Creating a Project Using an Existing Build System

In this section:

Introduction

If you have a custom build system (for example, Makefile-based), you can streamline the process of configuring build settings by using C++test's cpptestscan utility, which collects information about your build process. You can then point C++test to the generated "build data file" to configure build settings, as described in Use options from a build data file.

Proper Compiler Configuration is Critical

In most cases, C++test needs to invoke the compiler and linker in order to perform static analysis and runtime testing tasks, which commonly involve preprocessing, compiling, and linking programs.
To access C++test's full functionality, the machine where C++test is run must have the complete development environment and compiler tool chain.

About Build Data Files (.bdf)

Build information, such as the working directory, command line options for the compilation, and link processes of the original build, are stored in a file called the build data file. The following example is a fragment from a build data file:

```
------- cpptestscan v. 9.4.x.x -------
working_dir=/home/place/project/hypnos/pscom
project_name=pscom
arg=g++
arg=-c
arg=src/io/Path.cc
arg=-Iinclude
arg=-I.
arg=-o
arg=/home/place/project/hypnos/product/pscom/shared/io/Path.o
```

Using `cpptestscan` or `cpptesttrace` to Create a Build Data File

The `cpptestscan` and `cpptesttrace` executables are located in the C++test installation directory. They collect information from the build process of an existing code base, generate build data files with the information, and append information about each execution into a file.

The `cpptestscan` utility is used as a wrapper for the compiler and/or linker during the normal build. To use `cpptestscan` with an existing build, build the code base with `cpptestscan` as the prefix for the compiler / linker executable of an existing build to build the code base. This can be done in two ways:

- Modify the build command line to use `cpptestscan` as the wrapper for the compiler/linker executables
- If you don't want to (or cannot) override the compiler variable on the command line, embed `cpptestscan` in the actual make file or build script.

To use `cpptesttrace` with an existing build, build the code base with `cpptesttrace` as the prefix for the entire build command. `cpptesttrace` will trace the compiler and linker processes executed during the build and store them in the build data file.

In both cases, you need to either add the C++test installation directory to your PATH environment variable, or specify the full path to either utility.

Additional options for `cpptestscan` and `cpptesttrace` are summarized in the following table. Options can be set directly for the `cpptestscan` command or via environment variables. Most options can be applied to `cpptestscan` or `cpptesttrace` by changing the prefix in command line.

Basic `cpptestscan` usage:

- Windows: `cpptestscan.exe [options] [compile/link command]`
- Linux: `cpptestscan [options] [compile/link command]`

Basic `cpptesttrace` usage:

- Windows: `cpptesttrace.exe [options] [build command]`
- Linux: `cpptesttrace [options] [build command]`

Option	Environment Variable	Description	Default

`--cpptestscanOutputFile=<OUTPUT_FILE>`
 	
`--cpptesttraceOutputFile=<OUTPUT_FILE>`
	CPPTEST_SCAN_OUTPUT_FILE)	Defines file to append build information to.	cpptestscan.bdf

`--cpptestscanProjectName=<PROJECT_NAME>`
 	
`--cpptesttraceProjectName=<PROJECT_NAME>`
	CPPTEST_SCAN_PROJECT_NAME	Defines suggested name of the C++test project.	name of the current working directory

`--cpptestscanRunOrigCmd=[yes|no]`
 	
`--cpptesttraceRunOrigCmd=[yes|no]`
	CPPTEST_SCAN_RUN_ORIG_CMD	If set to "yes", original command line will be executed.	yes

`--cpptestscanQuoteCmdLineMode=[all|sq|none]`
 	
`--cpptesttraceQuoteCmdLineMode=[all|sq|none]`
	CPPTEST_SCAN_QUOTE_CMD_LINE_MODE	Determines the way C++test quotes parameters when preparing cmd line to run.
		
