NAME

scons − a software construction tool

SYNOPSIS

scons [*options...*] [*name=val...*] [*targets...*]

DESCRIPTION

The **scons** utility builds software (or other files) by determining which component pieces must be rebuilt and executing the necessary commands to rebuild them.

By default, **scons** searches for a file named *SConstruct*, *Sconstruct*, or *sconstruct* (in that order) in the current directory and reads its configuration from the first file found. An alternate file name may be specified via the **-f** option.

The *SConstruct* file can specify subsidiary configuration files using the **SConscript()** function. By convention, these subsidiary files are named *SConscript*, although any name may be used. (Because of this naming convention, the term "SConscript files" is sometimes used to refer generically to all **scons** configuration files, regardless of actual file name.)

The configuration files specify the target files to be built, and (optionally) the rules to build those targets. Reasonable default rules exist for building common software components (executable programs, object files, libraries), so that for most software projects, only the target and input files need be specified.

scons reads and executes the SConscript files as Python scripts, so you may use normal Python scripting capabilities (such as flow control, data manipulation, and imported Python libraries) to handle complicated build situations. **scons**, however, reads and executes all of the SConscript files *before* it begins building any targets. To make this obvious, **scons** prints the following messages about what it is doing:

```
$ scons foo.out
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
cp foo.in foo.out
scons: done building targets.
$
```

The status messages (everything except the line that reads "cp foo.in foo.out") may be suppressed using the **-Q** option.

scons does not automatically propagate the external environment used to execute **scons** to the commands used to build target files. This is so that builds will be guaranteed repeatable regardless of the environment variables set at the time **scons** is invoked. This also means that if the compiler or other commands that you want to use to build your target files are not in standard system locations, **scons** will not find them unless you explicitly set the PATH to include those locations. Whenever you create an **scons** construction environment, you can propagate the value of PATH from your external environment as follows:

```
import os
env = Environment(ENV = {'PATH' : os.environ['PATH']})
```

scons can scan known input files automatically for dependency information (for example, #include statements in C or C++ files) and will rebuild dependent files appropriately whenever any "included" input file changes. **scons** supports the ability to define new scanners for unknown input file types.

scons knows how to fetch files automatically from SCCS or RCS subdirectories using SCCS, RCS or BitKeeper.

scons is normally executed in a top-level directory containing a *SConstruct* file, optionally specifying as command-line arguments the target file or files to be built.

By default, the command

```
scons
```

will build all target files in or below the current directory. Explicit default targets (to be built when no targets are specified on the command line) may be defined the SConscript file(s) using the **Default()** function, described below.

Even when **Default()** targets are specified in the SConscript file(s), all target files in or below the current directory may be built by explicitly specifying the current directory (.) as a command-line target:

```
scons .
```

Building all target files, including any files outside of the current directory, may be specified by supplying a command-line target of the root directory (on POSIX systems):

```
scons /
```

or the path name(s) of the volume(s) in which all the targets should be built (on Windows systems):

```
scons C:\ D:\
```

To build only specific targets, supply them as command-line arguments:

```
scons foo bar
```

in which case only the specified targets will be built (along with any derived files on which they depend).

Specifying "cleanup" targets in SConscript files is not necessary. The **-c** flag removes all files necessary to build the specified target:

```
scons -c .
```

to remove all target files, or:

```
scons -c build export
```

to remove target files under build and export. Additional files or directories to remove can be specified using the Clean() function.

A subset of a hierarchical tree may be built by remaining at the top-level directory (where the *SConstruct* file lives) and specifying the subdirectory as the target to be built:

```
scons src/subdir
```

or by changing directory and invoking scons with the **-u** option, which traverses up the directory hierarchy until it finds the *SConstruct* file, and then builds targets relatively to the current subdirectory:

```
cd src/subdir
scons -u .
```

scons supports building multiple targets in parallel via a **-j** option that takes, as its argument, the number of simultaneous tasks that may be spawned:

```
scons -j 4
```

builds four targets in parallel, for example.

scons can maintain a cache of target (derived) files that can be shared between multiple builds. When caching is enabled in a SConscript file, any target files built by **scons** will be copied to the cache. If an up-to-date target file is found in the cache, it will be retrieved from the cache instead of being rebuilt locally. Caching behavior may be disabled and controlled in other ways by the **--cache-force**, **--cache-disable**, and **--cache-show** command-line options. The **--random** option is useful to prevent multiple builds from trying to update the cache simultaneously.

Values of variables to be passed to the SConscript file(s) may be specified on the command line:

```
scons debug=1 .
```

These variables are available in SConscript files through the ARGUMENTS dictionary, and can be used in the SConscript file(s) to modify the build in any way:

```
if ARGUMENTS.get('debug', 0):
   env = Environment(CCFLAGS = '-g')
else:
   env = Environment()
```

The command-line variable arguments are also available in the ARGLIST list, indexed by their order on the command line. This allows you to process them in order rather than by name, if necessary. ARGLIST[0] returns a tuple containing (argname, argvalue). A Python exception is thrown if you try to access a list member that does not exist.

scons requires Python version 1.5.2 or later. There should be no other dependencies or requirements to run **scons.**

By default, **scons** knows how to search for available programming tools on various systems. On WIN32 systems, **scons** searches in order for the Microsoft Visual C++ tools, the MinGW tool chain, the Intel compiler tools, and the PharLap ETS compiler. On OS/2 systems, **scons** searches in order for the OS/2 compiler, the GCC tool chain, and the Microsoft Visual C++ tools, On SGI IRIX, IBM AIX, Hewlett Packard HP-UX, and Sun Solaris systems, **scons** searches for the native compiler tools (MIPSpro, Visual Age, aCC, and Forte tools respectively) and the GCC tool chain. On all other platforms, including POSIX (Linux and UNIX) platforms, **scons** searches in order for the GCC tool chain, the Microsoft Visual C++ tools, and the Intel compiler tools. You may, of course, override these default values by appropriate configuration of Environment construction variables.

OPTIONS

In general, **scons** supports the same command-line options as GNU **make**, and many of those supported by **cons**.

-b Ignored for compatibility with non-GNU versions of **make.**

-c, --clean, --remove
: Clean up by removing all target files for which a construction command is specified. Also remove any files or directories associated to the construction command using the Clean() function.

--cache-disable, --no-cache
Disable the derived-file caching specified by **CacheDir**(). **scons** will neither retrieve files from the cache nor copy files to the cache.

--cache-force, --cache-populate
When using **CacheDir**(), populate a cache by copying any already-existing, up-to-date derived files to the cache, in addition to files built by this invocation. This is useful to populate a new cache with all the current derived files, or to add to the cache any derived files recently built with caching disabled via the **--cache-disable** option.

--cache-show
When using **CacheDir**() and retrieving a derived file from the cache, show the command that would have been executed to build the file, instead of the usual report, "Retrieved ‘file’ from cache." This will produce consistent output for build logs, regardless of whether a target file was rebuilt or retrieved from the cache.

-C *directory*, --directory=*directory*
Change to the specified *directory* before searching for the *SConstruct*, *Sconstruct*, or *sconstruct* file, or doing anything else. Multiple **-C** options are interpreted relative to the previous one, and the right-most **-C** option wins. (This option is nearly equivalent to **-f directory/SConstruct**, except that it will search for *SConstruct*, *Sconstruct*, or *sconstruct* in the specified directory.)

-D
Works exactly the same way as the **-u** option except for the way default targets are handled. When this option is used and no targets are specified on the command line, all default targets are built, whether or not they are below the current directory.

--debug=*type*
Debug the build process. *type* specifies what type of debugging:

--debug=count
Print a count of how many objects are created of the various classes used internally by SCons. This only works when run under Python 2.1 or later.

--debug=dtree
Print the dependency tree after each top-level target is built. This prints out only derived files.

--debug=includes
Print the include tree after each top-level target is built. This is generally used to find out what files are included by the sources of a given derived file:

```
$ scons --debug=includes foo.o
```

--debug=memory
Prints how much memory SCons uses before and after reading the SConscript files and before and after building.

--debug=objects
Prints a list of the various objects of the various classes used internally by SCons. This only works when run under Python 2.1 or later.

--debug=pdb
Re-run SCons under the control of the pdb Python debugger.

--debug=presub
Print the raw command line used to build each target before the construction environment variables are substituted. Also shows which targets are being built by this command. Output looks something like this:

```
$ scons --debug=presub
Building myprog.o with action(s):
  $SHCC $SHCCFLAGS $CPPFLAGS $_CPPINCFLAGS -c -o $TARGET $SOURCES
```

--debug=time
Prints various time profiling information: the time spent executing each build command, the total build time, the total time spent executing build commands, the total time spent executing SConstruct and SConscript files, and the total time spent executing SCons itself.

--debug=tree
Print the dependency tree after each top-level target is built. This prints out the complete dependency tree including implicit dependencies and ignored dependencies.

-f *file*, --file=*file*, --makefile=*file*, --sconstruct=*file*
Use *file* as the initial SConscript file.

-h, --help
Print a local help message for this build, if one is defined in the SConscript file(s), plus a line that describes the **-H** option for command-line option help. If no local help message is defined, prints the standard help message about command-line options. Exits after displaying the appropriate message.

-H, --help-options
Print the standard help message about command-line options and exit.

-i, --ignore-errors
Ignore all errors from commands executed to rebuild files.

-I *directory*, --include-dir=*directory*
Specifies a *directory* to search for imported Python modules. If several **-I** options are used, the directories are searched in the order specified.

--implicit-cache
Cache implicit dependencies. This can cause **scons** to miss changes in the implicit dependencies in cases where a new implicit dependency is added earlier in the implicit dependency search path (e.g. CPPPATH) than a current implicit dependency with the same name.

--implicit-deps-changed
Force SCons to ignore the cached implicit dependencies. This causes the implicit dependencies to be rescanned and recached. This implies **--implicit-cache**.

--implicit-deps-unchanged
: Force SCons to ignore changes in the implicit dependencies. This causes cached implicit dependencies to always be used. This implies **--implicit-cache**.

-j *N*, --jobs=*N*
: Specifies the number of jobs (commands) to run simultaneously. If there is more than one **-j** option, the last one is effective.

-k, --keep-going
: Continue as much as possible after an error. The target that failed and those that depend on it will not be remade, but other targets specified on the command line will still be processed.

--duplicate=*ORDER*
: There are three ways to duplicate files in a build tree: hard links, soft (symbolic) links and copies. The default behaviour of SCons is to prefer hard links to soft links to copies. You can specify different behaviours with this option. *ORDER* must be one of *hard-soft-copy* (the default), *soft-hard-copy*, *hard-copy*, *soft-copy* or *copy*. SCons will attempt to duplicate files using the mechanisms in the specified order.

-m
: Ignored for compatibility with non-GNU versions of **make**.

--max-drift=*SECONDS*
: Set the maximum expected drift in the modification time of files to *SECONDS*. This value determines how old a file must be before its content signature is cached. The default value is 2 days, which means a file must have a modification time of at least two days ago in order to have its content signature cached. A negative value means to never cache the content signature and to ignore the cached value if there already is one. A value of 0 means to always cache the signature, no matter how old the file is.

-n, --just-print, --dry-run, --recon
: No execute. Print the commands that would be executed to build any out-of-date target files, but do not execute the commands.

--profile=*file*
: Run SCons under the Python profiler and save the results in the specified *file*. The results may be analyzed using the Python pstats module.

-q, --question
: Do not run any commands, or print anything. Just return an exit status that is zero if the specified targets are already up to date, non-zero otherwise.

-Q
: Quiets SCons status messages about reading SConscript files, building targets and entering directories. Commands that are executed to rebuild target files are still printed.

--random
: Build dependencies in a random order. This is useful when building multiple trees simultaneously with caching enabled, to prevent multiple builds from simultaneously trying to build or retrieve the same target files.

-s, --silent, --quiet
Silent. Do not print commands that are executed to rebuild target files. Also suppresses SCons status messages.

-S, --no-keep-going, --stop
Ignored for compatibility with GNU **make**.

-t, --touch
Ignored for compatibility with GNU **make**. (Touching a file to make it appear up-to-date is unnecessary when using **scons**.)

-u, --up, --search-up
Walks up the directory structure until an *SConstruct* , *Sconstruct* or *sconstruct* file is found, and uses that as the top of the directory tree. If no targets are specified on the command line, only targets at or below the current directory will be built.

-U
Works exactly the same way as the **-u** option except for the way default targets are handled. When this option is used and no targets are specified on the command line, all default targets that are defined in the SConscript(s) in the current directory are built, regardless of what directory the resultant targets end up in.

-v, --version
Print the **scons** version, copyright information, list of authors, and any other relevant information. Then exit.

-w, --print-directory
Print a message containing the working directory before and after other processing.

--warn=*type*, --warn=no-*type*
Enable or disable warnings. *type* specifies the type of warnings to be enabled or disabled:

--warn=all, --warn=no-all
Enables or disables all warnings.

--warn=dependency, --warn=no-dependency
Enables or disables warnings about dependencies. These warnings are disabled by default.

--warn=deprecated, --warn=no-deprecated
Enables or disables warnings about use of deprecated features. These warnings are enabled by default.

--warn=missing-sconscript, --warn=no-missing-sconscript
Enables or disables warnings about missing SConscript files. These warnings are enabled by default.

--no-print-directory
Turn off -w, even if it was turned on implicitly.

-Y *repository*, --repository=*repository*
Search the specified repository for any input and target files not found in the local directory hierarchy. Multiple **-Y** options may specified, in which case the repositories are searched in the order specified.

CONFIGURATION FILE REFERENCE

Construction Environments

A construction environment is the basic means by which the SConscript files communicate build information to **scons**. A new construction environment is created using the **Environment** function:

```
env = Environment()
```

By default, a new construction environment is initialized with a set of builder methods and construction variables that are appropriate for the current platform. An optional platform keyword argument may be used to specify that an environment should be initialized for a different platform:

```
env = Environment(platform = 'cygwin')
env = Environment(platform = 'os2')
env = Environment(platform = 'posix')
env = Environment(platform = 'win32')
```

Specifying a platform initializes the appropriate construction variables in the environment to use and generate file names with prefixes and suffixes appropriate for the platform.

Note that the **win32** platform adds the **SYSTEMROOT** variable from the user's external environment to the construction environment's **ENV** dictionary. This is so that any executed commands that use sockets to connect with other systems (such as fetching source files from external CVS repository specifications like **:pserver:anonymous@cvs.sourceforge.net:/cvsroot/scons**) will work on Win32 systems.

The platform argument may be function or callable object, in which case the Environment() method will call the specified argument to update the new construction environment:

```
def my_platform(env):
    env['VAR'] = 'xyzzy'

env = Environment(platform = my_platform)
```

Additionally, a specific set of tools with which to initialize the environment may specified as an optional keyword argument:

```
env = Environment(tools = ['msvc', 'lex'])
```

Non-built-in tools may be specified using the toolpath argument:

```
env = Environment(tools = ['default', 'foo'], toolpath = ['tools'])
```

This looks for a tool specification in tools/foo.py (as well as using the ordinary default tools for the platform). foo.py should have two functions: generate(env) and exists(env). generate() modifies the passed in environment and exists() should return a true value if the tool is available. Tools in the toolpath are used before any of the built-in ones. For example, adding gcc.py to the toolpath would override the built-in gcc tool.

The elements of the tools list may also be functions or callable objects, in which case the Environment() method will call the specified elements to update the new construction environment:

```
def my_tool(env):
    env['XYZZY'] = 'xyzzy'

env = Environment(tools = [my_tool])
```

The tool definition (i.e. my_tool()) can use the PLATFORM variable from the environment it receives to customize the tool for different platforms.

If no tool list is specified, then SCons will auto-detect the installed tools using the PATH variable in the ENV construction variable and the platform name when the Environment is constructed. Changing the PATH variable after the Environment is constructed will not cause the tools to be redetected.

SCons supports the following tool specifications out of the box:

```
386asm
aixc++
aixcc
aixf77
aixlink
ar
as
bcc32
c++
cc
cvf
dmd
dvipdf
dvips
f77
f90
f95
fortran
g++
g77
gas
gcc
gnulink
gs
hpc++
hpcc
hplink
icc
icl
ifl
ifort
ilink
ilink32
jar
javac
javah
latex
lex
link
linkloc
```

m4
masm
midl
mingw
mslib
mslink
msvc
msvs
nasm
pdflatex
pdftex
qt
rmic
sgiar
sgic++
sgicc
sgilink
sunar
sunc++
suncc
sunlink
swig
tar
tex
tlib
yacc
zip

Additionally, there is a "tool" named **default** which configures the environment with a default set of tools for the current platform.

On posix and cygwin platforms the GNU tools (e.g. gcc) are preferred by SCons, on win32 the Microsoft tools (e.g. msvc) followed by MinGW are preferred by SCons, and in OS/2 the IBM tools (e.g. icc) are preferred by SCons.

Builder Methods

Build rules are specified by calling a construction environment's builder methods. The arguments to the builder methods are **target** (a list of target files) and **source** (a list of source files).

Because long lists of file names can lead to a lot of quoting, **scons** supplies a **Split()** global function and a same-named environment method that split a single string into a list, separated on strings of white-space characters. (These are similar to the string.split() method from the standard Python library, but work even if the input isn't a string.)

Like all Python arguments, the target and source arguments to a builder method can be specified either with or without the "target" and "source" keywords. When the keywords are omitted, the target is first, followed by the source. The following are equivalent examples of calling the Program builder method:

```
env.Program('bar', ['bar.c', 'foo.c'])
env.Program('bar', Split('bar.c foo.c'))
env.Program('bar', env.Split('bar.c foo.c'))
env.Program(source = ['bar.c', 'foo.c'], target = 'bar')
env.Program(target = 'bar', Split('bar.c foo.c'))
env.Program(target = 'bar', env.Split('bar.c foo.c'))
```

```
env.Program('bar', source = string.split('bar.c foo.c'))
```

When the target shares the same base name as the source and only the suffix varies, and if the builder method has a suffix defined for the target file type, then the target argument may be omitted completely, and **scons** will deduce the target file name from the source file name. The following examples all build the executable program **bar** (on POSIX systems) or **bar.exe** (on Windows systems) from the bar.c source file:

```
env.Program(target = 'bar', source = 'bar.c')
env.Program('bar', source = 'bar.c')
env.Program(source = 'bar.c')
env.Program('bar.c')
```

It is possible to override or add construction variables when calling a builder method by passing additional keyword arguments. These overridden or added variables will only be in effect when building the target, so they will not affect other parts of the build. For example, if you want to add additional libraries for just one program:

```
env.Program('hello', 'hello.c', LIBS=['gl', 'glut'])
```

or generate a shared library with a nonstandard suffix:

```
env.SharedLibrary('word', 'word.cpp', SHLIBSUFFIX='.ocx')
```

Although the builder methods defined by **scons** are, in fact, methods of a construction environment object, they may also be called without an explicit environment:

```
Program('hello', 'hello.c')
SharedLibrary('word', 'word.cpp')
```

In this case, the methods are called internally using a default construction environment that consists of the tools and values that **scons** has determined are appropriate for the local system.

All builder methods return a list of Nodes that represent the target or targets that will be built. A *Node* is an internal SCons object which represents build targets or sources.

The returned Node(s) can be passed to other builder methods as source(s) or passed to any SCons function or method where a filename would normally be accepted. For example, if it were necessary to add a specific **-D** flag when compiling one specific object file:

```
bar_obj_list = env.StaticObject('bar.c', CCFLAGS='-DBAR')
env.Program(source = ['foo.c', bar_obj_list, 'main.c'])
```

Using a Node in this way makes for a more portable build by avoiding having to specify a platform-specific object suffix when calling the Program() builder method.

Note that Builder calls will automatically "flatten" the source and target file lists, so it's all right to have the bar_obj list return by the StaticObject() call in the middle of the source file list. If you need to manipulate a list of lists returned by Builders directly using Python, you can either build the list by hand:

```
foo = Object('foo.c')
bar = Object('bar.c')
objects = ['begin.o'] + foo + ['middle.o'] + bar + ['end.o']
for object in objects:
    print str(object)
```

Or you can use the **Flatten**() supplied by scons to create a list containing just the Nodes, which may be more convenient:

```
foo = Object('foo.c')
bar = Object('bar.c')
objects = Flatten(['begin.o', foo, 'middle.o', bar, 'end.o'])
for object in objects:
    print str(object)
```

The path name for a Node's file may be used by passing the Node to the Python-builtin **str()** function:

```
bar_obj_list = env.StaticObject('bar.c', CCFLAGS='-DBAR')
print "The path to bar_obj is:", str(bar_obj_list[0])
```

Note again that because the Builder call returns a list, we have to access the first element in the list **(bar_obj_list[0])** to get at the Node that actually represents the object file.

scons provides the following builder methods:

CFile()

env.CFile()

Builds a C source file given a lex (.l) or yacc (.y) input file. The suffix specified by the $CFILESUFFIX construction variable (.c by default) is automatically added to the target if it is not already present. Example:

```
# builds foo.c
env.CFile(target = 'foo.c', source = 'foo.l')
# builds bar.c
env.CFile(target = 'bar', source = 'bar.y')
```

CXXFile()

env.CXXFile()

Builds a C++ source file given a lex (.ll) or yacc (.yy) input file. The suffix specified by the $CXXFILESUFFIX construction variable (.cc by default) is automatically added to the target if it is not already present. Example:

```
# builds foo.cc
env.CXXFile(target = 'foo.cc', source = 'foo.ll')
# builds bar.cc
env.CXXFile(target = 'bar', source = 'bar.yy')
```

DVI()

env.DVI()

Builds a .dvi file from a .tex, .ltx or .latex input file. If the source file suffix is .tex, **scons** will examine the contents of the file; if the string **ocumentclass** or **ocumentstyle** is found, the file is assumed to be a LaTeX file and the target is built by invoking the $LATEXCOM command line; otherwise, the $TEXCOM command line is used. If the file is a LaTeX file, the **DVI** builder method will also examine the contents of the **.aux file** and invoke the $BIBTEX command line if the string **bibdata** is found, and will examine the contents **.log** file and re-run the $LATEXCOM command if the log file says it is necessary.

