# SWTools Manual

**Version 1.0**

**January 2011**

# SWTools 1.0 Manual

Prepared for the
Office of Science
U.S. Department of Energy

Mark R. Fahey

January 2011

Prepared by

OAK RIDGE NATIONAL LABORATORY
Oak Ridge, Tennessee 37831-6070
managed by
UT-BATTELLE, LLC
for the
U.S. DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

# Abstract

SWTools is python code combined with a directory hierarchy and rules to create an infrastructure for software management; or SoftWare Tools. SWTools was created to help manage third-party software installations at supercomputer centers. It was designed to keep the installations consistent and up-to-date while trying to avoid problems encountered with previous software repositories.

After having gained significant experience with previous filesystems where third party software installations were located, the SWTools creators had several distinct goals in mind when they approached the task of designing the NCCS new software management system SWTools. The main goal of the new design was to have a system that was strongly hierarchical, with rules for naming, installing, and documenting that could be enforced in automated way. In addition, the authors sought to automate as much of the software maintenance process as possible.

Foremost, the authors wished to be able to build, link, or test any installed application on any system at any time. One problem that all software installers and maintainers face is the problem of upgrades, whether those happen to be compiler upgrades, operating system upgrades or application upgrades. Any time one of those events occurs, a huge amount of work has to be done. At the very least, after the upgrade is installed, all applications that depend on it must be retested. In the case of OS upgrades, the maintainer may have to relink a significant number of the applications on the system in order to fix problems relating to new system libraries. Compiler upgrades are somewhat simpler in that they usually do not break any of the existing system applications, but nonetheless, any third party libraries that are provided on that system should be compiled with the new compiler, which can be viewed as a simple regression test of the compiler.

The authors also sought to automate the collection of as much reporting data as possible. In the new system, all provided documentation will be written by the application installers. This theoretically will keep the documentation as up to date as possible. Also, inventory and user documentation will be kept up to date dynamically. That is, the same documentation that the installer writes for local users of the system will be made available online, and all web based inventory information will be dynamically updated. Locally, a site can institute a workflow to include a review process [by the core software administration team] for the documentation if deemed necessary. The core software administration team will ultimately be responsible for making sure all applications and packages conform to the rules that have been set out, and that all information presented is high quality in nature.

Highlights of SWTools are:

- helps manage installations of third-party software
- provides testing infrastructure
- uses a structured hierarchy
- helps ensure consistency of installations across packages
- checks for conformance to established rules
- creates web pages

These benefits make managing third-party software more efficient. The consistency alone is a huge payoff for organizations that have many people contributing to software maintenance.

# Contents

# Abbreviations

| | |
|---|---|
| centos | Centos operation system |
| cnl | Compute Node Linux (Cray operating system on XT compute nodes) |
| gnu | GNU compiler |
| intel | Intel compiler |
| NCCS | National Center for Computational Science |
| NICS | National Institute for Computational Science |
| pathscale | Pathscale compiler |
| pgi | Portlang Group compiler |
| sles | SuSE Linux Enterprise Server |

# Acknowledgments

The SWTools project is a product of the National Center for Computational Sciences (NCCS) at Oak Ridge National Laboratory (ORNL). This work was designed and led by Mark R. Fahey (ORNL). SWTools was implemented by Nicholas Jones (former ORNL intern and now graduate student at Princeton). Nick also wrote the original draft of the documentation. SWTools has been used in production at the NCCS and the National Institute for Computational Science (NICS) supercomputing centers since 2008.

SWTools Contributors:

- Mark Fahey
- Nick Jones
- Tom Barron

# Chapter 1

# Introduction

## 1.1 What is SWTools

SWTools is python code combined with a directory hierarchy and rules to create an infrastructure for software management; or SoftWare Tools. SWTools was created to help manage third-party software installations at supercomputer centers[1]. It was designed to keep the installations consistent and up-to-date while trying to avoid problems encountered with previous software repositories.

SWTools is infrastructure code that works with user-supplied build and test scripts resulting in a mostly-automated software build and test framework. The framework tries to simplify as much as possible the work required to deal with upgrades, whether they be compiler, operating system, or application upgrades. Further, the code also automates the collection of as much reporting data as possible including user-facing web pages. Also, inventory and user documentation will be kept up to date dynamically; that is, the same documentation that the installer writes for local users of the system will be made available online, and all web based inventory information will be dynamically updated.

Highlights of SWTools are:

- helps manage installations of third-party software
- provides testing infrastructure
- uses a structured hierarchy
- helps ensure consistency of installations across packages
- checks for conformance to established rules
- creates web pages

These benefits make managing third-party software more efficient. The consistency alone is a huge payoff for organizations that have many people contributing to software maintenance.

SWTools assumes you will use the following directory hierarchy structure within which to install software

```
swbase/machine/app/ver/build
```

where

| | |
|---|---|
| `swbase` | root for the installations and/or swtools themselves |
| `machine` | either an architecture or machine name |
| `app` | application to be installed |
| `ver` | version of the application |
| `build` | particular build of the application/ver |

## 1.2 Goals

After having gained significant experience with previous filesystems where third party software installations were located, the SWTools creators had several distinct goals in mind when they approached the task of designing the NCCS new software management system SWTools. The main goal of the new design was to have a system that was strongly hierarchical, with rules for naming, installing, and documenting that could be enforced in automated way. In addition, the authors sought to automate as much of the software maintenance process as possible.

Foremost, the authors wished to be able to build, link, or test any installed application on any system at any time. One problem that all software installers and maintainers face is the problem of upgrades, whether those happen to be compiler upgrades, operating system upgrades or application upgrades. Any time one of those events occurs, a huge amount of work has to be done. At the very least, after the upgrade is installed, all applications that depend on it must be retested. In the case of OS upgrades, the maintainer may have to relink a significant number of the applications on the system in order to fix problems relating to new system libraries. Compiler upgrades are somewhat simpler in that they usually do not break any of the existing system applications, but nonetheless, any third party libraries that are provided on that system should be compiled with the new compiler, which can be viewed as a simple regression test of the compiler.

The authors also sought to automate the collection of as much reporting data as possible. In the new system, all provided documentation will be written by the application installers. This theoretically will keep the documentation as up to date as possible. Also, inventory and user documentation will be kept up to date dynamically. That is, the same documentation that the installer writes for local users of the system will be made available online, and all web based inventory information will be dynamically updated. Locally, a site can institute a workflow to include a review process [by the core software administration team] for the documentation if deemed necessary. The core software administration team will ultimately be responsible for making sure all applications and packages conform to the rules that have been set out, and that all information presented is high quality in nature.

## 1.3 Future SWTools Improvements

- Dependency analysis
- automatic updating of modulefiles
- genericizing the pbs scripts, pbs settings in config

# Chapter 2

# Installation

This section describes the installation process for SWTools. The process is not automated, but is fairly straightforward.

1. Download the latest version of SWTools from `http://www.nccs.gov/user-support/swtools`
2. Untar the swtools.tar.gz file into the desired destination. For example, the destination could be `/sw/tools`.
3. Assuming SWTools is installed in `/sw/tools`, put `/sw/tools/bin` in your path. This could be done by creating a swtools modulefile (example is included), editing it appropriately, adding `/sw/tools/modulefiles` to the `MODULEPATH`, and finally doing a `module load swtools`.
4. Decide on the filesystem and root prefix where you plan to install your applications. This could be `/sw` or anywhere else. (The tools themselves can coexist with the installations in the same prefix directory if desired.) From now on `swbase` refers to the base directory. It is recommend that `swbase` actually be a link to the real location, like `/sw -> /autofs/a1/sw`.
5. Edit `sw_config` - many configuration parameters need to be set in this file. See Section 2.1.
6. Edit `reportconform.py` to make it do what you want.
7. Edit `log.py` to make it do what you want.

