SConsTools

Unknown macro: 'html'

Unknown macro: 'html'

Introduction

Below is some more detailed/advanced information about the offline build system. The offline build system is based on SCons. It is implemented as a set of scripts that import functionality from `SConsTools` (software developed internally). SConsTools introduces a package and release structure to support the build process. The top level directory is for the release and contains the `SConstruct` file. Each package is in a sub-directory to the release. Packages have a `SConscript` file. The structure of a release directory is explained in this page about Packages and Releases. SConsTools has a number of features including:

- Targets are automatically identified and built based on source files in the package directories
- Libraries that targets need from other packages are automatically added during the linking of the targets
- A release wide include directory is set up. Package headers are prefixed by package names to avoid file name collisions
- A release wide location for applications and libraries is created
- Release wide directories for `data` and `web` are set up with links to package `data` and `web` files.
- Circular dependencies between packages are identified. Errors are raised to stop the build in these cases.

To build a release, run scons from the release directory. The top level SConscruct file is responsible for building and installing software in each package directory. As `scons` descends into the package sub-directories, it identifies all targets based on source code in the package.

There are a number of targets that will be created based on what is in the package. For example, suppose one has the following files split between two packages in a release:

```
release/
release/packageA
release/packageA/src/A1.cpp
release/packageA/src/A2.cpp
release/packageA/src/pythonA.py
release/packageA/include/A.h
release/packageA/app/programA1.cpp
release/packageA/app/programA2.cpp
release/packageA/app/appA
release/packageB/src/B1.cpp
release/packageB/src/B2.cpp
release/packageB/include/B.h
release/packageB/src/pythonB1.py
release/packageB/src/pythonB2.py
```

The targets built will be:

- library for packageA built from A1.cpp, A2.cpp
- library for packageB built from B1.cpp, B2.cpp
- executable programA1 built from programA1.cpp
- executable programA2 built from programA2.cpp
- executable script appA built from appA
- a python module for packageA
- a python module for packageB

From the release directory, one can now run programA1, programA2 or appA. Rules for source code in a package app directory are as follows:

- each c++ (or c) file is compiled into its own application.

- each file with no extension is installed as a script.

Scripts need to start with the appropriate #! line for the language used. For python, you should start you script with

```
#!@PYTHON@
```

to use the same version of python that the release uses.

When C and C++ code includes a header file, it needs to be qualified by the package name. To use functions in the packageA library, files in packageB (as well as packageA itself) would do

```
#include "packageA/fileA.h"
```

The way to access python modules is simply

```
import packageA.pythonA
import packageB.pythonB1
import packageB.pythonB2
```

For example, from the release directory, you could bring up an interactive python shell and do the above to access functions from the two packages.

An important point is that your environment is built around you being in the release directory. If you run python from the packageA subdirectory, it will not see the packageA python module.

Details

Below we give more detail about the build system as it is currently implemented.

After you run `scons` on the above files, you will see three new directories:

```
release/include
release/build
release/arch
```

You may also see

```
release/data
release/web
```

depending on the packages being built. These directories belong to the build system - you should never put anything in these directories. They will be rebuilt each time `scons` runs if need be.
In `release/include` you will find two links to the package include directories:

```
release/include/packageA  --> release/packageA/include
release/include/packageB  --> release/packageB/include
```

This is the mechanism by which `#include "packageA/fileA.h"` works. The only thing in `release/include` will be soft links to package include directories.

In the `arch` directory, you will find all the executables, scripts, libraries (built as shared object libraries) and python modules for each package.

The `build` directory includes all intermediate files. Note - no compiled code is put in the package directories. It all goes into the `release/build` directory.

Package `data` and `web` directories

The same mechanism used to share package header files is also done for package `data` and `web` directories. That is, if one had the files:

```
release/packageA/data/data_fileA
release/packageB/web/introB.html
```

after running `scons`, the following directories would be created:

```
release/data
release/web
```

and within those directories, the following softlinks

```
release/data/packageA -> release/packageA/data
release/web/packageB  -> release/packageB/web
```

As with the directory `release/include`, do not create or put anything in `release/data` or `release/web` as they are cleaned out and recreated during an `scons` build.

As with `release/include`, `release/data` and `release/web` are created before any targets are built.

Useful Switches and Build Options

The top level `SConstruct` file implements several features that may be useful. Running

```
scons -h
```

from the release directory displays all the available options. A few notable things one might do with the switches:

Stop Build After First Error

When getting your code working it can be convenient to stop compiling after the first error - less it scroll away after all the subsequent errors. One can do

```
scons CCFLAGS=-Wfatal-errors
```

Check You Python Code

A useful tool for checking python code is `pylint`. `scons` will run `pylint` and report any errors it finds on your python code if you do

```
scons pylint
```

from the release directory.

External Libraries and Scons Build Customization

Using a Library Installed on the System

Most packages require no additional options beyond those in the default configuration. If a package requires additional build options, these can often be added by calling the `standardSConscript()` function in the SConscript file in the package directory. For instance, suppose a psana - Original Documentation user is developing a module in a package called `MyPackage` which needs to use functions from the Gnu Scientific Library. They would add the following line:

```
  standardSConscript(LIBS='gsl gslcblas')
```

To the file `MyPackage/SConscript`. Here is a complete list of options you can set with `standardSConscipt()`:

```
    LIBS - list of additional libraries needed by this package
    LIBPATH - list of directories for additional libraries
    BINS - dictionary of executables and their corresponding source files
    TESTS - dictionary of test applications and their corresponding source files
    SCRIPTS - list of scripts in app/ directory
    UTESTS - names of the unit tests to run, if not given then all tests are unit tests
    PYEXTMOD - name of the Python extension module, package name used by default
    CCFLAGS - additional flags passed to C/C++ compilers
    NEED_QT - set to True to enable Qt support
```

Note: when updating the `SConscript` file in your package directory, these options will apply to all targets built in your package. One should not modify the `SConscruct` file in the release directory.