The suffix .dvi (hard-coded within TeX itself) is automatically added to the target if it is not already present. Examples:

```
# builds from aaa.tex
env.DVI(target = 'aaa.dvi', source = 'aaa.tex')
# builds bbb.dvi
env.DVI(target = 'bbb', source = 'bbb.ltx')
# builds from ccc.latex
env.DVI(target = 'ccc.dvi', source = 'ccc.latex')
```

Jar()

env.Jar()

Builds a Java archive (.jar) file from a source tree of .class files. If the $JARCHDIR value is set, the **jar** command will change to the specified directory using the **−C** option. If the contents any of the source files begin with the string **Manifest-Version**, the file is assumed to be a manifest and is passed to the **jar** command with the **m** option set.

```
env.Jar(target = 'foo.jar', source = 'classes')
```

Java()

env.Java()

Builds one or more Java class files from one or more source trees of .java files. The class files will be placed underneath the specified target directory. SCons will parse each source .java file to find the classes (including inner classes) defined within that file, and from that figure out the target .class files that will be created. SCons will also search each Java file for the Java package name, which it assumes can be found on a line beginning with the string **package** in the first column; the resulting .class files will be placed in a directory reflecting the specified package name. For example, the file *Foo.java* defining a single public *Foo* class and containing a package name of *sub.dir* will generate a corresponding *sub/dir/Foo.class* class file.

Example:

```
env.Java(target = 'classes', source = 'src')
env.Java(target = 'classes', source = ['src1', 'src2'])
```

JavaH()

env.JavaH()

Builds C header and source files for implementing Java native methods. The target can be either a directory in which the header files will be written, or a header file name which will contain all of the definitions. The source can be either the names of .class files, or the objects returned from the **Java** builder method.

If the construction variable **JAVACLASSDIR** is set, either in the environment or in the call to the **JavaH** builder method itself, then the value of the variable will be stripped from the beginning of any .class file names.

Examples:

```
# builds java_native.h
classes = env.Java(target = 'classdir', source = 'src')
env.JavaH(target = 'java_native.h', source = classes)

# builds include/package_foo.h and include/package_bar.h
```

```
env.JavaH(target = 'include',
        source = ['package/foo.class', 'package/bar.class'])

# builds export/foo.h and export/bar.h
env.JavaH(target = 'export',
        source = ['classes/foo.class', 'classes/bar.class'],
        JAVACLASSDIR = 'classes')
```

Library()

env.Library()

A synonym for the **StaticLibrary** builder method.

M4()

env.M4()

Builds an output file from an M4 input file. This uses a default $M4FLAGS value of **-E**, which considers all warnings to be fatal and stops on the first warning when using the GNU version of m4. Example:

```
env.M4(target = 'foo.c', source = 'foo.c.m4')
```

Moc()

env.Moc()

Builds an output file from a moc input file. Moc input files are either header files or cxx files. This builder is only available after using the tool 'qt'. See the QTDIR variable for more information. Example:

```
env.Moc('foo.h') # generates moc_foo.cc
env.Moc('foo.cpp') # generates foo.moc
```

MSVSProject()

env.MSVSProject()

Builds Microsoft Visual Studio project files. This builds a Visual Studio project file, based on the version of Visual Studio that is configured (either the latest installed version, or the version set by **MSVS_VERSION** in the Environment constructor). For VS 6, it will generate **.dsp** and **.dsw** files, for VS 7, it will generate **.vcproj** and **.sln** files.

It takes several lists of filenames to be placed into the project file, currently these are limited to **srcs, incs, localincs, resources,** and **misc.** These are pretty self explanatory, but it should be noted that the 'srcs' list is NOT added to the $SOURCES environment variable. This is because it represents a list of files to be added to the project file, not the source used to build the project file (in this case, the 'source' is the SConscript file used to call MSVSProject).

In addition to these values (which are all optional, although not specifying any of them results in an empty project file), the following values must be specified:

target: The name of the target .dsp or .vcproj file. The correct suffix for the version of Visual Studio must be used, but the value

```
env['MSVSPROJECTSUFFIX']
```

will be defined to the correct value (see example below).

variant: The name of this particular variant. These are typically things like "Debug" or "Release", but really can be anything you want. Multiple calls to MSVSProject with different variants are allowed: all variants will be added to the project file with their appropriate build targets and sources.

buildtarget: A list of SCons.Node.FS objects which is returned from the command which builds the target. This is used to tell SCons what to build when the 'build' button is pressed inside of the IDE.

Example Usage:

```
barsrcs = ['bar.cpp'],
barincs = ['bar.h'],
barlocalincs = ['StdAfx.h']
barresources = ['bar.rc','resource.h']
barmisc = ['bar_readme.txt']

dll = local.SharedLibrary(target = 'bar.dll',
                source = barsrcs)

local.MSVSProject(target = 'Bar' + env['MSVSPROJECTSUFFIX'],
            srcs = barsrcs,
            incs = barincs,
            localincs = barlocalincs,
            resources = barresources,
            misc = barmisc,
            buildtarget = dll,
            variant = 'Release')
```

Object()

env.Object()

A synonym for the **StaticObject** builder method.

PCH()

env.PCH()

Builds a Microsoft Visual C++ precompiled header. Calling this builder method returns a list of two targets: the PCH as the first element, and the object file as the second element. Normally the object file is ignored. This builder method is only provided when Microsoft Visual C++ is being used as the compiler. The PCH builder method is generally used in conjuction with the PCH construction variable to force object files to use the precompiled header:

```
env['PCH'] = env.PCH('StdAfx.cpp')[0]
```

PDF()

env.PDF()

Builds a .pdf file from a .dvi input file (or, by extension, a .tex, .ltx, or .latex input file). The suffix specified by the $PDFSUFFIX construction variable (.pdf by default) is added automatically to the target if it is not already present. Example:

```
# builds from aaa.tex
env.PDF(target = 'aaa.pdf', source = 'aaa.tex')
# builds bbb.pdf from bbb.dvi
```

```
env.PDF(target = 'bbb', source = 'bbb.dvi')
```

PostScript()

env.PostScript()

Builds a .ps file from a .dvi input file (or, by extension, a .tex, .ltx, or .latex input file). The suffix specified by the $PSSUFFIX construction variable (.ps by default) is added automatically to the target if it is not already present. Example:

```
# builds from aaa.tex
env.PostScript(target = 'aaa.ps', source = 'aaa.tex')
# builds bbb.ps from bbb.dvi
env.PostScript(target = 'bbb', source = 'bbb.dvi')
```

Program()

env.Program()

Builds an executable given one or more object files or C, C++, D, or Fortran source files. If any C, C++, D or Fortran source files are specified, then they will be automatically compiled to object files using the **Object** builder method; see that builder method's description for a list of legal source file suffixes and how they are interpreted. The target executable file prefix (specified by the $PROGPREFIX construction variable; nothing by default) and suffix (specified by the $PROGSUFFIX construction variable; by default, .exe on Windows systems, nothing on POSIX systems) are automatically added to the target if not already present. Example:

```
env.Program(target = 'foo', source = ['foo.o', 'bar.c', 'baz.f'])
```

RES()

env.RES()

Builds a Microsoft Visual C++ resource file. This builder method is only provided when Microsoft Visual C++ or MinGW is being used as the compiler. The *.res* (or *.o* for MinGW) suffix is added to the target name if no other suffix is given. The source file is scanned for implicit dependencies as though it were a C file. Example:

```
env.RES('resource.rc')
```

RMIC()

env.RMIC()

Builds stub and skeleton class files for remote objects from Java .class files. The target is a directory relative to which the stub and skeleton class files will be written. The source can be the names of .class files, or the objects return from the **Java** builder method.

If the construction variable **JAVACLASSDIR** is set, either in the environment or in the call to the **RMIC** builder method itself, then the value of the variable will be stripped from the beginning of any .class file names.

```
classes = env.Java(target = 'classdir', source = 'src')
env.RMIC(target = 'outdir1', source = classes)

env.RMIC(target = 'outdir2',
         source = ['package/foo.class', 'package/bar.class'])

env.RMIC(target = 'outdir3',
```

```
source = ['classes/foo.class', 'classes/bar.class'],
JAVACLASSDIR = 'classes')
```

SharedLibrary()

env.SharedLibrary()

Builds a shared library (.so on a POSIX system, .dll on WIN32) given one or more object files or C, C++, D or Fortran source files. If any source files are given, then they will be automatically compiled to object files. The static library prefix and suffix (if any) are automatically added to the target. The target library file prefix (specified by the $SHLIBPREFIX construction variable; by default, lib on POSIX systems, nothing on Windows systems) and suffix (specified by the $SHLIBSUFFIX construction variable; by default, .dll on Windows systems, .so on POSIX systems) are automatically added to the target if not already present. Example:

```
env.SharedLibrary(target = 'bar', source = ['bar.c', 'foo.o'])
```

On WIN32 systems, the **SharedLibrary** builder method will always build an import (.lib) library in addition to the shared (.dll) library, adding a .lib library with the same basename if there is not already a .lib file explicitly listed in the targets.

Any object files listed in the **source** must have been built for a shared library (that is, using the **SharedObject** builder method). **scons** will raise an error if there is any mismatch.

On WIN32 systems, specifying "register=1" will cause the dll to be registered after it is built using REGSVR32. The command that is run ("regsvr32" by default) is determined by $REGSVR construction variable, and the flags passed are determined by $REGSVRFLAGS. By default, $REGSVRFLAGS includes "/s", to prevent dialogs from popping up and requiring user attention when it is run. If you change $REGSVRFLAGS, be sure to include "/s". For example,

```
env.SharedLibrary(target = 'bar',
          source = ['bar.cxx', 'foo.obj'],
          register=1)
```

will register "bar.dll" as a COM object when it is done linking it.

SharedObject()

env.SharedObject()

Builds an object file for inclusion in a shared library. Source files must have one of the same set of extensions specified above for the **StaticObject** builder method. On some platforms building a shared object requires additional compiler options (e.g. -fPIC for gcc) in addition to those needed to build a normal (static) object, but on some platforms there is no difference between a shared object and a normal (static) one. When there is a difference, SCons will only allow shared objects to be linked into a shared library, and will use a different suffix for shared objects. On platforms where there is no difference, SCons will allow both normal (static) and shared objects to be linked into a shared library, and will use the same suffix for shared and normal (static) objects. The target object file prefix (specified by the $SHOBJPREFIX construction variable; by default, the same as $OBJPREFIX) and suffix (specified by the $SHOBJSUFFIX construction variable) are automatically added to the target if not already present. Examples:

```
env.SharedObject(target = 'ddd', source = 'ddd.c')
env.SharedObject(target = 'eee.o', source = 'eee.cpp')
env.SharedObject(target = 'fff.obj', source = 'fff.for')
```

StaticLibrary()

env.StaticLibrary()

Builds a static library given one or more object files or C, C++, D or Fortran source files. If any source files are given, then they will be automatically compiled to object files. The static library prefix and suffix (if any) are automatically added to the target. The target library file prefix (specified by the $LIBPREFIX construction variable; by default, lib on POSIX systems, nothing on Windows systems) and suffix (specified by the $LIBSUFFIX construction variable; by default, .lib on Windows systems, .a on POSIX systems) are automatically added to the target if not already present. Example:

```
env.StaticLibrary(target = 'bar', source = ['bar.c', 'foo.o'])
```

Any object files listed in the **source** must have been built for a static library (that is, using the **StaticObject** builder method). **scons** will raise an error if there is any mismatch.

StaticObject()

env.StaticObject()

Builds a static object file from one or more C, C++, D, or Fortran source files. Source files must have one of the following extensions:

```
.asm   assembly language file
.ASM   assembly language file
.c     C file
.C     WIN32:  C file
       POSIX:  C++ file
.cc    C++ file
.cpp   C++ file
.cxx   C++ file
.cxx   C++ file
.c++   C++ file
.C++   C++ file
.d     D file
.f     Fortran file
.F     WIN32:  Fortran file
       POSIX:  Fortran file + C pre-processor
.for   Fortran file
.FOR   Fortran file
.fpp   Fortran file + C pre-processor
.FPP   Fortran file + C pre-processor
.s     assembly language file
.S     WIN32:  assembly language file
       POSIX:  assembly language file + C pre-processor
.spp   assembly language file + C pre-processor
.SPP   assembly language file + C pre-processor
```

The target object file prefix (specified by the $OBJPREFIX construction variable; nothing by default) and suffix (specified by the $OBJSUFFIX construction variable; are automatically added to the target if not already present. Examples:

```
env.StaticObject(target = 'aaa', source = 'aaa.c')
env.StaticObject(target = 'bbb.o', source = 'bbb.c++')
env.StaticObject(target = 'ccc.obj', source = 'ccc.f')
```

Tar()

env.Tar()

Builds a tar archive of the specified files and/or directories. Unlike most builder methods, the **Tar** builder method may be called multiple times for a given target; each additional call adds to the list of entries that will be built into the archive.

```
env.Tar('src.tar', 'src')

# Create the stuff.tar file.
env.Tar('stuff', ['subdir1', 'subdir2'])
# Also add "another" to the stuff.tar file.
env.Tar('stuff', 'another')

# Set TARFLAGS to create a gzip-filtered archive.
env = Environment(TARFLAGS = '-c -z')
env.Tar('foo.tar.gz', 'foo')

# Also set the suffix to .tgz.
env = Environment(TARFLAGS = '-c -z',
          TARSUFFIX = '.tgz')
env.Tar('foo')
```

TypeLibrary()

env.TypeLibrary()

Builds a Windows type library (.tlb) file from and input IDL file (.idl). In addition, it will build the associated inteface stub and proxy source files. It names them according to the base name of the .idl file.

For example,

```
env.TypeLibrary(source="foo.idl")
```

Will create foo.tlb, foo.h, foo_i.c, foo_p.c, and foo_data.c.

Uic()

env.Uic()

Builds a header file, an implementation file and a moc file from an ui file. and returns the corresponding nodes in the above order. This builder is only available after using the tool 'qt'. Note: you can specify .ui files directly as inputs for Program, Library and SharedLibrary without using this builder. Using the builder lets you override the standard naming conventions (be careful: prefixes are always prepended to names of built files; if you don't want prefixes, you may set them to ``). See the QTDIR variable for more information. Example:

```
env.Uic('foo.ui') # -> ['foo.h', 'uic_foo.cc', 'moc_foo.cc']
env.Uic(target = Split('include/foo.h gen/uicfoo.cc gen/mocfoo.cc'),
    source = 'foo.ui') # -> ['include/foo.h', 'gen/uicfoo.cc', 'gen/mocfoo.cc']
```

Zip()

env.Zip()

Builds a zip archive of the specified files and/or directories. Unlike most builder methods, the **Zip** builder method may be called multiple times for a given target; each additional call adds to the list of entries that will be built into the archive.

```
env.Zip('src.zip', 'src')

# Create the stuff.zip file.
env.Zip('stuff', ['subdir1', 'subdir2'])
# Also add "another" to the stuff.tar file.
env.Zip('stuff', 'another')
```

scons automatically scans C source files, C++ source files, Fortran source files with **.F** (POSIX systems only), **.fpp,** or **.FPP** file extensions, and assembly language files with **.S** (POSIX systems only), **.spp,** or **.SPP** files extensions for C preprocessor dependencies, so the dependencies do not need to be specified explicitly. In addition, all targets of builder methods automatically depend on their sources. An explicit dependency can be specified using the **Depends** method of a construction environment (see below).

Methods and Functions to Do Things

In addition to Builder methods, **scons** provides a number of other construction environment methods and global functions to manipulate the build configuration.

Usually, a construction environment method and global function with the same name both exist so that you don't have to remember whether to a specific bit of functionality must be called with or without a construction environment. In the following list, if you call something as a global function it looks like:

Function(*arguments*)

and if you call something through a construction environment it looks like:

env.Function(*arguments*)

If you can call the functionality in both ways, then both forms are listed.

Except where otherwise noted, the same-named construction environment method and global function provide the exact same functionality. The only difference is that, where appropriate, calling the functionality through a construction environment will substitute construction variables into any supplied strings. For example:

```
env = Environment(FOO = 'foo')
Default('$FOO')
env.Default('$FOO')
```

the first call to the global **Default()** function will actually add a target named **$FOO** to the list of default targets, while the second call to the **env.Default()** construction environment method will expand the value and add a target named **foo** to the list of default targets. For more on construction variable expansion, see the next section on construction variables.

Construction environment methods and global functions supported by **scons** include:

Action(*action*, [*strfunction*, *varlist*])

env.Action(*action*, [*strfunction*, *varlist*])

Creates an Action object for the specified *action*. See the section "Action Objects," below, for a complete explanation of the arguments and behavior.

AddPostAction(*target*, *action*)

env.AddPostAction(*target*, *action*)

Arranges for the specified *action* to be performed after the specified *target* has been built. The specified action(s) may be an Action object, or anything that can be converted into an Action object (see below).

AddPreAction(*target*, *action*)

env.AddPreAction(*target*, *action*)

Arranges for the specified *action* to be performed before the specified *target* is built. The specified action(s) may be an Action object, or anything that can be converted into an Action object (see below).

Alias(*alias*, [*targets*])

env.Alias(*alias*, [*targets*])

Creates one or more phony targets that expand to one or more other targets. Returns the Node object representing the alias, which exists outside of any file system. This Node object, or the alias name, may be used as a dependency of any other target, including another alias. **Alias** can be called multiple times for the same alias to add additional targets to the alias.

```
Alias('install')
Alias('install', '/usr/bin')
Alias(['install', 'install-lib'], '/usr/local/lib')

env.Alias('install', ['/usr/local/bin', '/usr/local/lib'])
env.Alias('install', ['/usr/local/man'])
```

AlwaysBuild(*target*, ...)

env.AlwaysBuild(*target*, ...)

Marks each given *target* so that it is always assumed to be out of date, and will always be rebuilt if needed. Note, however, that **AlwaysBuild**() does not add its target(s) to the default target list, so the targets will only be built if they are specified on the command line, or are a dependent of a target specified on the command line--but they will *always* be built if so specified. Multiple targets can be passed in to a single call to **AlwaysBuild**().

env.Append(*key*=*val*, [...])

Appends the specified keyword arguments to the end of construction variables in the environment. If the Environment does not have the specified construction variable, it is simply added to the environment. If the values of the construction variable and the keyword argument are the same type, then the two values will be simply added together. Otherwise, the construction variable and the value of the keyword argument are both coerced to lists, and the lists are added together. (See also the Prepend method, below.)

```
env.Append(CCFLAGS = ' -g', FOO = ['foo.yyy'])
```

env.AppendENVPath(*name*, *newpath*, [*envname*, *sep*])

This appends new path elements to the given path in the specified external environment (**ENV** by default). This will only add any particular path once (leaving the last one it encounters and ignoring the rest, to preserve path order), and to help assure this, will normalize all paths (using **os.path.normpath** and **os.path.normcase**). This can also handle the case where the given old path variable is a list instead of a string, in which case a list will be returned instead of a string. Example:

```
print 'before:',env['ENV']['INCLUDE']
include_path = '/foo/bar:/foo'
env.PrependENVPath('INCLUDE', include_path)
print 'after:',env['ENV']['INCLUDE']
```

yields:

```
before: /foo:/biz
after: /biz:/foo/bar:/foo
```

env.AppendUnique(*key*=*val*, [...])

Appends the specified keyword arguments to the end of construction variables in the environment. If the Environment does not have the specified construction variable, it is simply added to the environment. If the construction variable being appended to is a list, then any value(s) that already exist in the construction variable will *not* be added again to the list.

```
env.AppendUnique(CCFLAGS = '-g', FOO = ['foo.yyy'])
```

env.BitKeeper()

A factory function that returns a Builder object to be used to fetch source files using BitKeeper. The returned Builder is intended to be passed to the **SourceCode** function.

```
env.SourceCode('.', env.BitKeeper())
```

BuildDir(*build_dir*, *src_dir*, [*duplicate*])

env.BuildDir(*build_dir*, *src_dir*, [*duplicate*])

This specifies a build directory *build_dir* in which to build all derived files that would normally be built under *src_dir*. Multiple build directories can be set up for multiple build variants, for example. *src_dir* must be underneath the SConstruct file's directory, and *build_dir* may not be underneath the *src_dir* .

The default behavior is for **scons** to duplicate all of the files in the tree underneath *src_dir* into *build_dir*, and then build the derived files within the copied tree. (The duplication is performed by linking or copying, depending on the platform; see also the *--duplicate* option.) This guarantees correct builds regardless of whether intermediate source files are generated during the build, where preprocessors or other scanners search for included files, or whether individual compilers or other invoked tools are hard-coded to put derived files in the same directory as source files.

This behavior of making a complete copy of the source tree may be disabled by setting *duplicate* to 0. This will cause **scons** to invoke Builders using the path names of source files in *src_dir* and the path names of derived files within *build_dir*. This is always more efficient than *duplicate*=1, and is usually safe for most builds. Specifying *duplicate*=0, however, may cause build problems if source files are generated during the build, if any invoked tools are hard-coded to put derived files in the same directory as the source files.

Note that specifying a **BuildDir** works most naturally with a subsidiary SConscript file in the source directory. However, you would then call the subsidiary SConscript file not in the source directory, but in the *build_dir* , as if **scons** had made a virtual copy of the source tree regardless of the value of *duplicate*. This is how you tell **scons** which variant of a source tree to build. For example:

```
BuildDir('build-variant1', 'src')
SConscript('build-variant1/SConscript')
BuildDir('build-variant2', 'src')
SConscript('build-variant2/SConscript')
```

See also the **SConscript**() function, described below, for another way to specify a build directory in conjunction with calling a subsidiary SConscript file.)