## 2.1 sw_config

Most local settings for SWTools are contained in the `sw_config` file. This file resides in the tools/bin directory (after untarring the source.) An example `sw_config` file is included in the reference section. The intent is that the settings and comments are self-contained, but we include a brief discussion of the settings in Table 2.1.

Table 2.1: sw_config parameters

| | |
|---|---|
| `SOFTWARE_ROOT_DIRECTORY` | The base prefix for all software installations |
| `MACHINES` | A set of key-value pairs assigning machines names to a designation. Example: `jaguar:xt,kraken:xt, eugene:bgp, smoky:smoky` |
| `VALID_VERSIONS` | Possible version designations that can be used in the version file. Defaults are development, current, and deprecated. |
| `TEMPLATE_DIRECTORY` | Directory where templates reside. Example: `/sw/tools/templates` |
| `LOG_DIRECTORY` | Directory where log files are put. Example: `/sw/tools/log` |
| `ADDPACKAGE_TEMPLATES` | List of template files to be used when doing an addpackage. |
| `ADDBUILD_TEMPLATES` | List of template files to be used when doing an addbuild. |
| `BUILDDIR_PERMISSIONS` | Permission settings for build directory when doing an addbuild. |
| `APPDIR_PERMISSIONS` | Permission settings for application directory when doing an addpackage. |
| `DUPLICATE_ENVVAR` | The environment variable (currently `SW_MODFILE`) to use as a flag for swapping the programming environment between the ### markers during a duplicate operation. |
| `DUPLICATE_MODIFIED_FILES` | List of files to be modified when duplicated. |
| `DUPLICATE_MODIFY_START_FLAG` | Start marker for basic programming environment. |
| `DUPLICATE_MODIFY_END_FLAG` | End marker for basic programming environment. |
| `DEFAULT_MASK` | What is this for? Example: 0002. |
| `REBUILD_RELINK_RETEST_WORK_ENVVAR` | Name of the environment variable used for workspace. |
| `FROM_FIELD` | Email address that emails should come from when sent from reporting scripts. |
| `APPLICATION_LEVEL_EXCEPTIONS` | Directories at the application level that are exceptions (that is not application directories). |
| `VERSION_LEVEL_EXCEPTIONS` | Directories at the version level that are exceptions (that is not version directories). |
| `BULD_LEVEL_EXCEPTIONS` | Directories at the build level that are exceptions (that is not build directories). |
| `APPLICATION_REQUIRED_FILES` | List of files that are checked for by the conformance report at the application level. |
| `BUILD_REQUIRED_FILES` | List of files that are checked for by the conformance report at the build level. |
| `BUILD_EXIT3_FILES` | List of files to check for `exit 3` - a signal that the template has not been modified. Use is dependent on templates. |
| `WEBARCHS` | List of machine designations to create web pages for. |
| `WEBHOME` | Location to put web pages. Example: `/sw/tools/www`. |
| `COMPILERS` | List of compilers for each machine designation to list specifically on the software web pages. |
| `SHOW_OS_ON_WEB` | Show OS and compiler or just compiler in the names of builds on the software web pages. no or anything else for yes. |

# Chapter 3

# Using SWTools

In this chapter, you will find installation instructions for how you install a package in the `swbase` filesystem. SWTools assumes you will use the following directory hierarchy structure within which to install software

```
swbase/machine/app/ver/build
```

where

| | |
|---|---|
| `swbase` | root for the installations and/or swtools themselves |
| `machine` | either an architecture or machine name |
| `app` | application to be installed |
| `ver` | version of the application |
| `build` | particular build of the application/ver |

In all cases, it is highly suggested that only lowercase letters are used especially for application names. It is also highly suggested that a naming convention for the build directories be adhered to; a possible naming convention for builds is discussed later.

When first installing applications, one has to create this hierarchy and some tools are provided to help do this (discussed later.) Use of these tools helps keep the tree structure uniform and in conformance. However over time some of these tools (and some steps of the instructions below) may become less used.

The machine directories in which one will install software under `swbase` level is set in the tt sw_config file (which itself resides in the `swbase/tools/bin directory`.) You have to identify the appropriate machine directory in `swbase` to work in. An example list of machines are shown in Table 3.1.

In Table 3.1, notice that the machine designation can be the same as the machine name or it can be some arbitrary designation (like machine architecture or purpose.) Also note that multiple machines can resolve to the same machine designation.

Within each machine directory, there are application directories. It is desirable if there is consistency across machines in terms of what software is installed. However, if there are machines of wildly differing purposes using this infrastructure, then clearly the kinds of applications installed will not be consistent.

Table 3.1: Machine Designation Example

| Designation | Machine Name | OS (login,compute) |
|---|---|---|
| xt5 | kraken, jaguar | sles10.1, cnl2.2 |
| yona | yona | centos5.5 |
| analysis | nautilus | sles11.1 |

## 3.1 First Things First

Ensure default group is set: It is likely that all the installed software is supposed to be owned by the same linux group - lets say `install`. If so, it is best to first run "`newgrp install`" before setting up the rest of the SWTools environment.

Of course, this will probably only work if install is one of your groups. Note that you may have to re-initialize your environment (`source /etc/bashrc`) because newgrp "logs you in again".

Set up environment: This can be done a variety of ways. For now, the easiest method is to source an environment file that updates the `MODUELPATH` to expose the swtools modulefile. These environment files are optional, but might be set up as part of the installation. Assuming `swbase=/sw`, then the following commands would be used to set up the environment.

```
. /sw/tools/bin/environment.sh         (bourne shell syntax)
      OR
source /sw/tools/bin/environment.csh   (c shell syntax)
module load swtools
```

Or set the MODULEPATH manually.

```
export MODULEPATH=${MODULEPATH}:/sw/tools/modulefiles
module load swtools
```

While in setup or testing mode (before the application modulefiles have been made available by default), you might need to do add the machine modulefile directory to your path (to get a new python

```
export MODULEPATH=${MODULEPATH}:/sw/tools/modulefiles:/sw/<machine>/modulefiles
module load swtools
```

Note: The machine path name is the "machine designation" from Table 3.1. For example, if you are installing on a Cray XT5, it might look like

```
export MODULEPATH=${MODULEPATH}:/sw/tools/modulefiles:/sw/xt5/modulefiles
```

You might need to load python/2.5.2 or newer if it has not already been done for you!!!

*Note: Once the swtools modulefile is loaded, most of the tools that are used from now on will automatically detect which machine you are on and use the proper machine designation directory.*

## 3.2 Adding a package

Only need to read this section if you are adding a package to a machine tree for the first time. Otherwise skip down to "Section 3.3: Adding a version" or "Section 3.4: Adding a build".

