Collecting Application Coverage with cpptestcc

In this section:

Introduction

C/C++test's multi-metric coverage analysis allows you to monitor code coverage when executing a standalone application or library outside C/C++test.

C/C++test ships with a standalone coverage package that consists of the following components:

- `[INSTALL_DIR]/cpptestcc` - A coverage tool that integrates into your build process to instrument your application to collect raw coverage data.
- `[INSTALL_DIR]/engine/coverage/runtime` - A coverage runtime library that needs to be integrated with the instrumented application.

Collecting coverage with `cpptestcc` involves three phases:

1. Instrumenting the application by integrating the `cpptestcc` tool into your build.
2. Executing instrumented code and collecting raw coverage data.
3. Reviewing the coverage with C/C++test by importing the raw coverage data into C/C++test with a built-in test configuration.

Quick Start with `cpptestcc`

1. Add the path to the `[INSTALL_DIR]` to the `PATH` system variable to enable execution of the `cpptestcc` tool.
2. Update your compilation command to include the `cpptestcc` executable as a prefix to the compiler command using `--` as a separator. For example:

   Original compilation command line

   ```
   cc -I app/includes -D defines -c source.cpp
   ```

   Updated compilation command line

   ```
   cpptestcc -compiler gcc_7 -line-coverage -- cc -I app/includes -D defines -c source.cpp
   ```

   

   At minimum, the `cpptestcc` tool requires the following parameters to be configured on the command line:

   - a compiler identifier: `-compiler <COMPILER_ID>`

   - a coverage metric (for example, `-decision-coverage`)

   See Command Line Reference for cpptestcc for information about other options.

3. Update your linker command with the path to the pre-built coverage runtime library shipped with C/C++test to add the library to your application. For example:

   Original command line

   ```
   lxx -L app/lib app/source.o somelib.lib -o app.exe
   ```

   Updated command line

   ```
   lxx -L app/lib app/source.o somelib.lib [INSTALL_DIR]/engine/coverage/runtime/lib/cpptest.lib -o app.exe
   ```

> **Important**
>
> If the coverage runtime library is linked as a shared (dynamic-load) library, you must ensure that it can be loaded when the instrumented application is started. This typically requires adding `[INSTALL_DIR]/engine/coverage/runtime/bin` to the`PATH` environment variable (on Windows) or `[INSTALL_DIR]/engine/coverage/runtime/lib` to the `LD_LIBRARY_PATH` environment variable (on Linux).

> C/C++test provides the pre-built coverage runtime library for native Windows and Linux applications. For cross-platform and embedded testing, the runtime library needs to built from sources that are available in `[INSTALL_DIR]/engine/coverage/runtime`. See Coverage Runtime Library for details.

4. Build the application. When instrumenting the code, `cpptestcc` creates the `.cpptest/cpptestcc` folder where important coverage-related data ("coverage maps") are stored. By default, the folder is located in the working directory of the current compilation. You can change the default location using the `-workspace <path>` option; see Command Line Reference for cpptestcc for details.
5. Run the application. The coverage data ("coverage log") will be stored it the `cpptest_results.clog` file.
6. In your IDE where C/C++test is installed, create a new project that includes all the source files of the application.
   Ensure that the files and all the paths remain unchanged.
7. Select the project and choose **Parasoft> Test Configurations> Utilities> Load Application Coverage** from your IDE menu to import the coverage data (see Importing the Coverage Data for details).
8. Review the coverage information (see Reviewing Coverage Information).

Importing the Coverage Data

The Load Application Coverage test configuration assumes that both `.cpptest/cpptestcc` folder and `cpptest_results.clog` file are stored in the default location. To customize the location, configure the following execution details in the test configuration (**Execution> General> Execution details**):
- **Coverage map files root location** - default: `${project_loc}/.cpptest/cpptestcc`
- **Coverage log files** - default: `${project_loc}/*.clog`

By default, the Load Application Coverage test configuration tries to load information about all the supported coverage metrics. To customize the list of supported coverage metricts, go to **Execution> General> Execution details> Instrumentation Mode> Instrumentation features> C/C++ Code Coverage metrics** and select the metrics you want to report.