Builder(*action*, [*arguments*])

env.Builder(*action*, [*arguments*])

Creates a Builder object for the specified *action*. See the section "Builder Objects," below, for a complete explanation of the arguments and behavior.

CacheDir(*cache_dir*)

env.CacheDir(*cache_dir*)

Specifies that **scons** will maintain a cache of derived files in *cache_dir* . The derived files in the cache will be shared among all the builds using the same **CacheDir**() call.

When a **CacheDir**() is being used and **scons** finds a derived file that needs to be rebuilt, it will first look in the cache to see if a derived file has already been built from identical input files and an identical build action (as incorporated into the MD5 build signature). If so, **scons** will retrieve the file from the cache. If the derived file is not present in the cache, **scons** will rebuild it and then place a copy of the built file in the cache (identified by its MD5 build signature), so that it may be retrieved by other builds that need to build the same derived file from identical inputs.

Use of a specified **CacheDir()** may be disabled for any invocation by using the **--cache-disable** option.

If the **--cache-force** option is used, **scons** will place a copy of *all* derived files in the cache, even if they already existed and were not built by this invocation. This is useful to populate a cache the first time **CacheDir**() is added to a build, or after using the **--cache-disable** option.

When using **CacheDir**(), **scons** will report, "Retrieved ‘file’ from cache," unless the **--cache-show** option is being used. When the **--cache-show** option is used, **scons** will print the action that *would* have been used to build the file, without any indication that the file was actually retrieved from the cache. This is useful to generate build logs that are equivalent regardless of whether a given derived file has been built in-place or retrieved from the cache.

Clean(*targets*, *files_or_dirs*)

env.Clean(*targets*, *files_or_dirs*)

This specifies a list of files or directories which should be removed whenever the targets are specified with the **-c** command line option. The specified targets may be a list or an individual target. Multiple calls to **Clean**() are legal, and create new targets or add files and directories to the clean list for the specified targets.

Multiple files or directories should be specified either as separate arguments to the **Clean**() method, or as a list. **Clean**() will also accept the return value of any of the construction environment Builder methods. Examples:

```
Clean('foo', ['bar', 'baz'])
Clean('dist', env.Program('hello', 'hello.c'))
Clean(['foo', 'bar'], 'something_else_to_clean')
```

Command(*target*, *source*, *commands*, [*key*=*val*, ...])

env.Command(*target*, *source*, *commands*, [*key*=*val*, ...])

Executes a specific action (or list of actions) to build a target file or files. This is more convenient than defining a separate Builder object for a single special-case build. Any keyword arguments specified override any same-named existing construction variables.

Note that an action can be an external command, specified as a string, or a callable Python object;

see "Action Objects," below. Examples:

```
env.Command('foo.out', 'foo.in',
        "$FOO_BUILD < $SOURCES > $TARGET")

env.Command('bar.out', 'bar.in',
        ["rm -f $TARGET",
         "$BAR_BUILD < $SOURCES > $TARGET"],
        ENV = {'PATH' : '/usr/local/bin/'})

def rename(env, target, source):
    import os
    os.rename('.tmp', str(target[0]))

env.Command('baz.out', 'baz.in',
        ["$BAZ_BUILD < $SOURCES > .tmp",
            rename ])
```

Configure(*env*, [*custom_tests*, *conf_dir*, *log_file*])

env.Configure([*custom_tests*, *conf_dir*, *log_file*])

Creates a Configure object for integrated functionality similar to GNU autoconf. See the section "Configure Contexts," below, for a complete explanation of the arguments and behavior.

env.Copy([*key*=*val*, ...])

Return a separate copy of a construction environment. If there are any keyword arguments specified, they are added to the returned copy, overwriting any existing values for the keywords.

```
env2 = env.Copy()
env3 = env.Copy(CCFLAGS = '-g')
```

Additionally, a list of tools and a toolpath may be specified, as in the Environment constructor:

```
def MyTool(env): env['FOO'] = 'bar'
env4 = env.Copy(tools = ['msvc', MyTool])
```

env.CVS(*repository*, *module*)

A factory function that returns a Builder object to be used to fetch source files from the specified CVS *repository*. The returned Builder is intended to be passed to the **SourceCode** function.

The optional specified *module* will be added to the beginning of all repository path names; this can be used, in essence, to strip initial directory names from the repository path names, so that you only have to replicate part of the repository directory hierarchy in your local build directory:

```
# Will fetch foo/bar/src.c
# from /usr/local/CVSROOT/foo/bar/src.c.
env.SourceCode('.', env.CVS('/usr/local/CVSROOT'))

# Will fetch bar/src.c
# from /usr/local/CVSROOT/foo/bar/src.c.
env.SourceCode('.', env.CVS('/usr/local/CVSROOT', 'foo'))

# Will fetch src.c
# from /usr/local/CVSROOT/foo/bar/src.c.
```

```
env.SourceCode('.', env.CVS('/usr/local/CVSROOT', 'foo/bar'))
```

Default(*targets*)

env.Default(*targets*)

This specifies a list of default targets, which will be built by **scons** if no explicit targets are given on the command line. Multiple calls to **Default**() are legal, and add to the list of default targets.

Multiple targets should be specified as separate arguments to the **Default**() method, or as a list. **Default**() will also accept the Node returned by any of a construction environment's builder methods. Examples:

```
Default('foo', 'bar', 'baz')
env.Default(['a', 'b', 'c'])
hello = env.Program('hello', 'hello.c')
env.Default(hello)
```

An argument to **Default**() of **None** will clear all default targets. Later calls to **Default**() will add to the (now empty) default-target list like normal.

The current list of targets added using the **Default**() function or method is available in the **DEFAULT_TARGETS** list; see below.

DefaultEnvironment([*args*])

Creates and returns a default construction environment object. This construction environment is used internally by SCons in order to execute many of the global functions in this list, and to fetch source files transparently from source code management systems.

Depends(*target*, *dependency*)

env.Depends(*target*, *dependency*)

Specifies an explicit dependency; the target file(s) will be rebuilt whenever the dependency file(s) has changed. This should only be necessary for cases where the dependency is not caught by a Scanner for the file.

```
env.Depends('foo', 'other-input-file-for-foo')
```

env.Dictionary([*vars*])

Returns a dictionary object containing copies of all of the construction variables in the environment. If there are any variable names specified, only the specified construction variables are returned in the dictionary.

```
dict = env.Dictionary()
cc_dict = env.Dictionary('CC', 'CCFLAGS', 'CCCOM')
```

Dir(*name*, [*directory*])

env.Dir(*name*, [*directory*])

This returns a Directory Node, an object that represents the specified directory *name*. *name* can be a relative or absolute path. *directory* is an optional directory that will be used as the parent directory. If no *directory* is specified, the current script's directory is used as the parent.

Directory Nodes can be used anywhere you would supply a string as a directory name to a Builder method or function. Directory Nodes have attributes and methods that are useful in many situations; see "File and Directory Nodes," below.

EnsurePythonVersion(*major*, *minor*)

env.EnsurePythonVersion(*major*, *minor*)

Ensure that the Python version is at least *major.minor*. This function will print out an error message and exit SCons with a non-zero exit code if the actual Python version is not late enough.

```
EnsurePythonVersion(2,2)
```

EnsureSConsVersion(*major*, *minor*)

env.EnsureSConsVersion(*major*, *minor*)

Ensure that the SCons version is at least *major.minor*. This function will print out an error message and exit SCons with a non-zero exit code if the actual SCons version is not late enough.

```
EnsureSConsVersion(0,9)
```

Environment([*key*=*value*, ...])

env.Environment([*key*=*value*, ...])

Return a new construction environment initialized with the specified *key*=*value* pairs.

Execute(*action*, [*strfunction*, *varlist*])

env.Execute(*action*, [*strfunction*, *varlist*])

Executes an Action object. The specified *action* may be an Action object (see the section "Action Objects," below, for a complete explanation of the arguments and behavior), or it may be a command-line string, list of commands, or executable Python function, each of which will be converted into an Action object and then executed. The exit value of the command or return value of the Python function will be returned.

Exit([*value*])

env.Exit([*value*])

This tells **scons** to exit immediately with the specified *value*. A default exit value of **0** (zero) is used if no value is specified.

Export(*vars*)

env.Export(*vars*)

This tells **scons** to export a list of variables from the current SConscript file to all other SConscript files. The exported variables are kept in a global collection, so subsequent calls to **Export**() will over-write previous exports that have the same name. Multiple variable names can be passed to **Export**() as separate arguments or as a list. A dictionary can be used to map variables to a different name when exported. Both local variables and global variables can be exported. Examples:

```
env = Environment()
# Make env available for all SConscript files to Import().
Export("env")

package = 'my_name'
# Make env and package available for all SConscript files:.
Export("env", "package")

# Make env and package available for all SConscript files:
Export(["env", "package"])
```

```
# Make env available using the name debug:.
Export({"debug":env})
```

Note that the **SConscript**() function supports an *exports* argument that makes it easier to to export a variable or set of variables to a single SConscript file. See the description of the **SConscript**() function, below.

File(*name*, [*directory*])

env.File(*name*, [*directory*])

This returns a File Node, an object that represents the specified file *name*. *name* can be a relative or absolute path. *directory* is an optional directory that will be used as the parent directory.

File Nodes can be used anywhere you would supply a string as a file name to a Builder method or function. File Nodes have attributes and methods that are useful in many situations; see "File and Directory Nodes," below.

FindFile(*file*, *dirs*)

env.FindFile(*file*, *dirs*)

Search for *file* in the path specified by *dirs*. *file* may be a list of file names or a single file name. In addition to searching for files that exist in the filesytem, this function also searches for derived files that have not yet been built.

```
foo = env.FindFile('foo', ['dir1', 'dir2'])
```

Flatten(*sequence*)

env.Flatten(*sequence*)

Takes a sequence (that is, a Python list or tuple) that may contain nested sequences and returns a flattened list containing all of the individual elements in any sequence. This can be helpful for collecting the lists returned by calls to Builders; other Builders will automatically flatten lists specified as input, but direct Python manipulation of these lists does not:

```
foo = Object('foo.c')
bar = Object('bar.c')

# Because `foo' and `bar' are lists returned by the Object() Builder,
# `objects' will be a list containing nested lists:
objects = ['f1.o', foo, 'f2.o', bar, 'f3.o']

# Passing such a list to another Builder is all right because
# the Builder will flatten the list automatically:
Program(source = objects)

# If you need to manipulate the list directly using Python, you need to
# call Flatten() yourself, or otherwise handle nested lists:
for object in Flatten(objects):
    print str(object)
```

GetBuildPath(*file*, [...])

env.GetBuildPath(*file*, [...])

Returns the **scons** path name (or names) for the specified *file* (or files). The specified *file* or files may be **scons** Nodes or strings representing path names.

GetLaunchDir()

env.GetLaunchDir()

Returns the absolute path name of the directory from which **scons** was initially invoked. This can be useful when using the **−u**, **−U** or **−D** options, which internally change to the directory in which the **SConstruct** file is found.

GetOption(*name*)

env.GetOption(*name*)

This function provides a way to query a select subset of the scons command line options from a SConscript file. See *SetOption*() for a description of the options available.

Help(*text*)

env.Help(*text*)

This specifies help text to be printed if the **-h** argument is given to **scons**. **scons** will exit after printing out the help text.

Ignore(*target*, *dependency*)

env.Ignore(*target*, *dependency*)

The specified dependency file(s) will be ignored when deciding if the target file(s) need to be rebuilt.

```
env.Ignore('foo', 'foo.c')
env.Ignore('bar', ['bar1.h', 'bar2.h'])
```

Import(*vars*)

env.Import(*vars*)

This tells **scons** to import a list of variables into the current SConscript file. This will import variables that were exported with **Export**() or in the *exports* argument to **SConscript**(). Variables exported by **SConscript**() have precedence. Multiple variable names can be passed to **Import**() as separate arguments or as a list. The variable "*" can be used to import all variables. Examples:

```
Import("env")
Import("env", "variable")
Import(["env", "variable"])
Import("*")
```

Install(*dir*, *source*)

env.Install(*dir*, *source*)

Installs one or more files in a destination directory. The file names remain the same.

```
env.Install(dir = '/usr/local/bin', source = ['foo', 'bar'])
```

InstallAs(*target*, *source*)

env.InstallAs(*target*, *source*)

Installs one or more files as specific file names, allowing changing a file name as part of the installation. It is an error if the target and source list different numbers of files.

```
env.InstallAs(target = '/usr/local/bin/foo',
```

```
        source = 'foo_debug')
env.InstallAs(target = ['../lib/libfoo.a', '../lib/libbar.a'],
        source = ['libFOO.a', 'libBAR.a'])
```

Literal(*string*)

env.Literal(*string*)

The specified *string* will be preserved as-is and not have construction variables expanded.

Local(*targets*)

env.Local(*targets*)

The specified *targets* will have copies made in the local tree, even if an already up-to-date copy exists in a repository. Returns a list of the target Node or Nodes.

env.ParseConfig(*command*, [*function*])

Calls the specified *function* to modify the environment as specified by the output of *command* . The default *function* expects the output of a typical **-config command* (for example, **gtk-config**) and parses the returned **-L**, **-l**, **-Wa**, **-Wl**, **-Wp**, **-I** and other options into the **LIBPATH**, **LIBS**, **ASFLAGS**, **LINKFLAGS**, **CPPFLAGS**, **CPPPATH** and **CCFLAGS** variables, respectively. A returned **-pthread** option gets added to both the **CCFLAGS** and **LINKFLAGS** variables.

env.Perforce()

A factory function that returns a Builder object to be used to fetch source files from the Perforce source code management system. The returned Builder is intended to be passed to the **SourceCode** function:

```
env.SourceCode('.', env.Perforce())
```

Perforce uses a number of external environment variables for its operation. Consequently, this function adds the following variables from the user's external environment to the construction environment's ENV dictionary: P4CHARSET, P4CLIENT, P4LANGUAGE, P4PASSWD, P4PORT, P4USER, SYSTEMROOT, USER, and USERNAME.

Platform(*string*)

Returns a callable object that can be used to initialize a construction environment using the platform keyword of the Environment() method:

```
env = Environment(platform = Platform('win32'))
```

env.Platform(*string*)

Applies the callable object for the specified platform *string* to the environment through which the method was called.

```
env.Platform('posix')
```

Note that the **win32** platform adds the **SYSTEMROOT** variable from the user's external environment to the construction environment's **ENV** dictionary. This is so that any executed commands that use sockets to connect with other systems (such as fetching source files from external CVS repository specifications like **:pserver:anonymous@cvs.sourceforge.net:/cvsroot/scons**) will work on Win32 systems.

Precious(*target*, ...)

env.Precious(*target*, ...)

Marks each given *target* as precious so it is not deleted before it is rebuilt. Normally **scons** deletes a target before building it. Multiple targets can be passed in to a single call to **Precious**().

env.Prepend(*key*=*val*, [...])

Appends the specified keyword arguments to the beginning of construction variables in the environment. If the Environment does not have the specified construction variable, it is simply added to the environment. If the values of the construction variable and the keyword argument are the same type, then the two values will be simply added together. Otherwise, the construction variable and the value of the keyword argument are both coerced to lists, and the lists are added together. (See also the Append method, above.)

```
env.Prepend(CCFLAGS = '-g ', FOO = ['foo.yyy'])
```

env.PrependENVPath(*name*, *newpath*, [*envname*, *sep*])

This appends new path elements to the given path in the specified external environment (**ENV** by default). This will only add any particular path once (leaving the first one it encounters and ignoring the rest, to preserve path order), and to help assure this, will normalize all paths (using **os.path.normpath** and **os.path.normcase**). This can also handle the case where the given old path variable is a list instead of a string, in which case a list will be returned instead of a string. Example:

```
print 'before:',env['ENV']['INCLUDE']
include_path = '/foo/bar:/foo'
env.PrependENVPath('INCLUDE', include_path)
print 'after:',env['ENV']['INCLUDE']

yields:
before: /biz:/foo
after: /foo/bar:/foo:/biz
```

env.AppendUnique(*key*=*val*, [...])

Appends the specified keyword arguments to the beginning of construction variables in the environment. If the Environment does not have the specified construction variable, it is simply added to the environment. If the construction variable being appended to is a list, then any value(s) that already exist in the construction variable will *not* be added again to the list.

```
env.PrependUnique(CCFLAGS = '-g', FOO = ['foo.yyy'])
```

env.RCS()

A factory function that returns a Builder object to be used to fetch source files from RCS. The returned Builder is intended to be passed to the **SourceCode** function:

```
env.SourceCode('.', env.RCS())
```

Note that **scons** will fetch source files from RCS subdirectories automatically, so configuring RCS as demonstrated in the above example should only be necessary if you are fetching from RCS,v files in the same directory as the source files, or if you need to explicitly specify RCS for a specific subdirectory.

env.Replace(*key*=*val*, [...])

Replaces construction variables in the Environment with the specified keyword arguments.

env.Replace(CCFLAGS = '-g', FOO = 'foo.xxx')

Repository(*directory*)

env.Repository(*directory*)

Specifies that *directory* is a repository to be searched for files. Multiple calls to **Repository**() are legal, and each one adds to the list of repositories that will be searched.

To **scons**, a repository is a copy of the source tree, from the top-level directory on down, which may contain both source files and derived files that can be used to build targets in the local source tree. The canonical example would be an official source tree maintained by an integrator. If the repository contains derived files, then the derived files should have been built using **scons**, so that the repository contains the necessary signature information to allow **scons** to figure out when it is appropriate to use the repository copy of a derived file, instead of building one locally.

Note that if an up-to-date derived file already exists in a repository, **scons** will *not* make a copy in the local directory tree. In order to guarantee that a local copy will be made, use the **Local()** method.

Return(*vars*)

This tells **scons** what variable(s) to use as the return value(s) of the current SConscript file. These variables will be returned to the "calling" SConscript file as the return value(s) of **SConscript**(). Multiple variable names should be passed to **Return**() as a list. Example:

```
Return("foo")
Return(["foo", "bar"])
```

Scanner(*function*, [*argument*, *keys*, *path_function*, *node_class*, *node_factory*, *scan_check*, *recursive*])

env.Scanner(*function*, [*argument*, *keys*, *path_function*, *node_class*, *node_factory*, *scan_check*, *recursive*])

Creates a Scanner object for the specified *function*. See the section "Scanner Objects," below, for a complete explanation of the arguments and behavior.

env.SCCS()

A factory function that returns a Builder object to be used to fetch source files from SCCS. The returned Builder is intended to be passed to the **SourceCode** function:

```
env.SourceCode('.', env.SCCS())
```

Note that **scons** will fetch source files from SCCS subdirectories automatically, so configuring SCCS as demonstrated in the above example should only be necessary if you are fetching from *s.SCCS* files in the same directory as the source files, or if you need to explicitly specify SCCS for a specific subdirectory.

SConscript(*scripts*, [*exports*, *build_dir*, *src_dir*, *duplicate*])

env.SConscript(*scripts*, [*exports*, *build_dir*, *src_dir*, *duplicate*])

SConscript(dirs=*subdirs*, [name=*script*, *exports*, *build_dir*, *src_dir*, *duplicate*])

env.SConscript(dirs=*subdirs*, [name=*script*, *exports*, *build_dir*, *src_dir*, *duplicate*])

This tells **scons** to execute one or more subsidiary SConscript (configuration) files. There are two ways to call the **SConscript**() function.

The first way you can call **SConscript**() is to explicitly specify one or more *scripts* as the first argument. A single script may be specified as a string; multiple scripts must be specified as a list (either explicitly or as created by a function like **Split**()).

The second way you can call **SConscript**() is to specify a list of (sub)directory names as a dirs=*subdirs* keyword argument. In this case, **scons** will, by default, execute a subsidiary configuration file named **SConscript** in each of the specified directories. You may specify a name other than **SConscript** by supplying an optional name=*script* keyword argument.

The optional *exports* argument provides a list of variable names or a dictionary of named values to export to the *script(s)*. These variables are locally exported only to the specified *script(s)*, and do not affect the global pool of variables used by the **Export**() function. The subsidiary *script(s)* must use the **Import**() function to import the variables.

The optional *build_dir* argument specifies that all of the target files (for example, object files and executables) that would normally be built in the subdirectory in which *script* resides should actually be built in *build_dir*. *build_dir* is interpreted relative to the directory of the calling SConscript file.

The optional *src_dir* argument specifies that the source files from which the target files should be built can be found in *src_dir*. *src_dir* is interpreted relative to the directory of the calling SConscript file.

By default, **scons** will link or copy (depending on the platform) all the source files into the build directory. This behavior may be disabled by setting the optional *duplicate* argument to 0 (it is set to 1 by default), in which case **scons** will refer directly to the source files in their source directory when building target files. (Setting *duplicate*=0 is usually safe, and always more efficient than the default of *duplicate*=1, but it may cause build problems in certain end-cases, such as compiling from source files that are generated by the build.)

Any variables returned by *script* using **Return**() will be returned by the call to **SConscript**().

Examples:

```
SConscript('subdir/SConscript')
foo = SConscript('sub/SConscript', exports='env')
SConscript('dir/SConscript', exports=['env', 'variable'])
SConscript('src/SConscript', build_dir='build', duplicate=0)
SConscript('bld/SConscript', src_dir='src', exports='env variable')
SConscript(dirs=['sub1', 'sub2'])
SConscript(dirs=['sub3', 'sub4'], name='MySConscript')
```

SConscriptChdir(*value*)

env.SConscriptChdir(*value*)

By default, **scons** changes its working directory to the directory in which each subsidiary SConscript file lives. This behavior may be disabled by specifying either:

```
SConscriptChdir(0)
env.SConscriptChdir(0)
```

in which case **scons** will stay in the top-level directory while reading all SConscript files. (This may be necessary when building from repositories, when all the directories in which SConscript files may be found don't necessarily exist locally.)