Use `swaddpackage` to create the new package directory (it uses the machine directory based on what machine you are logged into, so you wont be able to use `swaddpackage` from a different machine unless they share the same machine tree). You can do it manually by copying an existing directory, but be sure to copy all the necessary text files over including the hidden dot files and set the permissions appropriately.

```
swaddpackage -a <app>
```

Please use all lowercase for the names of the packages (or all uppercase, just be consistent.) `swaddpackage` creates the following files and puts them in the new package directory: `description`, `support`, `versions`, `.exceptions`, and `.check4newver`. Please edit the first 4 appropriately and use `swversion` to update `.check4newver`.

## 3.3 Adding a version

When adding a new version, there isn't much to do other than create the version directory in the application directory and make sure the permissions on the new directory are 775 and of course owned by the proper group.

```
mkdir 1.6.7
chmod g+w 1.6.7
chown .install 1.6.7       # assuming install is proper group
```

Note that after installing the builds for this version, the `swversion` command should be run to update the `.chk4newver` file, but doing it now is premature. For example,

```
swversion installed -a paraview
```

Part of installing a new version of the application is to actually get the source code. For simplicity and consistency sake (and reducing proliferation of source tarballs), it is recommended to put all sources (tarballs) in `swbase/sources/app/ver` which can then be used for any machine.

### 3.3.1 Changing default version

If you are going to change the default version of the package after a new version is installed, there are several things to do (below), but these items should be done after all the builds for this new version have been completely installed (not before.)

- first, log worked done by running the `swlog` script; this script can be set to send the log message to a mailman list or a ticket system or whatever. The idea is that this action will both log the work and notify the appropriate people that a new installation has been completed.
- with the notification, one could for example then make sure the software installations appear in a weekly message to users
- when the timetable happens, somehow the owner needs to remember to update the default version (a ticket system or a project management package might be do this), this includes:

- update versions file
- update modules default setting

If at anytime a new version (or build) changes how one would use the package, make sure to update the `description` file. Make sure it explains how different versions would be used.

## 3.4 Adding a build

This assumes the application and version directories already exist, so if they don't, go back up to "Section 3.2: Adding a package" or "Section 3.3: Adding a version".

### 3.4.1 Step 1: New build directory

For this step, you either want to add a new build directory from scratch or you want to copy/duplicate an existing build directory. Go to the appropriate subsection below and follow the instructions.

Remember, there is a specific directory structure that should be followed. In short, when creating a build directory it should be of the form `os_compiler[_compileropts]` where

| keyword | description |
|---|---|
| OS | combination of OS name with OS version. We suggest using partial version numbering like cnl2.1 (rather than 2.1.31) or sl5.2 (scientific linux) or sles10.1. (include .) |
| compiler | Combination of compiler and full version. Acceptable compiler names are pgi, pathscale, gnu, intel, xlc, or xlf. Examples are pgi7.2.5, pathscale3.2, gnu4.2.1, intel11.1.038, xlc9.0xlf11.1(bgp) (include .) |
| compileropts | Additional (non-default) compilation options like par (for parallel), i8, r4, r8, |

The following table shows example naming conventions for OS.

| architecture | login OS | compute OS | description |
|---|---|---|---|
| xt5 | sles10.1 | cnl2.2 | sles when linking against sles system libraries and cnl when using Cray wrappers linking against compute node system libraries |
| analysis | sles11.1 | | check `/etc/*-release` |

#### Add build directory

If there is no existing build directory you want to duplicate, then add the build directory with `swaddbuild`. Otherwise skip down to "Duplicate existing build", which streamlines the installation of new a build based on an existing build.

An example of adding a build from scratch is:

```
> swaddbuild -a hdf5 -v 1.6.7 -b cnl2.2_pgi9.0.4
creating /sw/xt/hdf5/1.6.7/cnl2.2_pgi9.0.4/relink
creating /sw/xt/hdf5/1.6.7/cnl2.2_pgi9.0.4/rebuild
creating /sw/xt/hdf5/1.6.7/cnl2.2_pgi9.0.4/retest
creating /sw/xt/hdf5/1.6.7/cnl2.2_pgi9.0.4/status
creating /sw/xt/hdf5/1.6.7/cnl2.2_pgi9.0.4/build-notes
creating /sw/xt/hdf5/1.6.7/cnl2.2_pgi9.0.4/dependencies
creating /sw/xt/hdf5/1.6.7/cnl2.2_pgi9.0.4/.owners
  addbuild successful.
```

This step makes the build directory (in this case named `cnl2.2_pgi9.0.4`) and copies template files into the build directory. If the directory already exists, the command will fail. The templates provided are designed for fairly simple builds and tests. (Note that two sets of templates for rebuild, relink and retest templates are available in `/sw/tools/templates`. The second set are set up to deal with slightly more complicated scenarios and named `rebuild2.template`, `relink2.template`, and `retest2.template`.)

The build directory is where the particular build is designed to be installed in. Thus, you probably want to untar the tarball (or whatever the source comes as) into a subdir of the build directory. For example, hdf5 source would be in `swbase/machine/hdf5/1.6.7/cnl2.2_pgi9.0.4/hdf5-1.6.7` and the configure and make can then be done in-place. The configure and make dont have to be done in-place, but could be done in a separate directory - that is up to the installer. However, it is ideal if the installation prefix is set to be `swbase/machine/app/ver/build` so that the `bin, lib, man, etc` directories would be found in the build directory. It is highly suggested that the bin, include and lib directories are always installed into the build directory to ensure they are always at the same level for consistency.

### Duplicate existing build

The swduplicate tool can be use to duplicate an existing build. This assumes there is an existing build directory that you want to copy as a starting point. If there is not an existing build directory, go back up to Add build directory and skip this. This tool makes sure to set the permissions correctly even if the source directory is not quite right.

Note that this tool can only be used to copy a build of an existing app/ver to another build name. It cannot be used to copy builds from one version to another or from one app to another.

Example: Copy the existing cnl2.2_pgi9.0.4 build to the new build cnl2.2_gnu4.3.4 within the hdf5/1.6.7 directory.

```
swduplicate -a hdf5 -v 1.6.7 -b cnl2.2_pgi9.0.4 -d cnl2.2_gnu4.3.4
```

Since this duplicates an existing build (which should already conform to the standards but may not), the new directory should have all the necessary files to meet conformance. This will copy the source scripts as is - if you want to update the programming environment part of `remodule` automatically, then you need to set your programming environment module commands in a file and then set `SW_MODFILE` to point to that file. For example, suppose the file `env` has the contents

```
. ${MODULESHOME}/init/ksh
module unload PrgEnv-gnu
module unload PrgEnv-pathscale
module unload PrgENv-pgi
module load PrgEnv-gnu
module swap gnu gnu/4.3.4
```

and then we set SW_MODFILE to point to it

```
export SW_ENVFILE=~/env
```

Then `swduplicate` will automatically update the programming environment section of the `remodule` file. This is not particularly helpful for installing one build, but is intended for the scenario where someone wants to rebuild many packages with say a newer version of a compiler.

**remodule file**

A best practice of the swtools way of doing things is the `remodule` file (the name is immaterial.) As seen below, the SWTools infrastructure expects to find three separate scripts in a build directory: rebuild, relink, and retest (described later.) Instead of repeating module commands and environment variable settings in these files, all common commands and settings are done in the `remodule` file, like setting `SRCDIR` or `PACKAGE` variables and any dependent modules. If `remodule` is used (properly), any updates to a build or duplicates of a build only requires editing the `remodule` file, and the other three scripts (`rebuild`, `relink`, and `retest`) will ideally not need to be modified.

