Working with the C++test Runtime Library

This topic explains how to build and use the C++test runtime library.

Runtime libraries must be built with a cross compiler in order to use C++test for testing on a platform other than the host (the environment where C++test is installed). If you plan on running testing only on the host, you do not need to build the C++test runtime library (because C++test is shipped with prebuilt libraries for host-based testing).

C++test can build the runtime library automatically as a part of the test-executable preparation phase. Manual preparation of the C++test runtime library is not required unless you want to prepare a custom build of the C++test runtime library.

Sections include:

Helper Keywords

Throughout this topic, helper keywords are used to represent complete paths. These helper keywords are defined as follows:
<C++test_runtime_root> = <C++test Install Dir>/engine

<C++test_runtime_sources> = <C++test Install Dir>/engine/runtime/src

<C++test_includes> = <C++test Install Dir>/engine/runtime/include

Introduction

The C++test runtime library is distributed with the C++test installation in the form of C source and header files. The source for the runtime library is located in the directories referenced in the above box

In most cases, you can let C++test automatically build the runtime library. If desired, you can build the runtime library using the make utility and one of the pre-configured make configurations for the supported platforms (see Building the C++test Runtime Library with 'make'). C++test provides special project files that facilitate runtime library building for popular IDEs.

Please note that a discrete version of a runtime library needs to be built for every target. Thus, if your project is built for more than one architecture, then most likely you'll need to build more than one version of the C++test runtime library (one for each architecture).

The runtime library sources include configuration macros for all officially-supported embedded platforms, as well as lower-level function-based configurations. These macros need to be correctly set for a build configuration in the IDE.

The following steps describe that general approach:

1. Build the runtime library using the provided makefiles or one of the preconfigured IDE projects located in `<C++test_runtime_root>/runtime/projects`.
2. Add the resulting `.a` file to **Project Properties> C++test> Build Options> Linker Option**
    - Use an absolute path.
    - The best option is to have the library file in some fixed location so that multiple users can point to it without changes.
    - Using a regular environment variable for the runtime library is also supported.

General Information about the Runtime Library

C++test's instrumentation references many symbols that must be resolved during the linking of the test object. Those symbols are provided in the C++test runtime library.

C++test ships with a prebuilt, full-featured runtime shared library. However, this runtime is designed for rich host platforms. Considering the multitude of embedded platforms on which testing may occur, as well as their capabilities and limitations, the runtime typically needs to be adjusted before being built for different embedded environments. That's why the runtime library's pure-C sources are also available.

The library can be configured to support features that are available on the target platform, or to block those that are not. The default configuration is full-featured, but you might need to block troublesome features—especially if you experience compile-time or runtime crashes. Embedded developers are familiar with the pros and cons of the platform(s) they use, and should be able to build an appropriately-configured library and append its path to the linker command line.

The runtime library's sources are available at <C++test_runtime_sources> (`<C++testInstallDir>/<C++test Install Dir>/engine/runtime/src`). The header files needed to build the runtime library are located in <C++test_includes> (`<C++test Install Dir>/engine/runtime/include`).

For example, in the Windows C++test 7.3 distribution, the runtime source code would be located in:

```
C:\Program Files\ParaSoft\C++test7.3\engine\runtime\src
```

The corresponding include files would be located in:

```
C:\Program Files\ParaSoft\C++test7.3\engine\runtime\include
```

Modifying Sources - Best Practice

Never modify the files in <C++test Install Dir>. Instead, copy the sources into a different directory before modifying them. Then, specify this new path in the Runtime library source option in the C++test Project Options Panel (see Specifying Options in the C++test Project or File Options Panel for details). The Automated Runtime Library Building feature will use this directory to build the library as described in Building the C++test Runtime Library.

Understanding the Runtime Distribution Structure and Contents

The C++test runtime distribution consists of the following subdirectories or files placed in the <C++test_runtime_sources> directory:

- src/: Contains C source files.
- listeners/: Contains a socket and serial listener as a simple utility that can be used with the given communication channel.
- target/: Contains a set of target configu
- test/: Contains one quick test.
- Makefile: Makefile for building the runtime library.
- *.vcproj: Microsoft Visual Studio project files for building the runtime library.

The C source code is divided into parts, represented by the following subdirectories or files:

- src/common/: Common utility functions for string manipulations, etc.
- src/transport/: Communication channel implementations.
- src/presentation/CppTestMessagePlain.c: Plain text layer for results reporting.
- src/: Test case execution facilities.
- src/CppTestCoverage.c: Utility functions for handling coverage reporting.

In the target directory, there are multiple target configurations available for various platforms, architectures, and compilers. The target configuration defines variables used by the Makefile to compile the runtime source files (for example, CC, CFLAGS, etc.) and variables such as LIBTOOL, LIBTOOL_FLAGS, etc. that the Makefile uses to create a runtime library from compiled sources (see Building the C++test Runtime Library with 'make').

Building the C++test Runtime Library

Overview

C++test can build the testing runtime automatically as a part of the test-executable preparation phase. This approach is recommended. In most cases, manual preparation of the C++test runtime library is not required.