We recommend that you keep consistency with the metrics enabled for the `cpptestcc` tool in the compilation command line.

Command Line Reference for `cpptestcc`

You can run the following command to print out the available options to the console: `cpptestcc -help`

The following options are available:

-compiler <name|path>

Specifies the name of the compiler configuration you want to use for code analysis and instrumentation. See Compilers for the list of supported compilers or use the `-list-compilers` command line option to print out the list of supported compilers to the console.

Examples:

- `cpptestcc -compiler gcc_3_4`
- `cpptestcc -compiler vc_11_0`

Configuration file format (see -psrc): `cpptestcc.compiler <name>`

-list-compilers

Prints out the names of all supported compiler configurations.

Configuration file format (see -psrc): `cpptestcc.listCompilers`

-include <file|pattern> and -exclude <file|pattern>

Includes into or excludes from the instrumentation scope all the file(s) that match the specified pattern.

Final filtering is determined only after all include/exclude entries have been specified in the order of their specification.

The following wildcards are supported:

- ? - Any character
- * - Any sequence of characters

To prevent shells from expanding * wildcards to the list of files or directories, you can use the `regex:` prefix to specify the value.

These options can be specified multiple times.

Configuration file format (see -psrc): `cpptestcc.include <path|pattern>`

Example 1:

Sample project layout:

```
<project root>
 + external_libs
 + src
 + include
```

If your project has the above layout, the following command will exclude all the files in the `external_libs` directory from instrumentation scope:

```
cpptestcc -include regex:*/<project root>/* -exclude regex:*/<project root>/external_libs <other command line options>
```

Example 2:

Sample project layout:

```
<project root>
<sourcefiles>.cpp
<headerfiles>.hpp
```

If your project has the above layout, the following command will only instrument the header files (the source files will not be instrumented):

```
cpptestcc -include regex:* -exclude regex:*.cpp <remaining part of cmd>
```

-ignore <pattern>

Specifies the source files that will be ignored during processing. The files that match the specified pattern will be compiled, but they will not be parsed or instrumented.

-ignore vs. -exclude

The `-ignore` option completely removes the specified file from processing so that it is not parsed by the coverage engine.

The `-include/-exclude` filters are applied after source code is parsed, which allows you to selectively instrument or not instrument header files.

You can use the `-ignore` option to reduce build time overhead by ignoring coverage analysis on some sections of the code (such as external libraries) or to ignore specific file that expose parse errors or other problems during processing.

The following wildcards are supported:

- ? - Any character
- * - Any sequence of characters

To prevent shells from expanding * wildcards to the list of files or directories, you can use the `regex:` prefix to specify the value.

This option can be specified multiple times.

Configuration file format (see -psrc): `cpptestcc.ignore <path|pattern>`

Example:

```
cpptestcc -ignore "*/Lib/*" <remaining part of cmd>
cpptestcc -ignore regex:*/file.c <remaining part of cmd>
cpptestcc -ignore c:/proj/file.c <remaining part of cmd>
cpptestcc -ignore "*/MyLib/*.cpp" -ignore file.cpp <remaining part of cmd>
```

-line-coverage

Enables collecting line coverage.

Runtime coverage results are being written to the results log as the code is executed. This imposes some overhead on the tested code execution time, it but it allows you to ensure that that coverage data is collected even if the application crashes.

Configuration file format (see -psrc): `cpptestcc.lineCoverage [true|false]`

-optimized-line-coverage

Enables collecting optimized line coverage.

Runtime coverage results are stored in memory and then written to the results log either after the application finishes or on user request. This results in better performance, but results may be lost if the application crashes.