It is important that the `remodule` script (just like the `rebuild`, `relink` and `retest` scripts) contain the appropriate marker lines around the commands that set the basic environment.

Example remodule file:

```
  ### Set Environment (do not remove this line only change what is in between)
  . ${MODULESHOME}/init/ksh
  module unload PrgEnv-pgi
  module unload PrgEnv-pathscale
  module unload PrgEnv-gnu
  module unload PrgEnv-intel
  module unload PrgEnv-cray
  module load PrgEnv-pgi
  module swap pgi pgi/10.4.0
  ### End Environment (do not remove this line only change what is in between)

module load cmake/2.8.0
module load parmetis/3.1

PACKAGE=trilinos
SRCDIR=trilinos-10.0.2-Source
```

Notice in the example above that the module loads for cmake and parmetis are outside the `###` markers. That is because those are modules needed by a particular package. The lines between `###` should set the environment you want applied to each and every package (assuming you are doing multiple builds.) Once the `remodule` file is set, then the `rebuild`, `relink` and `retest` scripts should have this:

```
  ### Set Environment (do not remove this line only change what is in between)
  . ${MODULESHOME}/init/ksh
  . ${SW_BLDDIR}/remodule
  ### End Environment (do not remove this line only change what is in between)
```

### 3.4.2 Step 2: Build script

For the build, the installer needs to modify an existing (or create your own) `rebuild` script. This script takes the environment variables shown in Table 3.2 as input.

Table 3.2: Inputs to rebuild

| Env Vars (inputs) | Description |
|---|---|
| SW_BLDDIR | Location of the build directory of the form `/sw/<mach>/<ver>/<build>`. Any references to actual pathnames rather than the links have been removed if set up properly. Common use is in the configure step with `prefix=$SW_BLDDIR`. Always provided by the SWTools. |
| SW_ENVFILE | Optional: A file that sets all the entire programming environment including local environment variables. This is useful to use an alternate set of commands and variables to build a package rather than what is in `remodule`. |
| SW_MODFILE | Optional: A file with basic programming environment settings. This is only to be used in a special case (like `swduplicate`) where someone wants to override the basic module commands between the special markers you embed in the `remodule` file. |

With this input, the installer has to then modify/create the build script to `configure, make, make install` the application. The provided template or rebuild script should give you enough of an outline to follow the style. While editing the [template] script, there are some things to consider:

1. Remove "`exit 3`" line if it exists; the conformance checker looks for this to test if the script might still be the unchanged template script
2. Make sure **not** to remove the "`###`" lines that are markers to the sw* scripts. Even if you plan to use the default programming environment, please make sure to specifically load the desired compiler/version in the `remodule` script and that the `rebuild` script sources `remodule`.
3. The script should not assume the `rebuild` script starts in the build directory. It is suggested that the script use/keep the "`cd $SW_BLDDIR`" line.
4. The first step after setting up the environment should likely be "`make distclean`" or equivalent. The intent of the rebuild script is to rebuild everything from scratch.
5. Then do the `configure`, `make`, and `make install` commands and check the exit status from each one. If any of those commands fail, then the script should return 1 (failure). You have to figure out the `configure` options and how to check for success and failure. This is application dependent.
   (a) Note that if desired the configure can be run from the build directory using the sources residing in `swbase/sources/app/ver`.
   (b) It is suggested that the default configure options should be used for typical installations, with any additional flags to disable or enable options that are necessary to ensure a successful build.
6. If the script gets through configure, make and make install without error, return 0.

When doing the build, use

```
swbuild -s -a <app> -v <ver> -b <build>
```

to call your `rebuild` script - `swbuild` will ensure that all the proper permissions are set before it exits. *Note that* `swbuild`*,* `swlink` *and* `swtest` *assume you are working on the machine you are installing the application for.*

The only output from the script (other than installing the application) should be to return a 0 for success or 1 for failure.

### 3.4.3 Step 3: Link script

This step is to modify or create the `relink` script. This is often very similar to the `rebuild` script, except that it should not need to do the "`make distclean`" or redo the configure step. Its intent is to only rebuild binary executables (in-place), not to recompile everything. A "`make clean`" will work, but usually removes more files (executables and object files) than desired. Ideally, the goal is for installers to make a surgical strike at deleting binary executables, which is then followed by a "`make`"and "`make install`". Just like `rebuild`, it has the same "ins" and same conventions on return codes. (You might want to use `rebuild` as a starting point for `relink`.)

When testing it, use

```
swlink -s -a <app> -v <ver> -b <build>
```

to call your `relink` script.

Note that originally `swlink` was intended as an easy method to relink binaries after an OS change, but in practice this has basically never been used (other than as a debug tool during installation.) Future releases of SWTools may discontinue support for relink.

### 3.4.4 Step 4: Test script

The intent of the `retest` script is to run the "`make tests`" or "`make check`" that comes with a package. Often, there is no need to come up with your own tests, however occasionally there are exceptions. Typically, the expectation is that the installer will check the output of "`make check`" and report that all tests passed or not. It can be argued if the actual "make"ing of the test binaries should be done in this step or as part of `rebuild` and/or `relink`. A reason to not do the build of the binaries in the `retest` script would be so one could truly test old binaries against new OSes and new compilers. But often it is not easy to separate the build from the tests.

`retest` has the same "ins" as `rebuild` except a fourth environment variable is required: `SW_WORKDIR`, which points to a scratch directory where tests can be run if doing them locally in the build directory is not an option. As for "outs", 0 is success, 1 is failure, and 2 means a batch job was submitted. Note that some tests will have to be run in batch mode because of the machine setup, and for those cases it must return a different code, namely 2.

For `retest`, the following items should be considered:

1. This assumes the `make` and `make install` were already complete, and only plan to do a "`make check`".

2. Delete the "`exit 3`" line.

3. The hard part of retest is to come up with the logic to check the output from "`make check`" or similar and report if all tests pass or not. If all tests pass, then one should "`echo verified`" into the `status` file in the build directory; otherwise "`echo unverified`" in the `status` file. There is a large amount of leeway in how to determine if something passed or not.

4. And if you have to do a batch job, then your `retest` script should create a `.running` file which your batch job deletes when it finishes. And then your batch script must update the `status` file.
   (a) `swbase/tools/templates/retest2.template` and `swbase/tools/templates/test.pbs` are examples of how this works.

When testing, use

```
swtest -s -a <app> -v <ver> -b <build>
```

to call your `retest` script.

When `swtest` is done, there should be a `status` file with verified or unverified as its contents in the build directory.

### 3.4.5 Step 5: Documentation

If we now assume `swbuild` and `swtest` have passed for all the builds you plan to do for `swbase/machine/app/ver`, then one should return back to the app level directory and update the `description`, `support` and `versions` files.

The `description` file can be used on the external web pages, so make sure it is clear and up-to-date. It has a fairly simple structure and the intent is for the installers to follow this structure as close as possible. The web pages that are generated from this will turn out right if the installers do the following:

- put app name at top inside html header code
- use two-level categorization scheme (try to use existing fields rather than make up your own)
- keep the description and use sections
- one can add additional sections if desired

The versions file is used by the infrastructure software, so please keep it up-to-date as well. It is used by the sw* scripts to decide what is the default version if one does not provide the "`-v <ver>`" option.

