Creating Projects - Standalone or Eclipse Plugin

Before you can test code with C++test, it must be available in a C++test project within the workbench (a temporary work area-- unique for each user). This involves:

1. Creating or importing a project:
    - If you have **existing Eclipse CDT, Wind River Workbench, QNX Momentics, or ARM RVDS projects**, simply configure your existing projects for use with C++test. Proceed to step 2.
    - If you have **a build system that runs from the command line**, use the `cpptestscan` utility to collect information from the build process, then have C++test automatically create a project based on the collected information. Projects can be created from the command line or using a GUI wizard. For details, see "Creating a C++test Project Using an Existing Build System" on page 23.
    - If you want to **import existing Visual Studio 6.0 projects**, use the procedure described in the "Importing an Existing Visual Studio 6.0 Project" User's Guide section.
    - If you want to **import existing Wind River Tornado projects**, use the procedure described in the "Importing an Existing Wind River Tornado Project" User's Guide section.
    - If you want to **import existing IAR Embedded Workbench projects**, use the procedure described in the "IAR Embedded Workbench Support" User's Guide section.
    - If you want to **import existing Keil uVision projects**, use the procedure described in the "Keil RealView MDK Support" User's Guide section.
    - If you want to **import existing Renesas HEW projects**, use the procedure described in the "Renesas High-performance Embedded Workbench Support" User's Guide section.
    - If you want to **import existing Texas Instruments Code Composer Studio 3.x projects**, use the procedure described in the "Texas Instruments Code Composer Studio Plugin" User's Guide section.
    - If you want to **import existing Microsoft eMbedded Visual C++ projects**, use the procedure described in the "Microsoft eMbedded Visual C++ Support" User's Guide section.
    - Otherwise, configure your project from the GUI as described in the "Creating a C++test Project From the GUI" User's Guide section.
2. Configuring build settings within the C++test project options.
    - This is required for all C++test projects-- including existing Eclipse CDT, Wind River Workbench, QNX Momentics, Texas Instruments Code Composer Studio 4.x/5.x, and ARM RVDS projects that you want to test with C++test.
    - See "Verifying Build Settings" on page 34 for details.

These two tasks need to be performed only once. After one team member creates and configures a project, it can be checked into source control, then reused by all developers working on the code as described in the "Sharing the Project" User's Guide topic.

Creating a C++test Project Using an Existing Build System

You can create a C++test project by using the `cpptestscan` utility to collect information from the build process for an existing code base, then having C++test automatically create a project based on the collected information.

The Build Data File concept

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.

You can use the cpptestscan or cpptesttrace utility to create a C++test project that you would normally build using tools such as GNU make, CMake, and QMake. The utilities collect information from the build process of an existing code base that C++test can use to automatically create a project. You can also build a project and manually configure it using the information collected by the utilities.

This information can then be used to create a project (from the GUI or from the command line).You can then point C++test to the generated "build data file" to configure build settings.

An example fragment of this file is included below.

```
------- 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
```

The build data file can be used as a source of information about project source files, compiler executable, compiler options, linker executable, and options used to build the project. There are three ways to use the build data file to create a project:

- Manually setting up 'Use options from the build data file' as the options source for the project and selecting appropriate build data file (see "Creating a C++test Project From the GUI" section on page 187).
- Using the GUI to automatically import a project. See "Importing project using Build Data File with the GUI wizard" section on page 171.
- Using the command line to automatically import a project. See "Importing a Project from the Command Line" section on page 179.

Note

Required environment variables can also be stored in the build data file if the following apply:

- Your build system sets up the required environment variables for the compiler / linker to work correctly
- These variables are not available in the environment when run C++tests.

See description of the '--cpptestscanEnvInOutput' option below.

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 and Solaris: `cpptestscan [options] [compile/link command]`

Basic `cpptesttrace` usage:

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

To ensure that C++test's project options remain in sync with any changes to the build system, set up the build so that it always updates the build data file.

