

HCL® OneTest™ Embedded
8.3.0 Documentation
September 2020



Special notice

Before using this information and the product it supports, read the information in [Notices on page mcli](#).

Contents

Chapter 1. Release Notes.....	5	Testing software components.....	446
Description.....	5	Using the graphical user interface.....	657
What's new in HCL OneTest™ Embedded 8.3.0.....	5	Test script languages.....	715
Installing the product.....	7	Chapter 6. Test Manager Guide.....	924
Known issues.....	7	Generating test reports.....	924
Contacting HCL support.....	7	Generating 2D and 3D chart data.....	924
Chapter 2. System Requirements.....	8	Publishing HTML reports to the Server.....	925
Chapter 3. Getting Started Guide	9	Opening runtime analysis reports.....	926
Overview.....	9	About test reports.....	927
Source code instrumentation overview.....	11	About coverage reports.....	928
Target deployment port overview.....	12	About memory profiling reports.....	931
Chapter 4. Administrator Guide.....	15	About performance profiling reports.....	935
Installing.....	15	About metrics results.....	936
Installation requirements.....	15	Viewing 2D and 3D charts.....	938
Installing the product.....	16	Chapter 7. Reference Guide.....	939
Verifying the installation.....	20	UI reference.....	939
Starting HCL OneTest™ Embedded.....	20	HCL OneTest™ Embedded preferences.....	939
Managing Licenses.....	21	TDP configuration settings.....	944
Configuring	23	Build configuration settings.....	947
Target Deployment Port Editor overview.....	23	Data pool editor reference.....	957
Opening the Target Deployment Port Editor.....	24	UML sequence diagram reference.....	958
Creating a TDP.....	24	Memory profiling errors.....	958
Using the TDP Editor.....	25	Memory profiling warnings.....	960
Editing customization points in a TDP.....	26	Command Line Interface.....	962
Updating a Target Deployment Port.....	27	Running a Node from the Command Line....	963
Integrating.....	29	Command Line Runtime Analysis for C and C+ +.....	965
IBM® Rational® Quality Manager integration overview.....	29	Command Line Component Testing for C, Ada and C++.....	966
Integrating HCL OneTest™ Embedded with other development tools.....	32	Command Line System Testing for C.....	967
Command line examples.....	968		
Chapter 5. Test Execution Specialist Guide.....	45	Setting Environment Variables for CLI tools.....	969
Testing with HCL OneTest™ Embedded for Eclipse IDE.....	45	Creating an Options Header File for the Testing component for C++ or for System Testing... 971	971
Getting started with HCL OneTest™ Embedded for Eclipse IDE.....	45	Preparing a Products Header File.....	972
Importing C projects.....	46	Instrumenting and Compiling the Source Code.....	972
Importing HCL OneTest™ Embedded examples.....	47	Compiling the TDP Library.....	973
Analyzing source code.....	47	Compiling the Test Harness.....	974
Coupling Analysis overview.....	161	Linking the Application.....	975
Application Profiling.....	178	Running the Test Harness or Application.....	976
Testing software components.....	188	Troubleshooting Command Line Usage.....	976
Testing with Studio.....	224	Splitting the trace dump file.....	978
HCL OneTest™ Embedded Studio overview.....	224	Opening Reports from the Command Line.....	978
Analyzing static source code.....	224	Code Coverage HTML Report Generator.....	979
Analyzing running applications	305		

RTistFdcConv command line.....	980
Studio Reference.....	981
User interface reference.....	981
Runtime and static analysis reference.....	1040
Command line interface.....	1053
Output window preferences.....	1151
Notices.....	mclii
Index.....	1156

Chapter 1. Release Notes

This document contains information about new features and enhancements for HCL OneTest™ Embedded and links to useful information about the products.

Contents

- [Description on page 5](#)
- [What's new in HCL OneTest Embedded 8.3.0 on page 5](#)
- [Installing the product on page 7](#)
- [Known issues on page 7](#)
- [Contacting HCL support on page 7](#)

Description

HCL OneTest™ Embedded is a complete test and runtime analysis tool set for systems development created in any cross-development environment.

HCL OneTest™ Embedded provides tools for automated component testing, code coverage, memory leak detection, performance profiling, and UML sequence diagram tracing.

What's new in HCL OneTest™ Embedded 8.3.0

You can find information about the features introduced in this release of HCL OneTest™ Embedded.

The following sections list the new features, enhancements or other changes made in this release.

- Justification of non-coverage
 - It could be difficult to achieve 100% of code coverage during the testing phases because some pieces of code could be non-reachable by tests.
 - Testers can now justify the non-reachable branches with a pragma in the relevant pieces of source code.
 - This justification is taken into account in the coverage percentage and the justification is included in the report.

For more information, see [Justification of non-covered lines of code on page 58](#) for Eclipse IDE and for Studio.
- Coverage of Assembler language for ARM 32 bits processor
 - HCL OneTest™ Embedded includes now assembler code coverage for ARM 32 bits processor using the gcc for ARM tool chain.
 - Assembler source files are taken into account in the builds as the C/C++ source files.
 - Or, optionally, C/C++ source files could be instrumented at the assembler level instead of being instrumented at the C/C++ level.
 - The supported coverage levels are:

- Functions
- Functions and exits
- Statement blocks
- Calls
- Coverage of Assembler language for ARM processor is fully compatible with C/C++ and Ada coverage.
- A new TDP for Raspberry Pie is now delivered.

For more information, see [Code coverage for assembler source files on page 65](#) for Eclipse IDE and [Code coverage for assembler source files on page 361](#) for Studio.

- Support Eclipse 4.7.3 and 4.12 (2019.06) and CDT 9
 - HCL OneTest™ Embedded is now delivered with Eclipse 4.7.3 and CDT 9.
 - It can be also installed on Eclipse 4.12 (a.k.a. 2019.06) using the update site mechanism.
- Execute OTD test scripts in Eclipse IDE
 - A tester can now import and run OTD test scripts from Studio to Eclipse IDE and run them, exactly as he does with PTU test scripts. See [Testing with .otd test scripts on page 214](#)
 - Reports are generated in the same way as it is in Studio.
- New Code Review HTML report
 - Code review post-processor generates now a new HTML report additionally to the standard report.
 - This report is created from a template that the tester can modify to introduce his own logo, or titles for example. See [Customizing the code review report on page 156](#)
- MISRA 2004 and 2012 rules improvements

See [Code review MISRA 2004 rules on page 95](#) and [Code review MISRA 2012 rules on page 126](#)

- The rules related to the number of lines of code propose now different ways to compute this number of lines of code (with or without the empty lines, with or without the comments).
- New rules are available:
 - max # of functions in a compilation unit
 - max # of global variables in a compilation unit
 - max # of lines in a compilation unit
 - max # of parameters in a function
- Open Eclipse IDE to other testing tools
 - Eclipse IDE supports now test scripts based on Python, shell or windows command instructions. See [Testing with Python, Perl, Windows or Linux scripts on page 216](#) and [Configuring Python, Perl, Windows or Linux scripts on page 217](#)
 - Runtime analysis features can be used in parallel.
 - HTML reports generated by these tools can also be part of the global report.

For more details, see [Creating test suites on page 207](#).

- Publish HTML reports on HCL OneTest™ Embedded

OneTest Embedded HTML reports can now be published on OneTest Server. See [Publishing HTML reports to the Server on page 925](#).

Installing the product

You can find information about the installation and upgrade instructions for HCL OneTest™ Embedded.

For instructions about installing the software, see [Installing on page 15](#).

You cannot upgrade HCL OneTest™ Embedded from an earlier version of the product to version 8.3.0. If you have an earlier version of the product, you must uninstall it before installing HCL OneTest™ Embedded 8.3.0.

Known issues

You can find information about the known issues identified in this release of HCL OneTest™ Embedded

Product	Download document	Knowledge Base
HCL OneTest™ Embedded		Knowledge articles

The knowledge base is continually updated as problems are discovered and resolved. By searching the knowledge base, you can quickly find workarounds or solutions to problems.

Contacting HCL support

- For technical assistance, contact [HCL Customer Support](#).
- Before you contact HCL support, you must gather the background information that you might need to describe your problem. When you describe a problem to the HCL support specialist, be as specific as possible and include all relevant background information so that the specialist can help you solve the problem efficiently. To save time, know the answers to these questions:
 - What software versions were you running when the problem occurred?
 - Do you have logs, traces, or messages that are related to the problem?
 - Can you reproduce the problem? If so, what steps do you take to reproduce it?
 - Is there a workaround for the problem? If so, be prepared to describe the workaround.

Chapter 2. System Requirements

This document includes information about hardware and software requirements for HCL® OneTest™ Embedded.

For information about hardware and software compatibility for HCL OneTest™ Embedded, download the PDF document on [HCL License and Delivery portal](#) where the product files are hosted.

Chapter 3. Getting Started Guide

This guide provides an overview of HCL® OneTest™ Embedded. You can find the information to get you started with HCL OneTest™ Embedded. This guide is intended for new users.

Before you can perform the various tasks described in the *Getting Started Guide* and the other guides, you must install HCL OneTest™ Embedded. See [Installing on page 15](#).

Overview

HCL® OneTest™ Embedded is a complete test and runtime analysis tool set for systems development created in any cross-development environment. HCL OneTest™ Embedded provides tools for automated component testing, code coverage, memory leak detection, performance profiling, and UML sequence diagram tracing.

Systems development includes (but is not limited to) embedded, real-time and/or technical systems development. And this type of software is often performed in conjunction with the larger scope of a systems engineering activity. HCL OneTest™ Embedded is a cross-platform solution designed specifically for developers creating software applications for products of embedded (for example, mobile phone, medical device, handled global positioning system, and so on), real-time (for example, aerospace, automotive or telecommunications control system), and other technical systems applications for example, simulated research computation and advanced grid computing systems).

Implementing a practical, effective and professional testing process within your organization has become essential because of the increased risk that accompanies software complexity. The time and cost devoted to testing must be measured and managed accurately. Very often, lack of testing causes schedule and budget overruns with no guarantee of quality. Critical trends require software organizations to be structured and to automate their test processes. These trends include:

- Ever increasing quality and time to market constraints.
- Growing complexity, size and number of software-based equipment.
- Lack of skilled resources despite need for productivity gains
- Increasing interconnections of critical and complex embedded systems.
- Proliferation of quality and certification standards throughout critical software markets, including the avionics, medical, and telecommunications industries.

HCL OneTest™ Embedded provides a full range of answers to these challenges by enabling full automation of system and software test processes.

HCL OneTest™ Embedded is a complete test and runtime analysis tool set for embedded, real-time and networked systems created in any cross-development environment. Automated testing, code coverage, memory leak detection, performance profiling, UML tracing, code review - with HCL OneTest™ Embedded you fix your code before it breaks.

HCL OneTest™ Embedded covers runtime analysis and software testing, all in a fully integrated testing environment.

The latest release of HCL OneTest™ Embedded integrates with Rational Quality Manager to provide a more collaborative approach to product software development and testing. HCL OneTest™ Embedded is the most complete automated developer testing solution available on a wide range of host and target platforms. In addition, new integrations with other popular development tool environments allow developers to work in the environment of their choice. This enables the powerful testing capabilities of HCL OneTest™ Embedded to be used early in the product software development lifecycle because it is part of the developers daily work environment.

Target deployment port technology

Target deployment port (TDP) technology is a versatile, low-overhead mechanism that enables target-independent testing and runtime analysis with limitless target support. As a key component of HCL OneTest™ Embedded, TDP technology allows your tests be run directly on your target embedded hardware.

Each TDP is customized to accommodate your compiler, linker, debugger, and target architecture. Tests are independent of the TDP, so that the tests don't change when your environment changes. For example, you can run the same tests and code on the embedded hardware or on your local computer by switching the TDP and rebuilding the project.

Target deployment ports are designed to strongly reduce the data communication and runtime overhead that can affect your embedded systems when tested, while being versatile enough to adapt to any cross-development environment (RTOS, compiler, debugger, target communication) within a very short time.

DO-178B/C Qualification Kit

All HCL OneTest™ Embedded customers have access to the HCL OneTest™ Embedded DO-178B/C Qualification Kit, which can be submitted with your other project artifacts to meet DO-178B/C compliance requirements. The Qualification Kit covers unit testing for C and Ada languages, coverage for C and Ada languages and code review for C language (MISRA 2004).

For more information about DO-178bB/C support, contact the Products & Platforms Customer Support via this link: <https://www.hcltech.com/products-and-platforms/support>.

Related information

[Target deployment port overview on page 12](#)

[Source code instrumentation overview on page 11](#)

[Integrating HCL OneTest Embedded with other development tools on page 32](#)

[Analyzing static source code on page 224](#)

[Runtime analysis overview on page 306](#)

Source code instrumentation overview

Source code insertion (SCI) technology uses instrumentation techniques that automatically add specific code to the source files under analysis. After compilation, execution of the code produces dump data for runtime analysis or component testing.

HCL® OneTest™ Embedded makes extensive use of source code insertion technology to transparently produce test and analysis reports on both native and embedded target platforms.

Instrumentation overhead

Instrumentation overhead is the increase in the binary size or the execution time of the instrumented application, which is due to source code insertion (SCI) generated by the Runtime Analysis features. Source code insertion technology is designed to reduce both types of overhead to a bare minimum. However, this overhead may still impact your application. The following table provides a quick estimate of the overhead generated by the product.

- **Code Coverage Overhead:** Overhead generated by the Code Coverage feature depends largely on the coverage types selected for analysis.

A 48-byte structure is declared at the beginning of the instrumented file. Depending on the information mode selected, each covered branch is referenced by an array that uses

- 1 byte in Default mode
- 1 bit in Compact mode
- 4 bytes in Hit Count mode

The actual size of this array may be rounded up by the compiler, especially in Compact mode because of the 8-bit minimum integral type found in C. See Information Modes for more information. Other Specifics:

- Loops, switch and case statements: a 1-byte local variable is declared for each instance.
- Modified/multiple conditions: one n -byte local array is declared at the beginning of the enclosing routine, where n is the number of conditions belonging to a decision in the routine I/O is either performed at the end of the execution or when the end-user decides (please refer to Coverage Snapshots in the documentation).

In summary, hit count mode and modified/multiple conditions produce the greatest data and execution time overhead. In most cases you can select each coverage type independently and use pass mode by default in order to reduce this overhead. The source code can also be partially instrumented.

- **Memory and Performance Profiling and Runtime Tracing:** Any source file containing an instrumented routine receives a declaration for a 16 byte structure. Within each instrumented routine, a n byte structure is locally declared, where n is 16 bytes +4 bytes for Runtime Tracing, +4 bytes for Memory Profiling, and $+3*t$ bytes for Performance Profiling, where t is the size of the type returned by the clock-retrieving function.

For example, if t is 4 bytes, each instrumented routine is increased of 20 bytes for Memory Profiling only, 20 bytes for Runtime Tracing only, 28 bytes for Performance Profiling only, or 36 bytes for all Runtime Analysis features together

- **Memory Profiling Overhead:** Any call to an allocation function is replaced by a call to the Memory Profiling Library. These calls aim to track allocated blocks of memory. For each memory block, $16+12*n$ bytes are allocated to contain a reference to it, as well as to contain link references and the call stack observed at allocation time. n depends on the Call Stack Size Setting, which is 6 by default. If ABWL errors are to be detected, the size of each tracked, allocated block is increased by $2*s$ bytes where s is the Red Zone Size Setting (16 by default). If FFM or FMWL errors are to be detected, a Free Queue is created whose size depends on the Free Queue Length and Free Queue Size Settings. Queue Length is the maximum number of tracked memory blocks in the queue. Queue Size is the maximum number of bytes, which is the sum of the sizes of all tracked blocks in the queue.
- **Performance Profiling Overhead:** For any source file containing at least one observed routine, a 24 byte structure is declared at the beginning of the file. The size of the global data storing the profiling results of an instrumented routine is $4+3*t$ bytes where t is the size of the type returned by the clock retrieving function.
- **Runtime Tracing Overhead:** Implicit default constructors, implicit copy constructors and implicit destructors are explicitly declared in any instrumented classes that permits it. Where C++ rules forbid such explicit declarations, a 4 byte class is declared as an attribute at the end of the class.

Instrumentation technology is designed to reduce both performance and memory overhead to a minimum.

Nevertheless, for certain cross-platform targets, it may need to be reduced still further. There are three ways to do this.

- Limiting code coverage types: When using the Code Coverage feature, procedure input and simple and implicit block code coverage are enabled by default. You can reduce instrumentation overhead by limiting the number of coverage types.



Note: The Code Coverage report can only display coverage types among those selected for instrumentation.

- Limiting instrumented calls: When calls are instrumented, any instruction that calls a C user function or library function constitutes a branch and thus generates overhead. You can disable call instrumentation on a set of C functions using the Selective Code Coverage Instrumentation Settings. For example, you can usually exclude calls to standard C library functions such as *printf* or *fopen*.
- Optimizing the information mode: When using Code Coverage, you can specify the information mode, which defines how much coverage data is produced and therefore stored in memory.

Target deployment port overview

Target deployment port (TDP) technology is a versatile, low-overhead technology enabling target-independent tests and run-time analysis despite limitless target support.

As a key component of HCL® OneTest™ Embedded, TDP technology allows your test cases as well as test execution analysis to be applied directly to your target embedded system. It is constructed to accommodate your

compiler, linker, debugger, and target architecture. Tests are independent of the TDP, so tests don't change when the environment does. Test script deployment, execution and reporting remain easy to use.

TDPs are designed to strongly reduce the data communication and runtime overhead that can affect your embedded systems when tested, while being versatile enough to adapt to any cross-development environment (RTOS, compiler, debugger, target communication) within a very short time.

TDP technology includes the following capabilities and benefits:

- Compiler dialect-aware and linker-aware, for transparent test building.
- Easy download of the test harness environment onto the target via the user's IDE, debugger, simulator or emulator.
- Painless test and run-time analysis results download from the target environment using JTAG probes, emulators or any available communication link, such as serial, Ethernet or file system.
- Powerful test execution monitoring to distribute, start, synchronize and stop test harness components, as well as to implement communication and exception handling.
- Versatile communication protocol adaptation to send and receive test messages.
- XML-based Target Deployment Port Editor enabling simple, in-house TDP customization

Obtaining target deployment ports

TDP technology was designed to adapt to any embedded or native target platform. This means that you need a particular TDP to deploy HCL® OneTest™ Embedded to your target. A wide array of TDPs has already been developed to suit most target platforms. The following platforms are already supported:

- Native development platforms: Windows™ and Linux™, the development platforms that leading companies in the devices/embedded systems and infrastructure industries are using.
- Cross-development environments: From 8- to 64-bit cross-development environments from WindRiver, GreenHills, ARM, Sun, Montavista, TI, NEC, Hitachi, Nohau, and more.

If there is no existing TDP for your particular target platform, HCL Customer Support can provide the service of creating a tailored TDP for you.

To obtain a copy of an existing TDP or to inquire about custom development, follow one of the methods:

- Contact your HCL sales representative.
- If you do not know your sales representative, contact [HCL Customer Support](#) or create an [HCL Support case](#).

Creating new target deployment ports

You can choose to create, unassisted, a TDP tailored for your embedded environment. There are several requirements to consider before choosing this option:

- Perl language knowledge: The HCL OneTest™ Embedded compiler interface is written in perl
- Programming language and compiler knowledge: The HCL OneTest™ Embedded runtime library uses the same language as the code under test (C, C++, Ada)
- Knowledge of HCL OneTest™ Embedded: Improve your experience with the product before considering your first TDP. You will need to be familiar with the runtime analysis and component testing tools and how the TDP is used with them.

Before creating a TDP for a new target platform, determine whether the target platform is capable of running embedded tests. To create a TDP, see the documentation that is embedded in the Target Deployment Port Editor, which provides an overview and detailed information on setting up a TDP, and using the Target Deployment Port Editor.

Chapter 4. Administrator Guide

This guide describes how to install the HCL® OneTest™ Embedded software.

After you install the software, you can perform administration tasks such as license configuration, user management, security, memory and disk usage management, back up and restore user data, and other tasks that a server administrator can perform. This guide is intended for administrators.

Installing

This section provides the instructions for installing IBM® Installation Manager and the product as well as installation verification. To install your product, follow the procedures and information in these topics. Installing the product involves verifying requirements, planning, managing licenses, and configuring web-based help.



Note: Before installing, consult the HCL OneTest™ Embedded release note for late-breaking installation issues.

Installation requirements

This section details hardware, software, and user privilege requirements that must be met in order to successfully install and run HCL OneTest™ Embedded.

Software requirements

Before you begin the installation, verify that your system meets the software requirements.

For information about hardware and software compatibility for HCL OneTest™ Embedded, download the PDF document on [HCL License and Delivery portal](#) where the product files are hosted.

User privileges requirements

You must have a user ID that meets the following requirements before you can install HCL OneTest™ Embedded.