You may enable and disable this ability by calling SConscriptChdir() multiple times:

```
env = Environment()
SConscriptChdir(0)
SConscript('foo/SConscript')        # will not chdir to foo
env.SConscriptChdir(1)
SConscript('bar/SConscript')        # will chdir to bar
```

SConsignFile([*file*,*dbm_module*])

env.SConsignFile([*file*,*dbm_module*])

This tells **scons** to store all file signatures in the specified *file*. If the *file* is omitted, **.sconsign.dbm** is used by default. If *file* is not an absolute path name, the file is placed in the same directory as the top-level **SConstruct** file.

The optional *dbm_module* argument can be used to specify which Python database module The default is to use a custom **SCons.dblite** module that uses pickled Python data structures, and which works on all Python versions from 1.5.2 on.

Examples:

```
# Stores signatures in ".sconsign.dbm"
# in the top-level SConstruct directory.
SConsignFile()

# Stores signatures in the file "etc/scons-signatures"
# relative to the top-level SConstruct directory.
SConsignFile("etc/scons-signatures")

# Stores signatures in the specified absolute file name.
SConsignFile("/home/me/SCons/signatures")
```

SetOption(*name*, *value*)

env.SetOption(*name*, *value*)

This function provides a way to set a select subset of the scons command line options from a SConscript file. The options supported are: **clean** which corresponds to -c, --clean, and --remove; **duplicate** which corresponds to --duplicate; **implicit_cache** which corresponds to --implicit-cache; **max_drift** which corresponds to --max-drift; **num_jobs** which corresponds to -j and --jobs. See the documentation for the corresponding command line object for information about each specific option. Example:

```
SetOption('max_drift', 1)
```

SideEffect(*side_effect*, *target*)

env.SideEffect(*side_effect*, *target*)

Declares *side_effect* as a side effect of building *target*. Both *side_effect* and *target* can be a list, a file name, or a node. A side effect is a target that is created as a side effect of building other targets. For example, a Windows PDB file is created as a side effect of building the .obj files for a static library. If a target is a side effect of multiple build commands, **scons** will ensure that only one set of commands is executed at a time. Consequently, you only need to use this method for side-effect targets that are built as a result of multiple build commands.

SourceCode(*entries*, *builder*)

env.SourceCode(*entries*, *builder*)

Arrange for non-existent source files to be fetched from a source code management system using the specified *builder*. The specified *entries* may be a Node, string or list of both, and may represent either individual source files or directories in which source files can be found.

For any non-existent source files, **scons** will search up the directory tree and use the first **SourceCode** builder it finds. The specified *builder* may be **None**, in which case **scons** will not use a builder to fetch source files for the specified *entries*, even if a **SourceCode** builder has been specified for a directory higher up the tree.

scons will, by default, fetch files from SCCS or RCS subdirectories without explicit configuration. This takes some extra processing time to search for the necessary source code management files on disk. You can avoid these extra searches and speed up your build a little by disabling these searches as follows:

```
env.SourceCode('.', None)
```

Note that if the specified *builder* is one you create by hand, it must have an associated construction environment to use when fetching a source file.

scons provides a set of canned factory functions that return appropriate Builders for various popular source code management systems. Canonical examples of invocation include:

```
env.SourceCode('.', env.BitKeeper('/usr/local/BKsources'))
env.SourceCode('src', env.CVS('/usr/local/CVSROOT'))
env.SourceCode('/', env.RCS())
env.SourceCode(['f1.c', 'f2.c'], env.SCCS())
env.SourceCode('no_source.c', None)
```

SourceSignatures(*type*)

env.SourceSignatures(*type*)

This function tells SCons what type of signature to use for source files: **MD5** or **timestamp**. If the environment method is used, the specified type of source signature is only used when deciding whether targets built with that environment are up-to-date or must be rebuilt. If the global function is used, the specified type of source signature becomes the default used for all decisions about whether targets are up-to-date.

"MD5" means the signature of a source file is the MD5 checksum of its contents. "timestamp" means the signature of a source file is its timestamp (modification time). There is no different between the two behaviors for Python **Value**() node objects. "MD5" signatures take longer to compute, but are more accurate than "timestamp" signatures. The default is "MD5".

Split(*arg*)

env.Split(*arg*)

Returns a list of file names or other objects. If arg is a string, it will be split on strings of white-space characters within the string, making it easier to write long lists of file names. If arg is already a list, the list will be returned untouched. If arg is any other type of object, it will be returned as a list containing just the object.

```
files = Split("f1.c f2.c f3.c")
files = env.Split("f4.c f5.c f6.c")
files = Split("""
```

```
        f7.c
        f8.c
        f9.c
""")
```

TargetSignatures(*type*)

env.TargetSignatures(*type*)

This function tells SCons what type of signatures to use for target files: **build** or **content**. If the environment method is used, the specified type of signature is only used for targets built with that environment. If the global function is used, the specified type of signature becomes the default used for all target files that don't have an explicit target signature type specified for their environments.

"build" means the signature of a target file is made by concatenating all of the signatures of all its source files. "content" means the signature of a target file is an MD5 checksum of its contents. "build" signatures are usually faster to compute, but "content" signatures can prevent unnecessary rebuilds when a target file is rebuilt to the exact same contents as the previous build. The default is "build".

Tool(*string,*toolpath=[]*)*

Returns a callable object that can be used to initialize a construction environment using the tools keyword of the Environment() method. The object may be called with a construction environment as an argument, in which case the object will be add the necessary variables to the construction environment and the name of the tool will be added to the **$TOOLS** construction variable.

```
env = Environment(tools = [ Tool('msvc') ])

env = Environment()
t = Tool('msvc')
t(env)  # adds 'msvc' to the TOOLS variable
u = Tool('opengl', toolpath = ['tools'])
u(env)  # adds 'opengl' to the TOOLS variable
```

env.Tool(*string*[,*toolpath]*)

Applies the callable object for the specified tool *string* to the environment through which the method was called.

```
env.Tool('gcc')
env.Tool('opengl', toolpath = ['build/tools'])
```

Value(*value*)

env.Value(*value*)

Returns a Node object representing the specified Python value. Value nodes can be used as dependencies of targets. If the result of calling **str(**value**)** changes between SCons runs, any targets depending on **Value(**value**)** will be rebuilt. When using timestamp source signatures, Value nodes' timestamps are equal to the system time when the node is created.

```
def create(target, source, env):
   f = open(str(target[0]), 'wb')
   f.write('prefix=' + source[0].get_contents())

prefix = ARGUMENTS.get('prefix', '/usr/local')
env = Environment()
```

```
env['BUILDERS']['Config'] = Builder(action = create)
env.Config(target = 'package-config', source = Value(prefix))
```

WhereIs(*program*, [*path*, *pathext*, *reject*])

env.WhereIs(*program*, [*path*, *pathext*, *reject*])

Searches for the specified executable *program,* returning the full path name to the program if it is found, and returning None if not. Searches the specified *path,* the value of the calling environment's PATH (env['ENV']['PATH']), or the user's current external PATH (os.environ['PATH']) by default. On Win32 systems, searches for executable programs with any of the file extensions listed in the specified *pathext,* the calling environment's PATHEXT (env['ENV']['PATHEXT']) or the user's current PATHEXT (os.environ['PATHEXT']) by default. Will not select any path name or names in the specified *reject* list, if any.

SConscript Variables

In addition to the global functions and methods, **scons** supports a number of Python variables that can be used in SConscript files to affect how you want the build to be performed.

ARGLIST

A list *keyword*=*value* arguments specified on the command line. Each element in the list is a tuple containing the (*keyword*,*value*) of the argument. The separate *keyword* and *value* elements of the tuple can be accessed by subscripting for element **[0]** and **[1]** of the tuple, respectively.

```
print "first keyword, value =", ARGLIST[0][0], ARGLIST[0][1]
print "second keyword, value =", ARGLIST[1][0], ARGLIST[1][1]
third_tuple = ARGLIST[2]
print "third keyword, value =", third_tuple[0], third_tuple[1]
for key, value in ARGLIST:
    # process key and value
```

ARGUMENTS

A dictionary of all the *keyword*=*value* arguments specified on the command line. The dictionary is not in order, and if a given keyword has more than one value assigned to it on the command line, the last (right-most) value is the one in the **ARGUMENTS** dictionary.

```
if ARGUMENTS.get('debug', 0):
    env = Environment(CCFLAGS = '-g')
else:
    env = Environment()
```

BUILD_TARGETS

A list of the targets which **scons** will actually try to build, regardless of whether they were specified on the command line or via the **Default**() function or method. The elements of this list may be strings *or* nodes, so you should run the list through the Python **str** function to make sure any Node path names are converted to strings.

Because this list may be taken from the list of targets specified using the **Default**() function or method, the contents of the list may change on each successive call to **Default**(). See the **DEFAULT_TARGETS** list, below, for additional information.

```
if 'foo' in BUILD_TARGETS:
    print "Don't forget to test the `foo' program!"
```

```
if 'special/program' in BUILD_TARGETS:
    SConscript('special')
```

Note that the **BUILD_TARGETS** list only contains targets expected listed on the command line or via calls to the **Default**() function or method. It does *not* contain all dependent targets that will be built as a result of making the sure the explicitly-specified targets are up to date.

COMMAND_LINE_TARGETS
A list of the targets explicitly specified on the command line. If there are no targets specified on the command line, the list is empty. This can be used, for example, to take specific actions only when a certain target or targets is explicitly being built:

```
if 'foo' in COMMAND_LINE_TARGETS:
    print "Don't forget to test the `foo' program!"
if 'special/program' in COMMAND_LINE_TARGETS:
    SConscript('special')
```

DEFAULT_TARGETS
A list of the target *nodes* that have been specified using the **Default**() function or method. The elements of the list are nodes, so you need to run them through the Python **str** function to get at the path name for each Node.

```
print str(DEFAULT_TARGETS[0])
if 'foo' in map(str, DEFAULT_TARGETS):
    print "Don't forget to test the `foo' program!"
```

The contents of the **DEFAULT_TARGETS** list change on on each successive call to the **Default**() function:

```
print map(str, DEFAULT_TARGETS)   # originally []
Default('foo')
print map(str, DEFAULT_TARGETS)   # now a node ['foo']
Default('bar')
print map(str, DEFAULT_TARGETS)   # now a node ['foo', 'bar']
Default(None)
print map(str, DEFAULT_TARGETS)   # back to []
```

Consequently, be sure to use **DEFAULT_TARGETS** only after you've made all of your **Default**() calls, or else simply be careful of the order of these statements in your SConscript files so that you don't look for a specific default target before it's actually been added to the list.

Construction Variables

A construction environment has an associated dictionary of *construction variables* that are used by built-in or user-supplied build rules. Construction variables must follow the same rules for Python identifiers: the initial character must be an underscore or letter, followed by any number of underscores, letters, or digits.

A number of useful construction variables are automatically defined by scons for each supported platform, and additional construction variables can be defined by the user. The following is a list of the automatically defined construction variables:

AR The static library archiver.

ARCOM
The command line used to generate a static library from object files.

ARFLAGS
General options passed to the static library archiver.

AS
The assembler.

ASCOM
The command line used to generate an object file from an assembly-language source file.

ASFLAGS
General options passed to the assembler.

ASPPCOM
The command line used to assemble an assembly-language source file into an object file after first running the file through the C preprocessor. Any options specified in the $ASFLAGS and $CPPFLAGS construction variables are included on this command line.

BIBTEX
The bibliography generator for the TeX formatter and typesetter and the LaTeX structured formatter and typesetter.

BIBTEXCOM
The command line used to call the bibliography generator for the TeX formatter and typesetter and the LaTeX structured formatter and typesetter.

BIBTEXFLAGS
General options passed to the bibliography generator for the TeX formatter and typesetter and the LaTeX structured formatter and typesetter.

BITKEEPER
The BitKeeper executable.

BITKEEPERCOM
The command line for fetching source files using BitKEeper.

BITKEEPERGET
The command ($BITKEEPER) and subcommand for fetching source files using BitKeeper.

BITKEEPERGETFLAGS
Options that are passed to the BitKeeper **get** subcommand.

BUILDERS
A dictionary mapping the names of the builders available through this environment to underlying Builder objects. Builders named Alias, CFile, CXXFile, DVI, Library, Object, PDF, PostScript, and Program are available by default. If you initialize this variable when an Environment is created:

```
env = Environment(BUILDERS = {'NewBuilder' : foo})
```

the default Builders will no longer be available. To use a new Builder object in addition to the default Builders, add your new Builder object like this:

```
env = Environment()
env.Append(BUILDERS = {'NewBuilder' : foo})
```

or this:

```
env = Environment()
env['BUILDERS]['NewBuilder'] = foo
```

CC
The C compiler.

CCCOM
The command line used to compile a C source file to a (static) object file. Any options specified in the $CCFLAGS and $CPPFLAGS construction variables are included on this command line.

CCFLAGS
General options that are passed to the C compiler.

CFILESUFFIX
The suffix for C source files. This is used by the internal CFile builder when generating C files from Lex (.l) or YACC (.y) input files. The default suffix, of course, is *.c* (lower case). On case-insensitive systems (like Win32), SCons also treats *.C* (upper case) files as C files.

CCVERSION
The version number of the C compiler. This may or may not be set, depending on the specific C compiler being used.

_concat
A function used to produce variables like $_CPPINCFLAGS. It takes four or five arguments: a prefix to concatenate onto each element, a list of elements, a suffix to concatenate onto each element, an environment for variable interpolation, and an optional function that will be called to transform the list before concatenation.

```
env['_CPPINCFLAGS'] = '$( ${_concat(INCPREFIX, CPPPATH, INCSUFFIX, __env__, RDirs)} $)',
```

CPPDEFINES
A platform independent specification of C preprocessor definitions. The definitions will be added to command lines through the automatically-generated $_CPPDEFFLAGS construction variable (see below), which is constructed according to the type of value of $CPPDEFINES:

If $CPPDEFINES is a string, the values of the $CPPDEFPREFIX and $CPPDEFSUFFIX construction variables will be added to the beginning and end.

```
# Will add -Dxyz to POSIX compiler command lines,
# and /Dxyz to Microsoft Visual C++ command lines.
env = Environment(CPPDEFINES='xyz')
```

If $CPPDEFINES is a list, the values of the $CPPDEFPREFIX and $CPPDEFSUFFIX construction variables will be appended to the beginning and end of each element in the list. If any element is a list or tuple, then the first item is the name being defined and the second item is its value:

```
# Will add -DB=2 -DA to POSIX compiler command lines,
# and /DB=2 /DA to Microsoft Visual C++ command lines.
env = Environment(CPPDEFINES=[('B', 2), 'A'])
```

If $CPPDEFINES is a dictionary, the values of the $CPPDEFPREFIX and $CPPDEFSUFFIX construction variables will be appended to the beginning and end of each item from the dictionary. The key of each dictionary item is a name being defined to the dictionary item's corresponding value; if the value is **None**, then the name is defined without an explicit value. Note that the resulting flags are sorted by keyword to ensure that the order of the options on the command line is consistent each time **scons**
is run.

```
# Will add -DA -DB=2 to POSIX compiler command lines,
# and /DA /DB=2 to Microsoft Visual C++ command lines.
env = Environment(CPPDEFINES={'B':2, 'A':None})
```

_CPPDEFFLAGS
An automatically-generated construction variable containing the C preprocessor command-line options to define values. The value of $_CPPDEFFLAGS is created by appending $CPPDEFPREFIX and $CPPDEFSUFFIX to the beginning and end of each directory in $CPPDEFINES.

CPPDEFPREFIX
The prefix used to specify preprocessor definitions on the C compiler command line. This will be appended to the beginning of each definition in the $CPPDEFINES construction variable when the $_CPPDEFFLAGS variable is automatically generated.

CPPDEFSUFFIX
The suffix used to specify preprocessor definitions on the C compiler command line. This will be appended to the end of each definition in the $CPPDEFINES construction variable when the $_CPPDEFFLAGS variable is automatically generated.

CPPFLAGS
User-specified C preprocessor options. These will be included in any command that uses the C preprocessor, including not just compilation of C and C++ source files via the $CCCOM, $SHCCCOM, $CXXCOM and $SHCXXCOM command lines, but also the $FORTRANPPCOM, $SHFORTRANPPCOM, $F77PPCOM and $SHF77PPCOM command lines used to compile a Fortran source file, and the $ASPPCOM command line used to assemble an assembly language source file, after first running each file through the C preprocessor. Note that this variable does *not* contain **-I** (or similar) include search path options that scons generates automatically from $CPPPATH. See **_CPPINCFLAGS**, below, for the variable that expands to those options.

_CPPINCFLAGS
An automatically-generated construction variable containing the C preprocessor command-line options for specifying directories to be searched for include files. The value of $_CPPINCFLAGS is created by appending $INCPREFIX and $INCSUFFIX to the beginning and end of each directory in $CPPPATH.

CPPPATH
The list of directories that the C preprocessor will search for include directories. The C/C++ implicit dependency scanner will search these directories for include files. Don't explicitly put include directory arguments in CCFLAGS or CXXFLAGS because the result will be non-portable and the directories will not be searched by the dependency scanner. Note: directory names in CPPPATH will be looked-up relative to the SConscript directory when they are used in a command. To force **scons** to look-up a directory relative to the root of the source tree use #:

```
env = Environment(CPPPATH='#/include')
```

The directory look-up can also be forced using the **Dir**() function:

```
include = Dir('include')
env = Environment(CPPPATH=include)
```

The directory list will be added to command lines through the automatically-generated $_CPPINCFLAGS construction variable, which is constructed by appending the values of the $INCPREFIX and $INCSUFFIX construction variables to the beginning and end of each directory in $CPPPATH. Any command lines you define that need the CPPPATH directory list should include $_CPPINCFLAGS:

```
env = Environment(CCCOM="my_compiler $_CPPINCFLAGS -c -o $TARGET $SOURCE")
```

CPPSUFFIXES
The list of suffixes of files that will be scanned for C preprocessor implicit dependencies (#include lines). The default list is:

```
[".c", ".C", ".cxx", ".cpp", ".c++", ".cc",
 ".h", ".H", ".hxx", ".hpp", ".hh",
 ".F", ".fpp", ".FPP",
 ".S", ".spp", ".SPP"]
```

CVS
The CVS executable.

CVSCOFLAGS
Options that are passed to the CVS checkout subcommand.

CVSCOM
The command line used to fetch source files from a CVS repository.

CVSFLAGS
General options that are passed to CVS. By default, this is set to "-d $CVSREPOSITORY" to specify from where the files must be fetched.

CVSREPOSITORY
The path to the CVS repository. This is referenced in the default $CVSFLAGS value.

CXX
The C++ compiler.

CXXFILESUFFIX
The suffix for C++ source files. This is used by the internal CXXFile builder when generating C++ files from Lex (.ll) or YACC (.yy) input files. The default suffix is *.cc*. SCons also treats files with the suffixes *.cpp*, *.cxx*, *.c++*, and *.C++* as C++ files. On case-sensitive systems (Linux, UNIX, and other POSIX-alikes), SCons also treats *.C* (upper case) files as C++ files.

CXXCOM
The command line used to compile a C++ source file to an object file. Any options specified in the $CXXFLAGS and $CPPFLAGS construction variables are included on this command line.

CXXFLAGS
General options that are passed to the C++ compiler.

CXXVERSION
The version number of the C++ compiler. This may or may not be set, depending on the specific C++ compiler being used.

Dir A function that converts a file name into a Dir instance relative to the target being built.

DSUFFIXES
The list of suffixes of files that will be scanned for imported D package files. The default list is:

['.d']

DVIPDF
The TeX DVI file to PDF file converter.

DVIPDFFLAGS
General options passed to the TeX DVI file to PDF file converter.

DVIPDFCOM
The command line used to convert TeX DVI files into a PDF file.

DVIPS The TeX DVI file to PostScript converter.

DVIPSFLAGS
General options passed to the TeX DVI file to PostScript converter.

ENV A dictionary of environment variables to use when invoking commands. When ENV is used in a command all list values will be joined using the path separator and any other non-string values will simply be coerced to a string. Note that, by default, **scons** does *not* propagate the environment in force when you execute **scons** to the commands used to build target files. This is so that builds will be guaranteed repeatable regardless of the environment variables set at the time **scons** is invoked.

If you want to propagate your environment variables to the commands executed to build target files, you must do so explicitly:

```
import os
env = Environment(ENV = os.environ)
```

Note that you can choose only to propagate certain environment variables. A common example is the system **PATH** environment variable, so that **scons** uses the same utilities as the invoking shell (or other process):

```
import os
env = Environment(ENV = {'PATH' : os.environ['PATH']})
```

ESCAPE
A function that will be called to escape shell special characters in command lines. The function should take one argument: the command line string to escape; and should return the escaped command line.

F77 The Fortran 77 compiler. You should normally set the $FORTRAN variable, which specifies the default Fortran compiler for all Fortran versions. You only need to set $F77 if you need to use a specific compiler or compiler version for Fortran 77 files.

F77COM
The command line used to compile a Fortran 77 source file to an object file. You only need to set $F77COM if you need to use a specific command line for Fortran 77 files. You should normally set the $FORTRANCOM variable, which specifies the default command line for all Fortran versions.

F77FLAGS
General user-specified options that are passed to the Fortran 77 compiler. Note that this variable does *not* contain **-I** (or similar) include search path options that scons generates automatically from $F77PATH. See **_F77INCFLAGS**, below, for the variable that expands to those options. You only need to set $F77FLAGS if you need to define specific user options for Fortran 77 files. You should normally set the $FORTRANFLAGS variable, which specifies the user-specified options passed to the default Fortran compiler for all Fortran versions.