In some cases, you may want to prepare a custom build of the C++test runtime library. You can build the C++test runtime library using the Makefileor one of the project .vcproj filesin the <C++test_runtime_sources> directory. You can also use one of the provided preconfigured IDE projects to build the runtime library; these are available in <C++test_runtime_root>/runtime/projects. Optionally, you can create an IDE project on your own

Basically, building the runtime library involves selecting the communication channel, then running the IDE builder or 'make' in the directory containing the Makefile. You may also need to change the compiler and platform-specific configuration if the default one causes problems or if there is no default configuration for your compiler and target platform (see Configuring the C++test Runtime Library).

Automated Building

The following test execution step should be used for automated building of the C++test runtime:

```
<BuildRuntimeLibStep additionalFlags="<build options>" />
```

This execution step will compile all source files found in the C++test Runtime library source location using the environment (compiler executable, build options) of the tested project. The compiled objects will then be linked into a test executable. The `additionalFlags` attribute is optional and can be used to specify additional build options.

Important Note - Removing the Runtime Library from Build Settings

Before C++test builds the unit testing runtime automatically (as a part of the test execution flow), ensure that the C++test runtime library is removed from the **Build Settings> Linker** options.

Preparing a Custom Build

Selecting the Communication Channel

The first step in manually building the runtime library is to choose one of the communication channel implementations that will be used by the runtime. The following communication channel implementations are in the C++test runtime distribution:

- File communication:
  - src/transport/CppTestTransportFille.c
  - src/transport/CppTestTransportFileSplit.c
  - src/transport/CppTestTransportFileBuffered.c
- TCP/IP sockets:
  - src/transport/CppTestTransportUnixSocket.c: Communication channel based on Unix sockets
  - src/transport/CppTestTransportWinSocket.c: Communication channel based on Windows sockets
- Serial link (rs232):
  - src/transport/CppTestTransportRS232Common.c
  - src/transport/CppTestTransportRS232Common.h
  - src/transport/CppTestTransportRS232Unix.c: Communication channel based on Unix serial port API
  - src/transport/CppTestTransportRS232Win32.c: Communication channel based on Windows serial port API
  - src/transport/CppTestTransportRS232STM32F103ZE.c: Communication channel for STM32F103 MCU UART

You can select a communication channel implementations by adding the appropriate CHANNEL_TYPE parameter to the 'make' command line:

- CHANNEL_TYPE=file: Selects file-based communication (default)
- CHANNEL_TYPE=file-buffered: Selects file-based communication with buffered write operations
- CHANNEL_TYPE=file-split: Selects file-based communication with data split into a series of files with a configurable maximum size.
- CHANNEL_TYPE=unix-socket: Selects Unix socket-based communication
- CHANNEL_TYPE=win-socket: Selects Windows socket-based communication
- CHANNEL_TYPE=win-rs232: Selects Windows serial communication
- CHANNEL_TYPE=unix-rs232: Selects Unix serial communication
- CHANNEL_TYPE=win-stm32f103ze-rs232: Selects STM32F103 MCU uart communication

Or, if you use your IDE instead of a make tool, you can add one of the following defines to the compiler options

- CPPTEST_USE_FILE_COMMUNICATIONS
- CPPTEST_USE_FILE_SPLIT_COMMUNICATIONS
- CPPTEST_USE_FILE_BUFFERED_COMMUNICATION
- CPPTEST_USE_UNIX_SOCKET_COMMUNICATION
- CPPTEST_USE_WIN_SOCKET_COMMUNICATION
- CPPTEST_USE_RS232_WIN_COMMUNICATION
- CPPTEST_USE_RS232_UNIX_COMMUNICATION
- CPPTEST_USE_RS232_STM32F103ZE_COMMUNICATION
- CPPTEST_USE_CUSTOM_COMMUNICATION

Building the C++test Runtime Library with 'make'

Calling the 'make' command in the directory containing the Makefile will compile the runtime sources with the gcc compiler and will create the C++test runtime library using the ar library tool. To change the compiler and library tool, you need to specify the target configuration at the 'make' command line. For example:

```
make TARGET_CFG=WR_DKM_gcc3_4_simlinux_VxWorks6_4.mk
```

The following configurations are available in the C++test runtime distribution (in <C++test_runtime_sources>/target/ directory):

- ARM_armcc1_2_ADS1_2.mk
- ARM_armcc3_0_RVDS3_0.mk
- ARM_armcc3_1_RVDS3_1.mk
- ARM_armcc3_1_uV3.mk
- ARM_armcc4_0_RVDS4_0.mk
- ARM_armcc5_0.mk
- ICCARM5_3.mk
- MSMobile.mk
- QNX_2_9.mk
- QNX_3_3.mk
- QNX_4_2.mk
- QNX_4_4.mk
- ST20_os20.mk
- ST40_os21.mk
- TIARM_4.9.mk
- TIARM_5.1.mk
- TIARM_5.1_eabi.mk
- TIARM_5.1_gcc.mk
- TIC2000_4.1.mk
- TIC2000_5.2.mk
- TIC6000_5.1.mk
- TIC6000_6.0.mk
- WR_DKM_dcc5_5_simlinux_VxWorks6_4.mk
- WR_DKM_dcc5_5_simnt_VxWorks6_4.mk