- Your user ID must not contain double-byte characters.
- You must install Installation Manager as an administrator on Windows if the version of your operating system requires user privileges to install or update product offerings, or install license keys for your products.
- If you install Installation Manager as an administrator on Windows, all products installed from Installation Manager must be run with the administrator privilege. In this case, you must run HCL OneTest™ Embedded as an administrator.
- If you install Installation Manager as a non-administrator on Windows, HCL OneTest™ Embedded can be installed with the same User account as the one used to install Installation Manager.
- You can enable users who are not the administrator so that they can work with HCL OneTest™ Embedded on some versions of Windows. If you are in such a case:

- Do not install HCL OneTest™ Embedded into a package group (installation location) in the Program Files directory (C:\Program Files\) and do not choose a shared resources directory in the Program Files directory.
- If you are *extending* an existing Eclipse installation, then do not install Eclipse in the Program Files directory (C:\Program Files\).
- On Linux, you must be able to log in as root (with sudo) to install and run HCL OneTest™ Embedded.
- On Ubuntu, you must ensure that the environment variables that are set while installing the products are retained when you open HCL OneTest™ Embedded and the application-under-test.
- HCL License Server must be installed on Windows™ at the same time or prior to HCL OneTest™ Embedded so that the license information entered during HCL OneTest™ Embedded installation is valid. If you need to install HCL License Server on Windows™ with a User account, right-click the `launchpad.exe` file, and click **Run as Administrator** or install the HCL License Server separately with an Administrator account.

Installing software

Installing the product involves verifying requirements, planning, performing pre-installation tasks and managing licenses.

Installation roadmap

The installation roadmap lists the high-level steps for installing your product.

Roadmap for installing HCL OneTest™ Embedded

Perform these tasks to install HCL OneTest™ Embedded:

1. Review the [release notes on page 5](#).
2. Plan the installation.
 - a. Review [System Requirements on page 8](#).
 - b. Review [user privilege requirements on page 15](#).
 - c. Plan for installation locations.
 - d. Plan for product coexistence.
 - e. Install with your instance of Eclipse.
3. Set up and manage product licenses.

Pre-installation Tasks for Studio

Before you install the product, you need to prepare or configure your computer.

Pre-installation checklist

Before you install your product, review the following information and ensure that the preinstallation steps are completed as required.

About this task

To help ensure a smooth installation process, complete these tasks before starting the installation tasks.

1. For HCL OneTest™ Embedded Studio support, you must first install Exuberant Ctags. See [Installing Exuberant Ctags on page 17](#) for more information.
2. Ensure that your existing compilers and development environments are installed and run properly. In particular, if you are using Microsoft™ Visual Studio, install and run it at least once before installing HCL OneTest™ Embedded. See [Support for Microsoft Visual Studio on page 18](#) for more information.
3. For UNIX™: If you want the product to be used by users other than root, then set the umask variable to 0022 **before you install the product**. To set this variable, log in as root user, start a terminal session, and type `umask 0022`.

Installing Exuberant Ctags

Before using HCL OneTest™ Embedded Studio on Windows™, you must ensure that Exuberant Ctags is installed on your computer and that the directory containing Ctags binary files is set in the PATH environment variable.

To install Exuberant Ctags:

1. Go to the following website and download the latest package labeled Source and binary for Windows™: <http://ctags.sourceforge.net>.
If the latest binary package is not available for download, go to the **Download** section and download the binary package for the previous version of Ctags.
2. Extract the file to `C:\installation_directory\HCL\HCLOneTest\Embedded\ctags`.
3. From the **Start** menu, select **Parameters > Control Panel > System**.
4. Select the **Advanced** tab and click **Environment variables**.
5. Edit the *PATH* environment variable to add the `C:\installation_directory\HCL\HCLOneTest\Embedded\ctags` directory and click **OK**.

Installing Cygwin

Before using HCL OneTest™ Embedded Studio on Windows™, you must ensure that Cygwin is installed on your computer and that the directory containing Cygwin binary files is set in the PATH environment variable.

To install Cygwin:

1. Go to the following website, on the **Install Cygwin** page and download the latest package for 32 or 64 bits versions of Windows™: <http://www.cygwin.com>.
2. Run the setup program. Once the root install directory and local package are selected, select a download site.
3. Check **MAKE** box.
4. Then, select a packages to install. You must select gcc, gcc-core, gcc: GNU Compiler Collection (C) and (C++) and GNU version of the make utility.
If you want to use the Cygwin gcc compiler, make sure that the Cygwin installation options include the development tools category. If not, you can install a different gcc 3.2 compiler.

Update the PATH environment variable:

5. From the **Start** menu, select **Parameters > Control Panel > System**.
6. Select the **Advanced** tab and click **Environment variables**.
7. Edit the *PATH* environment variable to add the Cygwin installation directory, for example `c:\cygwin\bin;` and click **OK**.

Support for Microsoft™ Visual Studio

If you plan to use HCL OneTest™ Embedded on Windows™ with Microsoft™ Visual Studio you must install Visual Studio and execute it at least once before installing HCL OneTest™ Embedded in order to correctly initialize the Windows™ registry database.

About this task

If you omitted to run Visual Studio before installing HCL OneTest™ Embedded, the installation produces an error message. In this case, proceed with the installation and then execute the following steps.

To enable support of Microsoft™ Visual Studio after installation:

1. Run and close Visual Studio at least once.
2. Open a Windows™ Explorer and browse to the following directory:
`C:\installation_directory\HCL\HCLOneTest\Embedded\targets\xml\`
3. Double-click the `cvisual6.xdp` (for Visual 6.0) or `cvisual7.xdp` (for Visual .NET), or `cvisual8.xdp` (for Visual 2005). This opens the Target Deployment Port (TDP) in the Target Deployment Port Editor.
4. Save the TDP to regenerate the TDP directory.

Increasing the number of file handles on Linux™ workstations

For best product performance, increase the number of file handles above the default of 1024 handles.

About this task

 **Important:** Before you work with your product, increase the number of file handles. Most products use more than the default limit of 1024 file handles per process. A system administrator might need to make this change.

Exercise caution when using the following steps to increase your file descriptors on Linux™. If the instructions are not followed correctly, the computer might not start correctly.

To increase your file descriptors:

1. Log in as root. If you do not have root access, you will need to obtain it before continuing.
2. Change to the `/etc` directory

! **Attention:** If you decide to increase the number of file handles in the next step, *do not* leave an empty `initscript` file on your computer. If you do so, your computer will not start up the next time that you turn it on or restart.

- Use the vi editor to edit the `initscript` file in the `etc` directory. If this file does not exist, type `vi initscript` to create it.
- On the first line, type `ulimit -n 4096`. The point is that 4096 is significantly larger than 1024, the default on most Linux™ computers.

! **Important:** Do not set the number of handles too high, because doing so can negatively impact system-wide performance.

- On the second line, type `eval exec "$4"`.
- Save and close the file after making sure you have completed steps 4 and 5.

📌 **Note:** Ensure that you follow the steps correctly. If this procedure is not completed correctly, your computer will not start.

- Optional:** Restrict the number of handles available to users or groups by modifying the `limits.conf` file in the `/etc/security` directory. Both SUSE Linux™ Enterprise Server (SLES) Version 9 and Red Hat Enterprise Linux™ Version 4.0 have this file by default. If you do not have this file, consider using a smaller number in step 4 in the previous procedure (for example, 2048). Do this so that most users have a reasonably low limit on the number of open files that are allowed per process. If you use a relatively low number in step 4, it is less important to do this. However, if you set a high number in step 4 earlier and you do not establish limits in the `limits.conf` file, computer performance can be significantly reduced.

The following sample `limits.conf` file restricts all users, and then sets different limits for others afterwards. This sample assumes that you set handles to 8192 in step 4 earlier.

```
*      soft nofile 1024
*      hard nofile 2048
root   soft nofile 4096
root   hard nofile 8192
user1  soft nofile 2048
user1  hard nofile 2048
```

Note that the `*` in the preceding example sets the limits for all users first. These limits are lower than the limits that follow. The root user has a higher number of allowable handles open, while number available to user1 is between the two. Make sure that you read and understand the documentation contained in the `limits.conf` file before making changes.

Verifying the installation

When the installation process is complete, a message confirms the success of the process. You can open the log file to verify your installation of the product.

Before you begin

When the installation process is complete, a message confirms the success of the process.

To verify the installation:

1. Click **View log file**. The installation log file for the current session opens in a new window. To continue, close the Installation Log window to continue.
2. In the Install Package wizard, select whether you want HCL OneTest™ Embedded to start when you exit.
3. Click **Finish** to launch the selected package. The Install Package wizard closes and you are returned to the Start page of Installation Manager.

Starting HCL OneTest™ Embedded

You can start your product from the desktop environment or a command-line interface.

About this task

For Microsoft™ Windows™ operating systems:

- Click **Start > Programs > >** to start HCL OneTest™ Embedded for Eclipse IDE.
- Click **Start > Programs > >** to start HCL OneTest™ Embedded Studio for testing C, C++, Ada.
- To start HCL OneTest™ Embedded for Eclipse IDE from a command line, type this command:

```
<installation_directory>\eclipse.exe -product com.ibm.rational.testrealtime.product.ide.
```
- To start HCL OneTest™ Embedded Studio from a command line, type this command:

```
<installation_directory>\bin\intel\win32\studio.exe
```

If the installation location or Shared Resources directory for your product is in a directory in the path `C:\Program Files`, you can run the product only as the administrator. To run as administrator, right-click the program shortcut, and click **Run as administrator**.



Note: From Windows™ 7 and later versions, the `Program Files` directory is usually virtualized in order to allow users who are not running as the administrator to have write access to this protected directory. However, the virtualization workaround is not compatible with your product. If you selected an installation location or shared resources directory in the path `C:\Program Files\` and you do not want to require running your product as Administrator, complete one of these steps:



- Reinstall your product and any other programs that sharing the same installation location, and select an installation location that is not in the path `C:\Program Files\`.
- Reinstall your product and all Software Delivery Platform product packages (regardless of their installation location), and select a shared resources directory and installation locations that are not in the path `C:\Program Files\`

For UNIX™ operating systems:

- To start HCL OneTest™ Embedded for Eclipse IDE from a sh or bash shell, type this command:

```
<installation_directory>/start_visualtest.sh
```

- To start HCL OneTest™ Embedded Studio from a sh or bash shell, type this command:

```
<installation_directory>/start_testrt.sh
```

Managing Licenses

Licensing for your HCL software is administered through [HCL® License & Delivery portal](#). This portal is FlexNet-based web application to manage software entitlements and licenses.

You must have ordered software. When a software order is placed and acknowledged, a software entitlement is created. You can then create the devices and map the software entitlement with the devices. Every device is associated to a server ID. This server ID is applied in the product. Multiple software entitlements can be created based on your requirements. Follow the instructions in Software Order Acknowledgment document that you received to activate your entitlement, create devices, and download the software from the Portal.

If you do not have access to the Internet, you can install and configure a local license server.

Setting up licensing

To start using a product, you must first set up licensing.

Before you begin

If you use a cloud license server, you must have:

- The ID of the server ID on cloud license server where your entitlements have been mapped.
- If the FlexNet cloud server <https://hclsoftware.compliance.flexnetoperations.com> is not the default one, you need the URL of the cloud server you use

If you use a local license server, you need its URL.

About this task

You must set up either a cloud license server or a local license server. It is done by setting some environment variables according to the Operating System.

Set up a cloud license server:

Before starting the product, you must set up the following environment variable:

- **HCL_LICENSING_ID**: (mandatory): contains the server ID that was provided to you or copy the ID from the HCL® License & Delivery portal.
- **HCL_LICENSING_URL**: (optional): contains the cloud server URL. If it is not set, it points to the HCL cloud license.

When you launch the product, it connects to the HCL® License & Delivery portal to verify this server ID and if there is a license available, it is checked out so that you can use the product. If the license is not available, a message is displayed about it, in the console.



Note: If the license is not used for 15 minutes, the license is returned to the server for others to consume it.

Set up a local license server

Sometimes, you might be working from a lab that does not have access to the Internet. In such cases, you must install and set up a local license server behind your company's firewall. For more information about installing and configuring the local license server, see the documentation of the local license server on the same page from where you downloaded the product bits. As part of configuration, the local license server must already have mapped your entitlements and therefore can serve your requests.

Before starting the product, you must be set the following environment variable:

- **HCL_LICENSING_URL**: (mandatory) must be set up to point to your local server.

Example: `HCL_LICENSING_URL=http://myServer:portNumber.`

HCL_LICENSING_ID environment variable must be blank or undefined. If you enter a value, the local server will be used as a cloud server, and the licenses do not apply.



Note: The HCL® Local License Server is supported on 32 and 64-bit Windows, and Linux platforms. You must install the server on a physical computer and not on a Virtual Machine.

Setting environment variables

See "Set up cloud license server" or "Set up local license server" above to find the required variables and values.

• Procedure on Windows:

- Open the **System Properties** window. Select the **Advanced** tab then click **Environment variables**.
- Under **System variables**, click **New** to add one of these environment variables:

- **Procedure on Linux:**

On some occasions, your computer might not be connected to the open Internet. For such cases, you can set up a local proxy server that can be on cloud or local.

What to do next

You can now work with the product.

Configuring

Use these topics to configure the product.

Target Deployment Port Editor overview

The TDP Editor provides a user interface designed to help you customize and create Target Deployment Ports (TDP) for any platform on which you want to run tests or programs.

The Target Deployment Port Editor user interface is made up of 4 main sections:

- **Navigation:** Use the navigation explorer view to select customization points.
- **Help:** This area provides direct reference information for the selected customization point.
- **Edit:** Use this area to edit the customization point. The form of the **Edit** window depends on the nature of the customization point.
- **Comment:** Use this area to store comments or descriptions for each customization point.

In the **Navigation** view, you can click on any customization point to obtain detailed reference information for that parameter in the **Help** area. Use this information to customize the TDP to suit your requirements.

Target Deployment Port Editor overview

The TDP Editor provides a user interface designed to help you customize and create Target Deployment Ports (TDP) for any platform on which you want to run tests or programs.

The Target Deployment Port Editor user interface is made up of 4 main sections:

- **Navigation:** Use the navigation explorer view to select customization points.
- **Help:** This area provides direct reference information for the selected customization point.
- **Edit:** Use this area to edit the customization point. The form of the **Edit** window depends on the nature of the customization point.
- **Comment:** Use this area to store comments or descriptions for each customization point.

In the **Navigation** view, you can click on any customization point to obtain detailed reference information for that parameter in the **Help** area. Use this information to customize the TDP to suit your requirements.

Opening the Target Deployment Port Editor

Target Deployment Ports (TDP) are stored as XDP files, which can be viewed and edited with the Target Deployment Port Editor.

To open a TDP in the Target Deployment Port Editor:

1. From the **Start** menu, click **> Target Deployment Port Editor**, or from a shell or command window, type the command: `tdpeditor`.
2. Click **File > Open**
3. In the `targets` directory, select an XDP file and click **Open**.
4. Save your changes and reload the TDP in HCL OneTest™ Embedded:
 - In HCL OneTest™ Embedded for Eclipse IDE, right-click the project and click **Properties > C/C++ Build > Settings > TDP Build**, select another TDP and select the updated TDP again. Click **OK**.
 - In HCL OneTest™ Embedded Studio, restart HCL OneTest™ Embedded Studio, click **Project > Configuration**, select the TDP, click **Remove**. Click **New**, select the updated TDP again and click **OK**.

To open a TDP from HCL OneTest™ Embedded:

5. First you must have the Target Deployment Port view displayed in HCL OneTest™ Embedded. To open this view, in the toolbar associated with the HCL OneTest™ Embedded perspective, click **Window**, and select **Show View > Other > > Target Deployment Port**.
6. The **Target Deployment Port** view opens and displays the list of all the Target Deployment Ports that are installed in HCL OneTest™ Embedded. Select a Target Deployment Port and click the  button to edit the selected Target Deployment port.
From this view, you can also open the preferences panel and configure the Target Deployment Port search path.

Creating a TDP

This topic provides a typical example workflow for creating a new target deployment port (TDP) for a C compiler.

About this task

Creating a new TDP requires advanced familiarity with:

- HCL® OneTest™ Embedded and its underlying TDP technology.
- The target platform hardware and software architecture.
- The target development environment.

1. In the Target Deployment Port Editor, at the top of the **Navigation** area, right-click the TDP name and type a new name.
2. Specify all the Basic settings. Create intermediate keys to help with future changes and save the TDP.
3. In HCL OneTest™ Embedded Studio, open the `add.rtp` project which is located in `examples/TDP/tutorial`. This is a simple project that can be used for debugging target deployment ports.

4. Click **Edit > Preferences > Project** and select **Verbose**.
5. Click **Project > Configuration** to create a new configuration, and select the new TDP. Click **OK**.
6. Select the new configuration based on the new TDP.
7. Click **Settings > Build > Build Options > ...** and remove all instrumentation. At this point any modifications of the `DEFAULT_xxxx` in the Target Deployment Port Editor will be ignored in the project. Therefore, you must duplicate or copy any changes in the **Build > Build > Compiler/Link** configuration settings.
8. In the project browser, right click `add.c` and select **Compile**. Check that the object file is generated in the correct directory. If any problems occurred, open the Target Deployment Port Editor and correct the problems in **Build Settings > Compilation function**. Repeat this step until `add.c` is properly generated.
9. In the **Build > Build options > ...** settings, enable coverage instrumentation only and remove all files located in the `examples/TDP/tutorial/xdp_name` directory.
10. In the project browser, right click `add.c` and select **Compile**. The instrumentation occurs after the preprocessing and before compilation. Check the `.i` file is generated properly in the correct directory and that it contains `#line xx "fileName"` or `# xx "fileName"`. If any problems occurred, open the Target Deployment Port Editor and correct the problems in **Build Settings > Preprocessing function**. Repeat this step until the `.i` file is properly generated.
11. Check that `add.o` or `add.obj` is generated in the correct directory and not a file named `add_aug.o` or `add_aug.obj`. If any problems occurred, open the Target Deployment Port Editor and correct the problems in **Build Settings > Compilation function**. Repeat from step 9 until `add.o` or `add.obj` are properly generated.
12. In the project browser, right click `TP.c` and select **Compile**. Check that `TP.o` or `TP.obj` are generated in the correct directory. If any problems occurred, open the Target Deployment Port Editor and correct the problems in Library Settings. Repeat this step until `TP.o` or `TP.obj` are properly generated.
13. Check that `Test.exe` is generated in the correct directory. If any problems occurred, open the Target Deployment Port Editor and correct the problems in **Build Settings > Link function**. Repeat this step until `Test.exe` is properly generated.



Note: Any files added in the TDP Editor Build settings are located in `$TARGETDIR/cmd` by default.

Using the TDP Editor

The TDP Editor provides a user interface designed to help you customize and create unified Target Deployment Ports.

The TDP Editor is made up of 4 main sections:

- **A Navigation Tree:** Use the navigation tree on the left to select customization points.
- **A Help Window:** Provides direct reference information for the selected customization point.
- **An Edit Window:** The format of the **Edit** Window depends on the nature of the customization point.
- **A Comment Window:** Lets you to enter a personal comment for each customization point.

In the Navigation Tree, you can click on any customization point to obtain detailed reference information for that parameter in the **Help** Window. Use this information to customize the TDP to suit your requirements.

Note The TDP Editor is not included with the trial version of the product.

To learn about	See
Making changes to the TDP	Editing customization points in a TDP on page 26
Launching the TDP Editor	Opening a Target Deployment Port on page 24
Creating a new TDP	Creating a TDP on page 24
Applying changes made to a TDP	Updating a Target Deployment Port on page 27
Changing the way a TDP is generated	Using a Post-generation Script on page 27
Importing old TDPs from ATTOL Testware products	Migrating from Pre-v2002 Target Deployment Ports on page 28

Editing customization points in a TDP

Use the Target Deployment Port Editor to adapt an existing Target Deployment Port (TDP) to a specific target platform or development environment.

About this task

Target Deployment Ports can be subdivided into four primary sections:

- **Basic Settings:** This section specifies default file extensions, default compilation and link flags, environment variables and custom variables required for your target environment. This section allows you to set all the common settings and variables used by HCL OneTest™ Embedded and the different sections of the TDP. For example, the name and location of the cross compiler for your target is stored in a Basic Settings variable, which is used throughout the compilation, preprocessing and link functions. If the compiler changes, you only need to update this variable in the Basic Settings section.
- **Build Settings:** This section configures the functions required by the HCL OneTest™ Embedded build process. It defines compilation, link and execution Perl scripts, plus any user-defined scripts when needed. This section is the core of the TDP, as it drives all the actions needed to compile and execute a piece of code on the target. All files related to the Build settings are stored in the `cmd` subdirectory of the TDP folder.
- **Library Settings:** This section describes a set of source code files and a dedicated customization file (`custom.h`), which adapt the TDP to target platform requirements. This section is the most complex and usually only requires customization for specialized platforms (unknown RTOS, no RTOS, unknown simulator, emulator, etc.). These files are stored in the `lib` subdirectory of the TDP folder.
- **Parser Settings:** This section modifies the behavior of the parser in order to address non-standard compiler extensions (for example: non-ANSI extensions). This section allows HCL OneTest™ Embedded to properly parse your source code, either for instrumentation or code generation purposes. The resulting files are stored in the `ana` subdirectory of the TDP folder.

1. In the **Navigation** view of the Target Deployment Port Editor, select the customization point that you want to edit.
2. In the **Help** window, read the reference information pertaining to the selected customization point. Use this information fill out the **Edit** window.
3. Type any remarks or comments in the **Comments** window.
4. Save your changes and reload the TDP in HCL OneTest™ Embedded:

Choose from:

- In HCL OneTest™ Embedded for Eclipse IDE, right-click the project and click **Properties > C/C++ Build > Settings > TDP Build**, select another TDP and select the updated TDP again. Click **OK**.
- In HCL OneTest™ Embedded Studio, restart HCL OneTest™ Embedded Studio, click **Project > Configuration**, select the TDP, click **Remove**. Click **New**, select the updated TDP again and click **OK**.

Updating a Target Deployment Port

Target Deployment Technology

The Target Deployment Port (TDP) settings are read or loaded when a HCL OneTest™ Embedded project is opened, or when a new Configuration is used.