Option	Environment Variable	Description	Default
–cpptestscanOutput-File=<OUTPUT_FILE> –cpptesttraceOutput-File=<OUTPUT_FILE>	CPPTEST_SCAN_OUTPUT_FILE)	Defines file to append build information to.	cpptestscan.bdf
–cpptestscanProject-Name=<PROJECT_NAME> –cpptesttraceProject-Name=<PROJECT_NAME>	CPPTEST_SCAN_PROJECT_NAME	Defines suggested name of the C++test project.	name of the current working directory
–cpptestscanRun-OrigCmd= [yes|no] –cpptesttraceRun-OrigCmd=[yes|no]	CPPTEST_SCAN_RUN_ORIG_CMD	If set to "yes", original command line will be executed.	yes

Option	Environment Variable	Description	Default
–cpptestscanQuoteCmdLineMode=[all|sq|none] –cpptesttraceQuoteCmdLineMode=[all|sq|none]	CPPTEST_SCAN_QUOTE_CMD_LINE_MOD	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 cpptestscanQuoteCmdLineMode 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 --cpptestscanEnvInOutput enabled	[empty]

Option	Environment Variable	Description	Default
–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 --cpptestscanEnvInOutput 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]

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 OutputFile, 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 --cpptestscanOutputFile=/path/to/name.bdf --cpptestscanProjectName=<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 be then used to create a project from the GUI or from the command line.

For non-make based build systems, usage of `cpptestscan` is very similar. Typically, a compiler is defined as a variable somewhere in the build scripts. To create a project from that build system, replace the compiler variable with `cpptestscan <compiler_name>`.

Notes

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 multi-core machine, for example, you can pass the `-j <number_of_parallel_jobs>` parameter to the GNU make command to build your project and prepare the Build Data File faster.

Ensuring that the correct compiler is used

When using `cpptestcli` to create a project from a .bdf file, create an options file containing the line `bdf.import.compiler.family=<family>` and pass that file to `cpptestcli` using the `-localsettings` switch . For example:

```
cpptestcli -bdf name.bdf -localsettings myCompiler.properties ...
```

Example: Using cpptesttrace with CMake build

Assuming that you have a CMake-based build, you can produce a build data file using cpptesttrace:

1. Run the original CMake command to use CMake to generate make files. For example:

```
cmake -G "Unix Makefiles" ../project_root
```

2. Setup environment variables for `cpptestscan` making sure to use an absolute path for the output file:

```
export CPPTEST_SCAN_PROJECT_NAME=my_project
export CPPTEST_SCAN_OUTPUT_FILE=$PROJ_ROOT/cpptestscan.bdf
```

3. Make sure the `cpptesttrace` executable is available on the PATH.
4. Run the project build normally but with `cpptesttrace` as a wrapper. For example if normal build command is `make clean all` for the build with 'cpptesttrace' the command will be `cpptesttrace make clean all`

A build data file will be generated in the location defined by the CPPTEST_SCAN_OUTPUT_FILE variable. If the varialbe is isn't set, the build data file(s) will be generated in the Makefiles' locations.

Example: Using cpptestscan with CMake build

All scripts and commands are bash-based – adapt them as needed for different shells.

Assuming a CMake-based build, do the following to produce a build data file using cpptestscan:

1. Use CMake to re-generate make files with the 'cpptestscan' used as a compiler prefix. Make sure the ‘cpptestscan’ executable is available on the PATH.
    a. If original CMake command is `cmake -G "Unix Makefiles" ../project_root`, then you need to get rid of existing CMake cache and run cmake overriding compiler variables. The following example assumes 'gcc' is used as a C compiler and 'g++' as a C++ compiler executable:

```
rm CMakeCache.txt
CC="cpptestscan gcc" CXX="cpptestscan g++" cmake -G "Unix Makefiles" ../project_root
```

    b. Look in the CMakeCache.txt file to see if CMAKE_*_COMPILER variables point to cpptestscan.
    c. If the make files are re-generated, jump to step 5. Continue if `cmake` failed in the bootstrap phase because the compiler wasn’t recognized.
2. Prepare the `cpptestscan` wrapper scripts that will behave like a CMake compiler by creating the following BASH scripts. In this example, we assume 'gcc' is used as a C compiler and 'g++' as a C++ compiler executable:

```
>cat cpptest_gcc.sh
#!/bin/bash
cpptestscan gcc --cpptestscanRunOrigCmd=no $* > /dev/null 2>&1
gcc $*
exit $?

>cat cpptest_g++.sh
#!/bin/bash
cpptestscan g++ --cpptestscanRunOrigCmd=no $* > /dev/null 2>&1
g++ $*
exit $?
```