Configuration file format (see -psrc): `cpptestcc.optimizedLineCoverage [true|false]`

-function-coverage

Enables collecting function coverage.

Configuration file format (see -psrc): `cpptestcc.functionCoverage [true|false]`

-optimized-function-coverage

Enables collecting optimized function coverage. Configuration file format (see -psrc):

`cpptestcc.optimizedFunctionCoverage [true|false]`

-statement-coverage

Enables collecting statement coverage. Configuration file format (see -psrc):

`cpptestcc.statementCoverage [true|false]`

-optimized-statement-coverage

Enables collecting statement coverage.

Configuration file format (see -psrc): `cpptestcc.optimizedStatementCoverage [true|false]`

-block-coverage

Enables collecting block coverage.

Configuration file format (see -psrc): `cpptestcc.blockCoverage [true|false]`

-optimized-block-coverage

Enables collecting optimized block coverage.

Configuration file format (see -psrc): `cpptestcc.optimizedBlockCoverage [true|false]`

-path-coverage

Enables collecting path coverage.

Configuration file format (see -psrc): `cpptestcc.pathCoverage [true|false]`

-decision-coverage

Enables collecting decision coverage.

Configuration file format (see -psrc): `cpptestcc.decisionCoverage [true|false]`

-optimized-decision-coverage

Enables collecting optimized decision coverage.

Configuration file format (see -psrc): `cpptestcc.optimizedDecisionCoverage [true|false]`

-simple-condition-coverage

Enables collecting simple condition coverage.

Configuration file format (see -psrc): `cpptestcc.simpleConditionCoverage [true|false]`

-optimized-simple-condition-coverage

Enables collecting optimized simple condition coverage.

Configuration file format (see -psrc): `cpptestcc.optimizedSimpleConditionCoverage [true|false]`

-mcdc-coverage

Enables collecting MC/DC coverage.

Configuration file format (see -psrc): `cpptestcc.mcdcCoverage [true|false]`

-call-coverage

Enables collecting call coverage.

Configuration file format (see -psrc): `cpptestcc.callCoverage [true|false]`

-optimized-call-coverage

Enables collecting optimized call coverage.

Configuration file format (see -psrc): `cpptestcc.optimizedCallCoverage [true|false]`

-coverage-early-init

Enables initializing the coverage module at the beginning of the application entry point.

Configuration file format (see -psrc): `cpptestcc.coverageEarlyInit [true|false]`

-coverage-auto-finalization

If enabled, collecting coverage will be automatically finalized at application exit. This option is enabled by default.

Configuration file format (see -psrc): `cpptestcc.coverageAutoFinalization [true|false]`

-optimized-coverage-corruption-detection

Enables corruption detection algorithms for optimized coverage metrics.

Configuration file format (see -psrc): `cpptestcc.optimizedCoverageCorruptionDetection [true|false]`

-template-coverage

Enables collecting coverage for template classes and functions

Configuration file format (see -psrc): `cpptestcc.templateCoverage [true|false]`

-workspace <path>

Specifies a custom directory where information about code structure will be stored during code analysis and instrumentation. The `cpptestcc` tool will use the information to generate the final coverage report.

By default, the information is stored in the working directory of the current compilation. If your compilation uses more than one working directory, we recommend that you specify a custom directory to ensure that all coverage data is stored in the same location.

Configuration file format (see -psrc): `cpptestcc.workspace <path>`

-psrc <file>

Specifies the path to a configuration file where you can configure additional `cpptestcc` options.

By default, `cpptestcc` attempts to read the `.psrc` file located in either the current working directory or in the user HOME directory. This option allows you to specify a custom location of the configuration file.

If an option is configured both in the command line and in the configuration file, `cpptestcc` will use the value specified in the command line.

-version

Prints out information about the version

-help

Prints out the help message and exits.

