SCons

News

- **03/18/2008** - The new RM is now activated and commpiles ST LATEST builds after they have been triggered via the old RM. The new RM pages can be viewed from inside SLAC firewall by visiting: http://glast-tomcat03.slac.stanford.edu:8080/releasemanager/releaseManager.jsp
- **02/04/2008** - The old RM now automatically tags ScienceTools LATEST builds with SCons tags. Shortly after a CMT style RM compile starts one can check out ScienceTools for SCons now. Tags are converted as follows: A LATEST1.2222 build in the old RM would correspond to a tag of ScienceTools-LATEST-1-2222 in SCons.
- **01/31/2008** - The first ScienceTools tag for SCons has been created. The tag is called ScienceTools-LATEST-1-2220 and corresponds to LATEST1.2220 of CMT. It can be checked out with the command cvs co -r ScienceTools-LATEST-1-2220 ScienceTools-scons.

Introduction

"SCons is an Open Source software construction tool---that is, a next-generation build tool. Think of SCons as an improved, cross-platform substitute for the classic Make utility with integrated functionality similar to autoconf/automake and compiler caches such as ccache. In short, SCons is an easier, more reliable and faster way to build software." – http://scons.org/

This page is intended as a tutorial on getting people up to speed about the way SConscript files should be created. SConscript files are the equivalent of our requirements files currently in use. They define the targets that SCons will create during the build process.

Status

Currently SCons is fully functional with ScienceTools. The new RM is compiling packages Science Tools with SCons as the builds happen with CMT. These builds are performed for Linux in 32bit and 64bit format as well as in Windows 32bit format. The tasks that remain for the Science Tools side are:

Task	Status	Time Estimate	Comments
Fix of last few unit test failures. See Here	No progress	Unknown	Package owners need to take active role in this
New tag collector to use with SCons.	Work in Progress. 50% Done.	2-3 Weeks	
New MrStudio replacement for SCons.	No Progress	Several weeks to a Month	This will heavily rely on the tag collector
Port to new OSes	Some Progress but on hold for now	Unknown	

On the GlastRelease side there is no progress yet. ScienceTools design can be used under GlastRelease and will accomplish 90% of GlastRelease's needs. The other 10% of the tasks that remain are:

Task	Status	Time Estimate	Comments
Create special SCons builder for gaudi	No Progress	1-2 Weeks	
Create SConscript files for all packages	No Progress	2-3 Weeks	Significantly faster if package owners help

Modify new RM to build GlastRelease	No Progress	1-2 Weeks	

Tagging convention

With SCons we will be switching tagging conventions away from the current `vXrYpZ` standard to the new `packageName-XX-YY-ZZ` standard. What this means is that any package performing a tag will have to have their package name in the tag as well as use only 2 digit version numbering. For example if in the old convention the Likelihood package would apply the tag `v1r2p3`, in the new convention this tag would be `Likelihood-01-02-03`. This convention is currently automatically enforced by the old RM. Whenever a package is tagged with the `vXrYpZ` format, the old RM automatically tags the same package with `packageName-XX-YY-ZZ`. In the near future this convention will be reversed.

Minimal SConscript file

This example shows a minimal SConscript file. It will build nothing but will import some of the tools necessary for when we add targets to be built. It should be stored in the top level directory of the package.

```
# $Id$
import platform

Import('baseEnv')
Import('listFiles')
progEnv = baseEnv.Clone()
libEnv = baseEnv.Clone()
```

The first line is the familiar line from CVS that will add information about the file's CVS info. The next line will import the platform Python module. This module is only necessary if you wish to add conditions that depend on the OS on which SCons is running.

The two Import() calls import two SCons objects that have been defined at the top level. The first of this, baseEnv, is the basic compile environment created by the top level. It will include all necessary compile options such as debug, optimized, third party library locations, etc.

> **Warning**
>
> It is very important that the base environment is NEVER modified. Any changes to the base environment are applied to all packages independent on the order in which they were called.