_F77INCFLAGS
An automatically-generated construction variable containing the Fortran 77 compiler command-line options for specifying directories to be searched for include files. The value of $_F77INCFLAGS is created by appending $INCPREFIX and $INCSUFFIX to the beginning and end of each directory in $F77PATH.

F77PATH
The list of directories that the Fortran 77 compiler will search for include directories. The implicit dependency scanner will search these directories for include files. Don't explicitly put include directory arguments in $F77FLAGS because the result will be non-portable and the directories will not be searched by the dependency scanner. Note: directory names in $F77PATH will be looked-up relative to the SConscript directory when they are used in a command. To force **scons** to look-up a directory relative to the root of the source tree use #: You only need to set $F77PATH if you need to define a specific include path for Fortran 77 files. You should normally set the $FORTRANPATH variable, which specifies the include path for the default Fortran compiler for all Fortran versions.

```
env = Environment(F77PATH='#/include')
```

The directory look-up can also be forced using the **Dir**() function:

```
include = Dir('include')
```

env = Environment(F77PATH=include)

The directory list will be added to command lines through the automatically-generated $_F77INCFLAGS construction variable, which is constructed by appending the values of the $INCPREFIX and $INCSUFFIX construction variables to the beginning and end of each directory in $F77PATH. Any command lines you define that need the F77PATH directory list should include $_F77INCFLAGS:

env = Environment(F77COM="my_compiler $_F77INCFLAGS -c -o $TARGET $SOURCE")

F77PPCOM
The command line used to compile a Fortran 77 source file to an object file after first running the file through the C preprocessor. Any options specified in the $F77FLAGS and $CPPFLAGS construction variables are included on this command line. You only need to set $F77PPCOM if you need to use a specific C-preprocessor command line for Fortran 77 files. You should normally set the $FORTRANPPCOM variable, which specifies the default C-preprocessor command line for all Fortran versions.

F90 The Fortran 90 compiler. You should normally set the $FORTRAN variable, which specifies the default Fortran compiler for all Fortran versions. You only need to set $F90 if you need to use a specific compiler or compiler version for Fortran 90 files.

F90COM
The command line used to compile a Fortran 90 source file to an object file. You only need to set $F90COM if you need to use a specific command line for Fortran 90 files. You should normally set the $FORTRANCOM variable, which specifies the default command line for all Fortran versions.

F90FLAGS
General user-specified options that are passed to the Fortran 90 compiler. Note that this variable does *not* contain **-I** (or similar) include search path options that scons generates automatically from $F90PATH. See **_F90INCFLAGS**, below, for the variable that expands to those options. You only need to set $F90FLAGS if you need to define specific user options for Fortran 90 files. You should normally set the $FORTRANFLAGS variable, which specifies the user-specified options passed to the default Fortran compiler for all Fortran versions.

_F90INCFLAGS
An automatically-generated construction variable containing the Fortran 90 compiler command-line options for specifying directories to be searched for include files. The value of $_F90INCFLAGS is created by appending $INCPREFIX and $INCSUFFIX to the beginning and end of each directory in $F90PATH.

F90PATH
The list of directories that the Fortran 90 compiler will search for include directories. The implicit dependency scanner will search these directories for include files. Don't explicitly put include directory arguments in $F90FLAGS because the result will be non-portable and the directories will not be searched by the dependency scanner. Note: directory names in $F90PATH will be looked-up relative to the SConscript directory when they are used in a command. To force **scons** to look-up a directory relative to the root of the source tree use #: You only need to set $F90PATH if you need to define a specific include path for Fortran 90 files. You should normally set the $FORTRANPATH variable, which specifies the include path for the default Fortran compiler for all Fortran versions.

```
env = Environment(F90PATH='#/include')
```

The directory look-up can also be forced using the **Dir**() function:

```
include = Dir('include')
env = Environment(F90PATH=include)
```

The directory list will be added to command lines through the automatically-generated $_F90INCFLAGS construction variable, which is constructed by appending the values of the $INCPREFIX and $INCSUFFIX construction variables to the beginning and end of each directory in $F90PATH. Any command lines you define that need the F90PATH directory list should include $_F90INCFLAGS:

```
env = Environment(F90COM="my_compiler $_F90INCFLAGS -c -o $TARGET $SOURCE")
```

F90PPCOM
The command line used to compile a Fortran 90 source file to an object file after first running the file through the C preprocessor. Any options specified in the $F90FLAGS and $CPPFLAGS construction variables are included on this command line. You only need to set $F90PPCOM if you need to use a specific C-preprocessor command line for Fortran 90 files. You should normally set the $FORTRANPPCOM variable, which specifies the default C-preprocessor command line for all Fortran versions.

F95 The Fortran 95 compiler. You should normally set the $FORTRAN variable, which specifies the default Fortran compiler for all Fortran versions. You only need to set $F95 if you need to use a specific compiler or compiler version for Fortran 95 files.

F95COM
The command line used to compile a Fortran 95 source file to an object file. You only need to set $F95COM if you need to use a specific command line for Fortran 95 files. You should normally set the $FORTRANCOM variable, which specifies the default command line for all Fortran versions.

F95FLAGS
General user-specified options that are passed to the Fortran 95 compiler. Note that this variable does *not* contain **-I** (or similar) include search path options that scons generates automatically from $F95PATH. See **_F95INCFLAGS**, below, for the variable that expands to those options. You only need to set $F95FLAGS if you need to define specific user options for Fortran 95 files. You should normally set the $FORTRANFLAGS variable, which specifies the user-specified options passed to the default Fortran compiler for all Fortran versions.

_F95INCFLAGS
An automatically-generated construction variable containing the Fortran 95 compiler command-line options for specifying directories to be searched for include files. The value of $_F95INCFLAGS is created by appending $INCPREFIX and $INCSUFFIX to the beginning and end of each directory in $F95PATH.

F95PATH
The list of directories that the Fortran 95 compiler will search for include directories. The implicit dependency scanner will search these directories for include files. Don't explicitly put include directory arguments in $F95FLAGS because the result will be non-portable and the directories

will not be searched by the dependency scanner. Note: directory names in $F95PATH will be looked-up relative to the SConscript directory when they are used in a command. To force **scons** to look-up a directory relative to the root of the source tree use #: You only need to set $F95PATH if you need to define a specific include path for Fortran 95 files. You should normally set the $FORTRANPATH variable, which specifies the include path for the default Fortran compiler for all Fortran versions.

```
env = Environment(F95PATH='#/include')
```

The directory look-up can also be forced using the **Dir**() function:

```
include = Dir('include')
env = Environment(F95PATH=include)
```

The directory list will be added to command lines through the automatically-generated $_F95INCFLAGS construction variable, which is constructed by appending the values of the $INCPREFIX and $INCSUFFIX construction variables to the beginning and end of each directory in $F95PATH. Any command lines you define that need the F95PATH directory list should include $_F95INCFLAGS:

```
env = Environment(F95COM="my_compiler $_F95INCFLAGS -c -o $TARGET $SOURCE")
```

F95PPCOM
The command line used to compile a Fortran 95 source file to an object file after first running the file through the C preprocessor. Any options specified in the $F95FLAGS and $CPPFLAGS construction variables are included on this command line. You only need to set $F95PPCOM if you need to use a specific C-preprocessor command line for Fortran 95 files. You should normally set the $FORTRANPPCOM variable, which specifies the default C-preprocessor command line for all Fortran versions.

FORTRAN
The default Fortran compiler for all versions of Fortran.

FORTRANCOM
The command line used to compile a Fortran source file to an object file. By default, any options specified in the $FORTRANFLAGS, $CPPFLAGS, $_CPPDEFFLAGS, $_FORTRANMODFLAG, and $_FORTRANINCFLAGS construction variables are included on this command line.

FORTRANFLAGS
General user-specified options that are passed to the Fortran compiler. Note that this variable does *not* contain **-I** (or similar) include or module search path options that scons generates automatically from $FORTRANPATH. See **_FORTRANINCFLAGS**and**_FORTRANMODFLAGS,** below, for the variables that expand those options.

_FORTRANINCFLAGS
An automatically-generated construction variable containing the Fortran compiler command-line options for specifying directories to be searched for include files and module files. The value of $_FORTRANINCFLAGS is created by prepending/appending $INCPREFIX and $INCSUFFIX to the beginning and end of each directory in $FORTRANPATH.

FORTRANMODDIR

Directory location where the Fortran compiler should place any module files it generates. This variable is empty, by default. Some Fortran compilers will internally append this directory in the search path for module files, as well

FORTRANMODDIRPREFIX

The prefix used to specify a module directory on the Fortran compiler command line. This will be appended to the beginning of the directory in the $FORTRANMODDIR construction variables when the $_FORTRANMODFLAG variables is automatically generated.

FORTRANMODDIRSUFFIX

The suffix used to specify a module directory on the Fortran compiler command line. This will be appended to the beginning of the directory in the $FORTRANMODDIR construction variables when the $_FORTRANMODFLAG variables is automatically generated.

FORTRANMODFLAG

An automatically-generated construction variable containing the Fortran compiler command-line option for specifying the directory location where the Fortran compiler should place any module files that happen to get generated during compilation. The value of $_FORTRANMODFLAG is created by prepending/appending $FORTRANMODDIRPREFIX and $FORTRANMODDIRSUFFIX to the beginning and end of the directory in $FORTRANMODDIR.

FORTRANMODPREFIX

The module file prefix used by the Fortran compiler. SCons assumes that the Fortran compiler follows the quasi-standard naming convention for module files of *<module_name>.mod.* As a result, this variable is left empty, by default. For situations in which the compiler does not necessarily follow the normal convention, the user may use this variable. Its value will be appended to every module file name as scons attempts to resolve dependencies.

FORTRANMODSUFFIX

The module file suffix used by the Fortran compiler. SCons assumes that the Fortran compiler follows the quasi-standard naming convention for module files of *<module_name>.mod.* As a result, this variable is set to ".mod", by default. For situations in which the compiler does not necessarily follow the normal convention, the user may use this variable. Its value will be appended to every module file name as scons attempts to resolve dependencies.

FORTRANPATH

The list of directories that the Fortran compiler will search for include files and (for some compilers) module files. The Fortran implicit dependency scanner will search these directories for include files (but not module files since they are autogenerated and, as such, may not actually exist at the time the scan takes place). Don't explicitly put include directory arguments in FORTRANFLAGS because the result will be non-portable and the directories will not be searched by the dependency scanner. Note: directory names in FORTRANPATH will be looked-up relative to the SConscript directory when they are used in a command. To force **scons** to look-up a directory relative to the root of the source tree use #:

```
env = Environment(FORTRANPATH='#/include')
```

The directory look-up can also be forced using the **Dir**() function:

```
include = Dir('include')
env = Environment(FORTRANPATH=include)
```

The directory list will be added to command lines through the automatically-generated $_FORTRANINCFLAGS construction variable, which is constructed by appending the values of the $INCPREFIX and $INCSUFFIX construction variables to the beginning and end of each directory in $FORTRANPATH. Any command lines you define that need the FORTRANPATH directory list should include $_FORTRANINCFLAGS:

```
env = Environment(FORTRANCOM="my_compiler $_FORTRANINCFLAGS -c -o $TARGET $SOURCE")
```

FORTRANPPCOM
: The command line used to compile a Fortran source file to an object file after first running the file through the C preprocessor. By default, any options specified in the $FORTRANFLAGS, $CPPFLAGS, _CPPDEFFLAGS, $_FORTRANMODFLAG, and $_FORTRANINCFLAGS construction variables are included on this command line.

FORTRANSUFFIXES
: The list of suffixes of files that will be scanned for Fortran implicit dependencies (INCLUDE lines & USE statements). The default list is:

```
[".f", ".F", ".for", ".FOR", ".ftn", ".FTN", ".fpp", ".FPP",
".f77", ".F77", ".f90", ".F90", ".f95", ".F95"]
```

File
: A function that converts a file name into a File instance relative to the target being built.

GS
: The Ghostscript program used to convert PostScript to PDF files.

GSFLAGS
: General options passed to the Ghostscript program when converting PostScript to PDF files.

GSCOM
: The Ghostscript command line used to convert PostScript to PDF files.

IDLSUFFIXES
: The list of suffixes of files that will be scanned for IDL implicit dependencies (#include or import lines). The default list is:

```
[".idl", ".IDL"]
```

INCPREFIX
: The prefix used to specify an include directory on the C compiler command line. This will be appended to the beginning of each directory in the $CPPPATH and $FORTRANPATH construction variables when the $_CPPINCFLAGS and $_FORTRANINCFLAGS variables are automatically generated.

INCSUFFIX
: The suffix used to specify an include directory on the C compiler command line. This will be appended to the end of each directory in the $CPPPATH and $FORTRANPATH construction variables when the $_CPPINCFLAGS and $_FORTRANINCFLAGS variables are automatically generated.

INSTALL
A function to be called to install a file into a destination file name. The default function copies the file into the destination (and sets the destination file's mode and permission bits to match the source file's). The function takes the following arguments:

```
def install(dest, source, env):
```

dest is the path name of the destination file. *source* is the path name of the source file. *env* is the construction environment (a dictionary of construction values) in force for this file installation.

JAR The Java archive tool.

JARCHDIR
The directory to which the Java archive tool should change (using the **−C** option).

JARCOM
The command line used to call the Java archive tool.

JARFLAGS
General options passed to the Java archive tool. By default this is set to **cf** to create the necessary *jar* file.

JARSUFFIX
The suffix for Java archives: **.jar** by default.

JAVAC The Java compiler.

JAVACCOM
The command line used to compile a directory tree containing Java source files to corresponding Java class files. Any options specified in the $JAVACFLAGS construction variable are included on this command line.

JAVACFLAGS
General options that are passed to the Java compiler.

JAVACLASSDIR
The directory in which Java class files may be found. This is stripped from the beginning of any Java .class file names supplied to the **JavaH** builder.

JAVACLASSSUFFIX
The suffix for Java class files; **.class** by default.

JAVAH The Java generator for C header and stub files.

JAVAHCOM
The command line used to generate C header and stub files from Java classes. Any options specified in the $JAVAHFLAGS construction variable are included on this command line.

JAVAHFLAGS
General options passed to the C header and stub file generator for Java classes.

JAVASUFFIX
The suffix for Java files; **.java** by default.

LATEX
The LaTeX structured formatter and typesetter.

LATEXCOM
The command line used to call the LaTeX structured formatter and typesetter.

LATEXFLAGS
General options passed to the LaTeX structured formatter and typesetter.

LEX The lexical analyzer generator.

LEXFLAGS
General options passed to the lexical analyzer generator.

LEXCOM
The command line used to call the lexical analyzer generator to generate a source file.

_LIBDIRFLAGS
An automatically-generated construction variable containing the linker command-line options for specifying directories to be searched for library. The value of $_LIBDIRFLAGS is created by appending $LIBDIRPREFIX and $LIBDIRSUFFIX to the beginning and end of each directory in $LIBPATH.

LIBDIRPREFIX
The prefix used to specify a library directory on the linker command line. This will be appended to the beginning of each directory in the $LIBPATH construction variable when the $_LIBDIRFLAGS variable is automatically generated.

LIBDIRSUFFIX
The suffix used to specify a library directory on the linker command line. This will be appended to the end of each directory in the $LIBPATH construction variable when the $_LIBDIRFLAGS variable is automatically generated.

_LIBFLAGS
An automatically-generated construction variable containing the linker command-line options for specifying libraries to be linked with the resulting target. The value of $_LIBFLAGS is created by appending $LIBLINKPREFIX and $LIBLINKSUFFIX to the beginning and end of each directory in $LIBS.

LIBLINKPREFIX
The prefix used to specify a library to link on the linker command line. This will be appended to the beginning of each library in the $LIBS construction variable when the $_LIBFLAGS variable is automatically generated.

LIBLINKSUFFIX
The suffix used to specify a library to link on the linker command line. This will be appended to the end of each library in the $LIBS construction variable when the $_LIBFLAGS variable is automatically generated.

LIBPATH
The list of directories that will be searched for libraries. The implicit dependency scanner will search these directories for include files. Don't explicitly put include directory arguments in $LINKFLAGS or $SHLINKFLAGS because the result will be non-portable and the directories will not be searched by the dependency scanner. Note: directory names in LIBPATH will be looked-up relative to the SConscript directory when they are used in a command. To force **scons** to look-up a directory relative to the root of the source tree use #:

```
env = Environment(LIBPATH='#/libs')
```

The directory look-up can also be forced using the **Dir**() function:

```
libs = Dir('libs')
env = Environment(LIBPATH=libs)
```

The directory list will be added to command lines through the automatically-generated $_LIBDIRFLAGS construction variable, which is constructed by appending the values of the $LIBDIRPREFIX and $LIBDIRSUFFIX construction variables to the beginning and end of each directory in $LIBPATH. Any command lines you define that need the LIBPATH directory list should include $_LIBDIRFLAGS:

```
env = Environment(LINKCOM="my_linker $_LIBDIRFLAGS $_LIBFLAGS -o $TARGET $SOURCE")
```

LIBPREFIX
The prefix used for (static) library file names. A default value is set for each platform (posix, win32, os2, etc.), but the value is overridden by individual tools (ar, mslib, sgiar, sunar, tlib, etc.) to reflect the names of the libraries they create.

LIBPREFIXES
An array of legal prefixes for library file names.

LIBS
A list of one or more libraries that will be linked with any executable programs created by this environment.

The library list will be added to command lines through the automatically-generated $_LIBFLAGS construction variable, which is constructed by appending the values of the $LIBLINKPREFIX and $LIBLINKSUFFIX construction variables to the beginning and end of each directory in $LIBS. Any command lines you define that need the LIBS library list should include $_LIBFLAGS:

```
env = Environment(LINKCOM="my_linker $_LIBDIRFLAGS $_LIBFLAGS -o $TARGET $SOURCE")
```

LIBSUFFIX
The suffix used for (static) library file names. A default value is set for each platform (posix, win32, os2, etc.), but the value is overridden by individual tools (ar, mslib, sgiar, sunar, tlib, etc.) to reflect the names of the libraries they create.

LIBSUFFIXES
An array of legal suffixes for library file names.

LINK The linker.

LINKFLAGS
General user options passed to the linker. Note that this variable should *not* contain **-l** (or similar) options for linking with the libraries listed in $LIBS, nor **-L** (or similar) library search path options that scons generates automatically from $LIBPATH. See **_LIBFLAGS**, above, for the variable that expands to library-link options, and **_LIBDIRFLAGS**, above, for the variable that expands to library search path options.

LINKCOM
The command line used to link object files into an executable.

M4 The M4 macro preprocessor.

M4FLAGS
General options passed to the M4 macro preprocessor.

M4COM
The command line used to pass files through the macro preprocessor.

MAXLINELENGTH
The maximum number of characters allowed on an external command line. On Win32 systems, link lines longer than this many characters are linke via a temporary file name.

MSVS When the Microsoft Visual Studio tools are initialized, they set up this dictionary with the following keys:

VERSION: the version of MSVS being used (can be set via MSVS_VERSION)

VERSIONS: the available versions of MSVS installed

VCINSTALLDIR: installed directory of Visual C++

VSINSTALLDIR: installed directory of Visual Studio

FRAMEWORKDIR: installed directory of the .NET framework

FRAMEWORKVERSIONS: list of installed versions of the .NET framework, sorted latest to oldest.

FRAMEWORKVERSION: latest installed version of the .NET framework

FRAMEWORKSDKDIR: installed location of the .NET SDK.

PLATFORMSDKDIR: installed location of the Platform SDK.

PLATFORMSDK_MODULES: dictionary of installed Platform SDK modules, where the dictionary keys are keywords for the various modules, and the values are 2-tuples where the first is the release date, and the second is the version number.

If a value isn't set, it wasn't available in the registry.

MSVS_IGNORE_IDE_PATHS
Tells the MS Visual Studio tools to use minimal INCLUDE, LIB, and PATH settings, instead of the settings from the IDE.

For Visual Studio, SCons will (by default) automatically determine where MSVS is installed, and use the LIB, INCLUDE, and PATH variables set by the IDE. You can override this behavior by setting these variables after Environment initialization, or by setting **MSVS_IGNORE_IDE_PATHS = 1** in the Environment initialization. Specifying this will not leave these unset, but will set them to a minimal set of paths needed to run the tools successfully.

For VS6, the mininimal set is:
INCLUDE:'<VSDir>\VC98\ATL\include;<VSDir>\VC98\MFC\include;<VSDir>\VC98\include'
LIB:'<VSDir>\VC98\MFC\lib;<VSDir>\VC98\lib'
PATH:'<VSDir>\Common\MSDev98\bin;<VSDir>\VC98\bin'
For VS7, it is:
INCLUDE:'<VSDir>\Vc7\atlmfc\include;<VSDir>\Vc7\include'
LIB:'<VSDir>\Vc7\atlmfc\lib;<VSDir>\Vc7\lib'
PATH:'<VSDir>\Common7\Tools\bin;<VSDir>\Common7\Tools;<VSDir>\Vc7\bin'

Where '<VSDir>' is the installed location of Visual Studio.

MSVS_USE_MFC_DIRS
Tells the MS Visual Studio tool(s) to use the MFC directories in its default paths for compiling and linking. Under MSVS version 6, setting **MSVS_USE_MFC_DIRS** to a non-zero value adds the **ATL\include** and **MFC\include** directories to the default **INCLUDE** external environment variable, and adds the **MFC\lib** directory to the default **LIB** external environment variable. Under MSVS version 7, setting **MSVS_USE_MFC_DIRS** to a non-zero value adds the **atlmfc\include** directory to the default **INCLUDE** external environment variable, and adds the **atlmfc\lib** directory to the default **LIB** external environment variable. The current default value is **1**, which means these directories are added to the paths by default. This default value is likely to change in a future release, so users who want the ATL and MFC values included in their paths are encouraged to enable the **MSVS_USE_MFC_DIRS** value explicitly to avoid future incompatibility. This variable has no effect if the **INCLUDE** or **LIB** environment variables are set explictly.