Coverage Runtime Library

The coverage runtime library is a collection of helper functions and services used by source code instrumentation to emit coverage information at application runtime. Instrumented applications cannot be linked without the library. The runtime library can be linked to the final testable binary in multiple ways depending on tested project type.

In addition to providing basic services for instrumented code, the library is also used to adapt the code coverage solution to particular development environments, such as supporting non-standard transport for coverage results between tested embedded device and development host.

Pre-built Versions and Customized Builds

C/C++test ships with pre-built versions of the runtime library, which are suitable for use on the same platform on which C/C++Test is installed. In most of the cases, collecting code coverage information from natively developed applications can use pre-built versions of the runtime library.

All users developing cross-platform applications will need to prepare a custom build of the coverage runtime library using a suitable cross compiler and possibly linker. Source code of the code coverage runtime library is shipped with C/C++test.

The process of preparing the coverage runtime library custom build is typically limited to the compilation of coverage runtime library source code. In some situations, you may need to install some fragments of source code to adapt code coverage to a particular development platform. This process is described in the following sections.

Using the Pre-built Runtime Library

The following binary files are included with the C/C++test:

Windows (x86 and x86-64)

File	Description
`[INSTALL_DIR]/engine/coverage/runtime/lib/cpptest.a`	32-bit import library to be used with Cygwin GNU GCC compilers. To be added to linking command line.
`[INSTALL_DIR]/engine/coverage/runtime/lib/cpptest64.a`	64-bit import library to be used with Cygwin GNU GCC compilers. To be added to linking command line.
`[INSTALL_DIR]/engine/coverage/runtime/lib/cpptest.lib`	32-bit import library to be used with Microsoft Visual C++ compilers. To be added to linking command line.
`[INSTALL_DIR]/engine/coverage/runtime/lib/cpptest64.lib`	64-bit import library to be used with Microsoft Visual C++ compilers. To be added to linking command line.
`[INSTALL_DIR]/engine/coverage/runtime/bin/cpptest.dll`	32-bit dynamic-link library. `[INSTALL_DIR]/engine/coverage/runtime/bin` should be added to `PATH` environmental variable.
`[INSTALL_DIR]/engine/coverage/runtime/bin/cpptest64.dll`	64-bit dynamic-link library. `[INSTALL_DIR]/engine/coverage/runtime/bin` should be added to `PATH` environmental variable.

Linux (x86 and x86-64)

File	Description
`[INSTALL_DIR]/engine/coverage/runtime/lib/libcpptest.so`	32-bit shared library. To be added linking command line. `[INSTALL_DIR]/engine/coverage/runtime/lib` should be added to `LD_LIBRARY_PATH`
`[INSTALL_DIR]/engine/coverage/runtime/lib/libcpptest64.so`	64 bit shared library. To be added linking command line. `[INSTALL_DIR]/engine/coverage/runtime/lib` should be added to `LD_LIBRARY_PATH`

If you need to use the runtime library in a form not provided as an out-of-the-box solution, prepare a custom build of the coverage runtime library that matches specific development environment requirements. For more details, see Customizing the Runtime Library.

Integrating with the Linker Command Line

Integrating the coverage runtime library with a tested application linking process usually requires modifying the linker command line and, in some cases, the execution environment. This section describes how to modify the linking process when using the pre-built versions shipped with C/C++test.

Static library for Windows Cygwin GNU GCC compilers:

1. Locate the linker command line in your build scripts
2. Modify the build scripts so that the coverage runtime library is specified somewhere in the linker command line - preferably after all object files.