If you make any changes to the Basic Settings of a TDP with the TDP Editor, any project settings that are read from the TDP will not be taken into account until the TDP has been reloaded in the project.

To reload the TDP in HCL OneTest™ Embedded:

1. From the **Project** menu, select **Configurations**.
2. Select the TDP and click **Remove**.
3. Click **New**, select the TDP and click **OK**.

Related Topics

[Editing customization points in a TDP on page 26](#) | [Creating a TDP on page 24](#)

Using a Post-generation Script

Target Deployment Technology

In some cases, it can be necessary to make changes to the way the TDP is written to its directory beyond the possibilities offered by the TDP editor.

To do this, the TDP editor runs a post-generation Perl script called **postGen.pl**, which can be launched automatically at the end of the TDP directory generation process.

To use the postGen script:

1. In the TDP editor, right click on the **Build Settings** node and select **Add child** and **Ascii File**.
2. Name the new node **postGen.pl**.
3. Write a perl function performing the actions that you want to perform after the TDP directory is written by the TDP Editor.

Example

Here is a possible template for the **postGen.pl** script file:

```
sub postGen
{
    $d=shift;

    # the only parameter passed to this function is the path to the target directory

    # here any action to be taken can be added

}

1;
```

The parameter **\$d** contains `<tdp_dir>/<tdp_name>`, where `<tdp_dir>` is a chosen location for the TDP directory (by default, the **targets** subdirectory of the product installation directory), and `<tdp_name>` is the name of the current TDP directory

Related Topics

[Creating a TDP on page 24](#)

Migrating from v2001A Target Deployment Ports

Target Deployment Technology

This section describes the conversion of TDPs built for older versions (before v2002) of HCL OneTest™ Embedded to the current, unified format.

This section applies to TDPs and ATTOL Target Packages created for:

- ATTOL Coverage, UniTest and SystemTest
- HCL OneTest™ Embedded v2001A

TDPs created for later versions of HCL OneTest™ Embedded or **HCL OneTest™ Embedded** are compatible with the current version.

To migrate your old TDP to the current format:

1. In the TDP Editor, create a new Target Deployment Port based on the appropriate new template:
 - use **templatec.xdp** for C and C++ TDPs
 - use **templatea.xdp** for Ada TDPs
2. Item by item, recode or copy-paste information from your old TDP to the corresponding customization points in the TDP Editor, using the information in this section of the Target Deployment Guide to direct you.

Related Topics

[Updating a Target Deployment Port on page 27](#) | [Upgrading from a previous version](#)

Integrating

Read these topics to learn how the product works when integrated with other products.

IBM® Rational® Quality Manager integration overview

IBM® Rational® Quality Manager is a business-driven software quality environment for people seeking a collaborative and customizable solution for test planning, workflow control, tracking and metrics reporting capable of quantifying how project decisions and deliverables impact and align with business objectives.

Rational® Quality Manager allows you to:

- Create Rational® Quality Manager test environments that are linked to HCL OneTest™ Embedded target deployment ports
- Create Rational® Quality Manager test scripts that are linked to HCL OneTest™ Embedded test assets.
- Deploy and run HCL OneTest™ Embedded tests for the Rational® Quality Manager interface.
- View HTML reports in the Rational® Quality Manager interface.

Rational® Quality Manager uses the term *test script* to describe its basic test assets. Rational® Quality Manager test scripts are mapped to HCL OneTest™ Embedded test suites. A test suite contains multiple test harnesses that are run sequentially to provide global results for a project.

To use Rational® Quality Manager with a computer that uses HCL OneTest™ Embedded for Eclipse IDE, the HCL OneTest™ Embedded adapter service must be running on the computer.

With the adapter running, you can import test suites as Rational® Quality Manager test scripts, construct a new Rational® Quality Manager test case based on those test suites, and run the tests from Rational® Quality Manager. You can also view the results of the tests in Rational® Quality Manager as HTML reports.

Related information

[Initializing the Rational Quality Manager adapter on page 30](#)

[Importing test suites into Rational Quality Manager on page 31](#)

Initializing the Rational® Quality Manager adapter

To use Rational® Quality Manager with a computer that uses HCL OneTest™ Embedded for Eclipse IDE, the HCL OneTest™ Embedded adapter service must be properly running and configured on the computer.

Before you begin

You need administrator privilege to run Rational® Quality Manager adapter service on Windows and Linux.

As an RQM user, you must have write access to a valid RQM Public URL and project and the appropriate RQM CALs.

From HCL OneTest™ Embedded for Eclipse IDE V8.2.0, Rational® Quality Manager V6.0.5 is required.

To start the HCL OneTest™ Embedded for Eclipse IDE and Rational® Quality Manager adapter, follow these steps:

1. Run command prompt as an administrator user on Windows. On Linux open the command shell and enter `sudo` to have root rights.
2. Start the Rational® Quality Manager adapter service with the following command, located in the `\RQMAdapter\TestRTAdapter` folder of the product installation directory:

Choose from:

- On Windows™, enter the following command: `startTestRTAdapter.bat`

```
C:\Program Files\HCL\HCLOneTest\Embedded\RQMAdapter\TestRTAdapter\"startTestRTAdapter.bat"
```

- On UNIX™, enter `startTestRTAdapter.sh`

```
sudo startTestRTAdapter.sh
```



Note: The adapter requires access to a writable temporary directory. The `%TEMP%` variable is used to access to the default directory. If the adapter encounters permission problems with the default settings, add the following option to the command to specify a writable directory:

`-tempDir=temp_directory`. For example: `startTestRTAdapter.bat -tempDir=C:\temp`.

3. If this is the first time you run the adapter, you must configure the adapter by typing the following information, when prompted, in the command window:
 - a. Type the base URL of the Rational® Quality Manager server.

Example

For example: `https://hostname:9443/jazz`
 - b. Type your login and password for the Rational® Quality Manager account.

- c. Type the Rational® Quality Manager project area name.
- d. Type a name for the adapter, or press **Return** to use the default name.



Note: This step is not mandatory. If you don't enter any name, the default adapter name is taken into account.

The adapter only asks these questions the first time it is run. If you need to change the server URL or login information, run the adapter with the `-reconfigure` option as follows:

- On Windows, enter:

```
C:\Program Files\HCL\HCLOneTest\Embedded\RQMAadapter\TestRTAdapter\startTestRTAdapter.bat
-reconfigure
```

- On Linux, enter:

```
sudo startTestRTadapter.sh -reconfigure
```

Results

The Rational® Quality Manager adapter service starts.

Related information

[IBM Rational Quality Manager integration overview on page 29](#)

[Importing test suites into Rational Quality Manager on page 31](#)

Importing test suites into Rational® Quality Manager

The Rational® Quality Manager adapter for HCL OneTest™ Embedded enables you to import HCL OneTest™ Embedded test suites as Rational® Quality Manager test scripts.

To import a HCL OneTest™ Embedded test suite into Rational® Quality Manager:

1. Log in to Rational® Quality Manager and click **Construction > Import test scripts**.
2. In **Script type**, select .
3. Select **Use test resources that are local to a test machine** and click **Select Adapter**.
4. Select the HCL OneTest™ Embedded adapter that you want to use and click Next.
5. In Project Path, specify the path to the workspace project where the HCL OneTest™ Embedded test suite is located, and select **Go**.

The adapter parses all the subdirectories under the selected directory, therefore, if you specify a workspace path, it will find all the test suites in that workspace.

6. Select one or several test suites to import, click **Finish** and **Import**.

What to do next

Once the test scripts are imported, construct a new test case in Rational® Quality Manager with the HCL OneTest™ Embedded test suites. After running the Rational® Quality Manager test case, click **Close** and **Show results**. You can click the links in the **Result Details** section of Rational® Quality Manager to view the HTML reports.

Related information

[IBM Rational Quality Manager integration overview on page 29](#)

[Initializing the Rational Quality Manager adapter on page 30](#)

Integrating HCL OneTest™ Embedded with other development tools

HCL OneTest™ Embedded Studio is a versatile tool that is designed to integrate with your existing development environment.

To learn about	See
Rational ClearCase integration	Working with Rational ClearCase on page 33
Rational ClearQuest integration	Working with Rational ClearQuest on page 34
Microsoft Visual Studio integration	Configuring Microsoft Visual Studio on page 38
Using third party configuration management software	Working with Configuration Management on page 32

Integrating Studio with configuration management

The GUI provides an interface that allows you to control your project files through a configuration management (CM) system such as Rational® ClearCase® and submit software defect report to a Rational® ClearCase® system.

Note Before using any configuration management tool, you must first configure the CMS Preferences dialog box. See Customizing Configuration Management.

You can also set up the GUI to use a CM system of your choice.

To learn about	See
Configuration management with Rational ClearCase	Working with Rational ClearCase on page 33
Reporting defects with Rational ClearQuest	Working with Rational ClearQuest on page 34

Setting up the GUI to use a third-party configuration management tool.

[Customizing source control tools on page 35](#)

Related Topics

[CMS Preferences on page 1011](#) | [ClearQuest Preferences on page 1012](#) | [Working with Other Development Tools on page 32](#)

Integrating Studio with IBM Rational ClearCase

Rational® ClearCase® is a configuration management system (CMS) tool providing version control, workspace management, configuration process, and build management. With Rational® ClearCase®, your development team gets a scalable, best-practices-based development process that simplifies change management – shortening your development cycles, ensuring the accuracy of your releases, and delivering reliable builds and patches for your previously shipped products.

By default, HCL OneTest™ Embedded offers configuration management support for Rational® ClearCase®. You can however customize the product to support different configuration management software. When using Rational® ClearCase®, you can instantly control your files from the product **Tools** menu.



Note: Before using ClearCase commands, select Rational® ClearCase® as your CMS tool in the [CMS Preferences. on page 1011](#)

Source Control Commands.

For any file in the HCL OneTest™ Embedded project, Rational® ClearCase®, or any other CMS tool, can be accessed through a set of source control commands.

Source control can be applied to all files and nodes in the Project Browser or Asset Browser. When a source control command is applied to a project, group, application, test or results node, it affects all the files contained in that node.

The following source control commands are included to be used with Rational® ClearCase®:

- Add to Source Control
- Check Out
- Check In
- Undo Check Out
- Compare to Previous Version
- Show History
- Show Properties

Refer to the documentation provided with Rational® ClearCase® for more information about these commands.

Source control commands are fully configurable from the **Tools** menu.

To control files from the Tools menu:

1. Select one or several files in the **Project Explorer** window.
2. From the **Tools** menu, select Rational® ClearCase® and the source control command that you want to apply.

To control files from the Source Control popup menu:

1. Right-click one or several files in the **Project Explorer** window.
2. From the popup menu, select Source Control and the source control command that you want to apply.

Related Topics

[Working with Rational ClearQuest on page 34](#) | [CMS Preferences on page 1011](#) | [About the Tools Menu on page 712](#) | [Customizing source control tools on page 35](#)

Integrating Studio with IBM Rational ClearQuest

IBM® Rational® ClearQuest® is a defect and change tracking tool designed to operate in a client/server environment. It allows you to easily track defects and change requests, target your most important problems or enhancements to your product. Rational® ClearQuest® helps you determine the quality of your application or component during each phase of the development cycle and helps you track the release in which a feature, enhancement or bug fix appears.

By default, the product offers defect tracking support for Rational® ClearQuest®. When using ClearQuest with HCL OneTest™ Embedded Studio you can directly submit a report from a test or runtime analysis report.

To submit a ClearQuest report from HCL OneTest™ Embedded Studio:

1. In the **Report Explorer**, right-click a test.
2. From the pop-up menu, select **Submit ClearQuest Report**.
3. This opens the **ClearQuest Submit Defect** window, with information about the **Failed test**.
4. Enter any other necessary useful information, and click **OK**.

For more information, see the Rational® ClearQuest® documentation.

Related Topics

[ClearQuest Preferences on page 1012](#)

Customizing source control tools

Out of the box, the product offers configuration management support for [Rational ClearCase on page 33](#), but the product can be configured to use most other Configuration Management Software (CMS) that uses a vault and local repository architecture and that offers a command line interface.

To configure the product to work with your version control software:

1. Add a new CMS tool to the Toolbox with the command lines for checking files into and out of the configuration management software. This activates the **Check In** and **Check Out** commands in the [Project Explorer on page 1021](#) and the ClearCase Toolbar.
2. Set up version control repository in CMS Preferences.

Related Topics

[Working with Rational ClearCase on page 33](#) | [CMS Preferences on page 1011](#) | [About the Tools Menu on page 712](#)

Working with IBM Rational Quality Manager

Integrating Studio with IBM Rational Quality Manager

Rational® Quality Manager is a business-driven software quality environment for people seeking a collaborative and customizable solution for test planning, workflow control, tracking and metrics reporting capable of quantifying how project decisions and deliverable impact and align with business objectives.

HCL OneTest™ Embedded Studio integration with Rational Quality Manager enables you to:

- Create Rational Quality Manager test environments that are linked to HCL OneTest™ Embedded target deployment ports
- Create Rational Quality Manager test scripts that are linked to HCL OneTest™ Embedded Studio projects and tests or application nodes
- Deploy and run HCL OneTest™ Embedded Studio tests for the Rational Quality Manager interface
- View HTML reports in the Rational Quality Manager interface

To learn about

Enabling a computer with HCL OneTest™ Embedded to be used by Rational Quality Manager for running tests

Creating Rational Quality Manager test scripts with HCL OneTest™ Embedded projects

See

[Running the Rational Quality Manager adapter on page 36](#)

[Importing a HCL OneTest™ Embedded project on page 37](#)

Running tests with different Target Deployment Ports.

[Using Target Deployment Ports with Rational Quality Manager on page 37](#)

Running the Rational Quality Manager adapter

To use Rational Quality Manager with a computer that uses HCL OneTest™ Embedded Studio, the HCL OneTest™ Embedded adapter must be running on the computer.

Before running the adapter, ensure that both the PATH and JAVA_HOME environment variables are properly set to the correct location of a Java Runtime Environment (JRE) version 1.5 or later.

When you run the adapter for the first time, you are asked to type configuration information in the console window.

To run the HCL OneTest™ Embedded adapter for Rational Quality Manager:

1. Open a command line window and navigate to the run the adapter command line:

```
<installation directory>\RQMAadapter\TestRTadapter\
```

2. Run the adapter command:

- On Windows, type **startTestRTAdapter.bat**, or from the **Start** menu, select **> Tools > Start > Adapter for Rational Quality Manager**.
- On UNIX, enter the following command:

```
startTestRTAdapter.sh
```

3. If you run the adapter for the first time, enter the following information in the command window:

- **Server URL:** Enter the URL of the Rational® Quality Manager server.
- **Login:** Enter the login used to connect to Rational® Quality Manager.
- **Password:** Enter your password.
- **Project Area (Optional):** Enter the name of project area, if necessary.
- **Enter adapter name:** Enter the name of the HCL OneTest™ Embedded adapter on the current computer as it will appear in Rational® Quality Manager. By default the name is **TestRT on <hostname>**.

The adapter only asks these questions the first time it is run. If you need to change this server URL and login information, run the adapter with the **-reconfigure** option.

```
startTestRTAdapter.bat -reconfigure
```

Related Topics

[Importing a HCL OneTest™ Embedded project on page 37](#) | [Associating Target Deployment Ports with test environments on page 37](#)

Importing a HCL OneTest™ Embedded project into Rational Quality Manager

IBM Rational Quality Manager integration

Rational Quality Manager uses the term *test script* to describe its basic test assets. The HCL OneTest™ Embedded adapter for Rational Quality Manager enables you to import HCL OneTest™ Embedded projects as Rational Quality Manager test scripts.

When you select the HCL OneTest™ Embedded adapter, the **HCL OneTest™ Embedded** project will be run with the default Target Deployment Port selected in the project.

To import a HCL OneTest™ Embedded project into Rational Quality Manager:

1. Log in to Rational Quality Manager, click **Construction > Import test script**.
2. In **Script type**, select HCL OneTest™ Embedded.
3. Select **Use test resources that are local to a test machine** and click **click Select Adapter**.
4. Select the HCL OneTest™ Embedded adapter that you want to use.
5. In **Project Path**, specify the directory where the HCL OneTest™ Embedded **.rtp** project file is located, and select **Go**. The adapter will parse all the sub-directories under the selected directory.
6. Select one or several **.rtp** project files, click **OK**, and then click **Import**.

Once the test scripts are imported, construct a new test case with the HCL OneTest™ Embedded test scripts. After execution, click **Close and show results**. You can click the links in the Result Details section to view the HTML reports.

Related Topics

[Running the Rational Quality Manager adapter on page 36](#) | [Associating Target Deployment Ports with test environments on page 37](#)

Associating Target Deployment Ports with test environments

IBM Rational Quality Manager integration

When you select the HCL OneTest™ Embedded adapter in Rational Quality Manager, by default, the HCL OneTest™ Embedded project will be run with the Target Deployment Port (TDP) that is selected in the project. To run the same project with different TDPs, you can create different test environments in Rational Quality Manager.

To create a test execution record with a specific TDP:

1. Log in to Rational Quality Manager, click **Lab Management > Create Test Environment**.
2. Type a name for the test environment that applies to the name of the HCL OneTest™ Embedded configuration. The name must be exactly the same as the Configuration name in HCL OneTest™ Embedded, for example: **C Win32 - GNU**.

3. Click **Save**.
4. Click **Construction > Create Test Execution Record** and enter a name for the new test execution record.
5. Select the **Test Case** and the **Default Test Script**.
6. In **Available Test Environments**, select the test environment with the name of the TDP that you want to use.
7. Click **Save**.

Related Topics

[Running the Rational Quality Manager adapter on page 36](#) | [Associating Target Deployment Ports with test environments on page 37](#)

Integrating Studio with Microsoft Visual Studio

HCL OneTest™ Embedded provides a special setup tool to configure runtime analysis tools with Microsoft Visual Studio 6.0.

Note Integration with Microsoft Visual Studio is only available with the Windows version of HCL OneTest™ Embedded Studio.

Installation

Both HCL OneTest™ Embedded and Microsoft Visual Studio must be installed on the same machine.

To install the Microsoft Visual Studio 6.0 plug-in:

1. From the **Windows Start** menu, select **Programs HCL® Software > Software HCL OneTest™ Embedded, Tools and HCL OneTest™ Embedded Plug-in for Microsoft Visual Studio Install** to add the new menu items to Microsoft Visual Studio

To uninstall the plug-in:

1. From the Windows Start menu, select **Programs > HCL OneTest™ Embedded Software > HCL OneTest™ Embedded Software, HCL OneTest™ Embedded, Tools and HCL OneTest™ Embedded Plug-in for Microsoft Visual Studio Uninstall** to remove the plug-in from Microsoft Visual Studio.

To install the Microsoft Visual Studio .NET plug-in:

1. From the Windows Start menu, select **All Programs > HCL OneTest™ Embedded Software > HCL OneTest™ Embedded > Tools > TDP Editor**.
2. In the TDP Editor, select **File > Open** and open `cvisual7.xdp` located in the `<install_directory>/targets/xml` directory.
3. Under **Basic Settings > For All**, set the `INSTALL_PLUGIN` key to `TRUE`.
4. Save `cvisual7.xdp` and close the TDP Editor.

To uninstall the plug-in:

1. From the Windows Start menu, select All Programs > HCL OneTest™ Embedded Software > HCL OneTest™ Embedded > Tools > TDP Editor.
2. In the TDP Editor, select File > Open and open cvisual7.xdp located in the <install_directory>/ targets/xml directory.
3. Under Basic Settings > For All, set the INSTALL_PLUGIN key to FALSE.
4. Save cvisual7.xdp and close the TDP Editor.

Configuration

The HCL OneTest™ Embedded setup for Microsoft Visual Studio tool allows you to set up and activate coverage types and instrumentation options for HCL OneTest™ Embedded Studio runtime analysis features, without leaving Microsoft Visual Studio.

To run the product Setup for Microsoft Visual Studio:

In Microsoft Visual Studio, two new items are added to the Tools menu:

- **HCL OneTest™ Embedded Viewer:** this launches the Studio user interface, providing access to reports generated by HCL OneTest™ Embedded runtime analysis and test features.
- **HCL OneTest™ Embedded Options:** this launches the Setup for Microsoft Visual Studio tool.

The following commands are available:

- **Apply:** Applies the changes made
- **OK:** Apply the choices made and leave the window
- **Enable or Disable:** Enable or Disable the runtime analysis tools
- **Cancel:** Cancels modifications

Code Coverage Instrumentation Options

See [About Code Coverage on page 54](#) and the sections about coverage types.

- **Function instrumentation:**
 - Select **None** to disable instrumentation of function inputs, outputs and termination instructions.
 - Select **Function** to instrument function inputs only.
 - Select **Exit** to instrument function inputs, outputs and termination instructions.

- **Function calls instrumentation (C only):**

- Select **None** to disable function call instrumentation.
- Select **Call** to enable function call instrumentation.

- **Block instrumentation**

- Select **None** to disable block instrumentation.
- Select **Statement Blocks** to instrument simple blocks only.
- Select **Implicit Blocks** to instrument simple and implicit blocks.
- Select **Loops** to instrument implicit blocks and loops.

- **Condition instrumentation (C only)**

- Select **None** to disable condition instrumentation
- Select **Basic** to instrument basic conditions
- Select **Modified/Multiple** to instrument multiple
- Select **Forced** to instrument forced multiple conditions

- **No Ternaries Code Coverage:** when this option is selected, simple blocks corresponding for the ternary expression true and false branches are not instrumented

- **Instrumentation Mode:** see [Information Modes on page 309](#) for more information.