		`all`: all params will be quoted
		
		`none`: no params will be quoted
		
		`sq`: only params with space or quote character will be quoted
		
		cpptestscanQuoteCm dLineMode is not supported on Linux
			all

`--cpptestscanCmdLinePrefix=<PREFIX>`
 	
`--cpptesttraceCmdLinePrefix=<PREFIX>`
	CPPTEST_SCAN_CMD_LINE_PREFIX	If non-empty and running original executable is turned on, the specified command will be prefixed to the original command line.	[empty]

`--cpptestscanEnvInOutput=[yes|no]`
 	
`--cpptesttraceEnvInOutput=[yes|no]`
	CPPTEST_SCAN_ENV_IN_OUTPUT	Enabling dumps the selected environment variables and the command line arguments that outputs the file. For advanced settings use – cpptestscanEnvFile and – cpptestscanEnvars options	no

`--cpptestscanEnvFile=<ENV_FILE>`
 	
`--cpptesttraceEnvFile=<ENV_FILE>`
	CPPTEST_SCAN_ENV_FILE	If enabled, the specified file keeps common environment variables for all build commands; the main output file will only keep differences. Use this option to reduce the size of the main output file. Use this option with –cpptestscanEnvInOut put enabled	[empty]

`--cpptestscanEnvars=[*|<ENVAR_NAME>,...]`
 	
`--cpptesttraceEnvars=[*|<ENVAR_NAME>,...]`
	CPPTEST_SCAN_ENVARS	Selects the names of environment variables to be dumped or '*' to select them all. Use this option with – cpptestscanEnvInOut put enabled.	*

`--cpptestscanUseVariable=[VAR_NAME=VALUE,...]`
 	
`--cpptesttraceUseVariable=[VAR_NAME=VALUE,...]`
	CPPTEST_SCAN_USE_VARIABLE	Replaces each occurence of "VALUE" string in the scanned build information with the "${VAR_NAME}" variable usage.	[empty]

`--cpptesttraceTraceCommand`	CPPTEST_SCAN_TRACE_COMMAND	Defines the command names that will be traced when collecting build process information. These names, specified as regular expressions, should match original compiler / linker commands used in the build process.	

Example: Modifying GNU Make Build Command to Using cpptestscan

Assuming that a make-based build in which the compiler variable is CXX and the original compiler is g++:

```
make -f </path/to/makefile> <make target> [user-specific options] CXX="cpptestscan --cpptestscanOutputFile=/path/to/name.bdf --cpptestscanProjectName=<projectname> g++"
```

This will build the code as usual, as well as generate a build data file (name.bdf) in the specified directory.

> **Note**
>
> When the build runs in multiple directories:
>
> - If you do not specify output file, then each source build directory will have its own .bdf file. This is good for creating one project per source directory.
> - If you want a single project per source tree, then a single .bdf file needs to be specified, as shown in the above example.

Example: Modifying GNU Make Build Command Using cpptesttrace

Assume that a regular make-based build is executed with:

```
make clean all
```

you could use the following command line:

```
cpptesttrace --cpptesttraceOutputFile=/path/to/name.bdf --cpptesttraceProjectName=<projectname> make clean all
```

This will build the code as usual and generate a build data file (name.bdf) in the specified directory.

Note

If the compiler and/or linker executable names do not match default `cpptesttrace` command patterns, they you will need to use `--cpptesttraceTraceCommand` option described below to customize them. Default cpptestscan command trace patterns can be seen by running 'cpptesttrace --cpptesttraceHelp' command.

Example: Modifying GNU Makefile to use cpptestscan

If your Makefile uses CXX as a variable for the compiler executable and is normally defined as CXX=g++, you can redefine the variable:

```
     ifeq ($(BUILD_MODE), PARASOFT_CPPTEST)
     CXX="/usr/local/parasoft/cpptestscan --cpptestscanOutputFile=<selected_location>/MyProject.bdf --
cpptestscanProjectName=MyProject g++"
     else
     CXX=g++
     endif
```

Next, run the build as usual and specify an additional BUILD_MODE variable for make:

```
     make BUILD_MODE=PARASOFT_CPPTEST
```

The code will be built and a build data file (MyProject.bdf) will be created. The generated build data file can then be used to create a project from the GUI or from the command line.

Note

The `cpptestscan` and `cpptesttrace` utilities can be used in the parallel build systems where multiple compiler executions can be done concurrently. When preparing Build Data File on the multicore machine, for example, you can pass the `-j <number_of_parallel_jobs>` parameter to the GNU make command to build your project and quickly prepare the Build Data File.

When should I use cpptestscan?

It is highly recommended that the procedures to prepare a build data file are integrated with the build system. In this way, generating the build data file can be done when the normal build is performed without additional actions.

To achieve this, prefix your compiler and linker executables with the cpptestscan utility in your Makefiles / build scripts.

When should I use cpptesttrace?

Use cpptesttrace as the prefix for the whole build command when modifying your Makefiles / build scripts isn't possible or when prefixing your compiler / linker executables from the build command line is too complex.