Dynamic-link library for Microsoft Visual C++ compilers:

1. Locate the linker command line in your build scripts
2. Modify the build scripts so that the coverage runtime library is specified somewhere in the linker command line - preferably after all object files. For example:

```
$(LXX) $(PRODUCT_OBJ) $(OFLAG_EXE)$(PROJ_EXECUTABLE) $(LXXFLAGS) $(SYSLIB) $(EXECUTABLE_LIB_LXX_OPTS)
[INSTALL_DIR]/engine/coverage/runtime/lib/cpptest.lib
```

3. Make sure that the `[INSTALL_DIR]/engine/coverage/runtime/bin` directory is added to your `PATH` environment variable so that the library can be located when the tested program is started. You may also consider copying `cpptest.dll` (or `cpptest64.dll`) file to the same directory as your executable file or to another location that is scanned for dynamic-link libraries during tested application startup.

Shared library for Linux GNU GCC compilers:

1. Locate the linker command line in your build scripts
2. Modify the build scripts so that the coverage runtime library is specified somewhere in the linker command line - preferably after all object files. For example:

```
$(LXX) $(PRODUCT_OBJ) $(OFLAG_EXE)$(PROJ_EXECUTABLE) $(LXXFLAGS) $(SYSLIB) $(EXECUTABLE_LIB_LXX_OPTS) -L
[INSTALL_DIR]/engine/coverage/runtime/lib -lcpptest
```

Note the addition of the `-L [INSTALL_DIR]/engine/coverage/runtime/lib` and `-lcpptest` options.

3. Make sure that shared library can be found by tested executable by modifying `LD_LIBRARY_PATH` environmental variable to include `[INSTALL_DIR]/engine/coverage/runtime/lib` location.

Customizing the Runtime Library

You may need to customize the runtime library as a result of the following conditions:

- Different form of binary file is required
- Enabling a non-default communication channel for results transport
- Installing custom implementation of communication channel for results transport
- Enabling a non-default support for multithreaded applications
- Installing custom implementation of support for multithreaded applications

Library Source Code Structure

The runtime library source code is shipped with C/C++test in the `[INSTALL_DIR]/engine/coverage/runtime` directory. The following table describes the structure:

Component	Description
`include`	Directory that contains the library include files. `include/cpptest.h` - library public interface `include/cpptest/*` - library private interface The content of the include directory is not designed for environment specific modifications.
`src`	Directory that contains the library source code. `src/cpptest.c` - the main and single source file of the runtime library This file is designed for modifications and customizations.
`Makefile`	Basic Makefile provided for building the runtime library.
`target`	Directory that contains a set of Makefile include files with compiler specific options for preparing runtime library builds for most popular development environments.
`channel`	Directory that contains a set of Makefile include files with configuration for supported communication channels.

Switching Communication Channel Support

The runtime library supports data collection through various communication channels. The communication channel used depends on the development environment. In most cases, storing results in a file or files is appropriate, but in other TCP/IP sockets or RS232 transport may be required. Specific communication channels can be enabled by setting the value to a dedicated macro during `cpptest.c` library source file compilation. Add `-D<MACRO>` to the compilation command line to set the value. The following table provides the full list of communication channel control macros:

Channel	Description
`CPPTEST_NULL_COMMUNICATION`	Empty implementation. If enabled no results will be sent. Suitable for initial test builds and debugging.

`CPPTEST_FILE_COMMUNICATION`	File-based implementation. ANSI C File I/O interface is used. If enabled, results will be written to a local drive file.
	
	The following additional configuration macros are also provided:
	
	`CPPTEST_LOG_FILE_NAME:` Name of the results file; default `cpptest_results.clog`
	
	`CPPTEST_LOG_FILE_APPEND:` Creates new results file or appends to existing. Default value is `1 -> append`, alternative `0 -> create new`