    - The first script invokes `cpptestscan` to extract options without running the compiler. The second script runs the actual compiler so that the entire script looks and acts like a compiler in order to be “accepted” by CMake.
3. Give the scripts executable attributes and place them in a common location so they are accessible to everyone who needs to scan make files. Make sure the `cpptestscan` and the scripts are available on the PATH.
4. Use CMake to re-generate make files with the scripts used as compilers by extending the original CMake command with options that re-generate make files from the prepared scripts.
    - If original CMake command is `cmake -G "Unix Makefiles" ../project_root` then you need to get rid of existing CMake cache and run cmake overriding compiler variables. In the following example, we assume 'gcc' is used as a C compiler and 'g++' as a C++ compiler executable:

```
rm CMakeCache.txt
cmake -G "Unix Makefiles" -D CMAKE_C_COMPILER=cpptest_gcc.sh -D CMAKE_CXX_COMPILER=cpptest_g++.sh
../project_root
```

    - Look in the CMakeCache.txt file to see if CMAKE_*_COMPILER variables point to prepared wrappers.

5. Setup environment variables for `cpptestscan`; make sure to use an absolute path for the output BDF file:

```
export CPPTEST_SCAN_PROJECT_NAME=my_project
export CPPTEST_SCAN_OUTPUT_FILE=$PROJ_ROOT/cpptestscan.bdf
```

6. Run the project build normally without overwriting any make variables. Build data files(s) will be generated in location defined by the CPPTEST_SCAN_OUTPUT_FILE variable or - if not set - in location of Makefiles.

Note

By default CMake-generated make files only print information about performed actions without actual compiler/linker command lines. Add “VERBOSE=1” to the make command line to see executed compiler/linker command lines.

Using cpptestscan or cpptestrace with other Build Systems

For non-make based build systems, usage of cpptestscan and cpptesttrace is very similar to the examples shown above. Typically, a compiler is defined as a variable somewhere in the build scripts. To create a Build Data File from that build system using cpptestscan, prefix the original compiler executable with cpptestscan. To create a Build Data File from that build system using cpptesttrace, prefix whole build command line with cpptesttrace.

"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.

Creating a Project with the GUI Wizard

Once you have used `cpptestscan` to generate a build data file for code you want to test in C++test, you can use the Project Creation wizard to create a C++test project.

To create a project from a build data file:

1. Open the wizard by choosing **File> New> Project**, select **C++test> Create project from a build data file**, then click **Next**. The wizard’s first page will display.
2. Complete the first wizard page, then click **Next**.
    - In the **Build data file** field, enter or browse to the location of the build data file that was previously created.
    - In the **Project location** section, specify where you want the projects created. There are two possibilities: workspace and external location. If workspace is chosen, the projects will be created in subdirectories within the workspace location. If external location is chosen, single projects will be created directly in that location. If multiple projects are created, then subdirectories for each project will be created in the specified external location.

- In the **Compiler settings** section, specify the compiler family. The other options will be set automatically.

3. In the second wizard page, which displays a project tree with the projects that will be created, verify the project's structure and content, making modifications as needed.
   - The first level lists all projects that will be created. To change the project's name or link additional folders to the project, right-click the project name and choose the appropriate shortcut menu command.
   - The second level lists all folders that will be linked. To change a folder's name or prevent it from being included in the project, right-click the folder name and choose the appropriate shortcut menu command.
   - Deeper in the tree, you will see all folders and files which are present in linked folders. A green marker is used to indicate files referenced in the .bdf file.
4. (Optional) In the third wizard page, set Path Variables if needed.
   - If you want Path Variables used in linked folders, check **Use Path Variable to define linked folder locations (if applicable)**.
   - To use a pre-defined Path Variable, select it from the **Path Variable list** box.
   - To use a custom Path Variable, choose **Custom** from the **Path Variable list** box, then manually enter the Path Variable name and value in corresponding fields.
5. Click **Finish**. C++test will create the specified project(s) in the specified location. The project(s) will include all source files whose options were scanned, and project properties should be set up appropriately.

Creating a Project from the Command Line

You can also create a BDF-based projects in command line mode by using the `-bdf <cpptestscan.bdf>` switch to `cpptestcli`.