- WR_DKM_dcc5_5_simsparc_VxWorks6_4.mk
- WR_DKM_dcc5_6_simlinux_VxWorks6_6.mk
- WR_DKM_dcc5_6_simnt_VxWorks6_6.mk
- WR_DKM_dcc5_6_simsparc_VxWorks6_6.mk
- WR_DKM_gcc3_4_simlinux_VxWorks6_4.mk
- WR_DKM_gcc3_4_simnt_VxWorks6_4.mk
- WR_DKM_gcc3_4_simsparc_VxWorks6_4.mk
- WR_DKM_gcc4_1_simlinux_VxWorks6_6.mk
- WR_DKM_gcc4_1_simnt_VxWorks6_6.mk
- WR_DKM_gcc4_1_simsparc_VxWorks6_6.mk
- WR_RTP_dcc5_5_simpentium_VxWorks6_4.mk
- WR_RTP_dcc5_5_simsparc_VxWorks6_4.mk
- WR_RTP_dcc5_6_simpentium_VxWorks6_6.mk
- WR_RTP_gcc3_4_simpentium_VxWorks6_4.mk
- WR_RTP_gcc3_4_simsparc_VxWorks6_4.mk
- WR_RTP_gcc4_1_simpentium_VxWorks6_6.mk
- eVC4_0_arm.mk
- eVC4_0_mips.mk
- eVC4_0_sh.mk
- eVC4_0_x86.mk
- gcc-shared.mk
- gcc-static.mk
- msvc-shared.mk
- msvc-static.mk

The target configuration provides definitions for the following variables that are used when building the C++test runtime library:

Name	Default	Description
CC	gcc	Compiler executable used to compile C++test runtime sources.
CFLAGS		Flags used during compilation.
CC_OUT_DIR_FLAG		Flag used with a directory for storing compilation products: $(CC_OUT_DIR_FLAG)"<output-dir>"
CC_OUT_FLAG	-o	Flag used with output file name. The $(CC_OUT_FLAG)"<output-file-name>" is added to the compiler command line.
OBJ_EXT	o	Object file extension used when creating an output file name
LIBTOOL	ar	Executable that is able to create a library from compiled sources.
LIBTOOL_FLAGS	-ruv	Flags used with lib tool when creating a C++test runtime library
LIBTOOL_OUT_FLAG		Flag used with output file name. The $(LIBTOOL_OUT_FLAG)"<output-file-name>" is added to the lib tool command line.
LIB_PREFIX	lib	These variables are used when creating a C++test runtime library name: (LIB_PREFIX)(LIB_NAME).$(LIB_EXT)
LIB_NAME	cpptestruntime	
LIB_EXT	a	

The LD, LDFLAGS, LD_OUT_FLAG and OUT_EXT are also defined by available configurations, but they are not required to build the C++test runtime library. They are only used to build a simply test executable when the 'test' target from make file is called.

To create your own target configuration, make a copy of an existing one that best describes your target, then modify it as needed. To build the C++test runtime library with your new target configuration, add the TARGET_DIR parameter to the 'make' command line

```
    make TARGET_DIR=<directory with my target configuration>
TARGET_CFG=MyTargetConfig.mk
```

You can also change the directory in which the runtime library is created (by default, it is set to ./build directory). To change this, add the OUT_DIR parameter to the 'make' command line. For example:

```
    make OUT_DIR=<output directory for my C++test runtime library> ...
```

You can perform the build in the location where C++test is installed, or you can copy the C++test runtime source files and build them in a different location. In such cases, remember to adjust the path to C++test includes (which are stored in <C++test_includes>). To do this, modify the value of the CPPTEST_INC_DIR variable inside the make file to an absolute path. For example:

```
CPPTEST_INC_DIR=/usr/local/C++test/engine/runtime/include
```

Multi-Thread Support

To build a custom Runtime Library with the multi-thread support, add "-DCPPTEST_THREADS_ENABLED=1" to the compiler command line.

Configuring the C++test Runtime Library

Because different target platforms have different limitations, the C++test runtime library is designed to be easily configured to suit different environments. The C++test runtime library can be configured—mainly with the help of the available macro definitions. If your target has some non-standard limitations, it may also be necessary to make some additional changes to the runtime source code.

> **Note**
>
> Include files used for building the runtime library (located in <C++test_includes>) are also included by the generated test harness (test source files automatically generated by C++test). If you need to modify these files, be extremely careful. Even a small mistake could cause the test harness compilation to fail. It is typically safe to introduce target-specific changes using the preprocessor conditional instructions (#ifdef...#endif).

The available compiler configurations are placed in the `<C++test_includes>/config/` directory. There is also a `cpptest_portinfo.h` header file in the C++test include directory that determines which compiler configuration should be used when building the C++test runtime library. A compiler configuration header file defines a set of macros that adapt the runtime to the target platform. It also may define the macros that disable some C++test features that cannot be supported on target platform. The full list of features that can be disabled can be found in the `cpptest_features.h` header file.

In most cases, if you need to change the value of some configuration macros, you don't need to modify or create a new configuration file. You can add to the CFLAGS variable in your target configuration file (or to the compiler options, if you use the IDE for building the runtime) the proper -D options that set the value of the given macros to the required values.