The `swaddpackage` step will put "noweb" in the `.exceptions` file in the package directory. This is so the report generated for the web won't include this package until it is ready. When it is ready (which actually will become part of step 9 later), the `.exceptions` file should be deleted unless it needs to contain some other keyword (like vendor or controlled.)

### 3.4.6 Step 6: Checks

Assuming the above has been completed, then build directory permissions should be verified to ensure it is writable by the appropriate group, and readable by everyone. Then, the conformance checker `swreport` should be run on the build to make sure everything is ok.

```
swreport conform -m <mach> -a <app> -v <ver> -b <build>
```

Or run it on the application for all versions and builds:

```
swreport conform -m <mach> -a <app>
```

If everything passes, the app name should be followed by nothing else. If something did not pass, then the app name will be followed by messages about missing files or permissions set incorrectly.

### 3.4.7 Step 7: Modulefiles

Update (or create) the appropriate modulefile for the work completed above. In our experiences, we believe it is best if one modulefile works for all the different compiler builds of a given version of an application for a machine, but new modulefiles are added for each new version of the application. For example, one modulefile for pgi, pathscale, and gnu builds of hdf5/1.6.7. The modulefiles can be located in a path like `swbase/machine/modulefiles` or whatever is desired.

Upon completing the modulefile, please make sure the permissions are correct: readable by all and writable by the appropriate group. It is also necessary that the installer tests the modulefile by at least loading it, displaying it and viewing the help.

### 3.4.8 Step 8: Notification

Use `swlog` to log a message that will go to a mailist or a ticketing system (again whatever is set up in the installation phase) about what was done. It is suggested that a weekly digest be sent out to the appropriate team about the software installs that week.

### 3.4.9 Step 9: Update web pages

Use "`swreport html`" to [re]generate the web pages after one or more builds and/or versions have been installed.

## 3.5 Other things to know

In this section, we describe some other aspects of using SWTools to manage third party installations in practice.

### 3.5.1 Testing

The testing capabilities in SWTools can be used to do a variety of things:

- check a recent install - to potentially double check a new build someone else did
- check new compilers - regression testing for new compilers before they become defaults
- check OS upgrades - extra testing for the new OS

"`swtest -a <app> -v <ver> -b <build>`" can be used where one possibility for "app" is "all". In the case of "all", it is probably the case that you want all the default versions, thus no version argument is needed. The build option can be wildcarded - so something like "sl5.0_pgi*" or "cnl2.0*" or "*" are valid. (I don't think "*" is valid for build.)

When doing a large operation over many builds and applications, it is likely that a before and after report will be useful. For that, "`swreport text [-m <mach> [-a <app>]]`" is available - it generates a simple text report.

### 3.5.2 Using cron jobs to check for conformance on installs

It is fairly straightforward to set up a cron job that runs a conformance check (`swreport conform`) on the tree of installations. This is useful to make sure staff contributing to the installations are actually following the guidelines in a timely manner. It is not efficient for a center to stumble across partially finished installs or installs that dont conform at some time in the future when the knowledge of how it was installed may be lost or forgotten. The conformance check could be set up to email the individual owner instead and keep the responsibility on them rather than one person or a small team.

### 3.5.3 Building many applications with wildcards

There are two common ways the SWTools infrastructure will be used: (1) to install one application or a new version of an application or a new build of an application and (2) to rebuild/retest many applications (because a new compiler or new OS is available.) Much of the discourse above deals with the former - installing one application or build. For the latter, the abilities of SWTools are described and current best practices.

The `swbuild`, `swlink`, `swtest` and `swdriver` scripts allow for wildcards with some limitations. These are described in their documentation in the reference part of this document later. There is a tie in with the versions file that is manually updated. Whatever the versions file indicates is the "current" version, that is what the sw* scripts will use as a default when doing operations across many applications (with wildly different versioning.) Wildcards can be used for builds and applications as well - for example, if all SLES 11.1 builds are targeted, the one would use the option "`-s sles11.1*`" to the sw* scripts. The following example rebuilds and retests all applications on the current machine, picking the default version specified in the versions file, but only for builds that start with "sles"

```
swdriver build test -a all -b "sles*"
```

As another example, the following command would, for the default version of all apps, duplicate the pgi7.1.4 builds into a pgi7.2.3 build (updating the re* scripts with the module commands found in `SW_MODFILE`), and then rebuild and retest those new pgi7.2.3 builds.

```
export SW_MODFILE=<path to file with new module commands>
swdriver duplicate build test -a all -b "cnl2.1_pgi7.1.4*" -d "cnl2.1_pgi7.2.3"
```

If wildcard operations like this are performed, we highly suggest that the output from the "sw*" scripts be redirected to a log file that is inspected afterwards for failures.

After using SWTools in practice for nearly two years, we find that it actually works better to use a simple shell script that loops over the desired list read in from a text file (like all apps and their default current version) that then calls the desired sw* scripts (without using the -s option). This is opposed to using wildcards directly with the sw* scripts. If at any time the shell operation dies, it is real easy to restart from a new text file with the completed items deleted. There is an easy way with SWTools to generate a list of apps, versions, and builds - use "`swreport text`". If this output is saved in a text file, then it is very simple to edit the list for what you want (to keep or exclude) and to reorder (because of dependencies). A simple shell script is also included with SWTools that can read this simple text file and call the sw* scripts.

As was described in the "New build" section above, it is rarely the case that a rebuild operation is done in place (except while in the original installation phase.) Typically, like when doing a build with a new compiler, an existing build is "duplicated" or copied to a build name that includes the new compiler and then the build and test is performed in the new build directory. Thus, the SWTools includes a `swduplicate` script that not only does a copy, but can do the advanced operation of swapping out the basic programming environment set in the `remodule` file with a new programming environment provided in the file pointed to by `SW_MODFILE`. This is a very powerful component of SWTools and must be used with care, and requires that the `remodule` files follow the proper format.

### 3.5.4 Dependencies

When building third party packages, some are dependent on other third-party software. These dependencies should be resolved in the `remodule` file outside of the `###` markers. (Inside the `###` markers is for the basic programming environment.) However, the SWTools infrastructure does not make any attempt to update or fix dependencies on the other third party packages. For example, when doing a rebuild of HDF5 version 1.8.5 built with PGI 9.0.4 with a newer 10.4 version of PGI. The operation can be encapsulated with the command

```
swdriver duplicate build test -a hdf5 -v 1.8.5 -b sles11.1_pgi9.0.4 \
        -d sles11.1_pgi10.4
```

assuming `SW_MODFILE` points to a file that has the basic programming environment setup including pgi/10.4. This example has a potential problem if sles11.1_pgi9.0.4 build was performed including say szip/2.1/sles11.1_pgi9.0.4 and thus the remodule file has a module load statement of the form

```
module load szip/2.1
```

So when the hdf5 build is duplicated, the new build directory will have a remodule with the new basic environment, but it will still contain the module load statement for szip/2.1 which loads an older sles11.1_pgi9.0.4 build. So if hdf5 is rebuilt before the szip modulefile is updated, then the new hdf5 build sles11.1_pg10.4 will likely use the szip build with pgi 9.0.4 (or whatever the logic in the modulefile resovles to.)

As described in the previous section, it is highly suggested that the list of software to build in a large wildcard operation be first listed out in a text file, which is used as input. With this text file, someone can manually sort the packages in the proper order to resolve dependencies. However, as just described, this does not fix the modulefiles and thus the dependencies may be resolved with targets that arent desired.