The second Import() call imports a function defined at the top level. This function should be used any time you wish to pass a list of files to be included for creating a shared library, static library, program, etc.

The last two lines are what allows us to make our own customizations to the environment that will not affect other builds. By calling baseEnv.Clone() a copy is made of the basic environment and stored in the libraries progEnv and libEnv.

> **Warning**
>
> Due to technical limitations/bugs/features of SCons it is imperative that two copies of the base environment are made if both libraries of any kind and applications are created. This is almost always the case, so it is highly encouraged two copies of the base environment are always created.

Simple static library

After creating the above minimal SConscript file we can create targets to be compiled by SCons. In this example we will create a static library. We will only show the new code that should appear after the above code.

```
myLib = libEnv.StaticLibrary('myLib', listFiles(['src/*.cxx']))
progEnv.Tool('registerObjects', package = 'myPackage', libraries = [myLib], includes = ['myPackage/myLib.h'])
```

The first line is what creates a static library object containing all the information necessary to build the library at a later point. Notice we use the libEnv variable. This is one of the copies made of the base environment. This copy should be used for any libraries to be compiled. It should **NEVER** be used for creating an application. If this rule is violated it will cause errors in other packages that will be hard to track back down to this source.

The StaticLibrary function call takes two arguments. The first argument specifies the name that should be given to the library. This name should not include any prefixes/suffixes that are platform specific. SCons will take care of adding those automatically. This example on Unix based systems would create a library named `libmyLib.a`. The second argument to SCons is a list of files to be compiled into the library. In this case we specify that the files to be included are in the src directory, relative to the top directory of the package, and are named *.cxx (anything ending with .cxx). Should only a single file be needed for the compilation of that library we can either specify a single file to the listFiles function call `listFiles('src/myLib.cxx')` or we can simply skip the use of listFiles() call and replace the entire listFiles function call with ['src/myLib.cxx'].

The second line will register our objects at the top level to be compiled when appropriate. This is not a standard SCons ability but custom extension to SCons. Several things to note are that we use the progEnv copy of the environment to register objects even if the objects are libraries. This is a convention we strong suggest you abide by.

The arguments used by the Tool() call are as follows. The first argument is the name of the tool to be called. Without going into too much detail at this point, this argument needs to always be specified 'regsiterObjects'. The second argument is the name of the package. The third argument is a list of library objects to be registered for this package. These can be shared or static. The name of the variable used is the same as the one used to store the library object returned by libEnv.StaticLibrary() call on the previous line. The next argument is a list of include files to be registered. These are **ONLY** the public include files necessary to use the libraries that are going to be registered. Since we only need a single header file to be registered we specify it with `['myPackage/myLib.h']`. If a list of header files, such as *.h, needed to be specified we could use the listFiles() call again. Other arguments can be specified as well. A complete list is provided in a later section.

Simple Shared library

The steps to create a shared library are similar to those for a static library. The code looks as follows.

```
mySharedLib = libEnv.SharedLibrary('mySharedLib', ['src/file1.cxx', 'src/file2.cxx'])
```

We simply use libEnv.SharedLibrary() call instead of the libEnv.StaticLibrary() call from the previous section. The arguments to the SharedLibrary() call are identical to those of the static library version. This time we also chose to list the files to compile into the library individually. Our shared library will be compiled from the two source files src/flie1.cxx and src/file2.cxx.

Just like before, this object would need to be registered to be included in any SCons builds. Assuming the static library from the previous section and the shared library from this section are both created we would change the registration function call to be as follows

```
progEnv.Tool('registerObjects', package = 'myPackage', libraries = [myLib, mySharedLib], includes = ['myPackage
/myLib.h', 'myPackage/file1.h'])
```

This call is like the one from the previous section except we have added mySharedLib to the list of libraries to be registered. We have also added an additional header file that needs to be registered because in order to use the shared library that header file is needed.