- **Pass mode:** allows you to distinguish covered branches from those not covered.
- **Count mode:** The number of times each branch is executed is displayed in addition to the pass mode information in the coverage report.
- **Compact mode:** The compact mode is similar to the Pass mode. But each branch is stored in one bit instead of one byte to reduce overhead.

Other Options

- **Dump:** this specifies the dump mode:

- Select **None** to dump on exit of the application
- Select **Calling** to dump on call of the specified function
- Select **Incoming** to dump when entering the specified function
- Select **Returning** to dump when exiting from the specified function

- **Static Files Directory:**allows you to specify where the **.fdc**and **.tsf**files are to be generated
- **Runtime Tracing:**this option activates the Runtime Tracing runtime analysis feature
- **Memory Profiling:**this option activates the Memory Profiling runtime analysis feature
- **Performance Profiling:**this option activates the Performance Profiling runtime analysis feature
- **Other:**allows you to specify additional command-line options that are not available using the buttons. See the Reference help for a complete list of Instrumentor options.

Related Topics

[Using Runtime Analysis Features on page 306](#) | [Importing Files from a Microsoft Visual Studio Project file on page 689](#) | [Working with Rational ClearQuest on page 34](#) | [Working with Rational ClearCase on page 33](#)

Integrating HCL OneTest™ Embedded Studio with Microsoft Visual Studio

Integration with Microsoft Visual Studio is only available for the Windows versions of HCL OneTest™ Embedded Studio.

HCL OneTest™ Embedded Studio and Microsoft Visual Studio 6.0 must be installed on the same machine.

- To enable the integration with Visual Studio, from the Windows **Start** menu, select **Programs > HCL OneTest™ Embedded, Tools > HCL OneTest™ Embedded Plug-in for Microsoft Visual Studio 6.0 Install**to add the new menu items to Microsoft Visual Studio.
- To disable the integration with Visual Studio, from the Windows **Start** menu, select **Programs > HCL OneTest™ Embedded, Tools > HCL OneTest™ Embedded Plug-in for Microsoft Visual Studio 6.0 Uninstall**to add the new menu items to Microsoft Visual Studio.

Related Topics

[Configuring Microsoft Visual Studio Integration on page 38](#) | [Importing Files from a Microsoft Visual Studio Project file on page 689](#)

Integrating Studio with Microsoft Visual Studio

HCL OneTest™ Embedded provides a special setup tool to configure runtime analysis tools with Microsoft Visual Studio 6.0.

Note Integration with Microsoft Visual Studio is only available with the Windows version of HCL OneTest™ Embedded Studio.

Installation

Both HCL OneTest™ Embedded and Microsoft Visual Studio must be installed on the same machine.

To install the Microsoft Visual Studio 6.0 plug-in:

1. From the **Windows Start** menu, select **Programs HCL® Software > Software HCL OneTest™ Embedded, Tools and HCL OneTest™ EmbeddedPlug-in for Microsoft Visual Studio Install** to add the new menu items to Microsoft Visual Studio

To uninstall the plug-in:

1. From the Windows Start menu, select Programs > HCL OneTest™ Embedded Software > HCL OneTest™ Embedded Software, HCL OneTest™ Embedded, Tools and HCL OneTest™ Embedded Plug-in for Microsoft Visual Studio Uninstall to remove the plug-in from Microsoft Visual Studio.

To install the Microsoft Visual Studio .NET plug-in:

1. From the Windows Start menu, select All Programs > HCL OneTest™ Embedded Software > HCL OneTest™ Embedded > Tools > TDP Editor.
2. In the TDP Editor, select File > Open and open cvisual7.xdp located in the <install_directory>/ targets/xml directory.
3. Under Basic Settings > For All, set the INSTALL_PLUGIN key to TRUE.
4. Save cvisual7.xdp and close the TDP Editor.

To uninstall the plug-in:

1. From the Windows Start menu, select All Programs > HCL OneTest™ Embedded Software > HCL OneTest™ Embedded > Tools > TDP Editor.
2. In the TDP Editor, select File > Open and open cvisual7.xdp located in the <install_directory>/ targets/xml directory.
3. Under Basic Settings > For All, set the INSTALL_PLUGIN key to FALSE.
4. Save cvisual7.xdp and close the TDP Editor.

Configuration

The HCL OneTest™ Embedded setup for Microsoft Visual Studio tool allows you to set up and activate coverage types and instrumentation options for HCL OneTest™ Embedded Studio runtime analysis features, without leaving Microsoft Visual Studio.

To run the product Setup for Microsoft Visual Studio:

In Microsoft Visual Studio, two new items are added to the Tools menu:

- **HCL OneTest™ Embedded Viewer:** this launches the Studio user interface, providing access to reports generated by HCL OneTest™ Embedded runtime analysis and test features.
- **HCL OneTest™ Embedded Options:** this launches the Setup for Microsoft Visual Studio tool.

The following commands are available:

- **Apply:** Applies the changes made
- **OK:** Apply the choices made and leave the window
- **Enable or Disable:** Enable or Disable the runtime analysis tools
- **Cancel:** Cancels modifications

Code Coverage Instrumentation Options

See [About Code Coverage on page 54](#) and the sections about coverage types.

- **Function instrumentation:**

- Select **None** to disable instrumentation of function inputs, outputs and termination instructions.
- Select **Function** to instrument function inputs only.
- Select **Exit** to instrument function inputs, outputs and termination instructions.

- **Function calls instrumentation (C only):**

- Select **None** to disable function call instrumentation.
- Select **Call** to enable function call instrumentation.

- **Block instrumentation**

- Select **None** to disable block instrumentation.
- Select **Statement Blocks** to instrument simple blocks only.
- Select **Implicit Blocks** to instrument simple and implicit blocks.
- Select **Loops** to instrument implicit blocks and loops.

- **Condition instrumentation (C only)**

- Select **None** to disable condition instrumentation
- Select **Basic** to instrument basic conditions

- Select **Modified/Multiple** to instrument multiple
- Select **Forced** to instrument forced multiple conditions
- **No Ternaries Code Coverage**: when this option is selected, simple blocks corresponding for the ternary expression true and false branches are not instrumented
- **Instrumentation Mode**: see [Information Modes on page 309](#) for more information.
 - **Pass mode**: allows you to distinguish covered branches from those not covered.
 - **Count mode**: The number of times each branch is executed is displayed in addition to the pass mode information in the coverage report.
 - **Compact mode**: The compact mode is similar to the Pass mode. But each branch is stored in one bit instead of one byte to reduce overhead.

Other Options

- **Dump**: this specifies the dump mode:
 - Select **None** to dump on exit of the application
 - Select **Calling** to dump on call of the specified function
 - Select **Incoming** to dump when entering the specified function
 - Select **Returning** to dump when exiting from the specified function
- **Static Files Directory**: allows you to specify where the **.fd** and **.tsf** files are to be generated
- **Runtime Tracing**: this option activates the Runtime Tracing runtime analysis feature
- **Memory Profiling**: this option activates the Memory Profiling runtime analysis feature
- **Performance Profiling**: this option activates the Performance Profiling runtime analysis feature
- **Other**: allows you to specify additional command-line options that are not available using the buttons. See the Reference help for a complete list of Instrumentor options.

Related Topics

[Using Runtime Analysis Features on page 306](#) | [Importing Files from a Microsoft Visual Studio Project file on page 689](#) | [Working with Rational ClearQuest on page 34](#) | [Working with Rational ClearCase on page 33](#)

Chapter 5. Test Execution Specialist Guide

This guide describes tasks that you can perform to test application code in HCL OneTest™ Embedded for Eclipse IDE. This guide is intended for testers or test execution specialists.

Testing with HCL OneTest™ Embedded for Eclipse IDE

Read these topics to learn how to use the product.

Getting started with HCL OneTest™ Embedded for Eclipse IDE

HCL OneTest™ Embedded for Eclipse IDE is designed to integrate into your existing Eclipse-based tool chain. Use this section as a guide to a typical workflow for testing and evaluating your C source code in the Eclipse CDT environment.

Before you begin

These guidelines assume that you have some familiarity with the following concepts and tools:

- The Eclipse CDT development environment.
- The features and tools provided by HCL OneTest™ Embedded.
- The target platform on which you plan to run the tests.

About this task

It is important to understand the concepts and assets used by the product.

To start using HCL OneTest™ Embedded:

1. Familiarize yourself with the features and tools provided by the product. See [Overview on page 9](#) and [Test assets overview on page 189](#).
2. Choose whether you are going to create a new project or import an existing CDT project.

Choose from:

- If you already have an Eclipse CDT project, import the project into HCL OneTest™ Embedded for Eclipse IDE and convert it into a HCL OneTest™ Embedded project. See [Importing C projects on page 46](#).
- If you are starting a new C project or if want to import an unmanaged C project into Eclipse, create a new HCL OneTest™ Embedded project and import the source files. See [Creating test projects on page 191](#).



Note: There is currently no migration path from HCL OneTest™ Embedded Studio test scripts and command line tools into the HCL OneTest™ Embedded for Eclipse IDE environment.

3. Verify that the C source files build and run correctly.
Resolve any compilation errors if necessary.
4. Open the call graph to view the structure of your source code and create a new test harness. See [Creating test harnesses from the call graph on page 205](#).

The generated test harness contains a test case, and optionally a stub behavior.

5. Open the test case and edit the initial and expected expressions for the each variable check. See [Editing test cases on page 195](#).
6. Run the test harness and compare the obtained values with the expected values for each variable in the test case editor. See [Running a test harness on page 211](#).
If necessary, repeat from step 4 to ensure that you obtain a passed test result.
7. Generate a test report from the results. See [Generating test reports on page 924](#).
8. Deploy and run your test on the target platform by changing the test configuration and running the test again. See [Switching test configurations on page 210](#).

What to do next

Once your test harness is running correctly, you can use more of the features of the product:

- Measure code coverage, memory profiling, performance profiling, and static metrics with the runtime analysis tools. See [Runtime analysis overview on page 47](#).
- Include test data sets from data pools and create data dictionary to reuse data sets. See [Creating data pools on page 202](#) and [Data dictionary overview on page 201](#).
- Create more test harnesses or add test cases and stubs to existing test harnesses.
- Create test suites to run multiple test harnesses and compare their results. See [Creating test suites on page 207](#).
- Integrate test suites into IBM® Rational® Quality Manager. See [IBM Rational Quality Manager integration overview on page 29](#).

Importing C projects

You can either create a new C project with the Eclipse CDT tools or you can import your existing C source files or Eclipse projects into your HCL OneTest™ Embedded workspace.

About this task

HCL OneTest™ Embedded can only work with its own CDT managed build toolchain. Therefore, imported projects must be converted to HCL OneTest™ Embedded projects.

To import an existing C project:

1. Click **File > Import > General > Existing projects into workspace**.
2. Follow the wizard to import the project into the workspace.
3. After importing, right-click the project and select **Convert to HCL OneTest™ Embedded Project**.
4. Select the default target deployment port (TDP) for the project and click **Finish**.
5. If you have not already enabled CDT indexing, click **Window > Preferences > C/C++ > Indexer**, select **Index unused headers** and click **OK**.

Results

After conversion, the toolchain of the project is configured to use HCL OneTest™ Embedded instead of the default toolchain. If necessary, you can temporarily switch to the original toolchain in the project properties. However, you

must switch back to the HCL OneTest™ Embedded toolchain to use HCL OneTest™ Embedded runtime analysis and component testing tools.

 **Tip:** You can edit the CDT managed build toolchain to use environment variables with the UNIX notation `$$`. This can be useful when you are sharing projects with other users.

Related information

[Importing HCL OneTest Embedded examples on page 47](#)

[Creating test projects on page 191](#)

[Target deployment port overview on page 12](#)

Importing HCL OneTest™ Embedded examples

HCL OneTest™ Embedded is provided with several sample projects to help you get started.

To import the sample projects:

1. In the C/C++ perspective, click **File > Import > General > Existing projects into workspace** and click **Next**.
2. Click **Select root directory, Browse**, and choose a project folder in the following directory: `<product installation directory>/examplesEclipse/`.
3. Click **Select All** and select **Copy projects into workspace**.
4. Click **Finish**.

Related information

[Importing C projects on page 46](#)

[Creating test projects on page 191](#)

[Target deployment port overview on page 12](#)

Analyzing source code

Runtime analysis overview

The runtime analysis tools are designed to closely monitor the behavior of your application for debugging and validation purposes. These features use *source code insertion* to instrument the source code providing dynamic analysis of the application while it is running, either on a native or embedded target platform.

The following tools are available:

- *Code coverage* performs code coverage analysis.
- *Memory profiling* analyzes memory usage and detects memory leaks.

- *Performance profiling* provides metrics on execution time for each procedure/function/method of the application. For C language, it also provides an estimation of WCET.
- Control Coupling provides coverage information on Control Coupling that represent the interactions between modules (C language only).
- Data Coupling provides coverage information on def/use pairs identified in the application (C language only).
- Worst Stack Size computes an estimation of the maximum of the application stack size (C language only).
- *Runtime tracing* draws a real-time UML sequence diagram of your application.

Each of these runtime analysis tools can be used alone or together with the component testing features. When the source code is run with any of the runtime analysis tools engaged, either alone or in a component test, the source code is instrumented. The resulting instrumented code is then executed and the result is dynamically displayed in the corresponding reports.



Note: Instrumentation of the source code generates a certain amount of overhead, which can impact application size and performance.

Runtime analysis tools can analyze source code that complies with ANSI 89, ANSI 99, C99, and K&R C specifications.

Code coverage

Source code coverage consists of identifying which portions of a program are executed or not during a given test case. Source code coverage is recognized as one of the most effective ways of assessing the efficiency of the test cases applied to a software application.

The code coverage tool can provide the coverage information for the following source code elements:

- Statement blocks, decisions, and loops.
- Function or procedure calls.
- Basic conditions, modified conditions/decisions (MC/DC), multiple condition, and forced condition.
- Procedure entries and exits.
- Terminal or potentially terminal statements
- Statements that can't be covered in C.

For more information, see [Code review overview on page 90](#).

Memory profiling

Runtime memory errors and leaks are among the most difficult errors to locate and the most important to correct. The symptoms of incorrect memory use are unpredictable and typically appear far from the cause of the error. The errors often remain undetected until triggered by a random event, so that a program can seem to work correctly when in fact it's only working by accident. Memory profiling helps you detect HEAP memory allocation problems and leaks.

After execution of an instrumented application, the Memory Profiling report provides a summary diagram and a detailed report for both byte and HEAP memory block usage. The summary diagrams give you a quick overview of HEAP memory usage in blocks and bytes, including:

- The total HEAP memory allocated during the execution of the application.
- The HEAP memory that remains allocated after the application was terminated.
- The maximum HEAP memory usage encountered during execution

The detailed section of the report lists memory usage events identified as errors or warnings.

For more information, see [Memory profiling overview on page 69](#).



Restriction: With HCL OneTest™ Embedded for Eclipse IDE, static and stack memory are not checked, only dynamically allocated memory is checked.

Performance profiling

The performance profiling tool provides performance data for each software component so that you can locate the performance bottlenecks. With this information, you can concentrate your optimization efforts on those portions of code, which can lead to significant improvements in performance.

The Performance Profiling report provides function profiling data for your program and its components so that you can see exactly where your program spends most of its time. A **Top Functions** graph provides a high level view of the largest time consuming functions in your application. The **Performance Summary** section of the report indicates, for each instrumented function, procedure, or method (collectively referred to as functions), the number of calls and the time spent in the function and in its descendants. And for C language, it provides the Worst Case Estimation Time. For more information, see [Performance Profiling Results on page 78](#).

Runtime tracing

Runtime Tracing is a tool for monitoring real-time dynamic interaction analysis of your source code by generating trace data, which is dynamically turned into a UML sequence diagram. The diagram displays a lifeline of the interactions of the source code components. For more information, see [Runtime tracing overview on page 84](#).

Control Coupling

HCL OneTest™ Embedded introduces a new coverage level called "Control Coupling" for C language that consists in verifying that all the interactions between modules have been covered by at least one test. This new coverage level is implemented in HCL OneTest™ Embedded as follows:

- Modules on C language and compilation units (example: C files that are independently compiled).
- Interactions are calls between 2 functions that are in 2 different compilation units.
- Control Coupling is not a simple interaction. It is a control flow in the calling module that ends with an interaction with another module.

For more information, see [Control Coupling overview on page 162](#).

Data Coupling

HCL OneTest™ Embedded introduces a new coverage level called “Data Coupling” for C language that consists to verify that all the global variables of the application under test has been consumed in read (also called *use*) and write (also called *def*) during the tests, as following:

- For each global variable, HCL OneTest™ Embedded identifies the *def* and *use*. Then it considers all the possible *def/use* pair as a data coupling.
- To cover a Data Coupling, i.e. a *def/use* pair, it should exist at least one test that has executed this *def* and this *use*.

For more information, see [Overview on page 171](#).

Worst Stack Size

Static analysis and Dynamic analysis are used to provide an estimation of the worst stack size.

For more information, see [Worst Stack Size overview on page 179](#).

Enable runtime analysis tools

When the source code is run with any of the runtime analysis tools enabled, either alone or in a component test, the source code is instrumented and the results are displayed in a report.

Before you begin

Before running a test with any of the runtime analysis tools enabled, ensure that the correct Target Deployment Port (TDP) is selected.

To enable runtime analysis tools on your source code:

1. In the project explorer, right-click the project on which you want to enable the runtime analysis tools and click **Properties**.
2. Click **C/C++ Build > Settings** and select the **Build TDP** page to check that the correct TDP is selected.
If necessary, click the **Target Deployment Port** value to change the TDP.
3. Select the **Build Instrument** page and select **Settings > General > Selective instrumentation**.
4. Select the **Build Options** line and click **Edit**.
5. In the **Build Options** window, select the runtime analysis tools that you want to enable.
 - **Memory Profiling** detects memory leaks and allocation problems.
 - **Performance Profiling** locates performance issues and bottlenecks.
 - **Code Coverage** provides coverage information of the source code as it is run.
 - **Runtime Tracing** displays a dynamic UML sequence diagram of the run.
 - **Static Metrics** evaluates the complexity of the source code.
 - **Code Review** assesses compliance to coding rules.
 - **Debug** enables the workbench debug mode.

6. Click **OK, Apply** the changes and close the **Properties** window.
7. Click **Project > Clean > Clean all projects**.

Result

If the project is successfully built, in the project explorer, the **Binaries** folder contains the compiled binary executable for the project. If the project did not build successfully, see the Troubleshooting section for help on resolving compilation issues.

Related information

[Runtime analysis overview on page 47](#)

Running instrumented applications

To run a program with runtime analysis tools enabled, you must run it as an instrumented application.

About this task

If you run the program with a standard C/C++ run configuration, the program is not instrumented and the runtime analysis tools are not used.

To run an instrumented application:

1. Click **Project > Clean > Clean all projects**.

Result

If the project is successfully built, in the project explorer, the **Binaries** folder contains the compiled instrumented program for the project.

2. Right-click the instrumented program and click **Run As > Run Instrumented Application**.

Results

After running the instrumented application, in the **Project Explorer**, the **Application Result** folder contains the compiled binary executable for the project. To view the results of the run, see [Opening runtime analysis reports on page 926](#).

Estimating Instrumentation Overhead

Instrumentation overhead is the increase in the binary size or the execution time of the instrumented application, which is due to source code insertion (SCI) generated by the Runtime Analysis features.

Source code insertion technology is designed to reduce both types of overhead to a bare minimum. However, this overhead may still impact your application.

The following table provides a quick estimate of the overhead generated by the product.

Code Coverage Overhead

Overhead generated by the Code Coverage feature depends largely on the [coverage types on page 310](#) selected for analysis.

A **48-byte** structure is declared at the beginning of the instrumented file.

Depending on the information mode selected, each covered branch is referenced by an array that uses

- **1 byte** in **Default** mode
- **1 bit** in **Compact** mode
- **4 bytes** in **Hit Count** mode

The actual size of this array may be rounded up by the compiler, especially in **Compact** mode because of the 8-bit minimum integral type found in C and C++.

See [Information Modes on page 309](#) for more information.

Other Specifics:

- **Loops, switch and case statements:** a 1-byte local variable is declared for each instance
- **Modified/multiple conditions:** one n -byte local array is declared at the beginning of the enclosing routine, where n is the number of conditions belonging to a decision in the routine

I/O is either performed at the end of the execution or when the end-user decides (please refer to Coverage Snapshots in the documentation).

As a summary, **Hit Count** mode and modified/multiple conditions produce the greatest data and execution time overhead. In most cases you can select each coverage type independently and use Pass mode by default in order to reduce this overhead. The source code can also be partially instrumented.

Memory and Performance Profiling and Runtime Tracing

Any source file containing an instrumented routine receives a declaration for a 16 byte structure.

Within each instrumented routine, a n byte structure is locally declared, where n is:

- 16 bytes
- +4 bytes for Runtime Tracing
- +4 bytes for Memory Profiling
- +3*t bytes for Performance Profiling, where t is the size of the type returned by the clock-retrieving function

For example, if t is 4 bytes, each instrumented routine is increased of:

- 20 bytes for Memory Profiling only
- 20 bytes for Runtime Tracing only
- 28 bytes for Performance Profiling only
- 36 bytes for all Runtime Analysis features together

Memory Profiling Overhead

This applies to Memory Profiling for C and C++. Memory Profiling for Java does not use source code insertion.

Any call to an allocation function is replaced by a call to the Memory Profiling Library. See the **Target Deployment Guide** for more information.

