SCons and Windows To Do

A partial to-do list

1. Environment setter-upper file might need a little more work. For example, swig executable was not in path for Studio. That one is now (6/19) fixed, but more may be lurking.

1. Project files seem to want to rebuild everything all the time. For example, rebuilding a facilities test program from Studio compiles not just program-specific source, but also those files in the facilities library which the program depends on, even though the library was just built.

Already-done list

1. Figure out what needs to be in a library project file to get Studio to copy the library to the SCons variant lib directory (same place the library gets installed when built by SCons). **Done 3/10**
2. Currently only .cxx files are added to projects. Need .h, possibly also xml, python, etc. **Done 3/11** for includes, xml, python and pfiles. These files are already known if there is a suitable call to registerObjects in the package SConscript file.
3. The .cxx files in the project are specified using an absolute file path. This allows Studio to find the files, but means that the project files are not portable. They must be built where used. Should make a special SCons target for an installation which only regenerates all project and solution files. *[3/11 update: Probably this will entail enhancements in our SCons machinery to make it easy to identify various targets and a feature in GoGui making it easy to request build of a specific target.]* **Done 3/20**
4. Don't know how to make a solution file referencing all the projects in a package. When I try the obvious thing from within SCons, invoking MSVSSolution( ), the resulting file uses paths to the SCons build area rather than real destination of the file. The ones in the build area are not (and are not intended to be) usable as project files. When I tried adding a second project to a solution file from within Studio it balked. **Done 3/20** s olution file now uses correct absolute paths to reference project files. **Done 3/23** solution file is installed in a reasonable place instead of languishing in SCons build directory.
5. **Solutions files** created by SCons need to a) include references to project files for all libraries used, not just the ones in the current package and b) keep track of dependencies among the projects so that build order is correct. **Done 4/10**
6. Add **custom tool for swig** to project files when library is a wrapper and other special handling needed in project files for wrapper libraries **Done 4 /21**
7. Add **custom tool for rootcint** to project files when library needs to be dynamically loadable from ROOT. **Done 5/8**
8. Get SCons to create a usable **environment-setter-upper**, probably a .bat file that can be run before invoking devenv. **Partially done as of 3/13**. _setup.bat fixes up the PATH variable but doesn't yet handle other variables. **As of 3/20** also defines INST_DIR. Still need to add definition for ROOTSYS, add to PYTHONPATH and (the tough one) deal with pfiles. **As of 4/21** added stuff needed for swig and to determine which library dependencies are externals and which are not. **Done 6/16** using a two-part method: default build creates _setup.vbs which, when run, writes _setup.bat. The latter has hard-coded paths so should only be created by the end-user. (looks like Swig set-up got inadvertantly left out, though).

Solution files - a plan

The closest analog to cmt show uses in SCons is probably the LIBS contruction environment variable, used to construct the link command for a particular target. It has a list of all libraries (by name; does not include path information) needed in the link. In principal there could be a different LIBS variable for every target, but usually all or most applications in a single package will use the same construction environment (hence have the same value for LIBS). The LIBS variable for the primary shareable library is necessarily different since it doesn't include a reference to itself. LIBS includes references to external libraries as well as locally-built ones, so it will be necessary to discover which names refer to externals and exclude them from further processing. **Done as 4/2**

In order to make it easy to refer to "foreign" project files I will probably change the scheme I have now (in which project and solution files for a package are kept in a special directory under the package root) to one where project and solution files are installed in a top-level variant directory, as is the case with libraries, include files, etc. **Done as of 4/2**

Support for generating project and solution files is currently split between our locally-written tool, **registerObjects**, and SCons-supplied (but locally enhanced) **msvs**. **registerObjects**does more than just generate project and solution files, but it is beginning to look somewhat top-heavy (also the current call-interface for **registerObjects** isn't quite adequate since there is no place for per-target construction environment) so I plan to split off the project /solution file support to a separate module, say called ***makeStudio***. Since interface needs to change, made a replacement for **registerObjects** (working name is **registerStuff**) with the new interface. It handles non-Windows-specific registration, then invokes **makeStudio**. **Done as of 4/2**.

For each target belonging to the package, **makeStudio** will invoke msvs.MSVSProject() to make a project file. This is already being handled by the current version of **registerObjects**. The only change will be to install the project file in the new top-level variant directory. **Then makeStudio**will call msvs. MSVSSolution but instead of just passing a list of project files, as is currently done, it will also have to pass in the associated pruned (external libraries excluded) LIBS variables. Made mods to externals.scons to make set of external library names available as a set via construction variable. makeStudio uses this to compute set of non-external libraries required for each project. **Done as of 4/2**

msvs.MSVSSolution() will need considerable work to get the dependencies into the solution file *and*to do it efficiently. The latter will involve a suitable design of data structures to store derived information from previously-handled values for LIBS in a quickly retrievable and usable form. **Done 4/10** (Efficiently enough for now, anyway).