Simple Application

To create a simple application we add a section similar to this:

```
myApp = progEnv.Program('myProgram', ['src/myProgram.cxx'])
```

We use the progEnv copy of the base environment to create a program objection. The copy of the environment for creating libraries should **NEVER** be used for this task. It will generate problems in other packages and will be very difficult to trace back. The arguments provided to the progEnv.Program() call are similar to those for libraries. The first argument is the name of the executable making sure to exclude and platform specific prefixes or suffixes. On Windows SCons will create a program called myProgram.exe. The second argument is a list of source code files to compile the program. All three methods described in the previous two sections are valid here as well (listFiles(), ['single file'], or ['list', 'of', 'files']).

Just like in the previous two sections, we need to register the program object with the top level before it can be used. Assuming the static and shared libraries from before still exist and we want to add this program to the registration call we would modify the registration line as follows

```
progEnv.Tool('registerObjects', package = 'myPackage', libraries = [myLib, mySharedLib], includes = ['myPackage
/myLib.h', 'myPackage/file1.h']
            binaries = [myApp])
```

We added a new argument that will list all the binaries to be used.

> ⚠ **If creating test applications**
>
> You should register test applications with the `testApps = [myApp]` argument instead of the `binaries = [myApp]` argument.

OS specific conditions

If you wish to perform functions on certain platforms only you can use regular python conditionals around the functions.
For example to define the TRAP_FPE macro only on Linux platforms we would append:

```
if platform.system() == 'Linux':
        progEnv.Append(CPPDEFINES = 'TRAP_FPE')
```

The platform.system() call returns the name of the OS we are on. In this case we wish to know if we are running on a Linux platform. If that is the case, we wish to add a -DTRAP_FPE to the gcc command line. The progEnv.Append() call is explained later.

Libraries that depend on other libraries

SCons performs dependency computations at the source code level. It does not compute dependencies of various binary packages such as the dependency of library A on library B when compiling library A into application A. A package maintainer writing application A does not wish to know all dependencies of all libraries. He should only have to know the **DIRECT** dependencies of the application. Similarly the package maintainer of library A should not have to know all the dependencies of library B when creating library A. He should only have to know that library A depends on library B. SCons, by default, does not have this ability. The package maintainer for have to not only know that application A depends on library A but the owner would also have to know that library A depends on library B and so on until all dependencies have been met.

Luckily SCons provides something called a tool to simplify this problem to only specifying direct dependencies. When the package owner creates library A in the SConscript files as described above, the package owner also creates an additional file to record the **DIRECT** dependencies of library A. This file has to have a specific name called <package>Lib.py. It has to be located in the top level of the package along with the SConscript file. Continuing our example, our package owner of 'myPackage' has two libraries myLib and mySharedLib. Assuming myLib depends on some other library of some other package **DIR ECTLY**, let's call that package someOtherPackage, and myLib also depends on some external library xerces. The contents of myPackageLib.py would be as follows:

```
def generate(env, **kw):
    if not kw.get('depsOnly', 0):
        env.Tool('addLibrary', library = ['libA'])
    env.Tool('someOtherPackageLib')
    env.Tool('addLibrary', library = env['xercesLibs'])

def exists(env):
    return 1
```

Both these python functions need to exist at all times. The second of these functions is for features currently not used by use so it should always be specified as shown. The first function is what creates the recursive computation of library dependencies. The first line in the if statement adds libA to the dependencies. This line is put inside an if statement that determines if it should be added or not. The reason for this is that the dependencies of libA need to be specified when libA is created. However, when libA is being created we can't specify that it should include libA. This would create a recursive dependency. As a result, when we want to build libA and we want to catch all the dependencies we call this function but pass an additional argument setting depsOnly = 1 so that the recrusive dependency isn't created. The second (and more if needed) add the dependencies of libA to other libraries created by other packages. These must be **DIRECT** dependencies to keep computation fast. Unnecessary listings will slow SCons down considerably. The last line of the function lists one (or more, if needed) external libraries that libA depends on **DIRECTLY**.