These calls aim to track allocated blocks of memory. For each memory block, $16+12*n$ bytes are allocated to contain a reference to it, as well as to contain link references and the call stack observed at allocation time. n depends on the Call Stack Size Setting, which is 6 by default.

If [ABWL on page 367](#) errors are to be detected, the size of each tracked, allocated block is increased by $2*s$ bytes where s is the Red Zone Size Setting (16 by default).

If [FFM on page 366](#) or [FMWL on page 368](#) errors are to be detected, a Free Queue is created whose size depends on the Free Queue Length and Free Queue Size Settings. Queue Length is the maximum number of tracked memory blocks in the queue. Queue Size is the maximum number of bytes, which is the sum of the sizes of all tracked blocks in the queue.

Performance Profiling Overhead

For any source file containing at least one observed routine, a 24 byte structure is declared at the beginning of the file.

The size of the global data storing the profiling results of an instrumented routine is $4+3*t$ bytes where t is the size of the type returned by the clock retrieving function. See the **Target Deployment Guide** for more information.

Runtime Tracing Overhead

Implicit default constructors, implicit copy constructors and implicit destructors are explicitly declared in any instrumented classes that permits it. Where C++ rules forbid such explicit declarations, a 4 byte class is declared as an attribute at the end of the class.

Related Topics

[Reducing Instrumentation Overhead on page 54](#) | [Source code instrumentation overview on page 11](#)

Reducing Instrumentation Overhead

HCL OneTest™ Embedded Source Code Insertion (SCI) technology is designed to reduce both performance and memory overhead to a minimum. Nevertheless, for certain cross-platform targets, it may need to be reduced still further. There are three ways to do this.

Limiting Code Coverage Types

When using the Code Coverage feature, procedure input and simple and implicit block code coverage are enabled by default. You can reduce instrumentation overhead by limiting the number of coverage types.

Note The Code Coverage report can only display coverage types among those selected for instrumentation.

Instrumenting Calls (C Language)

When calls are instrumented, any instruction that calls a C user function or library function constitutes a *branch* and thus generates overhead. You can disable call instrumentation on a set of C functions using the Selective Code Coverage Instrumentation Settings.

For example, you can usually exclude calls to standard C library functions such as **printf** or **fopen**.

Optimizing the Information Mode

When using Code Coverage, you can specify the Information Mode which defines how much coverage data is produced, and therefore stored in memory.

Related Topics

[Estimating Instrumentation Overhead on page 51](#) | [Selecting Coverage Types on page 310](#) | [Information Modes on page 309](#)

Code coverage

Code coverage overview

Source code coverage consists of identifying which portions of a program are executed or not during a given test case. Source code coverage is recognized as one of the most effective ways of assessing the efficiency of the test cases applied to a software application.

The code coverage tool can provide the coverage information for the following source code elements:

- Statement blocks, decisions, and loops.
- Function or procedure calls.
- Basic conditions, modified conditions/decisions (MC/DC), multiple condition, and forced condition.
- Procedure entries and exits.
- Terminal or potentially terminal statements
- Statements that are considered non-coverable in C.

See [Coverage levels on page 55](#) for more details about each coverage level.

Information modes

The information mode is the method used to code the trace output. This has a direct impact of the size of the trace file as well as on CPU overhead. You can change the information mode in the coverage type settings. See [Changing a code coverage settings on page 63](#).

There are three information modes:

- Default mode: Each branch generates one byte of memory. This offers the best compromise between code size and speed overhead.
- Compact mode: This is functionally equivalent to Pass mode, except that each branch needs only one bit of storage instead of one byte. This implies a smaller requirement for data storage in memory, but produces a noticeable increase in code size (shift/bits masks) and execution time.
- Hit Count mode: In this mode, instead of storing a Boolean value indicating coverage of the branch, a specific count is maintained of the number of times each branch is executed. This information is displayed in the code coverage report.

Count totals are given for each branch, for all trace files transferred to the report generator as parameters. In the code coverage report, branches that have never been executed are highlighted with an asterisk '*'. The maximum count in the report generator depends on the amount of memory available on the computer running the tests. If this maximum count is reached, the report signals it with a Maximum reached message.



Note: The last bracket (}) in a function after a return statement is always displayed in red in the coverage report, even if the function reports 100% coverage.

On-the-fly display

By default, code coverage generates a report when the execution ends. The *on-the-fly* mode generates code coverage results dynamically during the execution. This is useful for applications that never exit or to interact with the execution during the analysis, for example if you want to stop the code coverage when you reach a specified coverage rate threshold.

Coverage levels

The product provides coverage information for various levels of statements, decisions, loops, calls, conditions.

Block coverage

When running the code coverage feature on C source code, the following coverage types are analyzed.

- Statement blocks (simple blocks): Simple blocks are the main blocks of the C function, introduced by decision statements:
 - THEN and ELSE FOR IF statements
 - FOR, WHILE and DO ... WHILE blocks

- Non-empty blocks introduced by switch case or default statements.
- True and false outcomes of ternary expressions (`<expression> ? <expression> : <expression>`).
- Blocks following a potentially terminal statement.

Each simple block is a branch. Every C function contains at least one simple block corresponding to its main body.

- Decisions (implicit blocks): Implicit blocks are introduced by an `IF` statement without an `ELSE` or a `SWITCH` statement without a `DEFAULT`. Each simple block is a branch. Every C function contains at least one simple block corresponding to its main body.
- Loops (logical blocks): Logical blocks are defined by loop statements `FOR`, `WHILE`, and `DO ... WHILE`.

A typical `FOR` or `WHILE` loop can reach three different conditions:

- The statement block contained within the loop is executed zero times. The output condition is *True* from the start
- The statement block is executed exactly once. The output condition is *False*, then *True* the next time
- The statement block is executed at least twice. The output condition is *False* at least twice, and becomes *True* at the end.

In a `DO ... WHILE` loop, because the output condition is tested after the block has been executed, two further branches are created:

- The statement block is executed exactly once. The output is condition *True* the first time.
- The statement block is executed at least twice. The output condition is *False* at least once, then *True* at the end.

Call coverage

Code coverage provides coverage of function or procedure calls by counting as many branches as it encounters function calls while running the program. This type of coverage ensures that all the call interfaces can be shown to have been exercised for each C function, which may be a pass or failure criterion in software integration test phases.

You can exclude specific C functions whose calls you do not want to measure coverage in the configuration settings of the project. This can be useful for C library functions for example.

Condition coverage

For conditions, the following coverage types are analyzed:

- Basic condition coverage: Conditions are operands of either `||` or `&&` operators wherever they appear in the body of a C function, `IF` statements and ternary expressions, and tests for `FOR`, `WHILE`, and `DO ... WHILE` statements even if these expressions do not contain `||` or `&&` operators.

Two branches are involved in each condition, causing the sub-condition to be *true* or *false*. In a switch statement, one basic condition is associated with every `CASE` and `DEFAULT`, whether implicit or not.

Two branches are enumerated for each condition, and one per `CASE` or `DEFAULT`.

- Modified condition/decision coverage (MC/DC): A modified condition (MC) is defined for each basic condition enclosed in a composition of `||` or `&&` operators, proving that the condition affects the result of the enclosing composition. For example, in a subset of values affected by the other conditions, if the value of this condition changes, the result of the entire expression changes. Because compound conditions list all possible cases, you must find the two cases that can result in changes to the entire expression. The modified condition is covered only if the two compound conditions are covered.

You can associate a modified condition with more than one case. Code Coverage calculates matching cases for each modified condition. The number of modified conditions matches the number of Boolean basic conditions in a composition of `||` and `&&` operators.

- Multiple condition coverage: A multiple (or compound) condition is one of all the available cases for the `||` and `&&` logical operator compositions, whenever it appears in a C function. It is defined by the simultaneous values of the enclosed Boolean basic conditions. Remember that the right operand of a `||` or `&&` operator is not evaluated if the evaluation of the left operand determines the result of the entire expression.

Code Coverage calculates every available case for each composition. The number of enumerated branches is the number of distinct available cases for each composition of the `||` or `&&` operators.

- Forced condition coverage: Forced conditions are multiple conditions in which the instrumentation replaces any occurrence of the `||` or `&&` logical operators in the code, with `|` and `&` binary operators. You can use this coverage type, after evaluating all modified conditions, to make sure that every basic condition has been evaluated. With this forced condition coverage, you can ensure that only the basic condition has changed between two tests.

Function coverage

When analyzing C source code, HCL® OneTest™ Embedded can provide the following function coverage:

- Procedure entries: Inputs identify the C functions that are executed. One branch is defined per C function.
- Procedure exits: These include the standard output (if coverable), and all return instructions, exits, and other terminal instructions that are instrumented, as well as the input. At least two branches are defined per C function.

The input is always enumerated, as is the output if it can be covered. If it cannot, it is preceded by a terminal instruction involving returns or an exit. In addition to the terminal instructions provided in the standard definition file, you can define other terminal instructions using the pragma `attol exit_instr`.

Additional statements

Some statements are identified as terminal, potentially terminal, or non-coverable.

A C statement is *terminal* if it transfers program control out of sequence (`RETURN`, `GOTO`, `BREAK`, `CONTINUE`), or if it stops the execution (`EXIT`). By extension, a decision statement (`IF` or `SWITCH`) is terminal if all branches are terminal; that is if the non-empty `THEN ... ELSE`, `CASE`, and `DEFAULT` blocks all contain terminal statements. An `IF` statement without an `ELSE` and a `SWITCH` statement without a `DEFAULT` are never terminal, because their empty blocks necessarily continue the program sequence.

The following decision statements are *potentially terminal* if they contain at least one statement that transfers program control out of their sequence (`RETURN`, `GOTO`, `BREAK`, `CONTINUE`), or that stops the execution (`EXIT`):

- `IF` without an `ELSE`
- `SWITCH`
- `FOR`
- `WHILE` OR `DO ... WHILE`

Some C statements are considered *non-coverable* if they follow either a terminal instruction, a `CONTINUE`, or a `BREAK`, and are not a `GOTO` label. Code coverage detects non-coverable statements during instrumentation and produces a warning message that specifies the source file and line location of each non-coverable statement.



Note: User functions whose purpose is to terminate execution unconditionally are not evaluated. Furthermore, code coverage does not statically analyze exit conditions for loops to check whether they are infinite. As a result, `FOR ... WHILE` and `DO ... WHILE` loops are always assumed to be non-terminal, able to resume program control in sequence.

Justification of non-covered lines of code

You can enter justification statements in uncovered branches of a program so that they are considered as exceptions to the coverage rules. Thus, you identify in the source code the branches that are not covered and explain why they are not covered. The justification text must be declared in the `attol cov_justify` pragma line of the uncovered branch with one or multiple attributes.



Note: This feature applies to C and C++ programming languages only.

SYNTAX:

The justification pragma syntax is the following one:

```
#pragma attol cov_justify (<lineOffset>, <type>, <what>, <justification text> ) [ (...) [ (...) ...]]
```

`#pragma attol cov_justify` is the pragma, and `<lineOffset>`, `<type>`, `<what>`, and `<justification text>` are the attributes.

Multiple statements can be specified in the same pragma line, with four attributes for each.

Each justification statement in a pragma line can cover only one branch of the code starting from a specified line of the source code.

Double-quotes can be added if the attribute includes commas.

The attributes are the following ones:

- **<type>** attribute:

This attribute is mandatory. It is used to determine what kind of code must be covered and how it must be covered.

You can use the following **<type>** values:

- **proc**: to justify that a function or a method is not covered.
 - **return**: to justify that a return statement is not covered.
 - **branch** or **block**: to justify that a block of code is not covered.
 - **implicit**: to justify that an implicit else statement is not covered.
 - **logical**, or **for**, or **while**: to justify that a loop is not covered (the loop number is given in the **<what>** attribute).
 - **case**: to justify that a case statement in a switch is not covered.
 - **call**: to justify that a function call or method call is not covered.
 - **cond**: to justify that a simple condition is not covered (the value true or false of the condition is given in the **<what>** attribute).
 - **mcdc**: to justify that a MC/DC is not covered (the description of the MD/DC is given in the **<what>** attribute).
- **<lineOffset>** attribute:

This attribute is optional.

<lineOffset> attribute represents the number of lines between the pragma and the branch that must be justified. The value can be '+' if the branch is located after the pragma, or '-' if the branch is located before the pragma.

If the **<lineOffset>** attribute is omitted **<lineOffset>** is considered as "0". It means that the justification applies to the closest type of branch (attribute **<type>**) from the pragma. If there are multiple branches at a same distance of the pragma, **<lineOffset>** helps distinguish between branches which branch should be justified.

The **<lineOffset>** attribute specifies the line where the branch or condition to justify starts, it is relative to the pragma line (+/-), allowing to write this pragma line anywhere in the source file.



Note: For the **<block>** value, the target block of lines is the block where the pragma is declared.

- **<what>** attribute:

The **<what>** values are used to help specify some of the branches to be justified. It depends on the attribute **<type>** values being used:

- For **block** or **branch** <type>: The value is a string that describes the logical position of the block in the function, like '/then/else/seq'.
 - For **logical**, **for** and **while** <type>: The value is a list of '0' (the loop is not executed), '1' (the loop is executed only once) or '2+' (the loop is executed more that once) separated by ','; each of them could be prefixed with the block description string.
 - For **cond** <type>: The value is <expression>:<value>, <value> is *true* or *false* and <expression>.
 - For **mcdc** <type>: The value is a list of impossible combinations of the conditions separated by ','; each value of the conditions are set with 'T' for true, 'F' for false, or 'X' if the condition is not evaluated. For example, if the MC/DC consists of 3 conditions, the <what> value could be the following one: "TFX;FXX".
- <justification text> is the reason why this part of code can't be covered by a test.

<justification> is mandatory. It is presented as a free text in the coverage report that justifies a uncovered branch. It explains why it is not covered.

Example:

```
#pragma attol cov_justify (call, "my justification") (block, "myjustification") (cond, ":true",
"my justification") (for, "0;1" ,"my justification")
```

The following table lists the parameters that can be entered in the "**<lineOffset>**" and "**<what>**" attributes depending on the parameters indicated in the "**<type>**" attribute.

<type> attribute	<lineOffset> attribute	<what> attribute
proc	For the <type>= proc , the pragma line declared above or inside the body, or just after the end of the body justifies the function/procedure entry.	
return	For <type>= return , the pragma line, must be just before or just after the return line.	
branch/block	For <type>= branch or block . It starts on the first { of that block or on the line of the unique statement.	<p>The "branch to cover" attribute is used for a branch=<type> The branch string format is a list of :</p> <p>/then/else/seq OR /</p> <p>It is used to indicate which branch to cover when there are multiple branches on the same line.</p>

<type> attribute	<lineOffset> attribute	<what> attribute
		It can be empty is there are no ambiguity with the line number.
implicit	For <type>=implicit , the pragma line must be just before the decision, or at the else place.	<p>The "branch to cover" attribute is used for <type>= <branch>.</p> <p>The branch string format is a list of: / then/else/seq OR /</p> <p>It is used to indicate which branch to cover when there are multiple branches on the same line.</p> <p>It can be empty is there are no ambiguity with the line number.</p>
logical/for/do/while	For <type>=logical/for/while , the pragma line must be just before the 'for' or 'while', or 'do' keyword.	<p>This attribute is used for <type>=logical, the branch string format looks like 'branch depth/instruction type/value' with :</p> <p>instruction type is:</p> <p>/for OR /while OR /do</p> <p>The value is /0 or /1 or /2+ to specify which part must be covered and justified. It is mandatory and can specify multiple parts if separated by ; e.g. "/0 ; /1"</p> <p>Branch depth is a suite of strings like /then /else /for /while etc. clarifying the code depth of the branch where the loop has been found.</p> <p>e.g. "/else/then/for/while/1" or "/while/1" or "/1"</p>
case	For <type>=case , the pragma line must be just before or just after the case line.	Used for <type>=case , the string details the case expression (between 'case' and the ':') to cover into the switch block.

<type> attribute	<lineOffset> attribute	<what> attribute
	A case is both a block and a condition. If you enter a justification for a case <type>, it is the condition that is justified. You need to declare another pragma to justify the corresponding block.	It can be empty if there are no ambiguities with the line number.
call	For <type>= call , the pragma line must be just before or just after the call line,	Used for type=call, the string details the called method name to cover. It can be empty if there are no ambiguities with the line number.
cond	For <type>= cond , the pragma line must be just before or just after the condition line.	"condition expression:value" is used for "<type>= cond ". The string indicates the condition expression to cover into the decision with the value to cover, true or false. The "condition expression" can be empty if there is no ambiguity with the line number but the value must always be specified after a colon at the end of the string. Example: " <code>var>5:false</code> " or " <code>:true</code> ". Multiple values can be justified, separated by ';' such as " <code>:true ; :false</code> "
mcddc	For <type>= mcddc , the pragma line must be just before or after the first condition line, or just before or after the last condition line.	"combinations" is used for type=mcddc . It is a series of patterns separated by a semi-colon ";". It cannot be empty.

Once the source code is built, you can see the results of the non-coverage justification statements in the Code Coverage report, on the Source page.

For more information about the code coverage reports, see [About coverage reports on page 928](#).

Changing code coverage settings

You can edit the configuration settings for code coverage to explicitly include or exclude files, change the information mode, coverage level, and other settings.

To change the code coverage settings:

1. In the project explorer, right-click the project on which you want to change the settings and click **Properties**.
2. Click **C/C++ Build > Settings** and select **Build Settings**.
3. Expand **Code coverage** to access the settings and set the required coverage level for functions, calls, blocks and conditions, as well as any other required settings.

Instrumentation control

You can use the coverage type settings to declare various types of coverage. See [Coverage levels on page 55](#) for more information about these settings.

Coverage level functions

Select between function **Entries**, **With exits**, or **None**.

Coverage level calls

Select **Yes** or **No** to toggle call code coverage.

Coverage level blocks

Select the desired block code coverage type. You can combine, enable, or disable any of these coverage types before running the application node. All coverage types selected for instrumentation can be filtered out in the coverage viewer.

Exclude for loops

Select **Yes** to exclude for loops from instrumentation. Only *while* and *do* loops are instrumented.

Coverage level conditions

Selects the condition level of code coverage to be included in the report:

- **None**: The coverage report ignores conditions.
- **Basic**: Only basic conditions are included in the coverage report.
- **Modified (MC/DC)**: Only modified conditions are included in the coverage section of the test report.
- **Modified and Multiple**: Both modified and multiple conditions are included in the coverage report.
- **Forced Modified (MC/DC)**: The report includes modified conditions where all operators are replaced with bitwise operators.
- **Forced Modified and Multiple**: The report includes modified and multiple conditions where all operators are replaced with bitwise operators.

Condition in expressions

Select **Yes** to consider relational operators in an expression (for example: $y = (a > 0)$) as conditions.

Bitwise as logical

Select **Yes** to instrument bitwise operators as logical when both operands are booleans.

Ternary coverage

When this option is selected, code coverage reports ternary expressions as statement blocks.

Information mode

This setting specifies the information modes to be used by code coverage.

- **Default (Optimized for Code Size and Speed)**: This setting uses one byte per branch to indicate branch coverage.
- **Compact (Optimized for Memory)**: This setting uses one bit per branch. This method saves target memory but uses more CPU time.
- **Report Hit Count**: This adds information about the number of times each branch was executed. This method uses one integer per branch.

Excluded function calls

Specifies a list of functions to be excluded from the call coverage instrumentation type, such as *printf* or *fopen*. Use the **Add**, **Remove** buttons specify the functions to be excluded.

Not returning functions

Type the identifiers (not signatures) of the functions that do not return (functions that execute a *longjmp* or *exit*).

Advanced options

Trace file name (.tio)

this allows you to specify a path and filename for the `.tio` dynamic coverage trace file.

Key ignore source file path

Identifies source files based only on the filename instead of the complete path. Use this option to consolidate test results when a same file can be located in different paths. This can be useful in some multi-user environments that use source control. If you use this option, make sure that the source file names used by your application are unique.

User comment

This adds a comment to the code coverage report. This can be useful for identifying reports produced under different configurations. To view the comment, click the a magnifying glass symbol that is displayed at the top of your source code in the coverage viewer.

Report summary

Select **Yes** to add the coverage summary to the summary text file of the selected node.

On-the-fly frequency dump

Specify the function call number after which the coverage results are updated dynamically during execution. 0 means no update during execution.

4. Click **OK, Apply** the changes and close the **Properties** window.

Related information

[Coverage levels on page 55](#)

[Engaging runtime analysis tools on page 50](#)

Code coverage for assembler source files

From HCL OneTest™ Embedded V8.3.0, you can collect coverage metrics for assembler source files. Coverage information is displayed in the coverage report.

**Note:**

This feature supports only ARM in 32 bits mode. It is an extension of the C language mode.

You must have configured HCL OneTest™ Embedded to recognize the .asm file extension used for assembler files.

Coverage Assembler language for ARM processor is fully compatible with C/C++ and Ada code coverage.

Assembler source files are taken into account in the build as C/C++ source files. Optionally, C/C++ source files could be instrumented at the assembler level instead of at the C/C++ level.

The supported coverage levels are:

- Functions
- Functions and exits



- Statement blocks
- Calls

CONFIGURATION

Code coverage for assembler source files requires the use of an appropriate TDP. You can use the `clinCrossRaspiRemote.xdp` and `cwinCrossRaspiRemote.xdp` that are delivered, for example.