Using a Library by Modifying SConscript

Suppose you have the files:

```
mycode/myheader.h
mycode/mysource.cpp
mycode/libmylib.so
```

That is you have a dynamic library built out of some source code. If you want to call functions in mylib from packageA, you could modify packageA /SConscript to have

```
standardSConscript(CCFLAGS="-I/reg/neh/home/username/mycode", LIBPATH='/reg/neh/home/username/mycode',
LIBS='mylib')
```

Then targets in packageA could do

```
#include "myheader.h"
```

To find the header file (given the use of CCFLAGS above) and the library mylib will be added to the link line.

While targets in packageA that use mylib can now be built correctly, to run them you will need to make sure that the operating system can find the shared object library mylib. This is done by setting the environment variable LD_LIBRARY_PATH to include the directory where mylib is located. This could be done in the shell initialization file (such as .bashrc, or .cshrc, depending on which shell you use).

Using a Library through the External Package Mechanism

What if you want to make an external library available to all the packages in your release? It would be tedious to modify each new package you develop as in the section above. A better solution is to create an external package that interfaces to the library. In your release directory, do the following:

newpkg MyCode

edit the file

~/release/MyCode/SConscript

This SConscript calls the funciton standardSConscript(), but this is not what we want. We want to use standardExternalPackage(). Change the file to look like:

sconstools_standard_external_package

```
# Do not delete following line, it must be present in
# SConscript file for any SIT project
Import('*')

import os
from SConsTools.standardExternalPackage import standardExternalPackage
#
# For the standard external packages which contain includes, libraries,
# and applications it is usually sufficient to call standardExternalPackage()
# giving some or all parameters.
#
PREFIX = os.path.expanduser('~username/mycode')
INCDIR = "include"
LIBDIR = "lib"
PKGLIBS = "mylib"
standardExternalPackage('MyCode', **locals())
```

This is the only file that need by in the MyCode package. After doing scons, any package in the release will be able to call functions in mycode by doing

```
#include "MyCode/myheader.h"
```

The build system will identify the dependency on the MyCode package. Given the PKGLIBS parameter, it will then add mylib to the link line. libmylib.so will be found because PREFIX and LIBDIR together give the directory where the MyCode shared object libraries are. This directory is not added to the LD_LOAD_LIBRARY path or explicitly used when building. By default, standardExternalPackage() will make soft links from the $SIT_ARCH/lib directory of your release to all the shared object libraries in the external package lib directory. When you ran sit_setup from your release directory, it placed the $SIT_ARCH/lib directory of your release first in the $LD_LIBRARY path. This is the mechanism by which MyCode libraries are made available to all packages in your release.

The above SConscript file for an external package is the simplest case. MyCode has a common directory structure: a base directory with an include and lib directory as sub-directories. Thus the PREFIX, INCDIR and LIBDIR parameters work well to describe it to SConsTools. If the include and lib directory were in different places, one could omit the PREFIX parameter and give absolute paths for INCDIR and LIBDIR:

```
INCDIR = os.path.expanduser('~username/mycode/include')
LIBDIR = os.path.expanduser('~username/mycode/lib')
```

Many external packages are more than just a library. They have stand alone programs or tools to run. You can add these using the BINDIR parameter. If an external package like MyCode were to in turn depend on another external package, that package could be defined in the same SConScript file and the DEPS parameter could be used when defining MyCode with standardExternalPackage. An example of using these options can be seen in SConscript file for the hdf5 package. One can do

```
addpkg hdf5
```

Within your release and take a look at hdf5/SConscript.

Building Part of the Release

`SConsTools` is designed to build a release as a whole. The system takes care to check the dependencies of all the packages and keep them all in sync with one another. This can prove inconvenient when a user is developing a module in packageA and is forced to fix compiler errors in packageB. There are several ways to address this:

- Move packageB outside the release directory.
  - This is the recommended method. Any unexpected dependencies between packageB and packageA will be brought to your attention through compiler or linker errors.
- Rename the `SConscript` file in packageB to something like `SConscript.skip`.
  - Now unexpected dependencies between packageB and packageA may manifest themselves as difficult runtime problems. For example, packageB may have a previously built binary that links to the library in packageA. After building packageA, it gets out of sync.
- Run `scons --ignore-errors`
  - This will build everything it can, but in the noise of compiler errors for packageB - you may not notice problems with packageA.
- Explicitly list the targets in packageA for `scons`to build.
  - `scons` provides a facility to specify individual targets but it is awkward to use in our build environment. One must specify the fully qualified target name (such as `arch/x86_64-rhel5-gcc41-opt/lib/libpackageA.so`)

All but the first option suffer from the risk of unexpected dependencies that lead to difficult runtime problems. In practice, there may be no risk of dependencies. PackageA and PackageB may simply represent independent experiments with no dependencies rather than packages in a release system. The `SConsTools` based build system will be more awkward then need be in these cases. However it is easy for dependencies to creep in during development, and some of these experiments will evolve into more complicated reusable software that will benefit from the package/release structure of the `SConsTools` based build system.

Upgrading the Release

To upgrade the version of a release, see Common development tasks.

Presentations

Below are slides from a talk given during the 2014 LCSL users meeting that covers the release/package structure of SConsTools, as well as how psana modules: Software_Development_Environment.pdf