As stated above, when we create libA we want to link all of libA's dependencies into libA without creating a recursive dependency. In order to achieve this, we added that if statement around the addition of libA to the link line. In our SConscript file, prior to creating libA we add the dependencies with this line

```
libEnv.Tool('myPackageLib', depsOnly = 1)
```

This is identical to the example below when we want to link in libraries into the application. The only difference is that when we create libA we don't want to link in libA so we add the extra argument of **depsOnly = 1**.
TODO: Currently there's no way to specify which library created by a single package we wish to use. This feature will be added at a later stage since the problem has not arisen yet in ScienceTools.

Application dependence on libraries

With the dependency tree generated by the previous section for libraries, package owners wishing to create dependencies on libraries for their applications need to only list **DIRECT** dependencies of their applications. Continuing our example, the owner of myPackage currently has one application myApp. Assuming this application depends on some library from myPackage as well as the external library ROOT **DIRECTLY**, he would add this line to his SConscript file:

```
progEnv.Tool('myPackageLib')
progEnv.Tool('addLibrary', library = env['ROOTLibs'])
```

Technically the ordering of these two calls does not matter. We highly recommend, however, that these calls be made prior to the call for generating the application. This will make it easier for a human to understand the code at a later point. The first line will call the generate function from myPackageLib created as shown in the previous section. That function will add the library from that package to the dependencies of myApp along with any other libraries that myPackage's library depends on. The second call adds ROOT to the libraries that myApp depends on **DIRECTLY**. Should myApp not need ROOT directly but through some other package's library, it should be left out here. It is the responsibility of that package to add the ROOT dependency.

Arguments to register objects

Registration functionality is an extension of SCons created by us. The registration is done by a call to progeEnv.Tool('registerObjects, 'mypackage', ...). The minimum arguments to that function is two. The first argument hast to always be 'registerObjects'. The second argument always has to be the name of the package performing the function call. Additional arguments can be from the following:

- libraries - List of Shared of Static libraries to be registered.
- binaries - List of applications to be registered. This does **NOT** include test applications.
- includes - List of header files to be registered. These are **ONLY** header files necessary for other packages to use. These should not be internally used header files.
- testApps - List of test applications to be regsitered.
- pfiles - List of pfiles to be registered.

Other arguments will be added as the need for them arises. The ordering of these arguments are not important.

Compiler options

SCons provides some compiler independent options. They should be used as much as possible. Less common options have to be specified in a compiler dependent way and, therefore, prior to setting a compiler option for a less common option one should perform a check for which OS is being executed. This sort of check was described in a previous section.

In almost all cases these compiler options are lists that need to be appended to. If they are simply assigned new values they will overwrite older options already defined. Additionally, in order to prevent repetition of the same compile options several times, one should only append if the option does not exist. Luckily these options are available through a single function call: AppendUnique().

For example, to add a unique pre processor definition to the compiler when compiling an application you would do

```
progeEnv.AppendUnique(CPPFLAGS = ['TRAP_FPE')
```

This would add a -DTRAPF_FPE to the compiler options if one didn't already exist.

A complete version of these compiler options is available in the SCons man pages online at http://scons.org.

Common platform independent compiler options

These compiler options appear common enough that SCons will take care of converting to the required format for the compiler being used. If possible one should use these as much as possible.

- CPPFLAGS - A platform independent specification of C preprocessor definitions.
- CPPPATH - The list of directories that the C preprocessor will search for include directories. You should never need to set this.
- LIBPATH - The list of directories that will be searched for libraries. You should never need to set this.
- LIBS - A list of one or more libraries that will be linked with any executable programs created by this environment. You should never need to set this.

Less common platform dependent compiler options

These compiler options need to be specified different depending on the compiler being used. As a result they should be wrapped around if statements.