CPPTEST_SPLIT_FILE_COMMUNICATION	File-based implementation. ANSI C File I/O interface is used. If enabled, results will be written into a series of local drive files. You can configure this channel with the following macros: CPPTEST_LOG_FILE_NAME: Name of the first results file in the series; the default is cpptest_results.clog. Other files will be named in sequence, for example, cpptest_results.clog.0001. Ensure that all the files in the series are placed in the same directory. CPPTEST_MAX_ALLOWED_NUMBER_OF_BYTES_PER_FILE: Specifies the maximum size of one file in the series; default 2000000000 bytes (2 GB).
CPPTEST_UNIX_SOCKET_COMMUNICATION	TCP/IP socket based implementation. POSIX API is used. If enabled results are sent to the specified TCP/IP port. The following additional configuration macros are provided: CPPTEST_LOG_SOCKET_HOST: Specifies host IP address string CPPTEST_LOG_SOCKET_PORT: Specifies the port number CPPTEST_GETHOSTBYNAME_ENABLED: If set to 1, the host can be specified by domain name (requires gethostbyname function to be present)
CPPTEST_WIN_SOCKET_COMMUNICATION	As above, Windows API is used.
CPPTEST_UNIX_SOCKET_UDP_COMMUNICATION	As above, UDP based implementation.
CPPTEST_RS232_UNIX_COMMUNICATION	RS232 based implementation. POSIX API is used. If enabled then results are sent via the specified RS232 system device. The following additional configuration macros are provided: CPPTEST_RS232_DEVICE_NAME: System device name CPPTEST_RS232_BAUD_RATE: Transmission baud rate CPPTEST_RS232_BYTE_SIZE: Byte size CPPTEST_RS232_PARITY: Parity control CPPTEST_RS232_STOP_BIT: Stop bit usage CPPTEST_RS232_TIMEOUT: Transmission timeout value
CPPTEST_RS232_WIN_COMMUNICATION	As above. Windows API is used.
CPPTEST_RS232_STM32F103ZE_COMMUNICATION	STM32F103x USART based implementation. STM Cortex library interface is used (ST/STM32F10x/stm32f10x.h header file is required)
CPPTEST_HEW_SIMIO_COMMUNICATION	Renesas HEW simulator specific implementation.
CPPTEST_LAUTERBACH_FDX_COMMUNICATION	Lauterbach TRACE32 based implementation (FDX used)
CPPTEST_ITM_COMMUNICATION	ARM CoreSight ITM unit based communication. Requires CMSIS header files.
CPPTEST_CUSTOM_COMMUNICATION	Enables empty template for custom implementation

If the coverage runtime library is being built with the provided Makefile, then one of the make configuration files provided in the [INSTALL_DIR]/engine/coverage/runtime/channel directory can be used.

Installing Support for Custom Communication Channel

If none of the communication channel implementations fit into your development environment, then a custom implementation can be provided. The following instructions describe how to customize the runtime library so that it uses a custom implementation of a communication channel:

1. Make a copy of [INSTALL_DIR]/engine/coverage/runtime/src/cpptest.c and open the file for editing.
2. Locate the section 1.13 "Custom Communication Implementation.
   The custom communication implementation section contains empty templates for four different methods:

Function	Description

`void cpptestInitializeStream (void)`	This function is responsible initializing the communication channel, for example creating and connecting to a socket or the initialization of UART device.
`void cpptestFinalizeStream (void)`	This function is responsible for finalizing the communication channel. For example, it may be responsible for closing TCP/IP socket.
`int cpptestSendData(const char *data, unsigned size)`	This function is responsible for sending size bytes from a data buffer.
`void cpptestFlushData(void)`	This function is responsible for flushing the data. Its meaning depends on the particular transport type. It may have a limited application in some implementations. In this case, it should be left empty.

3. Provide the implementation for these methods that match your environment requirements.
4. Compile `cpptest.c` with the following macro definition added to compilation command line:
   `-DCPPTEST_CUSTOM_COMMUNICATION`
5. If the generated object file is insufficient, you can process the file even further to meet your needs (e.g., to create a shared library).