The available common configuration defines are listed in the following table (note that the specified defaults depend on the specific compiler and target environment):

Name	Details
CPPTEST_EXPORT, CPPTEST_IMPORT	When building DLLs for Windows platform, these are expanded to __declspec(dllexport), __declspec(dllimport), Microsoft-specific extensions for importing/exporting functions and data to and from a DLL.
CDECL_CALL	When building with a Microsoft compiler, this is expanded to __cdecl Microsoft-specific extensions for calling convention.
CPPTEST_WCHAR_ENABLED	If defined as 1, then the runtime library will be built with support for wchar types. When defined as 0, the C++test runtime library will be built without support for wchar types.
CPPTEST_SETJMP_ENABLED	If defined as 1, signal handling routines are used to recover if an exception is thrown during test case execution. In this case, C++test logs the exception and tries to continue test cases execution.
CPPTEST_USE_ANSI_SETJMP	If defined as 1, then the following functions are used for signals handling: longjmp , setjmp. If 0, then siglongjmp and sigsetjmp are used. And analogically with data types: sigjmp_buf versus jmp_buf. Note that you need to have CPPTEST_SETJMP_ENABLED set to 1 for CPPTEST_USE_ANSI_SETJMP to take effect.
CPPTEST_USE_UNCAUGHT_EXCEPTION_CHECKING	If defined as 1, then the uncaught_exception function is used for uncaught exception handling.
CPPTEST_USE_STD_NS	Define to 0 when C++test should not use the std namespace or when your compiler doesn't support namespaces.
CPPTEST_TIME_MODE	This macro controls which functions are used for time measuring. The list of allowed values is available in the cpptest_time.h header file.

The available feature configuration defines are listed in the following table:

Name	Details
CPPTEST_C_STREAMS_REDIRECT_ENABLED	If defined to 1, the c streams (stdin, stdout, stderr) redirection is enabled.
CPPTEST_CPP_STREAMS_REDIRECT_ENABLED	If defined to 1, the cpp streams redirection is enabled
CPPTEST_USE_WSTREAMS	If defined to 1, the wcin, wcout, wcerr, wclog are used instead cin, cout, cerr and clog. Note that you need to have CPPTEST_CPP_STREAMS_REDIRECT_ENABL ED set to 1 for CPPTEST_USE_WSTREAMS to take effect.
CPPTEST_EXCEPTIONS_ENABLED	If defined to 1, the exception handling is enabled.
CPPTEST_SPECIAL_STD_EXCEPTIONS_HANDLING_ENABLED	If defined to 1, the std::exception is handled separately from other exceptions. Note that you need to have CPPTEST_EXCEPTIONS_ENABLED set to 1 for CPPTEST_SPECIAL_STD_EXCEPTIONS_HANDLING_ENABLED to take effect.
CPPTEST_DATA_SOURCES_ENABLED	If defined to 1, the data sources are enabled.

CPPTEST_CSV_DATA_SOURCE_ENABLED	If defined to 1, the csv data sources are enabled.
CPPTEST_USE_SETUP_FOR_C_SUITES	If defined to 1, the setUp and tearDown functions are allowed for c suites.
CPPTEST_THREADS_ENABLED	If defined to 1, the support for threads is enabled.

Communication Modes

This section covers:

File Communication Channel

File Communication Channel Implementation

The file communication channel utilizes standard file I/O operations to save test results. There are two data streams: one for testing results and one for coverage results. The implementation contains three variants:

- src/transport/CppTestTransportFile.c: With plain file I/O writes.
- src/transport/CppTestTransportFileSplit.c: With plain file I/O writes, splitting data into series of files with configurable maximal size
- src/transport/CppTestTransportFileBuffered.c - With buffered write operations.

In the first mode (CppTestTransportFile.c), each data packet produced during testing is immediately saved to a file. In this mode, it is up to the operating system to decide when the actual physical write will be performed.

The second mode (CppTestTransportFileSplit.c) is similar to the first mode in that it also uses plain file I/O writes. But in CppTestTransportFileSplit.c mode, data packets are saved to a series of files. The maximum size for each file is configured by setting a value using the CPPTEST_MAX_ALLOWED_NUMBER_OF_BYTES_PER_FILE macro. The default is size is 2000000000 bytes (2 GB). The files produced are named sequentially, for example: 'cpptest_results.clog', 'cpptest_results.clog.0001', 'cpptest_results.clog.0002'. See the descriptions of the ReadDynamicCoverage Step and ReadTestLogStep Test Execution Flow Steps for information about loading the generated result files into C++test.

In the third mode (CppTestTransportFileBuffered.c), the intermediate memory buffer is introduced to reduce the frequency of writing small data packets. This mode should be used for environments in which performing multiple writes of small data chunks is more expensive than writing a combined, larger portion of data. You can determine the size of the intermediate buffer by using the FILE_COMMUNICATION_BUFFER_SIZE define. The larger your buffer size, the less frequently write operations will occur.

Socket Communication Channel

Socket Communication Channel Implementation

The socket communication channel utilizes TCP/IP sockets to save test results. There are two sockets opened: one for sending test results and one for sending coverage results. There are two available implementations are:

- src/transport/CppTestTransportUnixSocket.c: For Unix/Linux-based systems.
- src/transport/CppTestTransportWin32Socket.c: For Windows-based system.