MSVS_VERSION
Sets the preferred version of MSVS to use.

SCons will (by default) select the latest version of MSVS installed on your machine. So, if you have version 6 and version 7 (MSVS .NET) installed, it will prefer version 7. You can override this by specifying the **MSVS_VERSION** variable in the Environment initialization, setting it to the appropriate version ('6.0' or '7.0', for example). If the given version isn't installed, tool initialization will fail.

MSVSPROJECTCOM
The action used to generate Microsoft Visual Studio project and solution files.

MSVSPROJECTSUFFIX
The suffix used for Microsoft Visual Studio project (DSP) files. The default value is **.vcproj** when using Visual Studio version 7.x (.NET), and **.dsp** when using earlier versions of Visual Studio.

MSVSSOLUTIONSUFFIX
The suffix used for Microsoft Visual Studio solution (DSW) files. The default value is **.sln** when using Visual Studio version 7.x (.NET), and **.dsw** when using earlier versions of Visual Studio.

no_import_lib
When set to non-zero, suppresses creation of a corresponding Win32 static import lib by the **SharedLibrary** builder when used with MinGW or Microsoft Visual Studio. This also suppresses creation of an export (.exp) file when using Microsoft Visual Studio.

OBJPREFIX
The prefix used for (static) object file names.

OBJSUFFIX
The suffix used for (static) object file names.

P4 The Perforce executable.

P4COM
The command line used to fetch source files from Perforce.

P4FLAGS
General options that are passed to Perforce.

PCH The Microsoft Visual C++ precompiled header that will be used when compiling object files. This variable is ignored by tools other than Microsoft Visual C++. When this variable is defined SCons will add options to the compiler command line to cause it to use the precompiled header, and will also set up the dependencies for the PCH file. Example:

```
env['PCH'] = 'StdAfx.pch'
```

PCHSTOP
This variable specifies how much of a source file is precompiled. This variable is ignored by tools other than Microsoft Visual C++, or when the PCH variable is not being used. When this variable is define it must be a string that is the name of the header that is included at the end of the precompiled portion of the source files, or the empty string if the "#pragma hrdstop" construct is being used:

```
env['PCHSTOP'] = 'StdAfx.h'
```

PDB The Microsoft Visual C++ PDB file that will store debugging information for object files, shared libraries, and programs. This variable is ignored by tools other than Microsoft Visual C++. When this variable is defined SCons will add options to the compiler and linker command line to cause them to generate external debugging information, and will also set up the dependencies for the PDB file. Example:

```
env['PDB'] = 'hello.pdb'
```

PDFCOM
A deprecated synonym for $DVIPDFCOM.

PDFPREFIX
The prefix used for PDF file names.

PDFSUFFIX
The suffix used for PDF file names.

PLATFORM
The name of the platform used to create the Environment. If no platform is specified when the Environment is created, **SCons** autodetects the platform.

```
env = Environment(tools = [])
if env['PLATFORM'] == 'cygwin':
   Tool('mingw')(env)
else:
   Tool('msvc')(env)
```

PROGPREFIX
The prefix used for executable file names.

PROGSUFFIX
The suffix used for executable file names.

PSCOM
The command line used to convert TeX DVI files into a PostScript file.

PSPREFIX
The prefix used for PostScript file names.

PSSUFFIX
The prefix used for PostScript file names.

QTDIR The qt tool tries to take this from os.environ. It also initializes all QT_* construction variables listed below. (Note that all paths are constructed with python's os.path.join() method, but are listed here with the '/' separator for easier reading.) In addition, the construction environment variables CPPPATH, LIBPATH, LIBS, PROGEMITTER, SHLIBEMITTER and LIBEMITTER are modified. Because the build-performance is affected when using this tool, you have to explicitly specify it at Environment creation:

```
Environment(tools=['default','qt'])
```

The qt tool supports the following operations:

Automatic moc file generation from header files. You do not have to specify moc files explicitly, the tool does it for you. However, there are a few preconditions to do so: Your header file must have the same filebase as your implementation file and must stay in the same directory. It must have one of the suffixes .h, .hpp, .H, .hxx, .hh. You can turn off automatic moc file generation by setting QT_AUTOSCAN to 0. See also the corresponding builder method **Moc()**

Automatic moc file generation from cxx files. As stated in the qt documentation, include the moc file at the end of the cxx file. Note that you have to include the file, which is generated by the transformation ${QT_MOCCXXPREFIX}<basename>${QT_MOCCXXSUFFIX}, by default <basename>.moc. A warning is generated after building the moc file, if you do not include the correct file. If you are using BuildDir, you may need to specify duplicate=1. You can turn off automatic moc file generation by setting QT_AUTOSCAN to 0. See also the corresponding builder method **Moc()**

Automatic handling of .ui files. The implementation files generated from .ui files are handled much the same as yacc or lex files. Each .ui file given as a source of Program, Library or SharedLibrary will generate three files, the declaration file, the implementation file and a moc file. Because there are also generated headers, you may need to specify duplicate=1 in calls to BuildDir. See also the corresponding builder method **Uic()**

QT_AUTOSCAN
: Turn off scanning for mocable files. Use the Moc Builder to explicitely specify files to run moc on.

QT_DEBUG
: Prints lots of debugging information while scanning for moc files.

QT_LIB
: Default value is 'qt'. You may want to set this to 'qt-mt'.

QT_MOC
: Default value is '$QTDIR/bin/moc'.

QT_MOCCXXPREFIX
: Default value is ''. Prefix for moc output files, when source is a cxx file.

QT_MOCCXXSUFFIX
: Default value is '.moc'. Suffix for moc output files, when source is a cxx file.

QT_MOCFROMCPPFLAGS
: Default value is '-i'. These flags are passed to moc, when moccing a cpp file.

QT_MOCFROMCXXCOM
: Command to generate a moc file from a cpp file.

QT_MOCFROMHCOM
: Command to generate a moc file from a header.

QT_MOCFROMHFLAGS
: Default value is ''. These flags are passed to moc, when moccing a header file.

QT_MOCHPREFIX
: Default value is 'moc_'. Prefix for moc output files, when source is a header.

QT_MOCHSUFFIX
: Default value is '$CXXFILESUFFIX'. Suffix for moc output files, when source is a header.

QT_UIC
Default value is '$QTDIR/bin/uic'.

QT_UICDECLCOM
Command to generate header files from .ui files.

QT_UICDECLFLAGS
Default value is ''. These flags are passed to uic, when creating a a h file from a .ui file.

QT_UICDECLPREFIX
Default value is ''. Prefix for uic generated header files.

QT_UICDECLSUFFIX
Default value is '.h'. Suffix for uic generated header files.

QT_UICIMPLCOM
Command to generate cxx files from .ui files.

QT_UICIMPLFLAGS
Default value is ''. These flags are passed to uic, when creating a cxx file from a .ui file.

QT_UICIMPLPREFIX
Default value is 'uic_'. Prefix for uic generated implementation files.

QT_UICIMPLSUFFIX
Default value is '$CXXFILESUFFIX'. Suffix for uic generated implementation files.

QT_UISUFFIX
Default value is '.ui'. Suffix of designer input files.

RANLIB
The archive indexer.

RANLIBFLAGS
General options passed to the archive indexer.

RC
The resource compiler used by the RES builder.

RCCOM
The command line used by the RES builder.

RCFLAGS
The flags passed to the resource compiler by the RES builder.

RCS
The RCS executable. Note that this variable is not actually used for the command to fetch source files from RCS; see the **RCS_CO** construction variable, below.

RCS_CO
The RCS "checkout" executable, used to fetch source files from RCS.

RCS_COCOM
The command line used to fetch (checkout) source files from RCS.

RCS_COFLAGS
Options that are passed to the $RCS_CO command.

RDirs A function that converts a file name into a list of Dir instances by searching the repositories.

RMIC The Java RMI stub compiler.

RMICCOM
The command line used to compile stub and skeleton class files from Java classes that contain RMI implementations. Any options specified in the $RMICFLAGS construction variable are included on this command line.

RMICFLAGS
General options passed to the Java RMI stub compiler.

RPATH A list of paths to search for shared libraries when running programs. Currently only used in the GNU linker (gnulink) and IRIX linker (sgilink). Ignored on platforms and toolchains that don't support it. Note that the paths added to RPATH are not transformed by **scons** in any way: if you want an absolute path, you must make it absolute yourself.

SCANNERS
A list of the available implicit dependency scanners. New file scanners may be added by appending to this list, although the more flexible approach is to associate scanners with a specific Builder. See the sections "Builder Objects" and "Scanner Objects," below, for more information.

SCCS The SCCS executable.

SCCSCOM
The command line used to fetch source files from SCCS.

SCCSFLAGS
General options that are passed to SCCS.

SCCSGETFLAGS
Options that are passed specifically to the SCCS "get" subcommand. This can be set, for example, to *-e* to check out editable files from SCCS.

SHCC The C compiler used for generating shared-library objects.

SHCCCOM
The command line used to compile a C source file to a shared-library object file. Any options specified in the $SHCCFLAGS and $CPPFLAGS construction variables are included on this command line.

SHCCFLAGS
Options that are passed to the C compiler to generate shared-library objects.

SHCXX
The C++ compiler used for generating shared-library objects.

SHCXXCOM
The command line used to compile a C++ source file to a shared-library object file. Any options specified in the $SHCXXFLAGS and $CPPFLAGS construction variables are included on this command line.

SHCXXFLAGS
Options that are passed to the C++ compiler to generate shared-library objects.

SHELL
A string naming the shell program that will be passed to the *SPAWN* function. See the *SPAWN* construction variable for more information.

SHF77 The Fortran 77 compiler used for generating shared-library objects. You should normally set the $SHFORTRANC variable, which specifies the default Fortran compiler for all Fortran versions. You only need to set $SHF77 if you need to use a specific compiler or compiler version for Fortran 77 files.

SHF77COM
The command line used to compile a Fortran 77 source file to a shared-library object file. You only need to set $SHF77COM if you need to use a specific command line for Fortran 77 files. You should normally set the $SHFORTRANCOM variable, which specifies the default command line for all Fortran versions.

SHF77FLAGS
Options that are passed to the Fortran 77 compiler to generated shared-library objects. You only need to set $SHF77FLAGS if you need to define specific user options for Fortran 77 files. You should normally set the $SHFORTRANFLAGS variable, which specifies the user-specified options passed to the default Fortran compiler for all Fortran versions.

SHF77PPCOM
The command line used to compile a Fortran 77 source file to a shared-library object file after first running the file through the C preprocessor. Any options specified in the $SHF77FLAGS and $CPPFLAGS construction variables are included on this command line. You only need to set $SHF77PPCOM if you need to use a specific C-preprocessor command line for Fortran 77 files. You should normally set the $SHFORTRANPPCOM variable, which specifies the default C-preprocessor command line for all Fortran versions.

SHF90 The Fortran 90 compiler used for generating shared-library objects. You should normally set the $SHFORTRANC variable, which specifies the default Fortran compiler for all Fortran versions. You only need to set $SHF90 if you need to use a specific compiler or compiler version for Fortran 90 files.

SHF90COM
The command line used to compile a Fortran 90 source file to a shared-library object file. You only need to set $SHF90COM if you need to use a specific command line for Fortran 90 files. You should normally set the $SHFORTRANCOM variable, which specifies the default command line for all Fortran versions.

SHF90FLAGS
Options that are passed to the Fortran 90 compiler to generated shared-library objects. You only need to set $SHF90FLAGS if you need to define specific user options for Fortran 90 files. You should normally set the $SHFORTRANFLAGS variable, which specifies the user-specified options passed to the default Fortran compiler for all Fortran versions.

SHF90PPCOM
The command line used to compile a Fortran 90 source file to a shared-library object file after first running the file through the C preprocessor. Any options specified in the $SHF90FLAGS and $CPPFLAGS construction variables are included on this command line. You only need to set $SHF90PPCOM if you need to use a specific C-preprocessor command line for Fortran 90 files. You should normally set the $SHFORTRANPPCOM variable, which specifies the default C-preprocessor command line for all Fortran versions.

SHF95
The Fortran 95 compiler used for generating shared-library objects. You should normally set the $SHFORTRANC variable, which specifies the default Fortran compiler for all Fortran versions. You only need to set $SHF95 if you need to use a specific compiler or compiler version for Fortran 95 files.

SHF95COM
The command line used to compile a Fortran 95 source file to a shared-library object file. You only need to set $SHF95COM if you need to use a specific command line for Fortran 95 files. You should normally set the $SHFORTRANCOM variable, which specifies the default command line for all Fortran versions.

SHF95FLAGS
Options that are passed to the Fortran 95 compiler to generated shared-library objects. You only need to set $SHF95FLAGS if you need to define specific user options for Fortran 95 files. You should normally set the $SHFORTRANFLAGS variable, which specifies the user-specified options passed to the default Fortran compiler for all Fortran versions.

SHF95PPCOM
The command line used to compile a Fortran 95 source file to a shared-library object file after first running the file through the C preprocessor. Any options specified in the $SHF95FLAGS and $CPPFLAGS construction variables are included on this command line. You only need to set $SHF95PPCOM if you need to use a specific C-preprocessor command line for Fortran 95 files. You should normally set the $SHFORTRANPPCOM variable, which specifies the default C-preprocessor command line for all Fortran versions.

SHFORTRAN
The default Fortran compiler used for generating shared-library objects.

SHFORTRANCOM
The command line used to compile a Fortran source file to a shared-library object file.

SHFORTRANFLAGS
Options that are passed to the Fortran compiler to generate shared-library objects.

SHFORTRANPPCOM
The command line used to compile a Fortran source file to a shared-library object file after first running the file through the C preprocessor. Any options specified in the $SHFORTRANFLAGS and $CPPFLAGS construction variables are included on this command line.

SHLIBPREFIX
The prefix used for shared library file names.

SHLIBSUFFIX
The suffix used for shared library file names.

SHLINK
The linker for programs that use shared libraries.

SHLINKFLAGS
General user options passed to the linker for programs using shared libraries. Note that this variable should *not* contain **-l** (or similar) options for linking with the libraries listed in $LIBS, nor **-L** (or similar) include search path options that scons generates automatically from $LIBPATH. See **_LIBFLAGS**, above, for the variable that expands to library-link options, and **_LIBDIRFLAGS**, above, for the variable that expands to library search path options.

SHOBJPREFIX
The prefix used for shared object file names.

SHOBJSUFFIX
The suffix used for shared object file names.

SOURCE
A reserved variable name that may not be set or used in a construction environment. (See "Variable Substitution," below.)

SOURCES
A reserved variable name that may not be set or used in a construction environment. (See "Variable Substitution," below.)

SPAWN
A command interpreter function that will be called to execute command line strings. The function must expect 4 arguments:

```
def spawn(shell, escape, cmd, args, env):
```

sh is a string naming the shell program to use. *escape* is a function that can be called to escape shell special characters in the command line. *cmd* is the path to the command to be executed. *args* is that arguments to the command. *env* is a dictionary of the environment variables in which the command should be executed.

SWIG The scripting language wrapper and interface generator.

SWIGCFILESUFFIX
The suffix that will be used for intermediate C source files generated by the scripting language wrapper and interface generator. The default value is **_wrap$CFILESUFFIX**. By default, this value is used whenever the **-c++** option is *not* specified as part of the **SWIGFLAGS** construction variable.

SWIGCOM
The command line used to call the scripting language wrapper and interface generator.

SWIGCXXFILESUFFIX
The suffix that will be used for intermediate C++ source files generated by the scripting language wrapper and interface generator. The default value is **_wrap$CFILESUFFIX**. By default, this value is used whenever the **-c++** option is specified as part of the **SWIGFLAGS** construction variable.

SWIGFLAGS
General options passed to the scripting language wrapper and interface generator. This is where you should set **-python**, **-perl5**, **-tcl**, or whatever other options you want to specify to SWIG. If you set the **-c++** option in this variable, **scons** will, by default, generate a C++ intermediate source file with the extension that is specified as the **$CXXFILESUFFIX** variable.

TAR The tar archiver.

TARCOM
The command line used to call the tar archiver.

TARFLAGS
General options passed to the tar archiver.

TARGET
A reserved variable name that may not be set or used in a construction environment. (See "Variable Substitution," below.)

TARGETS
A reserved variable name that may not be set or used in a construction environment. (See "Variable Substitution," below.)

TARSUFFIX
The suffix used for tar file names.

TEX The TeX formatter and typesetter.

TEXCOM
The command line used to call the TeX formatter and typesetter.

TEXFLAGS
General options passed to the TeX formatter and typesetter.

TOOLS
A list of the names of the Tool specifications that are part of this construction environment.

WIN32_INSERT_DEF
When this is set to true, a library build of a WIN32 shared library (.dll file) will also build a corresponding .def file at the same time, if a .def file is not already listed as a build target. The default is 0 (do not build a .def file).

WIN32DEFPREFIX
The prefix used for WIN32 .def file names.

WIN32DEFSUFFIX
The suffix used for WIN32 .def file names.

YACC The parser generator.

YACCCOM
The command line used to call the parser generator to generate a source file.

YACCFLAGS
General options passed to the parser generator. If $YACCFLAGS contains a −d option, SCons assumes that the call will also create a .h file (if the yacc source file ends in a .y suffix) or a .hpp file (if the yacc source file ends in a .yy suffix)

ZIP The zip compression and file packaging utility.

ZIPCOM
The command line used to call the zip utility, or the internal Python function used to create a zip archive.

ZIPCOMPRESSION
The *compression* flag from the Python **zipfile** module used by the internal Python function to control whether the zip archive is compressed or not. The default value is **zipfile.ZIP_DEFLATED**, which creates a compressed zip archive. This value has no effect when using Python 1.5.2 or if the **zipfile** module is otherwise unavailable.

ZIPFLAGS
General options passed to the zip utility.

Construction variables can be retrieved and set using the **Dictionary** method of the construction environment:

```
dict = env.Dictionary()
dict["CC"] = "cc"
```

or using the [] operator:

```
env["CC"] = "cc"
```

Construction variables can also be passed to the construction environment constructor:

```
env = Environment(CC="cc")
```

or when copying a construction environment using the **Copy** method:

```
env2 = env.Copy(CC="cl.exe")
```

Configure Contexts

scons supports *configure contexts,* an integrated mechanism similar to the various AC_CHECK macros in GNU autoconf for testing for the existence of C header files, libraries, etc. In contrast to autoconf, **scons** does not maintain an explicit cache of the tested values, but uses its normal dependency tracking to keep the checked values up to date. The following methods can be used to perform checks:

Configure(*env*, [*custom_tests*, *conf_dir*, *log_file*])

env.Configure([*custom_tests*, *conf_dir*, *log_file*])

This creates a configure context, which can be used to perform checks. *env* specifies the environment for building the tests. This environment may be modified when performing checks. *custom_tests* is a dictionary containing custom tests. See also the section about custom tests below. By default, no custom tests are added to the configure context. *conf_dir* specifies a directory where the test cases are built. Note that this directory is not used for building normal targets. The default value is the directory #/.sconf_temp. *log_file* specifies a file which collects the output from commands that are executed to check for the existence of header files, libraries, etc. The default is the file #/config.log. If you are using the **BuildDir** method, you may want to specify a subdirectory under your build directory.

A created **Configure** instance has the following associated methods:

Configure.Finish(*self*)

This method should be called after configuration is done. It returns the environment as modified by the configuration checks performed. After this method is called, no further checks can be performed with this configuration context. However, you can create a new Configure context to perform additional checks. Only one context should be active at a time.

The following Checks are predefined. (This list will likely grow larger as time goes by and developers contribute new useful tests.)

Configure.CheckHeader(*self*, *header*, [*include_quotes*, *language*])

Checks if *header* is usable in the specified language. *header* may be a list, in which case the last item in the list is the header file to be checked, and the previous list items are header files whose **#include** lines should precede the header line being checked for. The optional argument *include_quotes* must be a two character string, where the first character denotes the opening quote and the second character denotes the closing quote. By default, both characters are " (double quote). The optional argument *language* should be either **C** or **C++** and selects the compiler to be used for the check. Returns 1 on success and 0 on failure.

Configure.CheckCHeader(*self*, *header*, [*include_quotes*])

This is a wrapper around **Configure.CheckHeader** which checks if *header* is usable in the C language. *header* may be a list, in which case the last item in the list is the header file to be checked, and the previous list items are header files whose **#include** lines should precede the header line being checked for. The optional argument *include_quotes* must be a two character string, where the first character denotes the opening quote and the second character denotes the closing quote (both default to "). Returns 1 on success and 0 on failure.

Configure.CheckCXXHeader(*self*, *header*, [*include_quotes*])

This is a wrapper around **Configure.CheckHeader** which checks if *header* is usable in the C++ language. *header* may be a list, in which case the last item in the list is the header file to be checked, and the previous list items are header files whose **#include** lines should precede the header line being checked for. The optional argument *include_quotes* must be a two character string, where the first character denotes the opening quote and the second character denotes the closing quote (both default to "). Returns 1 on success and 0 on failure.

Configure.CheckFunc(*self*, *function_name*, [*language*])

Checks if the specified C or C++ function is available. *function_name* is the name of the function to check for. The optional *language* argument should be **C** or **C++** and selects the compiler to be used for the check; the default is "C".