There two use case scenarios:

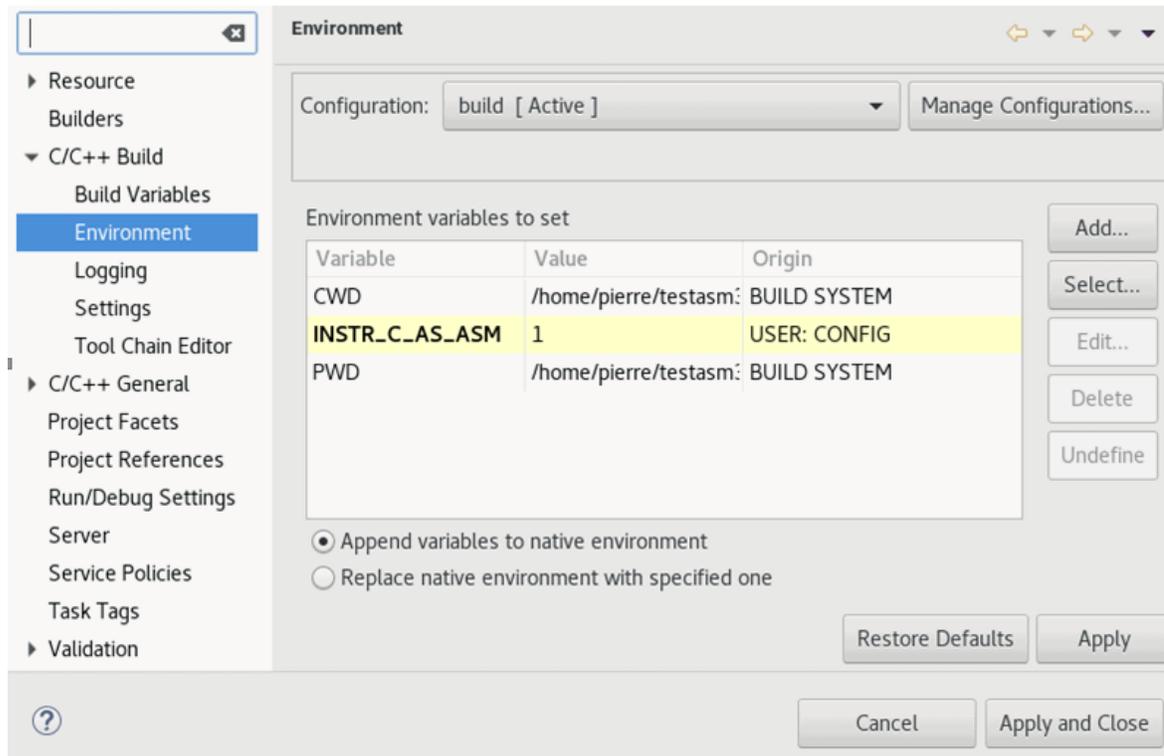
- For a project that uses both C and assembler source files, you only have to add the assembler `.asm` source files to the list of sources to be compiled. The `.asm` files are then instrumented, built, and linked with the other C sources to produce an executable file.
- For C code source files that are instrumented in assembler mode, the C source files are converted into assembler files by using the `gcc -S` command. Then, they are instrumented in assembler mode, they are converted into assembly language, and linked.

To implement this use case scenario, you must set the `INSTR_C_AS_ASM=1` environment variable.

To add this environment variable in Studio, proceed as follows:

- Right-click on your project in the **Project Explorer**, and select **Properties**.
- Select **C/C++ Build > Environment**.
- Click **Add**, give a name to the variable, and enter `INSTR_C_AS_ASM=1` in **Value**.
- Apply and close the window.

See the following example:



Note: In some case, when the assembler code increased due to the code coverage level, it might be necessary to re-organize the assembler code (Example: you can move data pools), or to decrease the level of code coverage (Example: you can delete code coverage for some functions calls in libraries).

Using the Code Coverage Viewer to view reports

Code Coverage for Ada, C and C++

The Code Coverage Viewer allows you to view code coverage reports generated by the Code Coverage feature. Select a tab at the top of the Code Coverage Viewer window to select the type of report:

- A [Source Report](#) that displays the source code under analysis, highlighted with the actual coverage information.
- A [Rates Report](#) that provides detailed coverage rates for each activated coverage type.

You can use the **Report Explorer** to navigate through the report. Click a source code component in the Report Explorer to go to the corresponding line in the Report Viewer.

You can jump directly to the next or previous Failed test in the report by using the **Next Failed Test** or **Previous Failed Test** buttons from the Code Coverage toolbar.

You can jump directly to the next or previous Uncovered line in the Source report by using the **Next Uncovered Line** or **Previous Uncovered Line** buttons in the Code Coverage feature bar.

When viewing a Source coverage report, the Code Coverage Viewer provides several additional viewing features for refined code coverage analysis.

To open a Code Coverage report:

1. Right-click a previously executed test or application node.
2. If a Code Coverage report was generated during execution of the node, select **View Report** and then **Code Coverage**.

Coverage types

Depending on the language selected, the Code Coverage feature offers the following choices:

- **Function or Method code coverage:** select between function **Entries**, **Entries and exits**, or **None**.
- **Call code coverage:** select **Yes** or **No** to toggle call coverage for Ada and C.
- **Block code coverage:** select the block coverage method you need.
- **Condition code coverage:** select condition coverage for Ada and C.

Any of the Code Coverage types selected for instrumentation can be filtered out in the Code Coverage report stage if necessary.

To filter coverage types from the report, proceed as follows:

1. From the **Code Coverage** menu, select **Code Coverage Type**.
2. Toggle each coverage type in the menu.

For example, to filter out multiple conditions (MC) from the report, select **Code Coverage > > Code Coverage Type**, and clear **Multiple conditions**.

3. Alternatively, you can filter out coverage types from the Code Coverage toolbar by toggling the Code Coverage type filter buttons.

Test by test analysis mode

The *test by test* analysis mode allows you to refine the coverage analysis by individually selecting the various tests that were generated during executions of the test or application node. In **Test-by-Test** mode, a **Tests** node is available in the **Report Explorer**.

When *test by test* analysis is disabled, the **Code Coverage Viewer** displays all traces as one global test.

To toggle *test by test* mode, proceed as follows:

1. In the **Code Coverage Viewer** window, select the **Source** tab.
2. From the **Code Coverage** menu, select **Test-by-Test**.

To select the Tests to display in Test-by-Test mode, follow these steps:

1. Expand the Tests node at the top of the **Report Explorer**.
2. Select one or several tests. The **Coverage Viewer** provides code coverage information for the selected tests.

Opening the HTML report

You can open the code coverage report in an HTML format if it is available.

To open the HTML report, select **Open HTML report** in the **Coverage Viewer** menu.

The HTML coverage report has been generated in the build folder with the extension `.cov.html`. The root name of the report is the name of the application, or the name of the test harness, or the name of the test suite result.



Note: The HTML report is created from a template file: `covreport.template` located in the folder `<installation folder>/lib/report`. You can modify this template. This is the default template that can be used as a string point for a new template.

To select a new template file, select **Code coverage > Advanced options** in the build settings. Then, click **Coverage Report Template** to select the new template file.

Memory profiling

Memory profiling overview

Memory profiling helps you detect memory allocation problems and leaks.

Runtime memory errors and leaks are among the most difficult errors to locate and the most important to correct. The symptoms of incorrect memory use are unpredictable and typically appear far from the cause of the error. The errors often remain undetected until triggered by a random event, so that a program can seem to work correctly when in fact it's only working by accident.

After execution of an instrumented application, the memory profiling report provides a summary diagram and a detailed report for both byte and memory block usage. The summary diagrams give you a quick overview of memory usage in blocks and bytes, including:

- The total memory allocated during the execution of the application.
- The memory that remains allocated after the application was terminated.
- The maximum memory usage encountered during execution

The detailed section of the report lists memory usage events identified as errors or warnings.

Related reference

[Memory profiling warnings on page 960](#)

[Memory profiling errors on page 958](#)

Checking for ABWL and FMWL errors

You can insert pragma macros into your source code to check for Late Detect Array Bounds Write (ABWL) and Late Detect Free Memory Write (FMWL)

About this task

By default, memory profiling checks for ABWL and FMWL errors whenever the routines are called or whenever the free queue is actually flushed. In some cases, you might want to manually specify when to check for ABWL and FMWL errors, and on which functions. You can use the ABWL and FMWL check frequency setting to perform a check at the following times:

- Each time the memory is dumped (by default).
- Each time a manual check macro is encountered in the code.
- Each function return.

The checks can be performed either on all memory blocks or only a selection of memory blocks.

1. To indicate where you want an ABWL or FMWL check to occur in your source code, insert an `_ATP_CHECK()` macro at the corresponding location.

Use the following syntax for the pragma macro:

```
#pragma attol insert _ATP_CHECK(@RELFLINE)
```

Each time this macro is encountered during execution, memory profiling checks for ABWL and FMWL errors on the specified blocks. The `@RELFLINE` parameter allows navigation from the memory profiling viewer to the corresponding line in the source code.

2. To create a selection of blocks that you explicitly want to verify, create a list in your source code using the `_ATP_TRACK()` macro variable. The syntax for this macro is:

Use the following syntax for the pragma macro:

```
#pragma attol insert _ATP_TRACK(<pointer>)
```

Related reference

[Memory profiling warnings on page 960](#)

[Memory profiling errors on page 958](#)

Related information

[Memory profiling overview on page 69](#)

[Engaging runtime analysis tools on page 50](#)

Memory profiling user heap

Memory profiling can use heap management routines on target hardware platforms where there are no or only partial provisions for memory management instructions.

When using Memory profiling on some embedded or real-time target platforms, you might encounter one of the following situations:

- Situation 1: There are no provisions for *malloc*, *calloc*, *realloc* or *free* functions on the target platform. The program uses custom heap management routines that may use a user API. Such routines could, for example, be based on a static buffer that performs memory allocation and free functions. In this case, you need to customize the memory heap parameters *RTRT_DO_MALLOC* and *RTRT_DO_FREE* in the target deployment port (TDP) to use the custom *malloc* and *free* functions. In this case, you can access the custom API functions.
- Situation 2: There are partial implementations of *malloc*, *calloc*, *realloc* or *free* functions on the target platform, but other functions provide methods of allocating or freeing heap memory. In this case, you do not have access to any custom API. This requires customization of the TDP. Refer to the documentation provided in the target deployment port editor for customization options.

In both of the above situations, memory profiling can use the heap management routines to detect memory leaks, array bounds and other memory-related defects.



Note: Application pointers and block sizes can be modified by memory profiling in order to detect Late Detect Array Bounds Write (ABWL) errors. *Actual-pointer* and *actual size* refer to the memory data handled by memory profiling, whereas *user pointer* and *user size* refer to the memory handled natively by the application-under-analysis. This distinction is important for the memory profiling ABWL and *red zone* settings.

Target deployment port API

The TDP library provides the following API for memory profiling:

```
void * _PurifyLTHeapAction ( _PurifyLT_API_ACTION, void *, RTRT_U_INT32, RTRT_U_INT8 );
```

In the function `_PurifyLTHeapAction`, the first parameter is the type of action that will be or has been performed on the memory block pointed by the second parameter. The following actions can be used:

```
typedef enum {
    _PurifyLT_API_ALLOC,
    _PurifyLT_API_BEFORE_REALLOC,
    _PurifyLT_API_FREE
} _PurifyLT_API_ACTION;
```

The third parameter is the size of the block. The fourth parameter is either of the following constants:

```
#define _PurifyLT_NO_DELAYED_FREE    0
#define _PurifyLT_DELAYED_FREE     1
```

If an allocation or free instruction has a size of 0, this fourth parameter indicates a delayed free in order to detect Late Detect Free Memory Write (FWML) and Freeing Freed Memory (FFM) errors. See the [Build configuration settings on page 947](#) section for the memory profiling settings.

A freed delay can only be performed if the block can be freed with `RTRT_DO_FREE` (for the situation 1 described previously) or ANSI C `free` (for situation 2). For example, if a function requires more parameters than the pointer to deallocate, then the FWML and FFM error detection cannot be supported and FFM errors will be indicated by a Freeing Unallocated Memory (FUM) error instead.

The following function returns the size of an allocated block, or 0 if the block was not declared to Memory Profiling. This allows you to implement a library function similar to the `msize` from Microsoft™ Visual 6.0.

```
RTRT_SIZE_T _PurifyLTHeapPtrSize ( void * );
```

The following function returns the actual-size of a memory block, depending on the size requested. Call this function before the actual allocation to find out the quantity of memory that is available for the block and the contiguous red zones that are to be monitored by memory profiling.

```
RTRT_SIZE_T _PurifyLTHeapActualSize ( RTRT_SIZE_T );
```

Example

Example

In the following examples, `my_malloc`, `my_realloc`, `my_free` and `my_msize` demonstrate the four supported memory heap behaviors. The following routine declares an allocation:

```
void *my_malloc ( int partId, size_t size )
{
    void *ret;
    size_t actual_size = _PurifyLTHeapActualSize(size);
    /* Here is any user code making ret a pointer to a heap or
       simulated heap memory block of actual_size bytes */
    ...
    /* Then comes the memory profiling action */
    return _PurifyLTHeapAction ( _PurifyLT_API_ALLOC, ret, size, 0 );
    /* The user-pointer is returned */
}
```

In situation 2, where you have access to a custom memory heap API, replace the "..." with the actual malloc API function.

For a `my_calloc(size_t nelem, size_t elsize)`, pass on `nelem*elsize` as the third parameter of the `_PurifyLTHeapAction` function. In this case, you might need to replace this operation with a function that takes into account the alignments of elements.

To declare a reallocation, two operations are required:

```
void *my_realloc ( int partId, void * ptr, size_t size )
{
```

```

void *ret;
size_t actual_size = _PurifyLHeapActualSize(size);
/* Before comes first Memory Profiling action */
ret = _PurifyLHeapAction ( _PurifyLT_API_BEFORE_REALLOC, ptr, size, 0 );
/* ret now contains the actual-pointer */
/* Here is any user code making ret a reallocated pointer to a heap or
   simulated heap memory block of actual_size bytes */
...
/* After comes second Memory Profiling action */
return _PurifyLHeapAction ( _PurifyLT_API_ALLOC, ret, size, 0 );
/* The user-pointer is returned */
}

```

To free memory without using the delay:

```

void my_free ( int partId, void * ptr )
{
  /* Memory Profiling action comes first */
  void *ret = _PurifyLHeapAction ( _PurifyLT_API_FREE, ptr, 0, 0 );
  /* Any code insuring actual deallocation of ret */
}

```

To free memory using a delay:

```

void my_free ( int partId, void * ptr )
{
  /* Memory Profiling action comes first */
  void *ret = _PurifyLHeapAction ( _PurifyLT_API_FREE, ptr, 0, 1 );
  /* Nothing to do here */
}

```

To obtain the user size of a block:

```

size_t my_msize ( int partId, void * ptr )
{
  return _PurifyLHeapPtrSize ( ptr );
}

```

Use the following macros to save customization time when dealing with functions that have the same prototypes as the standard ANSI functions:

```

#define _PurifyLT_MALLOC_LIKE(func) \
void *RTRT_CONCAT_MACRO(usr_,func) ( RTRT_SIZE_T size ) \
{ \
  void *ret; \
  ret = func ( _PurifyLHeapActualSize ( size ) ); \
  return _PurifyLHeapAction ( _PurifyLT_API_ALLOC, ret, size, 0 ); \
}
#define _PurifyLT_CALLOC_LIKE(func) \
void *RTRT_CONCAT_MACRO(usr_,func) ( RTRT_SIZE_T nelem, RTRT_SIZE_T elsize ) \
{ \
  void *ret; \
  ret = func ( _PurifyLHeapActualSize ( nelem * elsize ) ); \
  return _PurifyLHeapAction ( _PurifyLT_API_ALLOC, ret, nelem * elsize, 0 ); \
}
#define _PurifyLT_REALLOC_LIKE(func,delayed_free) \
void *RTRT_CONCAT_MACRO(usr_,func) ( void *ptr, RTRT_SIZE_T size ) \
{ \

```

```

void *ret; \
ret = func ( _PurifyLHeapAction ( _PurifyLT_API_BEFORE_REALLOC, \
                                ptr, size, delayed_free ), \
            _PurifyLHeapActualSize ( size ) ); \
return _PurifyLHeapAction ( _PurifyLT_API_ALLOC, ret, size, 0 ); \
}
#define _PurifyLT_FREE_LIKE(func,delayed_free) \
void RTRT_CONCAT_MACRO(usr_,func) ( void *ptr ) \
{ \
  if ( delayed_free ) \
  { \
    _PurifyLHeapAction ( _PurifyLT_API_FREE, ptr, 0, delayed_free ); \
  } \
  else \
  { \
    func ( _PurifyLHeapAction ( _PurifyLT_API_FREE, ptr, 0, delayed_free ) ); \
  } \
}

```

Changing memory profiling settings

You can edit the configuration settings for memory profiling to specify the errors and warnings that you want to detect and to set other advanced options.

To change the memory profiling settings:

1. In the project explorer, right-click the project on which you want to change the settings and click **Properties**.
2. Click **C/C++ Build > Settings** and select **Build Settings**.
3. Expand **Memory profiling** to access the settings and set the error and warning detection options as well as any other required options.

The following settings are available:

Instrumentation control

You can specify the type of memory errors and warnings that you want to detect. See [Memory profiling errors on page 958](#) and [Memory profiling warnings on page 960](#) for more information about these settings.

Detect File in Use (FIU)

When the application exits, this option reports any files left open.

Detect Memory in use (MIU)

When the application exits, this option reports allocated memory that is still referenced.

Free Invalid Memory (FIM)

This option activates the detection of invalid free memory instructions.

Detect Signal (SIG)

This option indicates the signal number received by the application forcing it to exit.

Detect Freeing Freed Memory (FFM) and Detect Free Memory Write (FMWL)

Select **Yes** to activate detection of these errors.

Free queue length (blocks)

Specifies the number of memory blocks that are kept free.

Free queue size (bytes)

Specifies the total buffer size for free queue blocks.

Largest free queue block size (bytes)

Specifies the size of the largest block to be kept in the free queue.

Detect Array Bounds Write (ABWL)

Select **Yes** to activate detection of ABWL errors.

Red zone length (bytes)

Specifies the number of bytes added by Memory Profiling around the memory range for bounds detection.

Number of functions in call stack

Specifies the maximum number of functions reported from the end of the CPU call stack. The default value is 6.

Only show memory leaks with call stack

Select this option to only record memory leaks that are associated with a call stack. Memory allocations that occurred before the application started do not have a call stack and are not included in the Memory Profiling report.

Line number link

Select **Statement** to link the line number in the report to the corresponding allocation or free statement in the function. Select **Function** to link only to the function entry and to improve performance.

Only show new memory leaks in each dump

In multi-dump report, Memory leaks (MLK) and potential leaks (MPK) are only reported once.

Advanced options**Trace File Name (.tpf)**

This setting allows you to specify a filename for the generated `.tpf` trace file.

Exclude block tracking before init

Disables memory profiling for any memory blocks allocated before the first execution of instrumented code. Use this option to prevent crashes when the system uses memory allocations that cannot be tracked.

Excluded global variables

Specifies a list of global variables that are not to be inspected for memory leaks. This option can be useful to save time and instrumentation overhead on trusted code. Use the **Add** and **Remove** buttons to add and remove global variables.

Exclude variables from directories

Specifies a list of directories from which any variables found in files are not to be inspected for memory leaks.

Break on error

Use this option to break the execution when an error is encountered. The break point must be set to *priv_check_failed* in debug mode.

ABWL and FMWL check frequency

Use this to check for ABWL and FMWL errors:

- Each time the memory is dumped (by default).
- Each time a manual check macro is encountered in the code.
- Each function return.

These checks can be performed either on all memory blocks or only a selection of memory blocks. See [Checking for ABWL and FMWL errors on page 70](#) for more information.

Preserve block content

Set this setting to **Yes** to preserve the content of memory blocks freed by the application. Use this setting to avoid application crashes when memory profiling is engaged. When this setting is enable, reads to freed blocks of memory are no longer detected.

4. Click **OK, Apply** the changes and close the **Properties** window.

Related reference

[Memory profiling errors on page 958](#)

[Memory profiling warnings on page 960](#)

Related information

[Memory profiling overview on page 69](#)

[Memory profiling user heap on page 71](#)

[Enable runtime analysis tools on page 50](#)

Performance profiling

Performance profiling overview

The performance profiling tool provides performance data for each software component so that you can locate the performance bottlenecks.

With performance profiling, you can concentrate your optimization efforts on those portions of code, which can lead to significant improvements in performance.

The performance profiling report provides function profiling data for your program and its components so that you can see exactly where your program spends most of its time. A **Top Functions** graph provides a high level view of the largest time consuming functions in your application. The **Performance Summary** section of the report indicates, for each instrumented function, procedure, or method (collectively referred to as functions), the number of calls and the time spent in the function and in its descendants. For C language, it also provide an estimation of WCET. Optionally, it includes the WCET calculation (Worst Case Execution Time) results.

Related information

[About performance profiling reports on page 935](#)

Performance profiling settings

You can configure the performance profiling settings before running your application in HCL OneTest™ Embedded for Eclipse IDE.

Build settings

All the following options must be set from the **Build settings** tab in the **Properties** window. To open this tab:

- In the **Project Explorer**, right-click on the project and click **Properties**.
- In the **Properties** window, click **C C++ Build > Settings**.

Enable the Performance Profiling

- In the **Build Settings** tab, click **Settings > General > Selective instrumentation**.
- In the right pane, click the **Value** field in **Build options** and click ... to open the **Build options** window.
- In the Build options list, click **Performance Profiling** to enable the feature.