If you want to perform analysis (e.g., static analysis and/or test generation) immediately after the project is created, ensure that the cpptestcli command uses -config to invoke the preferred Test Configuration. For example:

```
cpptestcli -data "</path/to/workspace>" -resource "<projectname>" -config "team://Team
Configuration"  -localsettings "</path/to/name.properties>" -bdf  "</path/to/name.bdf>"
```

If you simply want to create the project (without performing any analysis), omit -config. For example:

```
cpptestcli -data "</path/to/workspace>" -resource "<projectname>" -localsettings "</
path/to/name.properties>" -bdf  "</path/to/name.bdf>"
```

Note that -config "util/CreateProjectOnly", which was previously used for creating a project without testing, is no longer used in the current version of C++test. The fake Test Configuration "util/CreateProjectOnly" is no longer supported.

You can define custom project settings in a plain text options file, which is passed to `cpptestcli` using the `-localsettings` switch. The following settings can be specified in the options file:

Option	Description
`bdf.import.location=[WORKSPACE|ORIG|<path>]`	You can specify an external location, or use the keyword WORKSPACE. If WORKSPACE is used, projects will be created in subdirectories within the workspace directory. WORKSPACE is the default.
`bdf.import.pathvar.enabled=[true|false]`	Specifies if Path Variables should be used in linked folders that will be created in the new projects. The default is false.
`bdf.import.pathvar.name=<name>`	Specifies the name of the Path Variable (if Path Variables are used, per the bdf.import.pathvar.enabled property). The default Path Variable name is DEVEL_ROOT_DIR.
`bdf.import.pathvar.value=<path>`	Specifies the value of the Path Variable (if Path Variables are used, per the bdf.import.pathvar.enabled property). The default value is the most common root directory for all linked folders.
`bdf.import.compiler.family=<compiler_family>`	Specifies what compiler family will be used (for example, vc_6_0, vc_7_0, vc_7_1, vc_8_0, gcc_2_9, gcc_3_2, gcc_3_3, gcc_3_4, ghs_4_0). For a custom compiler, you need to use the custom compiler family identifier, which is the name of the directory containing gui.properties, c.psrc and cpp.psrc files). If this property is not specified, the default values will be used.
`bdf.import.c.compiler.exec=<exec>`	Specifies the executable of the C compiler that will be used in the created project.
`bdf.import.cpp.compiler.exec=<exec>`	Specifies the executable of the C++ compiler that will be used in the created project.
`bdf.import.linker.exec=<exec>`	Specifies the executable of the linker that will be used in the created project.
`bdf.import.project.<proj_name>=dir1;dir2;dir3`	Specifies the set of folders to link for the project prj_name. Folders should be specified as a value list of folder paths, separated with semicolons.

Example

The following is an example of how to create a C++test project from the command line using `cpptestscan`. The example uses a make-based build; however, a .bdf file can be produced from any build system:

1. Run a build with `cpptestscan` added as a prefix to your compiler/linker. Assuming that the compiler variable is CXX, and the original compiler is g++, use the following command line:
   ```
   make -f </path/to/makefile> <make target> [user-specific options]
   CXX="cpptestscan --cpptestscanOutputFile=/path/to/name.bdf --cpptestscanProjectName=<projectname> g++"
   ```
   `cpptestscan` will collect and store build information in the `name.bdf` file.
2. If you need to override the defaults for creating a project, create a plain text options file named `name.properties`. For instance, you could specify the following bdf options:
   - The compiler which is not default, but is listed among the supported compilers:
     `bdf.import.compiler.family=<compilername>`

- The external location where the project will be created:
  `bdf.import.location=<path/to/projectname>`
- The source directory for the project with linked sources:
  `bdf.import.project.<projectname>=</path/to/source>`

3. Create the project by running `cpptestcli` with the `-bdf` switch. For instance,
   `cpptestcli -data "</path/to/workspace>" -resource "<projectname>" -localsettings "</path/to/name.properties>" -bdf "</path/to/name.bdf>"`
   - Note that the `-config` switch must be included.
   - Omitting `-config` tells C++test to simply want to create the project (without performing any analysis).
   - If you want to perform analysis (e.g., static analysis and/or test generation) immediately after the project is created, do not omit `-config` Instead, use `-config` to invoke your preferred Test Configuration (e.g., `-config "team://Team Configuration")`.

The created project can later be tested with the desired Test Configuration by using
`cpptestcli -data "</path/to/workspace>" -resource "<projectname>" -config "<test configuration>"`