The Socket Listener

Data emitted from the target via the TCP/IP sockets should be captured with the help of a socket listener. The listener is a simple utility program shipped with C++test. It is located in <C++test install dir>/ engine/runtime/listeners/socket_listener, and it accepts the following parameters:

Name	Details
`-cl, --channel port_number@log_file`	Communication channel specification. Information about the port number to listen for results and the file to save data captured on this port.
`-sf, --sync_file filename`	The path to the file that will be used for synchronization. The listener will use this path to create the two "synchronization files": - <sync_file>.init when the listener initialization is finished. - <sync_file>.final when the connection from the target is closed.
`-to, --timeout timeout (in seconds)`	Timeout to wait for results. If this is exceeded, the program will stop.
`--help`	Displays the help information.

RS232 Communication Channel

Serial Communication Channel Implementation

The serial communication module is designed to handle testing results transport for devices where only serial communication is available. The implementation of this communication consists of two parts:

- The common layer: Responsible for data packaging, formulating frames, and verifying transfer correctness. The implementation is provided in the CppTestTransportRS232Common.c and CppTestTransportRS232Common.h files.
- The target-specific part: Responsible for interacting with the physical COM port. Contains implementations of setup and sending functions. C++test provides sample implementations for this layer for Windows-based targets and Unix-based targets; it also provides a simple bare metal implementation for STM32103ZE Cortex-M3 chip (Keil uVision3).

Raw and Safe Communication Modes

There are two communication modes available:

- Raw mode: In this mode, data is sent to a host machine without any verification. This is recommended for stable environments.
- Safe: In this mode, checksum is computed for every packet. If the checksum is incorrect, the transfer is stopped. This mode is activated by adding the compilation flag - DCPPTEST_RS232_SAFE_MODE=1.

Data sent via serial link is partitioned into packages. The packages are represented as shown below. Here is a frame example for "raw" mode:

```
     0      |     1    |  2  |                |  data size + 3  |  data size+4
STX (1 byte) |CHANNEL ID|Count|Data (data size) |       0         | ETX (1
byte).
--------------------------------- MAX 255 ----------------------------------
-----
```

Here is a frame example for "safe" mode:

```
      0      |     1    |  2  |                 |  data size + 3     | data
size+4
STX (1 byte) |CHANNEL ID|Count|Data (data size) |  Checksum (1 byte) | ETX
(1 byte).
--------------------------------- MAX 255 ----------------------------------
-------------
```

Here is a description of the symbols used:

Symbol	Description
STX (0x02)	Start of text
ETX (0x03)	End of text
EOT (0x04)	End of transmission
ENQ (0x05)	Enquiry
ACK (0x06)	Affirmative acknowledgement
NAK (0x15)	Negative acknowledgment

In "safe" mode, checksum is compute with exclusive-or. For "raw" mode, checksum is not computed and zero value is sent for all frames.

The default maximum amount of data is set to 250 bytes. This value can be changed using the MAX_PACKET_SIZE macro.

The Transmission Flow

In "safe mode", the client sends the ENQ to enquire about availability. It cannot start the transmission until it receives the ACK. After the client receives the ACK, it sends another frame. Every frame is confirmed with an ACK by the host machine. At the end, the client sends the EOT. In response, the host sends an ACK to complete the transmission. Checksum is calculated only in safe mode. In the normal mode, it is always 0.

In "raw" mode the transmission is one way only. The target platform emits data without waiting for an ACK at any data sending stage.

Adding a New Platform-Specific Implementation of Serial Communication

The following steps need to be accomplished in order to add another implementation of serial communication:

1. Add a new file in the <C++test Install dir>/engine/runtime/src/transport directory with the implementation of the custom serial communication. Follow this template:

```
#define CPPTEST_RUNTIME_IMPLEMENTATION
#include "cpptest_portinfo.h"
#ifdef CPPTEST_USE_RS232_MYPLATFORM_COMMUNICATION
#include "CppTestTransportRS232Common.h"
int localRsInitInternalSerial(CppTestStreamParameters *par)
{
  return RS_OK;
}
int localRsUninitInternalSerial(CppTestStreamParameters *par)
{
  return RS_OK;
}
int localRsSendInternalByte(CppTestStreamParameters *par, unsigned char
byte, unsigned char *is_send)
{
  return RS_OK;
}
int localRsSendInternalStr(CppTestStreamParameters *par, unsigned char
*bytes, int *nBytes)
{
  return RS_OK;
}
int localRsRecvInternalByte(CppTestStreamParameters *par, unsigned char
*pByte, unsigned char *is_recv)
{
  return RS_OK;
}
int localFlushInternalRS(CppTestStreamParameters *par)
{
  return RS_OK;
}
void localSetStop(int *stop_bit, int result)
{
}
void localSetParity(int *parity, char *result)
{
}
void localRsSleep(unsigned int msec)
{
}
#endif
```