### 3.5.5 Integrating SVN

Some pieces of the SWTools infrastructure could be integrated with SVN. The primary SWTools scripts are managed via SVN during development and it is suggested each center keep them under SVN as well. Other pieces of the tree could be integrated, but that is optional. All the files that are described at the end of this document in the reference section could be managed with SVN as well.

We do suggest however that not everything in the `swbase` hierarchy be put in SVN.

# Chapter 4

# Frequently Asked Questions

In this section, we deal with some common questions.

## 4.1 What if your default group does not match installation group?

`newgrp` can be used to change your group to the appropriate installation group before doing any of the work. If that is not done, then many of the files if not all that are worked on will be owned by the wrong group. A recursive `chown` can be used to fix the group permission problem after the installation is completed.

## 4.2 How do you install a precompiled binary package?

The instructions detailed in previous sections describe how to install a package from source. If however a precompiled binary is obtained, that can also be installed in a similar fashion. The build should be named something akin to `linux_binary` and the build directory should still have all the same files (`remodule`, `rebuild`, .....) that are created by `swaddbuild`. However, rebuild and relink for example should just do nothing and return 0. `remodule` will likely also be fairly empty. But `retest` should still attempt to test the installation for functionality - what it should be is package dependent.

## 4.3 What do you do about software provided by vendor [because you want a webpage for it]?

Some vendors like Cray, IBM and SGI provide their own math libraries and MPI libraries, for example. Centers would most likely want to advertise these libraries on their web pages just like any other third-party software installations. For this situation, you can use `swaddpackage` to create "placeholders" for these packages. In essence, you use `swaddpackage` as normal to create the application directory and then you make sure to edit the `.exceptions` file so that the vendor keyword is not commented out. (Comment out the keyword noweb when ready to make the web page.) Once this "exception" is in place the conformance checker will not look for version directories. The "`swreport html`" command will create web pages for these vendor applications.

## 4.4 What do you do about software that does not fit the directory hierarchy?

It is possible that an application may not fit the directory hierarchy of SWTools. Actually, there is really no reason that a package cannot be made to fit the directory structure because anything goes in the build directory level or lower; that is, the package can do whatever it wants in the build directory or in sub-directories of it. And then soft links can be made from the build directory where `bin`, `lib`, `include`, `etc`, `var`, ... reside to where they are actually installed.

Regardless, if a package is installed in a way that does not conform to the structure, then there are a couple hidden files that can be used to help the conformance checker and the web page generator to do the right thing. The two files `.notverdirs` and `.notbuilddirs` can be used in the app and ver directories, respectively, to list directories that should not be considered as a version or as a build.

## 4.5 What do you when the package does not have any tests that come with it?

Some packages do not come packaged with their own test suite. In these situations, there are some simple approaches that can be taken and of course more complicated approaches are also available. In the most simple of settings, the retest function can check if specific files exists after the installation process finishes. In the case of installing a binary application, this is insufficient since it does not check if it actually runs. For binary applications, it is suggested that the binary is at least run with a "`--help`" or "`--version`" option to make sure it actually runs. Some binaries have a demo or example mode that may be run in a batch mode, if that exists that is a good way to check a binary. If however, the package is a library with a variety of functions and subroutines and it does not come packaged with a test package, then this is a more complicated scenario and is up to the installer as to what to do.

# Chapter 5

# SWTools Reference

Here we include a list of the SWTools scripts and files each with a description.

## 5.1 License

## 5.2 Scripts

### 5.2.1 Python scripts

#### swbuild

```
> swbuild -h
Usage: swbuild -a APPLICATION -v VERSION -b BUILD
Example: swbuild -a hdf5 -v 1.6.5 -b cnl2.0_pgi6.2.5

Options:
  -h, --help            show this help message and exit
  -a APP, --app=APP     Application
  -v VERSION, --version=VERSION
                        Version
  -b BUILD, --build=BUILD
                        Build
  --debug
  -l, --log             Turn on logging. Logs are output to
                        /nics/e/sw/tools/logs
  -s, --stdout          Enable stdout from the build scripts
  -r RESUME, --resume=RESUME
```

### swlink

```
> swlink -h
Usage: swlink -a APPLICATION -v VERSION -b BUILD
Example: swlink -a hdf5 -v 1.6.5 -b cnl2.0_pgi6.2.5

Options:
  -h, --help            show this help message and exit
  -a APP, --app=APP     Application
  -v VERSION, --version=VERSION
                        Version
  -b BUILD, --build=BUILD
                        Build
  --debug
  -l, --log             Turn on logging. Logs are output to
                        /nics/e/sw/tools/logs
  -s, --stdout          Enable stdout from the build scripts
  -r RESUME, --resume=RESUME
```

### swtest

```
> swtest -h
Usage: swtest -a APPLICATION -v VERSION -b BUILD
Example: swtest -a hdf5 -v 1.6.5 -b cnl2.0_pgi6.2.5

Options:
  -h, --help            show this help message and exit
  -a APP, --app=APP     Application
  -v VERSION, --version=VERSION
                        Version
  -b BUILD, --build=BUILD
                        Build
  --debug
  -l, --log             Turn on logging. Logs are output to
                        /nics/e/sw/tools/logs
  -s, --stdout          Enable stdout from the build scripts
  -r RESUME, --resume=RESUME
```

### swdriver

```
> swdriver -h
Usage: swdriver action [options]
actions: duplicate, rebuild, retest, relink, report-conform, report-html


Options:
  -h, --help            show this help message and exit
  -a APP, --app=APP     application name
  -v VERSION, --version=VERSION
                        version number
  -b BUILD, --build=BUILD
                        build name
  -d DEST, --dest=DEST  destination directory
  -l, --log             Logging
  --debug               debugging
  -s, --stdout          enable stdout
  -m ARCHITECTURE, --architecture=ARCHITECTURE
                        Specify a machine
  -r RESUME, --resume=RESUME
  -n, --dryrun          show what would happen but don't do anything
```

### swaddpackage

```
> swaddpackage -h
Usage: swaddpackage -a APPLICATION
Example: swaddpackage -a hdf5

Options:
  -h, --help            show this help message and exit
  -a APP, --app=APP
  --debug
  -m ARCHITECTURE, --architecture=ARCHITECTURE
                        machine name
```

### swaddbuild

```
> swaddbuild -h
Usage: swaddbuild -a APPLICATION -v VERSION -b BUILD
Example: swaddbuild -a hdf5 -v 1.6.5 -b cnl2.0_pgi6.2.5

Options:
  -h, --help            show this help message and exit
  -a APP, --app=APP     application name
  -v VERSION, --version=VERSION
                        version number
  -b BUILD, --build=BUILD
                        build name
  --debug               debugging
  -m ARCHITECTURE, --architecture=ARCHITECTURE
                        machine name
```

### swduplicate

```
> swduplicate -h
Usage: swduplicate.py -a APPLICATION -v VERSION -b BUILD -d DESTINATION
```

```
Options:
  -h, --help            show this help message and exit
  -a APP, --app=APP     application name
  -v VERSION, --version=VERSION
                        version name (defaults to the current version if left
                        blank)
  -b BUILD, --build=BUILD
                        build name (wildcards are allowed)
  -d DEST, --dest=DEST  destination directory
  -f, --force           if duplication target exists, overwrite it
  --debug               debugging
  -l, --log
  -n, --dryrun
  -r RESUME, --resume=RESUME
```