- ARFLAGS - General options passed to the static library archiver. You should never need to set this.
- CCFLAGS - General options that are passed to the C and C++ compilers.
- CFLAGS - General options that are passed to the C compiler (C only; not C++).
- CXXFLAGS - General options that are passed to the C++ compiler. By default, this includes the value of CCFLAGS, so that setting CCFLAGS affects both C and C++ compilation.

External Libraries

The directory structure for the external libraries that SCons uses is different than those used by CMT. The directory structure in SCons was modified so that Unix and Windows based ones are more compatible than before. These external libraries only exist as 1 version for a particular OS (opt and debug builds use the same libraries). As a result the CMT external library structure can't be used with SCons anymore.

All external libraries used are automatically added to the library path of the compiler. A package only needs to add the libraries themselves to the library path. For example, to add CLHEP to list of libraries to link against you would use:

```
progEnv.Tool('addLibrary', library = progEnv['clhepLibs'])
```

Above, in the section "Libraries that depend on other libraries" is a similar example for external libraries as in this section. The same names for accessing the external libraries should be used in that section as the ones defined further down in this section.

Each external library the libraries required for linking against it stored in a variable like the above for clhepLibs. Currently these are the external libraries available to be linked against. The progEnv'clhepLibs' should be replaced with the name of the libraries you wish to add.

- CLHEP - clhepLibs
- FFTW - fftwLibs
- FITS - cfitsioLibs
- PIL - pilLibs
- ROOT - rootLibs and, additionally, rootGuiLibs if needed.
- SWIG - There are currently no libraries used as part of SWIG so none are defined.
- XERCES - xercesLibs
- CPPUnit - cppunitLibs
- Python - pythonLibs

Running SCons

SCons is installed at slac in /afs/slac/g/glast/applications/SCons/0.97.0d20070809/bin/scons. You have to be in the directory that contains the SConsctruct file when you run SCons. For example to check out ScienceTools version LATEST1.2220 you'd issue the cvs command:

```
cvs co -r ScienceTools-LATEST-1-2220 ScienceTools-scons
```

After the completion of this command you would enter the ScienceTools-scon directory that was created. In this directory there is the SConstruct file that is read by SCons.

To get a list of options added to SCons specifically for ScienceTools you can issue the command:

```
scons --help
```

These options have been programmed into the SConstruct and support files for ScienceTools. They are not written or supported by the SCons developers. To see the options that are written and supported by the SCons developers you need to issue the command:

```
scons -H
```

Building only a specific package (and its dependencies)

To build only a single package simply specify the package name as the target. For example, to build facilities (and its dependencies) issue the following command along with any other options necessary.

```
scons facilities
```

Ignoring errors

By default SCons will stop compilation after the first error it encounters. Since the first error is most likely not in a package of interest, one can tell SCons to continue building after it encounters an error. To do this you append the **-i** option to the SCons command.

Parallel builds

Since SCons reads every file it will build, it has very detailed knowledge of the build structure. As a result, SCons can safely perform parallel builds and not violate any dependency issues. If you wish to tell SCons to do parallel builds you need to specify the **-j** option. The -j option is followed by a number to specify how many concurrent builds should be done. For example, -j 2 would tell SCons to perform execute 2 g++ commands at the same time.

SCons build output

When SCons is performing the build process it will put files in the following subdirectories that are located in the same directory as the SConstruct file:

- include/[packageName] - all the globally shared header files for [packageName]
- bin/[variant] - all the binaries for the variant. A variant specifies the OS and compile options such as debug or optimized.
- lib/[variant] - all the libraries for the variant. This variant string is the same as above
- pfiles/[packageName] - all the pfiles for [packageName]

Other such output directories will be created in the future and they will follow the same convention. If the contents of the directory is dependent on the OS or the compile options, it will include the variant sub directory. If it is independent of such changes it will not include that directory.