2. Provide the implementation for the following functions:
   - int localFlushInternalRS(CppTestStreamParameters *par);
     Flush internal Serial port buffer. CppTestStreamParameters described below.
   - int localRsInitInternalSerial(CppTestStreamParameters *par);
     Init serial port.
   - localRsSendInternalByte(CppTestStreamParameters *par, unsigned char byte, unsigned char *is_send);
     Send one byte. Parameters are par (described below), byte to send and result.
   - int localRsSendInternalStr(CppTestStreamParameters *par, unsigned char bytes, int *nBytes);
     Send nBytes bytes. Parameters are par (described below), bytes to send and nBytes sent.
   - int localRsRecvInternalByte(CppTestStreamParameters *par, unsigned char *pByte, unsigned char *is_recv);
     Receiving one byte. Parameters are par (described below), received byte result.
   - int localRsUninitInternalSerial(CppTestStreamParameters *par);
     Close serial port.
   - void localSetStop(int *stop_bit, int result);
     Set stop bit transmission parameter.

- void localSetParity(int *parity, char *result);
  Set parity transmission parameter.
- char *localGetErrorText(void);
  Return buffer with ErrorText (if any).
- void localRsSleep(unsigned int msec);
  Sleep implementation.

```
Struct CppTestStreamParameters:

typedef struct {
  char serial_device[16]; Device name (COM1, /dev/ttyS0, cua0 etc). Depend
on platform
  int baud_rate; Device speed
  int byte_size; Byte size
  int parity; Parity
  int stop_bit; Stop bit
  int timeout; Timeout between two operations.
  long max_timeout; If exceeded tranmision broken.
} CppTestStreamParameters;
```

Integrating a New Implementation with the Runtime Library

To integrate a new implementation with the runtime library:

1. In the `channel` subdirectory, insert a new file named `myplatform-rs232.mk`. To this new file, add the line:
   `"CHANNEL_CONFIG_MACRO=-DCPPTEST_USE_RS232_MYPLATFORM_COMMUNICATION"`
2. In the `include/cpptest_portinfo.h` file, replace:

   ```
   #if !defined(CPPTEST_USE_FILE_COMMUNICATION) && \
       !defined(CPPTEST_USE_FILE_BUFFERED_COMMUNICATION) && \
       ...
       !defined(CPPTEST_USE_RS232_STM32F103ZE_COMMUNICATION) && \
       ...
       !defined(USE_CUSTOM_COMMUNICATION_CHANNEL)
   #  define CPPTEST_USE_FILE_COMMUNICATION
   #endif
   ```

   with

   ```
   #if !defined(CPPTEST_USE_FILE_COMMUNICATION) && \
       !defined(CPPTEST_USE_FILE_BUFFERED_COMMUNICATION) && \
       ...
       !defined(CPPTEST_USE_RS232_STM32F103ZE_COMMUNICATION) && \
       !defined(CPPTEST_USE_RS232_MYPLATFORM_COMMUNICATION) && \
       ...
       !defined(USE_CUSTOM_COMMUNICATION_CHANNEL)
   # define CPPTEST_USE_FILE_COMMUNICATION
   #endif
   ```

3. Build the library with the parameter `CHANNEL_TYPE=myplatform-rs232.`
4. If you use the Runtime Library Auto Build step, do one of the following:
   - Add `-DCPPTEST_USE_RS232_MYPLATFORM_COMMUNICATION` to the compiler flags in the C++test project.
   - Modify the Runtime Auto Build step's test execution flow to include the flag
     `<BuildRuntimeLibStep`
     `additionalFlags="-DCPPTEST_USE_RS232_MYPLATFORM_COMMUNICATION" />`

The RS232 Listener

Data emitted from the target via the serial link should be captured with the help of a serial port listener. The listener is a simple utility program shipped with C++test. It is located in <C++test install dir>/engine/runtime/listeners/rs232_listener, and it accepts the following parameters:

Parameter	Description
-d (--device) COM,SPEED,PARIT Y,STOP, BYTE_SIZE.	Used device. e.g. `-d 1,19200,N,1,8`. We use the COM1 port, at a rate of 19200, no parity, one stop bit and 8 byte size.
-sf (--syncfile) <sync file>	Filename to synchronize

-cn (--channel) number<mode>@<file path>	Communication channel. C++test uses two channel -test log and coverage log. The mode can be 'b' for binary or 't' for text (default)
-to (--timeout) <timeout in second>	Time to wait for results. If this is exceeded, the program stops.
-fi <file_name>	File mode. Reads results from binary file. All RS232 settings will be ignored.
-rm (--rawmode)	Raw mode (no ACK and checksum). This option is required if transport was used in non safe mode.
-v (--verbose)	Verbose mode.

Depending on the run mode (safe or raw), the listener may or may not compute the transmission check sum and send the ACK.

Initialization and Finalization

All data passed to the runtime library during test or application execution are processed, interpreted, and passed to the communication stream only if the runtime library is correctly initialized. If the runtime library is not initialized or has been finalized, then data such as coverage, unit test messages, asserts and runtime violations will be ignored.

The runtime library is initialized during the first call of the CppTest_InitializeRuntime() function, which initializes runtime library modules and opens a communication channel for messages and coverage data. The CppTest_InitializeRuntime() function is called as soon as possible by default. In projects that contain C++ code, this call is made before constructors for global objects are called. In C projects, this function is called at the beginning of the main() function. The runtime library is finalized during the first call to the CppTest_FinalizeRuntime() function, which finalizes runtime library modules and closes the communication channel. This function is called automatically after global objects destructors are called. In C projects, it is called before exiting from the main() function.