Generate a trace file

- In the **Build Settings** tab, click **Settings > Performance profiling**.
- In **Trace file name (.tqf)**, click ... to open the editor window and specify a filename for the generated .tqf trace file for performance profiling.

To get an evaluation of the Worst Case Execution Time in the report, you must set the WCET option.

Select the Worst Case Execution Time and/or the maximum execution time for each function and descendants:

- In the **Build Settings** tab, click **Settings > Performance profiling**.
- In the right pane, click **Compute F max and F+D max time** and select a value depending on the execution time that you want to be calculated for your project:
 - **No**: Does not calculate the maximum execution time for each function and descendants.
 - **Yes**: Calculate the maximum execution time for each function and its descendants.
 - **Yes + WCET**: Calculate the maximum execution time for each function and descendants, and the Worst Case Execution Time. With this option selected, the report indicates the number of time a function is called.

To get the performance profiling per entry point, you must enter the list of entry point threads of your application.

Entry points

To get the performance profiling per entry point, you must enter the list of entry points for each thread of your application.

- In the **Build Settings** tab, click **General > Multi-thread options**.
- Click ... to open the editor and enter the list of entry points for each thread of your application .
Use commas to separate the thread names.

Then, run the application and see the Performance report.

Performance Profiling Results

The Performance Profiling report provides function profiling data for your program and its components so that you can see exactly where your program spends most of its time. When the configuration settings are set and the test application is run, you can see the Performance Profiling report.

The default Performance report is in HTML format. It is generated from a template named **wcetreport.template** provided as text file that you can modify to customize the report. It uses four online JavaScript libraries:

- Bootstrap,
- JQuery,
- Font Awesome,
- VisJS.

These libraries are not provided. You need an internet connectivity when you open the report. If not, download the libraries (.css and .js files), copy them in the same folder than your report, and modify the template file as follows:

Replace the following lines:

```
<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css" integrity="sha384-MCw98/SFnGE8fJT3GXwEOngsV7Zt27NXFoaAPmYm81iuXoPkFOJwJ8ERdknLPMO" crossorigin="anonymous">
```

```

<link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.5.0/css/all.css"
  integrity="sha384-B4dIYHKNBt8Bc12p+WXckhzcICo0wtJAoU8YZTY5qE0Id1GSseTk6S+L3BlXeVIU"
  crossorigin="anonymous">
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/vis/4.21.0/vis.min.css">
...
<script src="https://code.jquery.com/jquery-3.3.1.slim.min.js"
  integrity="sha384-q8i/X+965Dz00rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8abtTE1Pi6jizo"
  crossorigin="anonymous"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.3/umd/popper.min.js"
  integrity="sha384-ZMP7rVo3mIykV+2+9J3UJ46jBk0WLaUAdn689aCwoqbBJiSnjAK/l8WvCWPIpM49"
  crossorigin="anonymous"></script>
<script src="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/js/bootstrap.min.js"
  integrity="sha384-ChfqquZUCnJSK3+MXmPNIyE6ZbWh2IMqE241rYiqJxyMiZ60W/JmZQ5stwEULTy"
  crossorigin="anonymous"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/vis/4.21.0/vis.js"></script>

```

With the following ones:

```

<link rel="stylesheet" href="./bootstrap.min.css">
<link rel="stylesheet" href="./all.css">
<link rel="stylesheet" href="./vis.min.css">
...
<script src="./jquery-3.3.1.slim.min.js"></script>
<script src="./popper.min.js"></script>
<script src="./bootstrap.min.js"></script>
<script src="./vis.js"></script>

```

The Performance profiling report is made of Summary, Functions and the Call Graph parts.

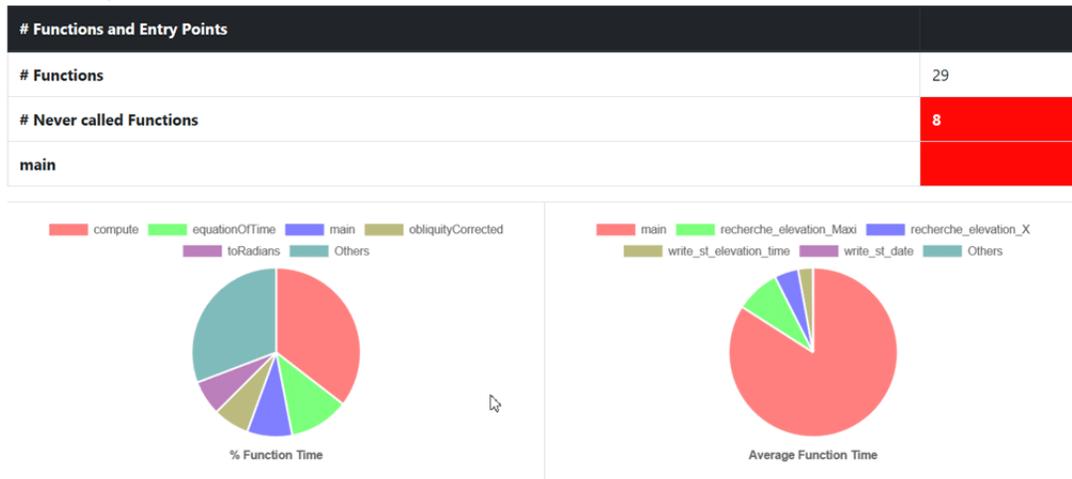
SUMMARY

Summary table

The Summary table displays the total number of functions and the number of functions that have never been executed and for which we have no data. If the instrumentation has been done with the WCET option (Worst Case Execution Time), then the table contains the list of the entry points with an evaluation of the WCET for each of them. This information can be empty (and the cell is red) if the WCET could not be computed. This can occur when one of the called functions in the call graph starting with this entry point has never been executed.

The WCET is given for each entry point if you have entered the list of entry point of your application in the Settings. For more details, see [Performance profiling settings on page 77](#).

Summary



Function time graphs

The Summary is followed by two graphs that provide a high level view of the largest time consumers detected by Performance Profiling in your application.

- **% Function Time:** It gives the five top functions with the greatest percentage of Function Time.
- **Average Function Time:** It gives the five top functions with the greatest Average Function Time.

FUNCTIONS

The Functions section of the report displays a table with the instrumented functions, procedures or methods (collectively referred to as functions) found in the application with the following information:

- **Functions:** Name of the function (in red if the function has never been executed).

If you have selected the WCET option, the chevron in front of the name allows the user to see how many times this function calls other functions. This can help to understand how the WCET is calculated.

- **EP:** Indicate if this function is an entry point or not. You can provide the list of the entry points, or, if not, they are deduced from the call graph (all the functions that are never called).
- **# Calls:** Number of times the function is called. If this value is 0, there is no more information for this function in the table because it has never been executed.
- **Function Time:** Total time spent for executing the function, excluding its descendants.
- **Function + Descendants Time:** Total time spent for executing the function, including its descendants.
- **% Function Time:** Percentage of time spent in this function against the total execution time.
- **% Function + Descendants Time:** Percentage of time spent for executing the function and its descendants against the total execution time.
- **Average Function Time:** Average time spent for executing this function, excluding its descendants.
- **Max Function Time:** Only if you set the option **Compute F max and F + D max**. Indicates the maximum time spent in a call while executing this function, excluding its descendants.

- **Max Function + Descendants Time:** Only if you set the option **Compute F max and F + D max time**, see [Performance profiling settings on page 77](#). This is the maximum time spent in a call while executing this function, including its descendants.
- **WCET:** Only if you set the option WCET, see [Performance profiling settings on page 77](#). It gives an evaluation of the Worst Case Execution Time. This information can be empty if the WCET could not be calculated during the execution. It is the case when one of the function and its descendants has never been executed. Click the chevron icon to deploy the list of functions that are not called.

Functions

Functions	EP	# Calls	Function Time	Function + Desc. Time	% Function Time	% Function + Desc. Time	Average Function Time	Max Function Time ^	Max Function + Desc. Time	WCET
> main	✓	1	13310us	153967us	8.64%	100%	13310us	13310us	153967us	
write_st_elevation_time		1	445us	445us	0.29%	0.29%	445us	445us	445us	445us

Call Graph

The Call Graph part displays all the functions in an interactive call graph that can be moved from left to right or from top to bottom. If the option WCET has been set, a tooltip on each function (node of the graph) gives the WCET. For more information, see [Performance profiling settings on page 77](#).

Customize the Performance Report

You can customize a Performance report.

The Performance report is based on a template called **wcetreport.template** that you can find in the following folder:

- In Windows:

```
<installation_directory>\HCL\HCLOneTest\Embedded\lib\reports
```

- In Unix:

```
<installation_directory>/HCL/HCLOneTest/Embedded/lib/reports
```

Raw data

This template is made of three sections:

- The HTML section that is the common part of all reports,
- A JavaScript section that sets the tables and call graph depending of 2 variables dynamically initialized while the report is creating:

```
var data = {{json}}; // the raw data
```

```
var d = new Date({{date}}) // the date of the generation
```

Raw data is composed of three sections at the top level:

- The list of the modules, each of them has the following information:
 - **Name** is the short name of the C file,
 - **Fullname** is the name and path of the C file,
 - **uuid** is a unique identifier of the module,
 - **unknown** is set to true is the module is not part of the information you provided (there is only one unknown module that gathers all the function calls that are not in the known modules),
 - **functions** is the list of the unique identifiers of functions of the module.

Modules are listed as hashmap with the uuid, as follows:

```
"modules": {
  "f5b5579edeaca82df478a6780c0c4c92": {
    "name": "USAGE.C",
    "fullname": "...",
    "uuid": "f5b5579edeaca82df478a6780c0c4c92",
    "unknown": false,
    "functions": [
      "ba9eb05ad703046fed306b4271b7ead7"
    ]
  },...
```

- The list of functions including following information:
 - **name** is the name of the C function,
 - **line** is the first line of the function in the module,
 - **id** is the number used in **.tsf** file to identify this function,
 - **stacksize** is the stack size computed during the execution if this option has been set (otherwise -1),
 - **uuid** is a unique identifier of the function,
 - **module** is a unique identifier of the module in which the function is declared,
 - **calls** is the list of the calls in this function. Each of them have the following information:
 - **calling_uuid** is the unique identifier of the calling function,
 - **called_uuid** is the unique identifier of the called function,
 - **line** is the line number of the call in the module,
 - **col** is the column number of the call in the module,
 - **same_module** is set to true id the called function is in the same module that the calling function.
 - **level** is a number that represent the level of the function in the call graph, starting to 0.
 - **calledby** is the list of unique identifiers of functions that call this one.
 - **maxLocal** is the maximum time spent in the function, excluding its descendants.
 - **maxTotal** is the maximum time spent in the function, including its descendants.
 - **sumLocal** is the time spent in the function, excluding its descendants.
 - **sumTotal** is the time spent in the function, excluding its descendants.
 - **wcet** is the Worst Case Execution Time of the function (this value is negative if it has not been calculated).

- Functions are listed as hashmap with the uuid, as following:

```

"functions": {
  "ba9eb05ad703046fed306b4271b7ead7": {
    "name": "write_usage",
    "line": 9,
    "id": 1,
    "stacksize": -1,
    "uuid": "ba9eb05ad703046fed306b4271b7ead7",
    "module": "f5b5579edeaca82df478a6780c0c4c92",
    "calls": [
      {
        "calling_uuid": "ba9eb05ad703046fed306b4271b7ead7",
        "called_uuid": "7b6cd643b5b44e1e0510f30f62729eba",
        "line": 10,
        "col": 2,
        "same_module": false
      }
    ],
    "level": 1,
    "calledby": [
      "3fb6b20659c9b70fc6d01ba797abae1f"
    ],
    "maxLocal": 27,
    "maxTotal": 28,
    "sumLocal": 3190,
    "sumTotal": 3853,
    "averageLocal": 0,
    "wcet": 60
  },...

```

- The final section contains the following information:
 - **entrypoints** is the list of entry points of the application; each of them contains:
 - **name** is the name of the entry points.
 - **module** is the uuid of the module where is the entry point.
 - **wcet** is the Worst Case Execution Time of the entry points (this value is negative if it has not been calculated).
 - **timeunit** is the unit of time used in the report (**us** is for micro-second, **ms** for millisecond, **s** for second).
 - **level** is the setting for performance (**0** when there is no "compute F max + D max time", **1** when this option is not set to yes, **2** when it is set to yes + WCET).

An example of this section:

```

"entrypoints": [
  {
    "name": "main",
    "module": "57f1afe89e0a74b786aab75cd448db9b",
    "wcet": -10
  }
],
"timeunit": "us",
"level": 2

```

Runtime tracing

Runtime tracing overview

Runtime Tracing is a feature for monitoring realtime interaction of your code in a dynamic UML sequence diagram. Runtime tracing uses source code instrumentation to generate trace data, which produces a UML sequence diagram.

UML sequence diagrams

The lifeline of an object is represented in the trace viewer as shown below. The instance creation box displays the name of the instance. For more information about UML sequence diagrams, see the [UML sequence diagram reference on page 958](#).

Step-by-step mode

When tracing large applications, it may be useful to slow down the display of the UML sequence diagram. You can do this by enabling the *step-by-step* mode in the trace viewer.

Triggers

Sequence diagram triggers allow you to predefine automatic start and stop parameters for the trace viewer. The trigger capability is useful if you only want to trace a specific portion of an instrumented application. Triggers can be inactive, time-dependent, or event-dependent.

Notes®

You can manually add your own notes inside your source code to make them display in the UML sequence diagram when runtime tracing is enabled. To do this, you must insert the following instrumentation pragma macro, into the C source code of the program:

```
#pragma attol att_insert_ATT_USER_NOTE("Text")
```

Advanced runtime tracing

On some platforms or for some specific applications, these settings might be useful.

Multithreaded programs

Runtime tracing can be configured for use in a multithreaded environment such as Posix, Solaris and Windows™. Multithread mode protects target deployment port global variables against concurrent access. This causes a significant increase in target deployment port size as well as an impact on performance.

To enable multithreaded mode, change the **Maximum number of threads** and **Record and display thread info** configuration settings. See [Changing runtime tracing settings on page 86](#) for more information about these settings.

Partial trace flush

When using this mode, the target deployment port only sends messages related to instance creation and destruction, or user notes. All other events are ignored. This can be useful to reduce the volume of the trace dump file. When partial trace flush mode is enabled, message dump can be toggled on and off during trace execution. The partial trace flush settings are located in the runtime tracing settings.

To do this manually, use the runtime tracing pragma user directives:

- `_ATT_START_DUMP`
- `_ATT_STOP_DUMP`
- `_ATT_TOGGLE_DUMP`
- `_ATT_DUMP_STACK`

For example, add the following line to the source code to toggle the trace dump on or off:

```
#pragma attol insert _ATT_TOGGLE_DUMP
```

Trace item buffer

Buffering allows you to reduce formatting and processing at time-critical steps by instructing the target deployment port to only output trace information when its buffer is full or at explicitly specified points in the program. This can prove useful when using runtime tracing on embedded platforms with limited resources.

A smaller buffer optimizes memory usage on the target platform, whereas a larger buffer improves performance of the real-time trace. The default value is 64.

It can be useful to flush the buffer before entering a time-critical part of the application-under-trace. You can do this by adding the `_ATT_FLUSH_ITEMS` user directive to the source-under-trace. For example:

```
#pragma attol insert _ATT_FLUSH_ITEMS
```

Splitting trace files

During execution, runtime tracing generates a dynamic trace file (`.tdf`). When a large application is instrumented, the size of the trace file can impact the display of the sequence diagram.

Splitting trace files allows you to produce multiple smaller files, resulting in better performance of the sequence diagram viewer and lower memory usage. However, split trace files cannot be used simultaneously with the *on-the-fly* tracing mode.

When displaying split trace files, synchronization elements are added to the UML sequence diagram to ensure that all instance lifelines are synchronized.



Note: The total size of split trace files is slightly larger than the size of a single trace file because each file contains additional context information.

Related reference

[UML sequence diagram reference on page 958](#)

Related information

[Runtime tracing overview on page 84](#)

Changing runtime tracing settings

You can edit the configuration settings for runtime tracing to specify how the trace dumps are performed and how the UML sequence diagram is drawn during or after the execution of the program.

To change the runtime tracing settings:

1. In the project explorer, right-click the project on which you want to change the settings and click **Properties**.
2. Click **C/C++ Build > Settings** and select **Build Settings**.
3. Expand **Runtime tracing** to access the runtime tracing settings and set the required options for dumping trace data and drawing UML sequence diagrams.

Instrumentation control

Runtime Tracing file name (.tdf)

This allows you to force a filename and path for the dynamic `.tdf` file. By default, the `.tdf` carries the name of the application node.

Show data classes

When this option is disabled, structures or classes that do not contain methods are excluded from instrumentation. Disable this option to reduce instrumentation overhead.

Trace control

Split trace file

When you use several runtime analysis tools together, the executable produces a multiplexed trace file, containing the output data for each tool. Use this option to split the generated `at1out.spt` output trace file into multiple files.

Maximum size (Kbytes)

This specifies the maximum size for a split `.tdf` file. When this size is reached, a new split `.tdf` file is created.

File name prefix:

By default, split files are named as `att_<number>.tdf`, where `<number>` is a 4-digit sequence number. This setting allows you to replace the `att_` prefix with the prefix of your choice.

Automatic loop detection

Loop detection simplifies UML sequence diagrams by summarizing repeating traces into a loop symbol. Loops are an extension to the UML sequence diagram standard and are not supported by UML.

Additional options

This setting allows you to add command line options. Normally, this line should be left blank.

Display maximum call level

When selected, the target deployment port records the highest level attained by the call stack during the trace. This information is displayed at the end of the UML sequence diagram in the runtime tracing viewer as **Maximum calling level reached**.

Runtime options**Disable on-the-fly mode**

When selected, this setting stops on-the-fly updating of the dynamic `.tdf` file. This option is primarily for target deployment ports that use *printf* output.

Runtime tracing buffer and Partial Runtime Tracing flush

See [Advanced runtime tracing on page 84](#) for more information about these settings.

Maximum buffer size (events)

The maximum number of events recorded in the buffer before it is flushed.

User signal action

Specify an action to be performed when a user signal is detected:

- **No action:** nothing.
- **Flush call stack:** the call stack is flushed to the trace file.
- **Runtime tracing on/off:** toggles the runtime tracing feature on or off.

Record and display time stamp

This setting adds timestamp information to each element in the UML sequence diagram generated by runtime tracing.

Record and display heap size

This setting enables the heap size bar in the UML sequence diagram generated by runtime tracing.

Record and display thread info

This setting enables the Thread Bar in the UML sequence diagram generated by runtime tracing.

4. Click **OK, Apply** the changes and close the **Properties** window.

Related reference

[Build configuration settings on page 947](#)

Related information

[Runtime tracing overview on page 84](#)

[Advanced runtime tracing on page 84](#)

Static metrics

Static metrics overview

Statistical measurement of source code properties is an extremely important matter when you are planning your test work for a software project.

HCL® OneTest™ Embedded provides a static metrics report, which displays complexity data and statistics for your source code.

Halstead metrics

Halstead complexity measurement was developed to measure a program module's complexity directly from source code, with emphasis on computational complexity. The measures were developed by the late Maurice Halstead as a means of determining a quantitative measure of complexity directly from the operators and operands in the module.

Halstead provides various indicators of the module's complexity

Halstead metrics allow you to evaluate the testing time of any C/C++ source code. These only make sense at the source file level and vary with the following parameters:

Table 1.

Parameter	Meaning
n_1	Number of distinct operators
n_2	Number of distinct operands

Table 1. (continued)

Parameter	Meaning
N_1	Number of operator instances
N_2	Number of operand instances

When a source file is selected in the metrics report, the following results are displayed:

Table 2.

Metric	Meaning	Formula
n	Vocabulary	$n_1 + n_2$
N	Size	$N_1 + N_2$
V	Volume	$N * \log_2 n$
D	Difficulty	$n_1/2 * N_2/n_2$
E	Effort	$V * D$
B	Errors	$V / 3000$
T	Testing time	E / k

In the above formulae, k is the stroud number, which has an arbitrary default value of 18. With experience, you can adjust the stroud number to adapt the calculation of the estimated testing time (T) to your own testing conditions: team background, criticality level, and so on.

When the project is selected, the metrics viewer displays the total testing time for the entire project.

V(g) or cyclomatic number

The V(g) or cyclomatic number is a measure of the complexity of a source code function that is correlated with difficulty in testing. The standard value is between 1 and 10. A value of 1 means the code has no branching. A function's cyclomatic complexity should not exceed 10.

The static metrics report displays the V(g) of a function in the Metrics tab when a source file or function is selected. When the type of the selected node is a source file, the sum of the V(g) of the contained function, the mean, the maximum and the standard deviation are calculated.

At the project level, the same statistical treatment is provided for every function in any source file.

Changing static metric settings

You can edit the configuration settings for static metrics to specify how the source code statistics are generated.

To change the static metric settings:

1. In the project explorer, right-click the project on which you want to change the settings and click **Properties**.
2. Click **C/C++ Build > Settings** and select **Build Settings**.
3. Expand **Static metrics** to access the runtime tracing settings and set the required options for dumping trace data and drawing UML sequence diagrams.

One level metrics

By default, `.met` static metric files are produced for source files as well as all dependency files that are found when parsing the source code. Set to **Yes** to restrict the calculation of static metrics only to the source files displayed in the navigator.

Analyzed directories

This setting allows you to restrict the generation of `.met` metric files only to files which are located in the specified directories.