Switching Multithreading API Support

The runtime library contains support for multithreaded applications. POSIX, Windows, and VxWorks APIs are supported. You can enable support for a specific multithreading API by adding `-D<MACRO>` to the compilation command line during `cpptest.c` compilation. The following table describes the full list of multithreading API support control macros:

Macro	Description
`CPPTEST_NO_THREADS`	Empty implementation. Coverage runtime is not prepared to be used together with multithreaded applications
`CPPTEST_WINDOWS_THREADS`	Windows multithreading API implementation
`CPPTEST_UNIX_THREADS`	POSIX multithreading API implementation
`CPPTEST_VXWORKS_THREADS`	VxWorks multithreading API implementation

Installing Support for Custom Threading API

If you are using C/C++test's coverage engine with multithreaded applications that do not use a supported multithreading API, you can customize the runtime library to work with your multithreading API. There are following steps required:

1. Make a copy of `[INSTALL_DIR]/engine/coverage/runtime/src/cpptest.c` and open the file for editing
2. Locate the section 2.5 "Custom Multithreading Implementation"
   Custom multithreading implementation section contains empty templates for two different methods:

Function	Description
`static int cpptestLock (void)`	This function ensures synchronized operations inside the coverage tool runtime library. If a thread locks access to the runtime library service, it means an atomic operation is in progress and no other thread can use runtime library services. Once the lock is released other threads can use runtime library services
`static int cpptestUnlock(void)`	Releases the lock on runtime library services.

3. Provide the implementation for the methods that matches your environment requirements.
4. Compile `cpptest.c` with the following macro added to compilation command line:
   `-DCPPTEST_CUSTOM_THREADS`
5. If the generated object file is insufficient, you can process the file even further to meet your needs (e.g., to create a shared library).

Building the Runtime Library

C/C++test ships with a simple Makefile (see Library Source Code Structure) which simplifies the process of building the runtime library. In many instances, however, the make file provided will not be required because the source code is already optimized for the building process. The only step that is always required is the compilation of the main `cpptest.c` source file. Any additional processing of the produced object file will depend on the particular development environment and its requirements, such as providing the runtime library as a shared library.

Building the Runtime Library Using the Provided Makefile

1. Copy `[INSTALL_DIR]/engine/coverage/runtime` to a local directory.
2. If compilation flags need to be modified (e.g., adding specific cross-compiler specific or definitions to enforce runtime library reconfiguration), provide a new make configuration file in the target subdirectory. For convenience, copy one of the existing target configuration files and modify its contents to fit your needs.
3. Invoke the following command line to create a `build` subdirectory that contains a single object `cpptest.<OBJ_EXT>`, which can be used to link with the instrumented application.

```
make TARGET_CFG=<target config file name> CHANNEL_FILE=<channel config file name>
```

Your command line may resemble the following:

```
make TARGET_CFG=gcc-static.mk CHANNEL_FILE=channel/unix-socket.mk
```

Alternatively, you can provide the channel type:

```
make TARGET_CFG=gcc-static.mk CHANNEL_TYPE=unix-socket
```

4. If the coverage runtime library needs to be linked to from a shared library, dynamic link library, or any other type of binary, the Makefile needs to be customized for this purpose or a custom build needs to be setup.

User Build of the Runtime Library

To setup a user build of coverage runtime library perform the following steps:

1. Copy the cpptest.c file from `[INSTALL_DIR]/engine/coverage/runtime/src/cpptest.c` to your preferred location.
2. Introduce any customizations as described in Customizing the Runtime Library.
3. Set up a build system of your preference (e.g., IAR Embedded Workbench project or any other type of source code builder).
4. Modify the compilation flags to contain the compiler include a flag (typically `-I`) with the following value:
   `-I[INSTALL_DIR]/engine/coverage/runtime/include`
5. Add any required configuration defines (typically `-D`), for example:
   `-DCPPTEST_FILE_COMMUNICATION -DCPPTEST_NO_THREADS`
6. Invoke the command to run your builder (for example, select build command in the IDE).
7. Locate the resulting object file and use it to link with your instrumented application.