There are two special functions that you can implement to contribute custom initialization and finalization code to the test harness execution:

void CppTest_Initialize(void)

This function is called at the *beginning* of the CppTest_InitializeRuntime function, even before C++test runtime library modules are initialized.

void CppTest_Finalize(void)

This function is called at the *end* of CppTest_FinalizeRuntime after the test results communication channel is closed.

These functions are typically used if your project requires early initialization code to be executed, such as hardware-specific initialization code (e.g. disabling the watchdog timer) or any other type of initialization that is expected to happen very early in execution sequence. Analogously for finalization code

Usage of CppTest_Initialize/Finalize in Unit Testing and Application Monitoring

When running unit tests, initialization code existing in your original project will be omitted from execution because original application entry point (typically the main function) is replaced by C++test with automatically generated test harness entry point. This is why the initialization/finalization code may need to be hooked via the CppTest_Initialize/Finalize function.

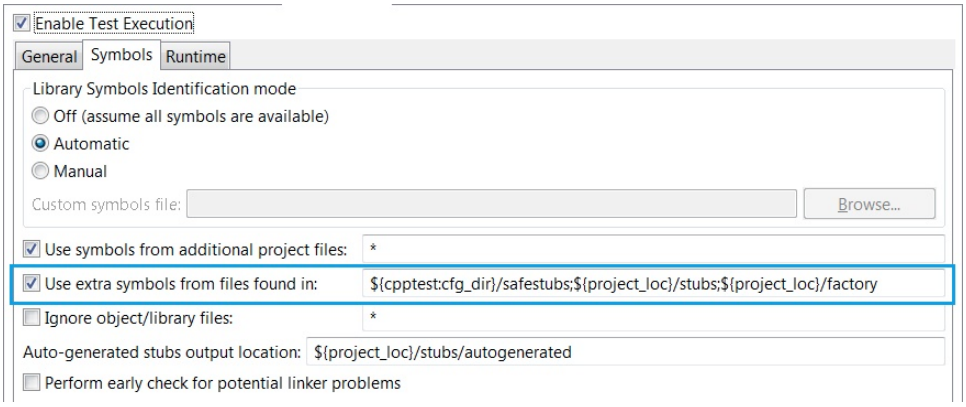
In Application Memory Monitoring mode, by contrast, there is usually no need to provide hardware initialization code in form of a C++test-specific function. This is because the original initialization code should be called for this purpose, since C++test does not replace the application entry point in Application Monitoring mode. However, if you do need to perform a C++test-specific initialization/finalization in Application Monitoring mode, the CppTest_Initialize/Finalize functions can be used the same way.

Adding CppTest_Initialize and CppTest_Finalize Functions

CppTest_Initialize and CppTest_Finalize functions can be added directly to your project. You can also place them in a source file that does not belong to your project. The source file must be placed in a dedicated directory that is not a part of your originial project and matches one of the locations specified in your test configuration:

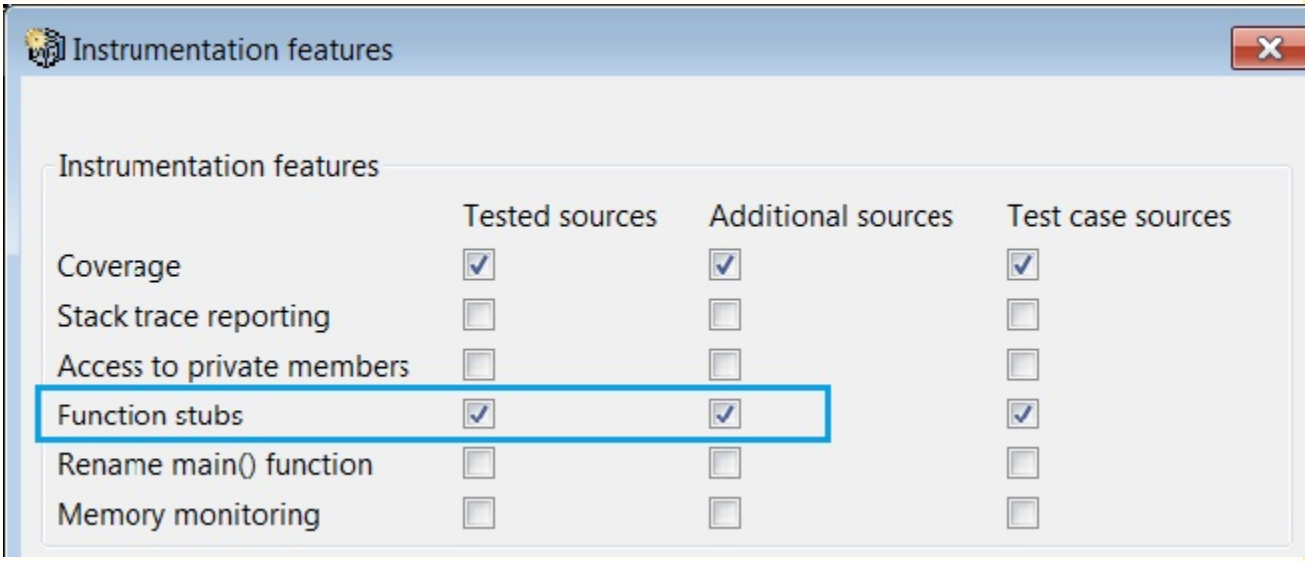
1. Open the Test Configuration.
2. Choose **Execution> Symbols** tabs.
3. Specify the folder containing the source file in the **Use extra symbols from files found in**: field.