Configure.CheckLib(*self*, [*library*, *symbol*, *header*, *language*, *autoadd*])

Checks if *library* provides *symbol*. If the value of *autoadd* is 1 and the library provides the specified *symbol*, appends the library to the LIBS construction environment variable. *library* may also be None (the default), in which case *symbol* is checked with the current LIBS variable, or a list of library names, in which case each library in the list will be checked for *symbol*. The default *symbol* is "main", which just check if you can link against the specified *library*. The optional *language* argument should be **C** or **C++** and selects the compiler to be used for the check; the default is "C". The default value for *autoadd* is 1. It is assumed, that the C-language is used. This method returns 1 on success and 0 on error.

Configure.CheckLibWithHeader(*self*, *library*, *header*, *language*, [*call*, *autoadd*])

In contrast to the Configure.CheckLib call, this call provides a more sophisticated way to check against libraries. Again, *library* specifies the library or a list of libraries to check. *header* specifies a header to check for. *header* may be a list, in which case the last item in the list is the header file to be checked, and the previous list items are header files whose **#include** lines should precede the header line being checked for. *language* may be one of 'C','c','CXX','cxx','C++' and 'c++'. *call* can be any valid expression (with a trailing ';'). The default is 'main();'. *autoadd* specifies whether to add the library to the environment (only if the check succeeds). This method returns 1 on success and 0 on error.

Configure.CheckType(*self*, *type_name*, [*includes*, *language*])

Checks for the existence of a type defined by **typedef**. *type_name* specifies the typedef name to check for. *includes* is a string containing one or more **#include** lines that will be inserted into the program that will be run to test for the existence of the type. The optional *language* argument should be **C** or **C++** and selects the compiler to be used for the check; the default is "C".

Example of a typical Configure usage:

```
env = Environment()
conf = Configure( env )
if not conf.CheckCHeader( 'math.h' ):
    print 'We really need math.h!'
    Exit(1)
if conf.CheckLibWithHeader( 'qt', 'qapp.h', 'c++', 'QApplication qapp(0,0);' ):
    # do stuff for qt - usage, e.g.
    conf.env.Append( CPPFLAGS = '-DWITH_QT' )
env = conf.Finish()
```

You can define your own custom checks. in addition to the predefined checks. These are passed in a dictionary to the Configure function. This dictionary maps the names of the checks to user defined Python callables (either Python functions or class instances implementing the *__call__* method). The first argument of the call is always a *CheckContext* instance followed by the arguments, which must be supplied by the user of the check. These CheckContext instances define the following methods:

CheckContext.Message(*self*, *text*)

Usually called before the check is started. *text* will be displayed to the user, e.g. 'Checking for library X...'

CheckContext.Result(*self,*, *res*)

Usually called after the check is done. *res* can be either an integer or a string. In the former case, 'ok' (res != 0) or 'failed' (res == 0) is displayed to the user, in the latter case the given string is displayed.

CheckContext.TryCompile(*self*, *text*, *extension*)
Checks if a file with the specified *extension* (e.g. '.c') containing *text* can be compiled using the environment's **Object** builder. Returns 1 on success and 0 on failure.

CheckContext.TryLink(*self*, *text*, *extension*)
Checks, if a file with the specified *extension* (e.g. '.c') containing *text* can be compiled using the environment's **Program** builder. Returns 1 on success and 0 on failure.

CheckContext.TryRun(*self*, *text*, *extension*)
Checks, if a file with the specified *extension* (e.g. '.c') containing *text* can be compiled using the environment's **Program** builder. On success, the program is run. If the program executes successfully (that is, its return status is 0), a tuple *(1, outputStr)* is returned, where *outputStr* is the standard output of the program. If the program fails execution (its return status is non-zero), then (0, '') is returned.

CheckContext.TryAction(*self*, *action*, [*text*, *extension*])
Checks if the specified *action* with an optional source file (contents *text* , extension *extension* = '') can be executed. *action* may be anything which can be converted to a **scons** Action. On success, *(1, outputStr)* is returned, where *outputStr* is the content of the target file. On failure *(0, '')* is returned.

CheckContext.TryBuild(*self*, *builder*, [*text*, *extension*])
Low level implementation for testing specific builds; the methods above are based on this method. Given the Builder instance *builder* and the optional *text* of a source file with optional *extension*, this method returns 1 on success and 0 on failure. In addition, *self.lastTarget* is set to the build target node, if the build was successful.

Example for implementing and using custom tests:

```
def CheckQt(context, qtdir):
    context.Message( 'Checking for qt ...' )
    lastLIBS = context.env['LIBS']
    lastLIBPATH = context.env['LIBPATH']
    lastCPPPATH= context.env['CPPPATH']
    context.env.Append(LIBS = 'qt', LIBPATH = qtdir + '/lib', CPPPATH = qtdir + '/include' )
    ret = context.TryLink("""
```

```
#include <qapp.h>
int main(int argc, char **argv) {
  QApplication qapp(argc, argv);
  return 0;
}
""")
    if not ret:
        context.env.Replace(LIBS = lastLIBS, LIBPATH=lastLIBPATH, CPPPATH=lastCPPPATH)
    context.Result( ret )
    return ret

env = Environment()
conf = Configure( env, custom_tests = { 'CheckQt' : CheckQt } )
if not conf.CheckQt('/usr/lib/qt'):
    print 'We really need qt!'
    Exit(1)
env = conf.Finish()
```

Construction Variable Options

Often when building software, various options need to be specified at build time that are not known when the SConstruct/SConscript files are written. For example, libraries needed for the build may be in non-standard locations, or site-specific compiler options may need to be passed to the compiler. **scons** provides a mechanism for overridding construction variables from the command line or a text-based SConscript file through an Options object. To create an Options object, call the Options() function:

Options([*files*], [*args*])

This creates an Options object that will read construction variables from the file or list of filenames specified in *files*. If no files are specified, or the *files* argument is **None**, then no files will be read. The optional argument *args* is a dictionary of values that will override anything read from the specified files; it is primarily intended to be passed the **ARGUMENTS** dictionary that holds variables specified on the command line. Example:

```
opts = Options('custom.py')
opts = Options('overrides.py', ARGUMENTS)
opts = Options(None, {FOO:'expansion', BAR:7})
```

Options objects have the following methods:

Add(*key*, [*help*, *default*, *validator*, *converter*])

This adds a customizable construction variable to the Options object. *key* is the name of the variable. *help* is the help text for the variable. *default* is the default value of the variable. *validator* is called to validate the value of the variable, and should take three arguments: key, value, and environment *converter* is called to convert the value before putting it in the environment, and should take a single argument: value. Example:

```
opts.Add('CC', 'The C compiler')
```

AddOptions(*list*)

A wrapper script that adds multiple customizable construction variables to an Options object. *list* is a list of tuple or list objects that contain the arguments for an individual call to the **Add** method.

```
opt.AddOptions(
      ('debug', '', 0),
```

```
  ('CC', 'The C compiler'),
  ('VALIDATE', 'An option for testing validation',
   'notset', validator, None),
)
```

Update(*env*, [*args*])

This updates a construction environment *env* with the customized construction variables. Normally this method is not called directly, but is called indirectly by passing the Options object to the Environment() function:

```
env = Environment(options=opts)
```

Save(*filename*, *env*)

This saves the currently set options into a script file named *filename* that can be used on the next invocation to automatically load the current settings. This method combined with the Options method can be used to support caching of options between runs.

```
env = Environment()
opts = Options(['options.cache', 'custom.py'])
opts.Add(...)
opts.Update(env)
opts.Save('options.cache', env)
```

GenerateHelpText(*env*, [*sort*])

This generates help text documenting the customizable construction variables suitable to passing in to the Help() function. *env* is the construction environment that will be used to get the actual values of customizable variables. Calling with an optional *sort* function will cause the output to be sorted by the specified argument. The specific *sort* function should take two arguments and return -1, 0 or 1 (like the standard Python *cmp* function).

```
Help(opts.GenerateHelpText(env))
Help(opts.GenerateHelpText(env, sort=cmp))
```

The text based SConscript file is executed as a Python script, and the global variables are queried for customizable construction variables. Example:

```
CC = 'my_cc'
```

To make it more convenient to work with customizable Options, **scons** provides a number of functions that make it easy to set up various types of Options:

BoolOption(*key*, *help*, *default*)

Return a tuple of arguments to set up a Boolean option. The option will use the specified name *key*, have a default value of *default*, and display the specified *help* text. The option will interpret the values **y**, **yes**, **t**, **true**, **1**, **on** and **all** as true, and the values **n**, **no**, **f**, **false**, **0**, **off** and **none** as false.

EnumOption(*key*, *help*, *default*, *allowed_values*, [*map*, *ignorecase*])

Return a tuple of arguments to set up an option whose value may be one of a specified list of legal enumerated values. The option will use the specified name *key*, have a default value of *default*, and display the specified *help* text. The option will only support those values in the *allowed_values* list. The optional *map* argument is a dictionary that can be used to convert input values into specific legal values in the *allowed_values* list. If the value of *ignore_case* is **0** (the default), then the values are case-sensitive. If the value of *ignore_case* is **1**, then values will be matched case-insensitive. If the value of *ignore_case* is **1**, then values will be matched case-insensitive, and all input values will be converted to lower case.

ListOption(*key*, *help*, *default*, *names*)
Return a tuple of arguments to set up an option whose value may be one or more of a specified list of legal enumerated values. The option will use the specified name *key*, have a default value of *default*, and display the specified *help* text. The option will only support the values **all**, **none**, or the values in the *names* list. More than one value may be specified, with all values separated by commas.

PackageOption(*key*, *help*, *default*)
Return a tuple of arguments to set up an option whose value is a path name of a package that may be enabled, disabled or given an explicit path name. The option will use the specified name *key*, have a default value of *default*, and display the specified *help* text. The option will support the values **yes**, **true**, **on**, **enable** or **search**, in which case the specified *default* will be used, or the option may be set to an arbitrary string (typically the path name to a package that is being enabled). The option will also support the values **no**, **false**, **off** or **disable** to disable use of the specified option.

PathOption(*key*, *help*, *default*)
Return a tuple of arguments to set up an option whose value is expected to be a path name. The option will use the specified name *key*, have a default value of *default*, and display the specified *help* text.

These functions make it convenient to create a number of options with consistent behavior in a single call to the **AddOptions** method:

```
opts.AddOptions(
   BoolOption('warnings', 'compilation with -Wall and similiar', 1),
   EnumOption('debug', 'debug output and symbols', 'no'
              allowed_values=('yes', 'no', 'full'),
              map={}, ignorecase=0),  # case sensitive
   ListOption('shared',
              'libraries to build as shared libraries',
              'all',
              names = list_of_libs),
   PackageOption('x11',
                 'use X11 installed here (yes = search some places)',
                 'yes'),
   PathOption('qtdir', 'where the root of Qt is installed', qtdir),
)
```

File and Directory Nodes

The *File*() and *Dir*() functions return *File* and *Dir* Nodes, respectively. python objects, respectively. Those objects have several user-visible attributes and methods that are often useful:

path The build path of the given file or directory. This path is relative to the top-level directory (where the **SConstruct** file is found). The build path is the same as the source path if *build_dir* is not being used.

abspath The absolute build path of the given file or directory.

srcnode()
The *srcnode*() method returns another *File* or *Dir* object representing the *source* path of the given *File* or *Dir*. The

```
# Get the current build dir's path, relative to top.
Dir('.').path
# Current dir's absolute path
Dir('.').abspath
# Next line is always '.', because it is the top dir's path relative to itself.
Dir('#.').path
File('foo.c').srcnode().path   # source path of the given source file.

# Builders also return File objects:
foo = env.Program('foo.c')
print "foo will be built in %s"%foo.path
```

EXTENDING SCONS

Builder Objects

scons can be extended to build different types of targets by adding new Builder objects to a construction environment. *In general*, you should only need to add a new Builder object when you want to build a new type of file or other external target. If you just want to invoke a different compiler or other tool to build a Program, Object, Library, or any other type of output file for which **scons** already has an existing Builder, it is generally much easier to use those existing Builders in a construction environment that sets the appropriate construction variables (CC, LINK, etc.).

Builder objects are created using the **Builder** function. The **Builder** function accepts the following arguments:

action The command line string used to build the target from the source. **action** can also be: a list of strings representing the command to be executed and its arguments (suitable for enclosing white space in an argument), a dictionary mapping source file name suffixes to any combination of command line strings (if the builder should accept multiple source file extensions), a Python function; an Action object (see the next section); or a list of any of the above.

An action function takes three arguments: *source* - a list of source nodes, *target* - a list of target nodes, *env* - the construction environment.

prefix The prefix that will be prepended to the target file name. This may be a simple string, or a callable object that takes two arguments, a construction environment and a list of sources, and returns a prefix.

```
b = Builder("build_it < $SOURCE > $TARGET"
        prefix = "file-")

def gen_prefix(env, sources):
    return "file-" + env['PLATFORM'] + '-'
b = Builder("build_it < $SOURCE > $TARGET"
        prefix = gen_prefix)
```

suffix The suffix that will be appended to the target file name. This may be a simple string, or a callable object that takes two arguments, a construction environment and a list of sources, and returns a suffix. If the suffix is a string, then **scons** will append a '.' to the beginning of the suffix if it's not already there. The string returned by callable object is untouched and must append its own '.' to the beginning if one is desired.

```
b = Builder("build_it < $SOURCE > $TARGET"
        suffix = "file-"

def gen_suffix(env, sources):
    return "." + env['PLATFORM'] + "-file"
b = Builder("build_it < $SOURCE > $TARGET"
        suffix = gen_suffix)
```

src_suffix
The expected source file name suffix.

target_scanner
A Scanner object that will be invoked to find implicit dependencies for this target file. This keyword argument should be used for Scanner objects that find implicit dependencies based only on the target file and the construction environment, *not* for implicit (See the section "Scanner Objects," below, for information about creating Scanner objects.)

source_scanner
A Scanner object that will be invoked to find implicit dependences in any source files used to build this target file. This is where you would specify a scanner to find things like **#include** lines in source files. (See the section "Scanner Objects," below, for information about creating Scanner objects.)

target_factory
A factory function that the Builder will use to turn any targets specified as strings into SCons Nodes. By default, SCons assumes that all targets are files. Other useful target_factory values include **Dir**, for when a Builder creates a directory target, and **Entry**, for when a Builder can create either a file or directory target.

Example:

```
MakeDirectoryBuilder = Builder(action=my_mkdir, target_factory=Dir)
env = Environment()
env.Append(BUILDERS = {'MakeDirectory':MakeDirectoryBuilder})
env.MakeDirectory('new_directory')
```

source_factory
A factory function that the Builder will use to turn any sources specified as strings into SCons Nodes. By default, SCons assumes that all source are files. Other useful source_factory values include **Dir**, for when a Builder uses a directory as a source, and **Entry**, for when a Builder can use files or directories (or both) as sources.

Example:

```
CollectBuilder = Builder(action=my_mkdir, source_factory=Entry)
env = Environment()
```

```
env.Append(BUILDERS = {'Collect':CollectBuilder})
env.Collect('archive', ['directory_name', 'file_name'])
```

emitter A function or list of functions to manipulate the target and source lists before dependencies are established and the target(s) are actually built. **emitter** can also be a string containing a construction variable to expand to an emitter function or list of functions, or a dictionary mapping source file suffixes to emitter functions. (Only the suffix of the first source file is used to select the actual emitter function from an emitter dictionary.)

An emitter function takes three arguments: *source* - a list of source nodes, *target* - a list of target nodes, *env* - the construction environment. An emitter must return a tuple containing two lists, the list of targets to be built by this builder, and the list of sources for this builder.

Example:

```
def e(target, source, env):
    return (target + ['foo.foo'], source + ['foo.src'])

# Simple association of an emitter function with a Builder.
b = Builder("my_build < $TARGET > $SOURCE",
        emitter = e)

def e2(target, source, env):
    return (target + ['bar.foo'], source + ['bar.src'])

# Simple association of a list of emitter functions with a Builder.
b = Builder("my_build < $TARGET > $SOURCE",
        emitter = [e, e2])

# Calling an emitter function through a construction variable.
env = Environment(MY_EMITTER = e)
b = Builder("my_build < $TARGET > $SOURCE",
        emitter = '$MY_EMITTER')

# Calling a list of emitter functions through a construction variable.
env = Environment(EMITTER_LIST = [e, e2])
b = Builder("my_build < $TARGET > $SOURCE",
        emitter = '$EMITTER_LIST')

# Associating multiple emitters with different file
# suffixes using a dictionary.
def e_suf1(target, source, env):
    return (target + ['another_target_file'], source)
def e_suf2(target, source, env):
    return (target, source + ['another_source_file'])
b = Builder("my_build < $TARGET > $SOURCE",
        emitter = {'.suf1' : e_suf1,
                   '.suf2' : e_suf2})
```

The *generator* and *action* arguments must not both be used for the same Builder.

multi Specifies whether this builder is allowed to be called multiple times for the same target file(s). The default is 0, which means the builder can not be called multiple times for the same target file(s). Calling a builder multiple times for the same target simply adds additional source files to the target; it is not allowed to change the environment associated with the target, specify addition environment overrides, or associate a different builder with the target.

env
A construction environment that can be used to fetch source code using this Builder. (Note that this environment is *not* used for normal builds of normal target files, which use the environment that was used to call the Builder for the target file.)

generator
A function that returns a list of actions that will be executed to build the target(s) from the source(s). The returned action(s) may be an Action object, or anything that can be converted into an Action object (see the next section).

The generator function takes four arguments: *source* - a list of source nodes, *target* - a list of target nodes, *env* - the construction environment, *for_signature* - a Boolean value that specifies whether the generator is being called for generating a build signature (as opposed to actually executing the command). Example:

```
def g(source, target, env, for_signature):
   return [["gcc", "-c", "-o"] + target + source]

b = Builder(generator=g)
```

src_builder
Specifies a builder to use when a source file name suffix does not match any of the suffixes of the builder. Using this argument produces a multi-stage builder.

single_source
Specifies that this builder expects exactly one source file per call. Giving more than one source files without target files results in implicitely calling the builder multiple times (once for each source given). Giving multiple source files together with target files results in a UserError exception.

The *generator* and *action* arguments must not both be used for the same Builder.

env
A construction environment that can be used to fetch source code using this Builder. (Note that this environment is *not* used for normal builds of normal target files, which use the environment that was used to call the Builder for the target file.)

```
b = Builder(action="build < $SOURCE > $TARGET")
env = Environment(BUILDERS = {'MyBuild' : b})
env.MyBuild('foo.out', 'foo.in', my_arg = 'xyzzy')
```

Any additional keyword arguments supplied when a Builder object is created (that is, when the Builder() function is called) will be set in the executing construction environment when the Builder object is called. The canonical example here would be to set a construction variable to the repository of a source code system.

Any additional keyword arguments supplied when a Builder *object* is called will only be associated with the target created by that particular Builder call (and any other files built as a result of the call).

These extra keyword arguments are passed to the following functions: command generator functions, function Actions, and emitter functions.

Action Objects

The **Builder()** function will turn its **action** keyword argument into an appropriate internal Action object. You can also explicity create Action objects using the **Action**() global function, which can then be passed to the **Builder**() function. This can be used to configure an Action object more flexibly, or it may simply be more efficient than letting each separate Builder object create a separate Action when multiple Builder objects need to do the same thing.

The **Action**() global function returns an appropriate object for the action represented by the type of the first argument:

Action
: If the first argument is already an Action object, the object is simply returned.

String
: If the first argument is a string, a command-line Action is returned.

```
Action('$CC -c -o $TARGET $SOURCES')
```

List
: If the first argument is a list, then a list of Action objects is returned. An Action object is created as necessary for each element in the list. If an element *within* the list is itself a list, the internal list is the command and arguments to be executed via the command line. This allows white space to be enclosed in an argument by defining a command in a list within a list:

```
Action([['cc', '-c', '-DWHITE SPACE', '-o', '$TARGET', '$SOURCES']])
```

Function
: If the first argument is a Python function, a function Action is returned. The Python function takes three keyword arguments, **target** (a Node object representing the target file), **source** (a Node object representing the source file) and **env** (the construction environment used for building the target file). The **target** and **source** arguments may be lists of Node objects if there is more than one target file or source file. The actual target and source file name(s) may be retrieved from their Node objects via the built-in Python str() function:

```
target_file_name = str(target)
source_file_names = map(lambda x: str(x), source)
```

The function should return **0** or **None** to indicate a successful build of the target file(s). The function may raise an exception or return a non-zero exit status to indicate an unsuccessful build.

```
def build_it(target = None, source = None, env = None):
    # build the target from the source
    return 0

a = Action(build_it)
```

The second, optional argument is a Python function that returns a string to be printed to describe the action being executed. Like a function to build a file, this function takes three arguments: **target** (a Node object representing the target file), **source** (a Node object representing the source file) and **env** (a construction environment). The **target** and **source** arguments may be lists of Node objects if there is more than one target file or source file. Examples:

```
def build_it(target, source, env):
    # build the target from the source
```

```
    return 0

def string_it(target, source, env):
    return "building '%s' from '%s'" % (target[0], source[0])

# Use a positional argument.
a = Action(build_it, string_it)

# Alternatively, use a keyword argument.
a = Action(build_it, strfunction=string_it)
```

The third, also optional argument is a list of construction variables whose values will be included in the signature of the Action when deciding whether a target should be rebuilt because the action changed. This is necessary whenever you want a target to be rebuilt when a specific construction variable changes, because the underlying Python code for a function will not change when the value of the construction variable does.

```
def build_it(target, source, env):
    # build the target from the 'XXX' construction variable
    open(target[0], 'w').write(env['XXX'])
    return 0

def string_it(target, source):
    return "building '%s' from '%s'" % (target[0], source[0])

# Use positional arguments.
a = Action(build_it, string_it, ['XXX'])

# Alternatively, use a keyword argument.
a = Action(build_it, varlist=['XXX'])
```

If the action argument is not one of the above, None is returned.