### swreport

```
> swreport -h
Usage: swreport.py ACTION
actions: conform conformdist text html

Options:
  -h, --help            show this help message and exit
  -a APP, --app=APP     application name
  -d DIFF, --diff=DIFF  diff against previous report
  --debug               debugging
  -m ARCHITECTURE, --architecture=ARCHITECTURE
                        architecture name
  -b BUILD, --build=BUILD
                        build name
  -v VERSION, --version=VERSION
                        version name
  -r, --recurse         recurse into build directories to check permissions
                        when running conform
  -l, --long
  -i, --hidestatus      hide verified/unverified status online
  -e, --hidesupported   hide the support status in generated html pages
```

### swlog

```
> swlog -h
Usage: swlog [-d]
Example: swlog [-d]

Options:
  -h, --help    show this help message and exit
  -d, --dialog  use dialog (not editor) format
```

### 5.2.2 Auxiliary shell scripts

**chdelim**

```
# This will traverse the /sw tree finding all build directories of the form
#    <os>+<comp>[+<options>]
# and rename them to
#    <os>_<comp>[_<options>].
#
# Originally, the delimiter was "+" but the SLES man command does find
# man pages found in paths that have "+".  So the "+" delimiter was changed
# to "_"
```

**listallapps**

```
# This will traverse the /sw tree finding all build directories of the form
#    <os>+<comp>[+<options>]
# and print out a list for each machine (for use as input to swbuild and swtest)

# Could be useful if you need a list of stuff to do that is not easily done with
#  in the sw scripts.
```

## 5.3 Files

### 5.3.1 Configuration

**sw_config - example**

```
# SW TOOLS CONFIG FILE
# Version 1.0

# NO TRAILING SLASHES on the directories

# General Settings

#SOFTWARE_ROOT_DIRECTORY=/nics/e/sw
SOFTWARE_ROOT_DIRECTORY=/sw

# machines:designations
MACHINES=krakenpf:xt,kraken-pwd3:xt,login:xt,nautilus:analysis

# Valid tags for use with the versions files
VALID_VERSIONS=current,development,deprecated
TEMPLATE_DIRECTORY=/nics/e/sw/tools/templates
LOG_DIRECTORY=/nics/e/sw/tools/logs

# Addbuild / Addpackage settings

# The templates that will be copied along with the corresponding octal permissions string
```

```
ADDPACKAGE_TEMPLATES=support:0664,versions:0664,.exceptions:0644
ADDBUILD_TEMPLATES=remodule:0664,relink:0774,rebuild:0774,retest:0774,status:0664,build-notes:0
BUILDDIR_PERMISSIONS=775
APPDIR_PERMISSIONS=775

# Duplicate Settings

# The name of the environment variable that
DUPLICATE_ENVVAR=SW_MODFILE
# Files to be modified when a duplicate is performed and SW_MODFILE is defined
DUPLICATE_MODIFIED_FILES=remodule

# Flags to signify what text should be replaced when using SW_MODFILE
DUPLICATE_MODIFY_START_FLAG=### Set Environment
DUPLICATE_MODIFY_END_FLAG=### End Environment

# Rebuild / Relink / Retest Settings
DEFAULT_UMASK=0002
REBUILD_RELINK_RETEST_WORK_ENVVAR=SW_WORKDIR

# Version Settings

# Who should the emails come from?
FROM_FIELD = swadm@nics.utk.edu

# Report Settings

# We do not use architecture level exceptions
# This is because report will only do run on architectures
# defined in MACHINES usinng the hostname:architecture forma

# Directories to ignore at the application level
APPLICATION_LEVEL_EXCEPTIONS=modulefiles,bin,etc,share,include,man,info,lib,acct
VERSION_LEVEL_EXCEPTIONS=
BUILD_LEVEL_EXCEPTIONS=

# Files to look for when running report conform
APPLICATION_REQUIRED_FILES=support,description,.check4newver,versions
BUILD_REQUIRED_FILES=relink,rebuild,retest,status,dependencies,.owners

# Files to check for "exit 3" in
BUILD_EXIT3_FILES=rebuild,relink,retest

# Architectures to appear online
WEBARCHS=xt,analysis
WEBHOME=/nics/e/sw/tools/www

# The list of compilers for each machine for the website
```

```
COMPILERS=xt:pgi,intel,pathscale,gnu;analysis:intel,pgi,gnu

# Show os_comp or just comp on webpage for each app
SHOW_OS_ON_WEB=yes
```

### 5.3.2 Application level files

#### description

The description file is used by the `swreport html` tool to generate the base web page for each application. Thus it is important to write good text for a description and usage. Also, the application name needs to go on the first line between the html markers. The installer also puts a category and a subcategory on the Category line - these should be somewhat consistent across packages so the "category" view is useful.

```
<h1>Put Application Here</h1>

<p> Category: categoryhere-subcategoryhere

<h2>Description</h2>

<p> Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text

<h2>Use</h2>

<p> Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text

<pre>
code
code
code
code
code
</pre>
```

#### versions

The versions file is used by the infrastructure to know what the current, development and deprecated versions of the application are, where these demarcations are actually specified in the `sw_config` file. When the SWTools infrastructure tools like `swbuild` or `swtest` are used to do multiple builds or tests across many applications, these tools need some way (obtained from the `versions` file) to determine what versions of the application to build or test.

```
> more versions
```

```
development:
current:
deprecated:
```

### support

This file is intended to be used as a way to indicate the application is officially supported by the center or by the vendor or both, or if the application is provided in an unsupported manner. The template version of this file contains all the keywords shown below, and the installer picks one of them.

```
> more support
site or vendor or site+vendor or unsupported   # pick one
```

### .chk4newver

This file is updated when the `swversion` command is run. The following is an example of what this log file will look like over time.

```
> more .chk4newver
2009-01-27: new version of /sw/xt5/hdf5 installed
2009-01-27: next check on 2009-06-25
2009-06-10: new version of /nics/e/sw/xt-cle2.2/hdf5 installed
2009-06-10: next check on 2009-12-08
2009-07-22: new version of /sw/xt/hdf5 installed
2009-07-22: next check on 2009-12-20
2009-09-17: no new version of /sw/xt-cle2.2/hdf5 available
2009-09-17: next check on 2009-12-16
2010-01-08: new version of /sw/xt/hdf5 installed
2010-01-08: next check on 2010-04-08
```

### .exceptions

This file is created by default by `swaddpackage`. And by default, the “noweb” keyword is not commented out so that a web page is not created for the application until the installer is ready. This can also be used to tell the SWTools infrastructure that the app is a placeholder for some vendor provided software and as such do not check for version and build directories.

```
> more .exceptions
noweb              # do not create web page
#protected 0022
#vendor            # do not check for version directories
#module            # do not check for a corresponding modulefile
```

### .notverdirs

This is an optional file that is only needed in rare cases as explained in Section 4.4. It typically contains a list of directories you want the conformance checker to skip because they are not “version” directories. For example,

```
> more .notverdirs
bin
data
current
```

### 5.3.3 Version level files

#### .exceptions

At the version level, the `.exceptions` file is used primarily to indicate that the version directory is just a placeholder and there are no build directories inside the version directory. This can be used to list versions of an application on the web page without having to have any build directories (for example if you want to show versions of vendor provided software.)

```
> more .exceptions
vendor
```