4. Click **Apply**.



When a definition of this function(s) is provided, C++test will detect it during test preparation and generate a call to it at an early stage of the test executable startup.

Stubs instrumentation must be enabled for Tested sources and Additional sources:



Creating CppTest_Initialize and CppTest_Finalize Skeletons

C++test ships with a dedicated wizard for automating the installation of CppTest_Initialize and CppTest_Finalize funtion skeletons. If preferred, users can still manually add these functions.

To access the "Runtime - Custom initialization & finalization" wizard:

1. Choose **File > New > Other**.
2. Choose **C++test > Runtime - Custom initialization & finalization** and click **Next**.
3. Specify the source file name and location in the fields provided. The wizard should be synchronized with the value specified in the **Use extra symbols from files found in**: field in your preferred Test Configuration settings (see Adding CppTest_Initialize and CppTest_Finalize Functions).
4. Choose a file type (C or C++) from the drop-down menu and click **Finish**.

Applying the settings generates a source file in the specified location. The new file will contain the following empty definitions:

- void CppTest_Initialize(void)
- void CppTest_Finalize(void)

You can then extend these functions with your custom initialization and finalization code.

Tips

- If the runtime library is initialized too early, causing application monitoring issues, you can disable automatic runtime library initialization by using the advanced option testrunner.earlyRuntimeInitialization (see Advanced Instrumentation Configuration Options for details) and call to CppTest_InitializeRuntime() in the appropriate place in the source code. We recommend having this call guarded with #ifdef PARASOFT_CPPTEST.
- To finalize the runtime library early, you can place a call to CppTest_FinalizeRuntime() directly in the source code. After this function is called, data such as coverage, unit test messages, asserts, and runtime violation will be ignored.

Symbols Used By the C++test Runtime Library

The following external symbols are used by the C++test runtime library:

- close
- dup
- dup2
- exit
- fclose
- fflush
- fopen
- fprintf
- fputc
- free
- fwrite
- getenv
- malloc
- memcmp
- memcpy
- memset
- mkstemp
- open
- perror
- pthread_exit
- pthread_mutex_destroy
- pthread_mutex_init
- pthread_mutex_lock
- pthread_mutex_unlock
- pthread_self
- read
- siglongjmp
- signal
- sprint
- stdout
- strcmp
- strcpy
- strlen
- strncmp
- strstr
- tolower
- vprintf
- vsprintf
- write

Notes

- If "CPPTEST_USE_ANSI_SETJMP" define is set to 1, the following symbols will be referenced from system libraries:
  - longjmp
  - setjmp
- If "CPPTEST_USE_ANSI_SETJMP" define is set to 0, then the symbols listed below are used instead:
  - siglongjmp
  - sigsetjmp
- If "CPPTEST_TIME_MODE" define is set to CPPTEST_TIME_ANSI, the following symbol will be referenced from system libraries:
  - time
- If "CPPTEST_TIME_MODE" define is set to CPPTEST_TIME_GETTIMEOFDAY, the following symbol will be referenced from system libraries:
  - gettimeofday
- If "CPPTEST_TIME_MODE" define is set to CPPTEST_TIME_TIMEB, the following symbol will be referenced from system libraries:
  - ftime
- If "CPPTEST_TIME_MODE" define is set to CPPTEST_TIME_TICKLIB, the following symbols will be referenced from system libraries:
  - tickGet
  - sysClkRateGet
- If "CPPTEST_USE_FILE_COMMUNICATION" or "CPPTEST_USE_FILE_SPLIT_COMMUNICATION" is defined, the following symbols will be referenced from system libraries:
  - fopen
  - fclose
  - fputc
  - fwrite
  - fflush
- If "CPPTEST_USE_UNIX_SOCKET_COMMUNICATION" is defined, the following symbols will be referenced from system libraries:
  - socket

    - gethostbyname
    - htons
    - inet_addr
    - connect
    - close
    - send
- If "CPPTEST_USE_WIN_SOCKET_COMMUNICATION" is defined, the following symbols will be referenced from system libraries:
    - WSAStartup
    - socket
    - gethostbyname
    - htons
    - inet_addr
    - connect
    - closesocket
    - shutdown
    - WSACleanup
    - send
- If "CPPTEST_C_STREAMS_REDIRECT_ENABLED" or "CPPTEST_CPP_STREAMS_REDIRECT_ENABLED" is defined, the following symbols will be referenced from system libraries:
    - dup
    - dup2
    - _get_osfhandle
    - perror
    - strstr
    - unlink
    - write
    - open
    - close
    - GetTempFileName (_WIN32)
    - GetTempPath (_WIN32)
    - mkstemp (!_WIN32)