Miscellaneous Action Functions

scons supplies a number of functions that arrange for various common file and directory manipulations to be performed. These are similar in concept to "tasks" in the Ant build tool, although the implementation is slightly different. These functions do not actually perform the specified action at the time the function is called, but instead return an Action object that can be executed at the appropriate time. (In Object-Oriented terminology, these are actually Action *Factory* functions that return Action objects.)

In practice, there are two natural ways that these Action Functions are intended to be used.

First, if you need to perform the action at the time the SConscript file is being read, you can use the **Execute** global function to do so:

```
Execute(Touch('file'))
```

Second, you can use these functions to supply Actions in a list for use by the **Command** method. This can allow you to perform more complicated sequences of file manipulation without relying on platform-specific external commands: that

```
env = Environment(TMPBUILD = '/tmp/builddir')
env.Command('foo.out', 'foo.in',
            [Mkdir('$TMPBUILD'),
             Copy('${SOURCE.dir}', '$TMPBUILD')
             "cd $TMPBUILD && make",
             Delete('$TMPBUILD')])
```

Chmod(*dest*, *mode*)

Returns an Action object that changes the permissions on the specified *dest* file or directory to the specified *mode*. Examples:

```
Execute(Chmod('file', 0755))

env.Command('foo.out', 'foo.in',
        [Copy('$TARGET', '$SOURCE'),
         Chmod('$TARGET', 0755)])
```

Copy(*dest*, *src*)

Returns an Action object that will copy the *src* source file or directory to the *dest* destination file or directory. Examples:

```
Execute(Copy('foo.output', 'foo.input'))

env.Command('bar.out', 'bar.in',
        Copy('$TARGET', '$SOURCE'))
```

Delete(*entry*, [*must_exist*])

Returns an Action that deletes the specified *entry*, which may be a file or a directory tree. If a directory is specified, the entire directory tree will be removed. If the *must_exist* flag is set, then a Python error will be thrown if the specified entry does not exist; the default is **must_exist=0**, that is, the Action will silently do nothing if the entry does not exist. Examples:

```
Execute(Delete('/tmp/buildroot'))

env.Command('foo.out', 'foo.in',
        [Delete('${TARGET.dir}'),
         MyBuildAction])

Execute(Delete('file_that_must_exist', must_exist=1))
```

Mkdir(*dir*)

Returns an Action that creates the specified directory *dir* . Examples:

```
Execute(Mkdir('/tmp/outputdir'))

env.Command('foo.out', 'foo.in',
        [Mkdir('/tmp/builddir',
         Copy('$SOURCE', '/tmp/builddir')
         "cd /tmp/builddir && ])
```

Move(*dest*, *src*)

Returns an Action that moves the specified *src* file or directory to the specified *dest* file or directory. Examples:

```
Execute(Move('file.destination', 'file.source'))

env.Command('output_file', 'input_file',
        [MyBuildAction,
         Move('$TARGET', 'file_created_by_MyBuildAction')])
```

Touch(*file*)
Returns an Action that updates the modification time on the specified *file*. Examples:

```
Execute(Touch('file_to_be_touched'))

env.Command('marker', 'input_file',
        [MyBuildAction,
         Touch('$TARGET')])
```

Variable Substitution

Before executing a command, **scons** performs construction variable interpolation on the strings that make up the command line of builders. Variables are introduced by a **$** prefix. Besides construction variables, scons provides the following variables for each command execution:

TARGET
The file name of the target being built, or the file name of the first target if multiple targets are being built.

TARGETS
The file names of all targets being built.

SOURCE
The file name of the source of the build command, or the file name of the first source if multiple sources are being built.

SOURCES
The file names of the sources of the build command.

(Note that the above variables are reserved and may not be set in a construction environment.)

For example, given the construction variable CC='cc', targets=['foo'], and sources=['foo.c', 'bar.c']:

```
action='$CC -c -o $TARGET $SOURCES'
```

would produce the command line:

```
cc -c -o foo foo.c bar.c
```

Variable names may be surrounded by curly braces ({}) to separate the name from the trailing characters. Within the curly braces, a variable name may have a Python slice subscript appended to select one or more items from a list. In the previous example, the string:

```
${SOURCES[1]}
```

would produce:

```
bar.c
```

Additionally, a variable name may have the following special modifiers appended within the enclosing curly braces to modify the interpolated string:

base The base path of the file name, including the directory path but excluding any suffix.

dir The name of the directory in which the file exists.

file The file name, minus any directory portion.

filebase Just the basename of the file, minus any suffix and minus the directory.

suffix Just the file suffix.

abspath The absolute path name of the file.

posix The POSIX form of the path, with directories separated by / (forward slashes) not backslashes. This is sometimes necessary on Win32 systems when a path references a file on other (POSIX) systems.

srcpath The directory and file name to the source file linked to this file through BuildDir. If this file isn't linked, it just returns the directory and filename unchanged.

srcdir The directory containing the source file linked to this file through BuildDir. If this file isn't linked, it just returns the directory part of the filename.

rsrcpath
The directory and file name to the source file linked to this file through BuildDir. If the file does not exist locally but exists in a Repository, the path in the Repository is returned. If this file isn't linked, it just returns the directory and filename unchanged.

rsrcdir The Repository directory containing the source file linked to this file through BuildDir. If this file isn't linked, it just returns the directory part of the filename.

For example, the specified target will expand as follows for the corresponding modifiers:

```
$TARGET              => sub/dir/file.x
${TARGET.base}       => sub/dir/file
${TARGET.dir}        => sub/dir
${TARGET.file}       => file.x
${TARGET.filebase}   => file
${TARGET.suffix}     => .x
${TARGET.abspath}    => /top/dir/sub/dir/file.x

BuildDir('sub/dir','src')
$SOURCE              => sub/dir/file.x
${SOURCE.srcpath}    => src/file.x
${SOURCE.srcdir}     => src

Repository('/usr/repository')
$SOURCE              => sub/dir/file.x
${SOURCE.rsrcpath}   => /usr/repository/src/file.x
${SOURCE.rsrcdir}    => /usr/repository/src
```

Lastly, a variable name may be a callable Python function associated with a construction variable in the environment. The function should take four arguments: *target* - a list of target nodes, *source* - a list of source nodes, *env* - the construction environment, *for_signature* - a Boolean value that specifies whether the function is being called for generating a build signature. SCons will insert whatever the called function returns into the expanded string:

```
def foo(target, source, env, for_signature):
    return "bar"

# Will expand $BAR to "bar baz"
env=Environment(FOO=foo, BAR="$FOO baz")
```

You can use this feature to pass arguments to a Python function by creating a callable class that stores one or more arguments in an object, and then uses them when the **__call__()** method is called. Note that in this case, the entire variable expansion must be enclosed by curly braces so that the arguments will be associated with the instantiation of the class:

```
class foo:
    def __init__(self, arg):
        self.arg = arg

    def __call__(self, target, source, env, for_signature):
        return arg + " bar"

# Will expand $BAR to "my argument bar baz"
env=Environment(FOO=foo, BAR="${FOO('my argument')} baz")
```

The special pseudo-variables **$(** and **$)** may be used to surround parts of a command line that may change *without* causing a rebuild--that is, which are not included in the signature of target files built with this command. All text between **$(** and **$)** will be removed from the command line before it is added to file signatures, and the **$(** and **$)** will be removed before the command is executed. For example, the command line:

```
echo Last build occurred $( $TODAY $). > $TARGET
```

would execute the command:

```
echo Last build occurred $TODAY. > $TARGET
```

but the command signature added to any target files would be:

```
echo Last build occurred  . > $TARGET
```

SCons uses the following rules when converting construction variables into command lines:

String
: When the value is a string it is interpreted as a space delimited list of command line arguments.

List
: When the value is a list it is interpreted as a list of command line arguments. Each element of the list is converted to a string.

Other
: Anything that is not a list or string is converted to a string and interpreted as a single command line argument.

Newline
Newline characters (\n) delimit lines. The newline parsing is done after all other parsing, so it is not possible for arguments (e.g. file names) to contain embedded newline characters. This limitation will likely go away in a future version of SCons.

Scanner Objects

You can use the **Scanner** function to define objects to scan new file types for implicit dependencies. Scanner accepts the following arguments:

function
A Python function that will process the Node (file) and return a list of strings (file names) representing the implicit dependencies found in the contents. The function takes three or four arguments:

```
def scanner_function(node, env, path):

def scanner_function(node, env, path, arg):
```

The **node** argument is the internal SCons node representing the file. Use **str(node)** to fetch the name of the file, and **node.get_contents()** to fetch contents of the file.

The **env** argument is the construction environment for the scan. Fetch values from it using the **env.Dictionary()** method.

The **path** argument is a tuple (or list) of directories that can be searched for files. This will usually be the tuple returned by the **path_function** argument (see below).

The **arg** argument is the argument supplied when the scanner was created, if any.

name The name of the Scanner. This is mainly used to identify the Scanner internally.

argument
An optional argument that, if specified, will be passed to the scanner function (described above) and the path function (specified below).

skeys An optional list that can be used to determine which scanner should be used for a given Node. In the usual case of scanning for file names, this argument will be a list of suffixes for the different file types that this Scanner knows how to scan. If the argument is a string, then it will be expanded into a list by the current environment.

path_function
A Python function that takes two or three arguments: a construction environment, directory Node, and optional argument supplied when the scanner was created. The **path_function** returns a tuple of directories that can be searched for files to be returned by this Scanner object.

node_class
The class of Node that should be returned by this Scanner object. Any strings or other objects returned by the scanner function that are not of this class will be run through the **node_factory** function.

node_factory
A Python function that will take a string or other object and turn it into the appropriate class of Node to be returned by this Scanner object.

scan_check
An optional Python function that takes two arguments, a Node (file) and a construction environment, and returns whether the Node should, in fact, be scanned for dependencies. This check can be used to eliminate unnecessary calls to the scanner function when, for example, the underlying file represented by a Node does not yet exist.

recursive
An optional flag that specifies whether this scanner should be re-invoked on the dependency files returned by the scanner. When this flag is not set, the Node subsystem will only invoke the scanner on the file being scanned, and not (for example) also on the files specified by the #include lines in the file being scanned.

SYSTEM-SPECIFIC BEHAVIOR

SCons and its configuration files are very portable, due largely to its implementation in Python. There are, however, a few portability issues waiting to trap the unwary.

.C file suffix

SCons handles the upper-case **.C** file suffix differently, depending on the capabilities of the underlying system. On a case-sensitive system such as Linux or UNIX, SCons treats a file with a **.C** suffix as a C++ source file. On a case-insensitive system such as Windows, SCons treats a file with a **.C** suffix as a C source file.

.F file suffix

SCons handles the upper-case **.F** file suffix differently, depending on the capabilities of the underlying system. On a case-sensitive system such as Linux or UNIX, SCons treats a file with a **.F** suffix as a Fortran source file that is to be first run through the standard C preprocessor. On a case-insensitive system such as Windows, SCons treats a file with a **.F** suffix as a Fortran source file that should *not* be run through the C preprocessor.

WIN32: Cygwin Tools and Cygwin Python vs. Windows Pythons

Cygwin supplies a set of tools and utilities that let users work on a Windows system using a more POSIX-like environment. The Cygwin tools, including Cygwin Python, do this, in part, by sharing an ability to interpret UNIX-like path names. For example, the Cygwin tools will internally translate a Cygwin path name like /cygdrive/c/mydir to an equivalent Windows pathname of C:/mydir (equivalent to C:\mydir).

Versions of Python that are built for native Windows execution, such as the python.org and ActiveState versions, do not have the Cygwin path name semantics. This means that using a native Windows version of Python to build compiled programs using Cygwin tools (such as gcc, bison, and flex) may yield unpredictable results. "Mixing and matching" in this way can be made to work, but it requires careful attention to the use of path names in your SConscript files.

In practice, users can sidestep the issue by adopting the following rules: When using gcc, use the Cygwin-supplied Python interpreter to run SCons; when using Microsoft Visual C/C++ (or some other Windows compiler) use the python.org or ActiveState version of Python to run SCons.

WIN32: scons.bat file

On WIN32 systems, SCons is executed via a wrapper **scons.bat** file. This has (at least) two ramifications:

First, Windows command-line users that want to use variable assignment on the command line may have to put double quotes around the assignments:

```
scons "FOO=BAR" "BAZ=BLEH"
```

Second, the Cygwin shell does not recognize this file as being the same as an **scons** command issued at the command-line prompt. You can work around this either by executing **scons.bat** from the Cygwin command line, or by creating a wrapper shell script named **scons .**

MinGW

The MinGW bin directory must be in your PATH environment variable or the PATH variable under the ENV construction variable for SCons to detect and use the MinGW tools. When running under the native Windows Python interpreter, SCons will prefer the MinGW tools over the Cygwin tools, if they are both installed, regardless of the order of the bin directories in the PATH variable. If you have both MSVC and MinGW installed and you want to use MinGW instead of MSVC, then you must explictly tell SCons to use MinGW by passing

```
tools=['mingw']
```

to the Environment() function, because SCons will prefer the MSVC tools over the MinGW tools.

EXAMPLES

To help you get started using SCons, this section contains a brief overview of some common tasks.

Basic Compilation From a Single Source File

```
env = Environment()
env.Program(target = 'foo', source = 'foo.c')
```

Note: Build the file by specifying the target as an argument ("scons foo" or "scons foo.exe"). or by specifying a dot ("scons .").

Basic Compilation From Multiple Source Files

```
env = Environment()
env.Program(target = 'foo', source = Split('f1.c f2.c f3.c'))
```

Setting a Compilation Flag

```
env = Environment(CCFLAGS = '-g')
env.Program(target = 'foo', source = 'foo.c')
```

Search The Local Directory For .h Files

Note: You do *not* need to set CCFLAGS to specify -I options by hand. SCons will construct the right -I options from CPPPATH.

```
env = Environment(CPPPATH = ['.'])
env.Program(target = 'foo', source = 'foo.c')
```

Search Multiple Directories For .h Files

```
env = Environment(CPPPATH = ['include1', 'include2'])
env.Program(target = 'foo', source = 'foo.c')
```

Building a Static Library

```
env = Environment()
env.StaticLibrary(target = 'foo', source = Split('l1.c l2.c'))
env.StaticLibrary(target = 'bar', source = ['l3.c', 'l4.c'])
```

Building a Shared Library

```
env = Environment()
env.SharedLibrary(target = 'foo', source = ['l5.c', 'l6.c'])
env.SharedLibrary(target = 'bar', source = Split('l7.c l8.c'))
```

Linking a Local Library Into a Program

```
env = Environment(LIBS = 'mylib', LIBPATH = ['.'])
env.Library(target = 'mylib', source = Split('l1.c l2.c'))
env.Program(target = 'prog', source = ['p1.c', 'p2.c'])
```

Defining Your Own Builder Object

Notice that when you invoke the Builder, you can leave off the target file suffix, and SCons will add it automatically.

```
bld = Builder(action = 'pdftex < $SOURCES > $TARGET'
        suffix = '.pdf',
        src_suffix = '.tex')
env = Environment(BUILDERS = {'PDFBuilder' : bld})
env.PDFBuilder(target = 'foo.pdf', source = 'foo.tex')

# The following creates "bar.pdf" from "bar.tex"
env.PDFBuilder(target = 'bar', source = 'bar')
```

Note also that the above initialization overwrites the default Builder objects, so the Environment created above can not be used call Builders like env.Program(), env.Object(), env.StaticLibrary(), etc.

Adding Your Own Builder Object to an Environment

```
bld = Builder(action = 'pdftex < $SOURCES > $TARGET'
        suffix = '.pdf',
        src_suffix = '.tex')
env = Environment()
env.Append(BUILDERS = {'PDFBuilder' : bld})
env.PDFBuilder(target = 'foo.pdf', source = 'foo.tex')
env.Program(target = 'bar', source = 'bar.c')
```

You also can use other Pythonic techniques to add to the BUILDERS construction variable, such as:

```
env = Environment()
env['BUILDERS]['PDFBuilder'] = bld
```

Defining Your Own Scanner Object

```
import re

include_re = re.compile(r'^include\s+(\S+)$', re.M)

def kfile_scan(node, env, path, arg):
  contents = node.get_contents()
  includes = include_re.findall(contents)
  return includes

kscan = Scanner(name = 'kfile',
          function = kfile_scan,
          argument = None,
```

```
            skeys = ['.k'])
scanners = Environment().Dictionary('SCANNERS')
env = Environment(SCANNERS = scanners + [kscan])

env.Command('foo', 'foo.k', 'kprocess < $SOURCES > $TARGET')

bar_in = File('bar.in')
env.Command('bar', bar_in, 'kprocess $SOURCES > $TARGET')
bar_in.target_scanner = kscan
```

Creating a Hierarchical Build

Notice that the file names specified in a subdirectory's SConscript file are relative to that subdirectory.

```
SConstruct:

    env = Environment()
    env.Program(target = 'foo', source = 'foo.c')

    SConscript('sub/SConscript')

sub/SConscript:

    env = Environment()
    # Builds sub/foo from sub/foo.c
    env.Program(target = 'foo', source = 'foo.c')

    SConscript('dir/SConscript')

sub/dir/SConscript:

    env = Environment()
    # Builds sub/dir/foo from sub/dir/foo.c
    env.Program(target = 'foo', source = 'foo.c')
```

Sharing Variables Between SConscript Files

You must explicitly Export() and Import() variables that you want to share between SConscript files.

```
SConstruct:

    env = Environment()
    env.Program(target = 'foo', source = 'foo.c')

    Export("env")
    SConscript('subdirectory/SConscript')

subdirectory/SConscript:

    Import("env")
    env.Program(target = 'foo', source = 'foo.c')
```

Building Multiple Variants From the Same Source

Use the BuildDir() method to establish one or more separate build directories for a given source directory, then use the SConscript() method to specify the SConscript files in the build directories:

SConstruct:

```
ccflags = '-DFOO'
Export("ccflags")
BuildDir('foo', 'src')
SConscript('foo/SConscript')

ccflags = '-DBAR'
Export("ccflags")
BuildDir('bar', 'src')
SConscript('bar/SConscript')
```

src/SConscript:

```
Import("ccflags")
env = Environment(CCFLAGS = ccflags)
env.Program(target = 'src', source = 'src.c')
```

Note the use of the Export() method to set the "ccflags" variable to a different value for each variant build.

Hierarchical Build of Two Libraries Linked With a Program

SConstruct:

```
env = Environment(LIBPATH = ['#libA', '#libB'])
Export('env')
SConscript('libA/SConscript')
SConscript('libB/SConscript')
SConscript('Main/SConscript')
```

libA/SConscript:

```
Import('env')
env.Library('a', Split('a1.c a2.c a3.c'))
```

libB/SConscript:

```
Import('env')
env.Library('b', Split('b1.c b2.c b3.c'))
```

Main/SConscript:

```
Import('env')
e = env.Copy(LIBS = ['a', 'b'])
e.Program('foo', Split('m1.c m2.c m3.c'))
```

The '#' in the LIBPATH directories specify that they're relative to the top-level directory, so they don't turn into "Main/libA" when they're used in Main/SConscript.

Specifying only 'a' and 'b' for the library names allows SCons to append the appropriate library prefix and suffix for the current platform (for example, 'liba.a' on POSIX systems,

Customizing contruction variables from the command line.

The following would allow the C compiler to be specified on the command line or in the file custom.py.

```
opts = Options('custom.py')
opts.Add('CC', 'The C compiler.')
env = Environment(options=opts)
Help(opts.GenerateHelpText(env))
```

The user could specify the C compiler on the command line:

```
scons "CC=my_cc"
```

or in the custom.py file:

```
CC = 'my_cc'
```

or get documentation on the options:

```
$ scons -h

CC: The C compiler.
    default: None
    actual: cc
```

Using Microsoft Visual C++ precompiled headers

Since windows.h includes everything and the kitchen sink, it can take quite some time to compile it over and over again for a bunch of object files, so Microsoft provides a mechanism to compile a set of headers once and then include the previously compiled headers in any object file. This technology is called precompiled headers. The general recipe is to create a file named "StdAfx.cpp" that includes a single header named "StdAfx.h", and then include every header you want to precompile in "StdAfx.h", and finally include "StdAfx.h" as the first header in all the source files you are compiling to object files. For example:

```
StdAfx.h:
    #include <windows.h>
    #include <my_big_header.h>

StdAfx.cpp:
    #include <StdAfx.h>

Foo.cpp:
    #include <StdAfx.h>

    /* do some stuff */

Bar.cpp:
    #include <StdAfx.h>

    /* do some other stuff */

SConstruct:
    env=Environment()
    env['PCHSTOP'] = 'StdAfx.h'
    env['PCH'] = env.PCH('StdAfx.cpp')[0]
    env.Program('MyApp', ['Foo.cpp', 'Bar.cpp'])
```

For more information see the document for the PCH builder, and the PCH and PCHSTOP construction variables. To learn about the details of precompiled headers consult the MSDN documention for /Yc, /Yu, and /Yp.

Using Microsoft Visual C++ external debugging information

Since including debugging information in programs and shared libraries can cause their size to increase significantly, Microsoft provides a mechanism for including the debugging information in an external file called a PDB file. SCons supports PDB files through the PDB construction variable.

SConstruct:

```
env=Environment()
env['PDB'] = 'MyApp.pdb'
env.Program('MyApp', ['Foo.cpp', 'Bar.cpp'])
```

For more information see the document for the PDB construction variable.

ENVIRONMENT

SCONS_LIB_DIR

Specifies the directory that contains the SCons Python module directory (e.g. /home/aroach/scons-src-0.01/src/engine).

SCONSFLAGS

A string of options that will be used by scons in addition to those passed on the command line.

SEE ALSO

scons User Manual, **scons** Design Document, **scons** source code.

AUTHORS

Steven Knight <knight@baldmt.com>
Anthony Roach <aroach@electriceyeball.com>