#### .notbuilddirs

This is an optional file that is only needed in rare cases as explained in Section 4.4. It typically contains a list of directories you want the conformance checker to skip because they are not “build” directories. For example,

```
> more .notbuilddirs
bin
data
current
```

### 5.3.4 Build level files

#### remodule

```
> more remodule
  ### Set Environment (do not remove this line only change what is in between)
  . ${MODULESHOME}/init/ksh
  module unload PrgEnv-pgi
  module unload PrgEnv-pathscale
  module unload PrgEnv-gnu
  module load PrgEnv=pgi
  module swap pgi pgi/10.4.0
  ### End Environment (do not remove this line only change what is in between)

# load any other dependent modules here
# module load hdf5/1.6.7

# env vars here
  PACKAGE=package_name
  SRCDIR=app-version

# some of these may need to go up in the special section between ###
```

```
#export CC=cc
#export CXX="CC -DMPICH_IGNORE_CXX_SEEK"
export CC=pgcc
export CXX=pgCC
#export F77=ftn
#export F90=ftn
#export F9C=ftn
export F77=pgf90
export F90=pgf90
export FC=pgf90
#export CPPFLAGS=-DpgiFortran
```

**rebuild**

```
#!/bin/ksh

############################## standard interface to /sw tools
# Input:
#   Environment variables
#     SW_BLDDIR    current directory (PWD) minus /autofs/na1_ stuff
#     SW_ENVFILE   file to be sourced which has alternate prog environment
#                     only to be used in special circumstances
#     SW_WORKDIR   unique work dir that local script can use
# Output
#   Return code of 0=success or 1=failure
##############################

# exit 3 is a signal to the sw infrastructure that this template has not
# been updated; please delete it when ready
exit 3

if [ -z $SW_BLDDIR ]; then
  echo "Error: SW_BLDDIR not set!"
  exit 1
else
  cd $SW_BLDDIR
fi

if [ -z $SW_ENVFILE ]; then
  ### Set Environment (do not remove this line only change what is in between)
  . ${MODULESHOME}/init/ksh
  . ${SW_BLDDIR}/remodule
  ### End Environment (do not remove this line only change what is in between)
else
  . $SW_ENVFILE
fi

############################## app specific section
```

```
#

# clear out old installation to prevent potential libtool chmod
# commands from failing when reinstalled by another person
rm -rf bin lib include doc share man etc libexec info

#clear out status file since re-making
rm -f status

cd $SRCDIR

make distclean

./configure --prefix=$SW_BLDDIR \
--disable-shared
#--disable-fortran-compiler-check
#--host=x86_64-unknown-linux-gnu
if [ $? -ne 0 ] ; then
  echo "$PACKAGE configure failed"
  exit 1
fi

make all
  if [ $? -ne 0 ] ; then
    echo "$PACKAGE make failed"
    exit 1
  fi

make install
if [ $? -ne 0 ] ; then
  echo "$PACKAGE install failed"
  exit 1
fi

cd ../

############################### if this far, return 0
exit 0
```

### relink

```
#!/bin/ksh

############################## standard interface to /sw tools
# Input:
#   Environment variables
#     SW_BLDDIR    current directory (PWD) minus /autofs/na1_ stuff
#     SW_ENVFILE   file to be sourced which has alternate prog environment
#                     only to be used in special circumstances
#     SW_WORKDIR   unique work dir that local script can use
# Output
#   Return code of 0=success or 1=failure
##############################

# exit 3 is a signal to the sw infrastructure that this template has not
# been updated; please delete it when ready
exit 3

if [ -z $SW_BLDDIR ]; then
  echo "Error: SW_BLDDIR not set!"
  exit 1
else
  cd $SW_BLDDIR
fi

if [ -z $SW_ENVFILE ]; then
  ### Set Environment (do not remove this line only change what is in between)
  . ${MODULESHOME}/init/ksh
  . ${SW_BLDDIR}/remodule
  ### End Environment (do not remove this line only change what is in between)
else
  . $SW_ENVFILE
fi

############################## app specific section
#

# clear out old installation to prevent potential libtool chmod
# commands from failing when reinstalled by another person
rm -rf bin lib include doc share man etc libexec info

#clear out status file since re-making
rm -f status

cd $SRCDIR

#probably overkill, need surgical strike on binary executables
make clean

make all
  if [ $? -ne 0 ] ; then
    echo "$PACKAGE make failed"
    exit 1
  fi
```

```
make install
if [ $? -ne 0 ] ; then
  echo "$PACKAGE install failed"
  exit 1
fi

cd ../

############################### if this far, return 0
exit 0
```

**retest**

```
#!/bin/ksh

############################### standard interface to /sw tools
# Input:
#   Environment variables
#     SW_BLDDIR    current directory (PWD) minus /autofs/na1_ stuff
#     SW_ENVFILE   file to be sourced which has alternate prog environment
#                     only to be used in special circumstances
#     SW_WORKDIR   work dir that local script can use
# Output:
#   Return code of 0=success or 1=failure   or 2=job submitted
#
# Notes:
#   If this script is called from swtest, then swtest requires
#   SW_WORKDIR to be set.  Then swtest adds a unique path to what
#   user gave swtest (action+timestamp+build) and provides this
#   script with a uniquely valued SW_WORKDIR.  swtest will
#   automatically remove this unique workspace when retest is done.
###################################################################

# exit 3 is a signal to the sw infrastructure that this template has not
# been updated; please delete it when ready
exit 3

if [ -z $SW_BLDDIR ]; then
  echo "Error: SW_BLDDIR not set!"
  exit 1
else
  cd $SW_BLDDIR
fi

if [ -z $SW_ENVFILE ]; then
  ### Set Environment (do not remove this line only change what is in between)
  . ${MODULESHOME}/init/ksh
  . ${SW_BLDDIR}/remodule
  ### End Environment (do not remove this line only change what is in between)
else
  . $SW_ENVFILE
fi

############################### app specific section
#

#clear out status file since re-testing
rm -f status

cd $SRCDIR

make test > $SW_BLDDIR/test.log 2>&1
  if [ $? -ne 0 ] ; then
    echo "$PACKAGE make test failed "
    exit 1
  fi

testspassed=‘grep -G "All [0123456789]" ../test.log | awk ’{total += $2} END {print total}’‘
if [[ $testspassed -ne 49 ]]; then
  # error
  echo $testspassed tests passed!
  echo unverified > $SW_BLDDIR/status
  exit 1
else
  echo $testspassed tests passed!
  echo verified > $SW_BLDDIR/status
  exit 0
fi

cd ../

################################ if this far, return 0
exit 0
```

### .owners

This file contains the username (or usernames) of the installers of the particular build.

### status

This file is updated by running `swtest` (which calls `retest` to determine whether the build has passed or not its tests and the file must contain either `verified` or `unverified`.

**build-notes**

This file contains any special notes about how to install the package for the particular compiler; that means any changes to source code should be documented here as well as any changes to the configure or make scripts. (This could be viewed as a README file.)

# Bibliography

[1] Nick Jones and Mark R. Fahey, "Design, Implementation, and Experiences of Third-Party Software Administration at the ORNL NCCS," *Proceedings of the 50$^{th}$ Cray User Group (CUG08)*, Helsinki, Finland, May 2008.