject* command-line tool further automates and simplifies the creation of new projects based on a project template.

config/	Package configuration files.
data/	Data files required by the software.
doc/	Documentation source files.
example/	Example files for users to try out the software.
include/	Header files of the public API of libraries.
lib/	Module files for scripting languages.
modules/	Project *Modules* (i.e., subprojects).
src/	Source code files.
test/	Implementations of unit and regression tests.
AUTHORS (.txt|.md)	A list of the people who contributed to this sofware.
BasisProject.cmake	Calls basis_project() to set basic project information.
CMakeLists.txt	Root CMake configuration file.
COPYING (.txt|.md)	The copyright and license notices.
INSTALL (.txt|.md)	Build and installation instructions.
README (.txt|.md)	Basic summary and references to the documentation.

See also:

The *Project Template* for a complete list of required and other standard project files. The CMake BASIS Package itself also serves as an example of a project following this standard layout.

Note: Not all of the named subdirectories must exist in every project.

---

7 Support

7.1 Report Issue

Please report any issues with BASIS, including bug reports, feature requests, or support questions, on GitHub. Before opening a new issue, we recommend a look at the frequently asked questions below and a search of the already reported open issues.

7.2 Frequently Asked Questions

Standard CMake Commands

Can I still use standard CMake calls such as add_library, or is some BASIS functionality lost?

Probably. However, you will definitely lose much of the useful functionality that BASIS was created to provide. This kind of usage has also not been heavily tested so it is not recommended. The BASIS philosophy is definitely that a project that switches to BASIS uses the basis_* CMake commands wherever possible. Consider BASIS an extension to CMake, but if you run into issues you can file a ticket and we will attempt to address the problem.

CMake Package Configuration

Can I use the <Package>Config.cmake files of projects that do not use BASIS?

In <Package>Config.cmake files of other projects, it is fine that there will be standard CMake commands add include/library directories or import targets. BASIS is "smart" enough to extract this information properly by overriding the standard CMake commands.

Export of Build Targets

Do library targets have to be manually exported?

No. This is taken care of by the functions found in the internal ExportTools.cmake module, including executable targets which correspond to executable Python, Perl, BASH scripts, or executable binaries generated by the MATLAB Compiler.

Does the BASISConfig.cmake file define all of the exported library targets?

As typical for CMake, import statements for exported targets are written to "export files". These are included by the BASISUse.cmake file which should be included by other packages which use BASIS as follows:

```
find_package(BASIS REQUIRED)
include(${BASIS_USE_FILE})
```

This is done already by the basis_use_package() function which in turn is called by basis_find_packages() for all project dependencies right after the respective basis_find_package() call. In case of BASIS itself, the `basis_use_package(BASIS)` is called by the basis_project_begin() command which also calls basis_find_packages().

Thus, all you need to do is add a call to basis_project_begin() to the root CMakeLists.txt file of your project. See the root CMakeLists.txt of the default *project template* included with BASIS for an example.

Is there an easy way for users to get a list of the exported targets in a module?

No. This should probably be part of the documentation of each respective package/module. Generally with CMake, you would have a look at the exports file of a package that can usually be found right next to the CMake package configuration file (`<Package>Config.cmake`).

Look into the build directory of your BASIS build for an example. There you find the following files:

File name	Description
BASISConfig.cmake	Package configuration file which is included by CMake's `find_package` command.
BASISExports.cmake	Import statements for exported targets.
BASISCustomExports.cmake	Import statements for exported custom targets.
BASISUse.cmake	File to be included by users. Imports the exported targets.

Note: These file are generated by BASIS for every project that uses it, where `BASIS` is replaced by package name.

Note that these files contain the paths to the libraries and executables in the build tree. For each of these, BASIS configures also a second version which contains then the paths for the installation tree which are all relative to the location of the `<Package>Config.cmake` file and made absolute upon inclusion of these files.

The export files are generated by the internal CMake function basis_export_targets(). This function not only exports the custom targets of a project, but also calls CMake's export and install(EXPORT) commands for built-in targets, i.e., C/C++ executables and libraries. This happens upon project "finalization", i.e., basis_project_end(), which must be called at the end of each root CMakeLists.txt, including the CMakeLists.txt file in the top-level directory of each *project module*.

The exported target names are all the "fully qualified target UIDs" as used internally by BASIS to avoid target name conflicts between packages. The target name specified as argument to the `basis_add_*` target commands is prepended by the name of the package (i.e., top-level project name in case of modules, respectively, the `PACKAGE_NAME` specified in the `BasisProject.cmake` file) and separated by a dot (`.`). For example, the BASIS Utilities library of the CMake BASIS package has the exported target name `basis.utilities`.

When two modules belong to the same package, the package name prefix of the target names can be omitted when calling basis_target_link_libraries(), for example.

8 People

Software Development

- Andreas Schuh
- Andrew Hundt

Contributors

The following people notably helped to define and shape BASIS.

- Dominique Belhachemi
- Kayhan N. Batmanghelich
- Luke Bloy
- Yangming Ou

Former Advisors at SBIA

- Christos Davatzikos
- Kilian M. Pohl