Generate metrics in source directories

By default, all `.met` files are generated in the project directory, and use the same name as the source file. Select **Yes** on this setting to compute metrics for source files that have the same name but are located in different directories. In this case, each `.met` is generated in the source directory of each file.

Additional options

Use this setting to specify extra command line options. In most cases, this should be empty.

4. Click **OK, Apply** the changes and close the **Properties** window.

Related information

[Static metrics overview on page 88](#)

Code review

Code review overview

Automated source code review is a method of analyzing code against a set of predefined rules to ensure that the source adheres to guidelines and standards that are part of any efficient quality control strategy.

HCL® OneTest™ Embedded provides an automated code review tool, which reports adherence to guidelines for your C source code. Among other guidelines, the code review tool implements rules from the MISRA-C:2004 standard, Guidelines for the use of the C language in critical systems.

Code review is part of the runtime analysis tools and can be engaged during a test run or in the project configuration settings.

After the run, the code review report lists the following elements:

- The **Outline** window displays a list of rules that were not compliant for each source file and function. You can use the elements in this view to navigate through the report.
- A summary provides information about the general configuration, the date and the number of analyzed files. It also lists the number of errors and warnings that were encountered.
- The code review report lists the rules for which errors or warnings were detected. It also provides information about the location of the error. You can click the title to go directly to the corresponding line in the source code.

Related reference

[Code review MISRA 2004 rules on page 95](#)

Related information

[Enable runtime analysis tools on page 50](#)

Configuring code review rules

The code review tool uses a set of predefined rules. You can select the default rule configuration file for the code review tool. MISRA 2004 and MISRA 2012 from HCL OneTest™ Embedded V8.2.0 are the default installed rule configuration files. You can disable or set the severity level to Warning. You can also configure the entry-point option if your application is multi-threaded.

About this task

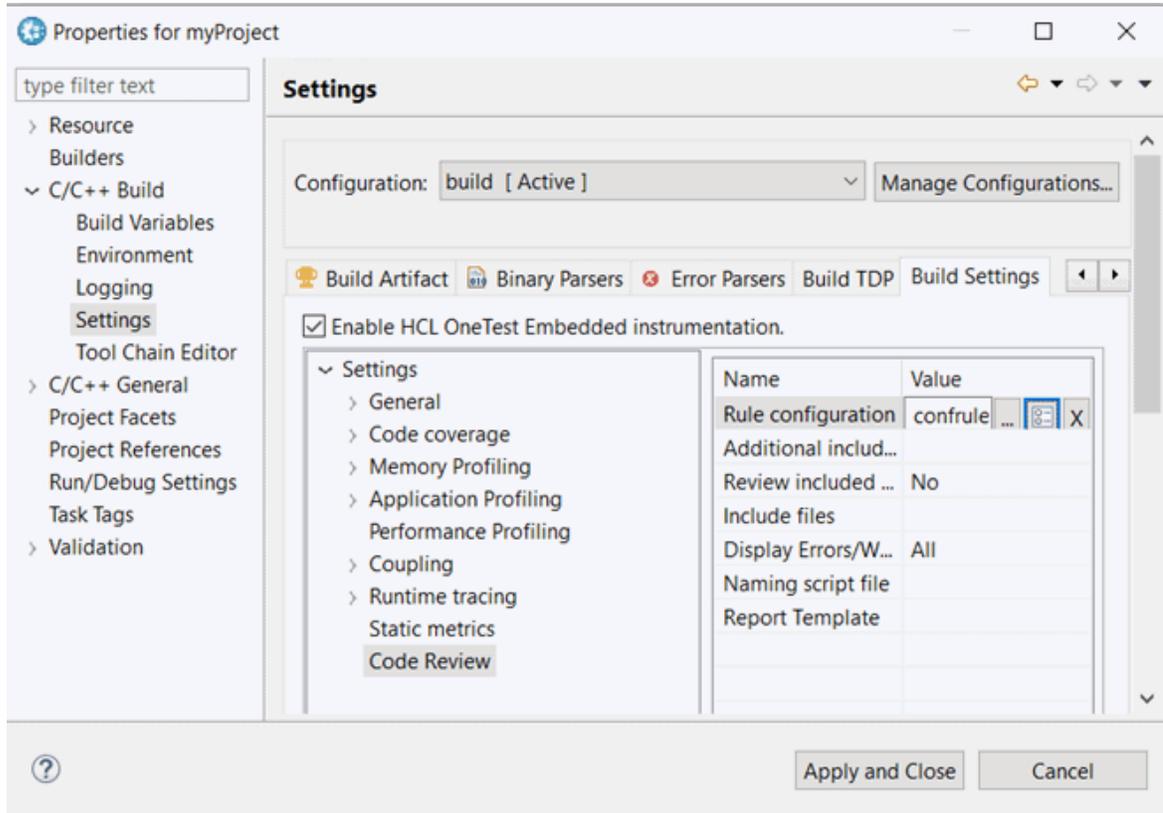
By default all rules are enabled and produce either an error or a warning in the code review report. You can save multiple customized rule policies. The default rule policy files MISRA 2004 and MISRA 2012 are located in: `<installation directory>/plugins/Common/lib/confrule.xml`. Do not modify the default rule configuration files. The only change that can be done in the default rule configuration files is to change or disable the severity level of the rule.



Note: For all new projects, you must select the configuration file that must be used. When you make any changes to the rule policy file, you can save the new policy file in the project.

To select the configuration file and disable or set the severity level of code review rules:

1. In the project explorer, right-click the project on which you want to change the settings and click **Properties**.
2. Expand **C/C++ Build** in the left panel, select **Settings**.
3. In the right panel, in the **Build** tab, expand **Settings** and select **Code Review**.
4. Expand **Code review** to access the code review settings.

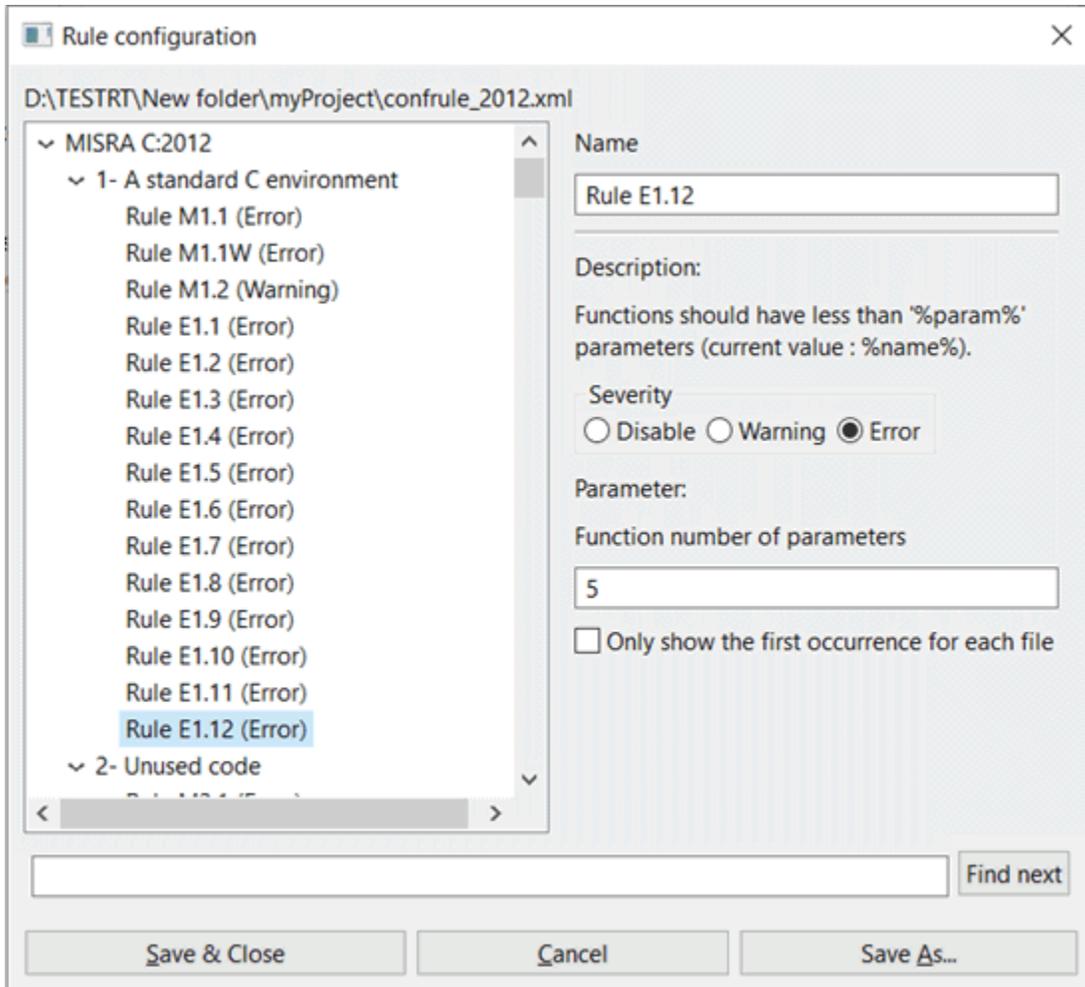


5. Click in the value on the **Rule configuration** row and click ... to select a rule configuration file.
6. Select the default MISRA 2004 or MISRA 2012 rule configuration file that are installed with the product. Alternatively, click + to select a new rule configuration in your browser.
7. Click **OK**.

Result

The path to the selected rule configuration displays in the value column on the Rule Configuration line.

8. On the **Rule configuration** row, click the  to open the **Rule Configuration** window.



9. In the **Rule configuration** window, select the rule that you want to configure.
Rules are grouped into categories. You can filter the rules by labels from the **Find** field. Search is not case sensitive.
When a rule is selected, its description is displayed on the right panel with the parameter description and value if any parameter is available for the selected rule.
10. On the right panel, select the severity level:
 - a. **Disabled**: The selected rule is ignored and is not displayed in the code review report.
 - b. **Warning**: When the rule condition is found, a warning is displayed in the code review report.
 - c. **Error**: When the rule condition is found, an error is displayed in the code review report.
11. Select **Show only the first occurrence** to only show the first occurrence of a rule condition in a file. Any subsequent occurrences of the condition are not reported.



Note: If your application is multi-threaded, you can provide the list of entry points to avoid that the rules about 'non-used functions' are raised.

To configure the **Multi_thread** option, follow these steps:

12. In the **Project Explorer**, right-click the project on which you want to change the settings and click **Properties**.
13. Click **C/C++ Build > Settings** and select **Build Settings**.
14. In the right panel, click **SettingsGeneral > Multi-Thread option**.
15. Click ... in the value field of the **Entry points** option to open the editor.
16. In the editor, enter the list of entry points for each thread and click **OK**.
17. Click **OK, Apply** the changes and close the **Properties** window.



Note: The Entry point option applies to rule E16.50 (MISRA_2004) and M2.2.2 (MISRA 2012).

Related reference

[Code review MISRA 2004 rules on page 95](#)

Related information

[Code review overview on page 90](#)

Using a customized Naming script file

In HCL OneTest™ Embedded for Eclipse IDE, you can edit and customize a Perl Naming script file to check your own naming rules (code custom naming rules U99.1). You must set the path to this customized naming script file in the code review settings to check your naming rules.

To set the path to a customized Naming script file:

1. From the **Project Explorer** view, select the project node.
2. Right-click and select **Properties**.
3. In the window that opens, select **Settings** in the left panel.
4. In the right panel, click **Settings > Code Review**.
5. Click in **Value** cell of the **Naming script file** option and click
6. Select the sample file that you installed: Example "NamingRules1.pl".
7. Click **Apply**.

Code review deviations

In some cases, it can be useful to temporarily ignore a rule non-compliance on a short portion of source code, while documenting the reason why you are allowing this deviation.

About this task

You can justify why you are allowing the deviation in a text. The text is added to the non-compliance in the source code. You can declare a deviation in the source code, for a specified number of lines and for the first or all occurrences of the error, by adding pragma lines to your source code.

1. Open the source file in the Text editor and find the lines of code that you want the rule to ignore.
2. Before the section of code for which compliance to the rule should be ignored, add one of the following lines:

- To justify non-compliance of a rule to the following pragma statement in the first occurrence:

```
#pragma attol crc_justify (<rule>[,<lines>], "<text>")
```

- To justify non-compliance of a rule to the following pragma statement in all occurrences:

```
#pragma attol crc_justify_all (<rule>,<lines>,"<text>")
```

- To justify the first occurrence of non-compliance of a rule in all the files of the current project, including in traps located before the pragma statement:

```
#pragma attol crc_justify_everywhere (<rule>,"<text>")
```

For all the pragma statements: <rule>

- <rule> is the name of the code review rule (for example: 'Rule M8.5').
- <lines> is the number of lines.
- <text> is the reason why the rule is ignored.

The recommended usage for `crc_justify_everywhere` is to create a specific source file containing only the list of pragma statements and add this file to the project.

Code review MISRA 2004 rules

The code review tool covers rules from the lists the rules that produced an error or a warning. Each rule can be individually disabled or assigned a Warning or Error severity by using the Rule configuration window. Some rules also have parameters that can be changed. Among other guidelines, the code review tool implements most rules from the MISRA-C:2004 standard, "Guidelines for the use of the C language in critical systems". These rules are referenced with an M prefix. In addition to the industry standard rules, HCL OneTest™ Embedded provides some additional coding guidelines, which are referenced with an E prefix.

Code Review for C - MISRA 2004 rules

Table 3. MISRA 2004 rules

Code review reference	MISRA-C: 2004 reference	Code review message	Description
Code compliance			
M1.1	Rule 1.1	ANSI C error: <error>	All code shall conform to ISO 9899:1990 Required
M1.1w	Rule 1.1	ANSI C warning: <warning>	Required
Language extensions			Required

Table 3. MISRA 2004 rules**(continued)**

Code review reference	MISRA-C: 2004 reference	Code review message	Description
M2.2	Rule 2.2	Source code shall only use /* ... */ style comments	Source code shall only use /* ... */ style comments Required
M2.3	Rule 2.3	The character sequence /* shall not be used within a comment	The character sequence /* shall not be used within a comment Required
E2.3.1			The character sequence // should not be used within a 'C-style' comment Advisory
E2.3.2			Line-splicing shall not be used in // comments Advisory
E2.6			A function should not contain unused label declarations Advisory
E2.7		There should be no unused parameters in functions	Advisory
E2.8		Macro %name% is never used	Advisory
E2.9		Type %name% is never used	Advisory
E2.10		Tag %name% is never used	Advisory
E2.50		Functions should have less than '100' lines. Note The number of lines can be specified.	Advisory
E2.51		Functions should have less than '15' V(g) complexity. Note: The	Advisory

Table 3. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
		complexity limit of lines can be specified.	
E2.52		Functions should have less than '%param%' lines, outside empty lines (current value: %name%).	
E2.53		Functions should have less than '%param%' lines, outside empty lines or comment lines (current value : %name%).	
E2.54		Functions should have less than '%param%' lines, outside empty lines, comment lines or bracket lines (current value : %name%).	<p>Lines are not counted in the following cases:</p> <ul style="list-style-type: none"> • If they contain spaces (including \t, \r, \n), • If they contain only brackets (there might be several brackets on same line), • If they contain comments only, or if they contain brackets and comments only.
E2.55		Compilation units should define have less than '%param%' functions (current value : %name%).	<p>Optional</p> <p>Compilation unit max number of functions.</p> <p>Default parameter value: 10.</p>
E2.56		Compilation units should have less than '%param%' functions (current value: %name%).	<p>Optional</p> <p>Compilation unit max number of variables.</p> <p>Default parameter value: 10.</p>
E2.57		Compilation unit should have less than '%param%' lines (current value: %name%).	<p>Optional</p> <p>Compilation unit max number of lines.</p>

Table 3. MISRA 2004 rules**(continued)**

Code review reference	MISRA-C: 2004 reference	Code review message	Description
			Default parameter value : 200.
E2.58		Compilation unit should have less than '%param%' lines, not counting empty lines (current value : %name%).	Optional Compilation unit max number of lines. Default parameter value : 200.
E2.59		Compilation unit should have less than '%param%' lines, not counting empty lines or comments (current value: %name%).	Optional Compilation unit max number of lines. Empty lines or comments (current value: %name%) are not counted. Default parameter value : 200.
E2.60		Compilation units should have less than '%param%' lines, not counting empty lines, comments or brackets (current value: %name%) are not counted.	Optional Compilation unit max number of lines. Empty lines, comments or brackets (current value : %name%) are not counted. Default parameter value : 200.
E2.61		Functions should have less than '%param%' parameters (current value: %name%).	
Documentation			
M3.4	Rule 3.4	All uses of the #pragma directive shall be documented and explained.	Required
Character sets			
M4.1.1	Rule 4.1	Only escape sequences that are defined in the ISO C standard shall be used	Only escape sequences that are defined in the ISO C standard shall be used Required

Table 3. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
M4.1.2	Rule 4.1	Only ISO C escape sequences are allowed(\v)	Only ISO C escape sequences are allowed(\v) Required
M4.2	Rule 4.2	Trigraphs shall not be used	Trigraphs shall not be used Required
Identifiers			
M5.1	Rule 5.1	Identifiers %name% and %name% are identical in the first <value> characters. Note The number of characters can be specified.	Identifiers (internal and external) shall not rely on the significance of more than 31 characters Required
E5.1.1		Identifiers %name% and %name% are ambiguous because of possible character confusion. Note that ambiguous characters can be parameterized.	Advisory
E5.1.2		Possible typing mistakes between the variables %name% or %name% because of repeating character.	Advisory
E5.1.3		Identifiers %name% and %name% are identical in the first %param% characters ignoring case	Advisory
E5.1.4		Macros %name% and %name% are identical in the first %param% characters	Advisory
E5.1.5		Macro %name% and identifier %name% are identical in the first %param% characters	Advisory

Table 3. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
E5.1.6		Macros %name% and %name% are identical in the first %param% characters ignoring case	Advisory
E5.1.7		Macro %name% and identifier %name% are identical in the first %param% characters ignoring case	Advisory
M5.2	Rule 5.2	Identifier %name% in an inner scope hides the same identifier in an outer scope : %location%	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier Required
E5.3		The tag name %name% should not be reused. Name already found in %location%	Advisory
M5.3.1	Rule 5.3		The typedef name %name% should not be reused except for its tag. Name already found in %location% Required
M5.3.2	Rule 5.3		The typedef name '%name%' should not be reused even for its tag. Name already found in %location% Required
M5.4	Rule 5.4	A struct and union cannot use the same tag name	A tag name shall be a unique identifier Required
M5.5	Rule 5.5	The static object or function %name% should not be reused. Static object or function already found in %location%.	No object or function identifier with static storage duration should be reused Advisory

Table 3. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
M5.6	Rule 5.6	Avoid using the same identifier %name% in two different name spaces. Identifier already found in %location%	No identifier in one name space should have the same spelling as an identifier in another name space, with the exception of structure and union member names Advisory
M5.7	Rule 5.7	The identifier %name% should not be reused. Identifier already found in %location%.	Advisory
Types			
M6.1.1	Rule 6.1	The C language plain char type should only be used for character values.	The C language plain char type should only be used for character values. Required
M6.1.2	Rule 6.1	Case char value is applicable only if the switch statement value is plain character variable	Required
M6.1.3	Rule 6.1	Avoid using comparison operators on plain char.	Required
M6.2	Rule 6.2	The C language signed char or unsigned char types should only be used for numeric values.	The C language signed char or unsigned char types should only be used for numeric values. Required
M6.3	Rule 6.3	The C language numeric type %name% should not be used directly but instead used to define typedef.	typedefs that indicate size and signedness should be used in place of the basic types Advisory
E6.3		The implicit 'int' type should not be used.	Required
M6.4.1	Rule 6.4	Bit fields should only be of type 'unsigned int' or 'signed int'.	Required

Table 3. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
M6.4.2	Rule 6.4	Bit fields should not be of type 'enum'	Required
M6.4.3	Rule 6.4	Bit fields should only be of explicitly signed or unsigned type	Required
M6.4.4	Rule 6.4	Bit fields should not be of type 'bool' under c99	Required
M6.4.5	Rule 6.4	Bit fields should not be of type 'boolean' outside c99	Required
M6.5	Rule 6.5	Bit fields of type 'signed int' must be at least 2 bits long.	Required
Constants			
M7.1	Rule 7.1	Octal constants and escape sequences should not be used.	Octal constants (other than zero) and octal escape sequences shall not be used Required
E7.1		Octal and hexadecimal escape sequences shall be terminated	Required
E7.2		The lowercase character 'l' shall not be used in a literal suffix	Required
E7.3		A string literal shall not be assigned to an object unless the object's type is pointer to a const-qualified char	Required
Declarations and definitions			
M8.1.1	Rule 8.1	A prototype for the function %name% should be declared before defining the function.	Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call Required

Table 3. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
E8.1.1		A prototype for the global object %name% should be declared before defining the object	Required
M8.1.2	Rule 8.1	A prototype for the function %name% should be declared before calling the function.	Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call Required
M8.1.3	Rule 8.1	A prototype for the function %name% should be declared before calling the function	Required
M8.2.1	Rule 8.2	The type of %name% should be explicitly stated.	Whenever an object or function is declared or defined, its type shall be explicitly stated Required
M8.2.2	Rule 8.2	The type of parameter %name% should be explicitly stated	Required
M8.3	Rule 8.3	Parameters and return types should use the same type names in the declaration and in the definition, even if basic types are the same.	For each function parameter the type given in the declaration and definition shall be identical, and the return types shall also be identical Required
E8.3		Parameters and return types should use compatible type in the declaration and in the definition	Required
M8.4	Rule 8.4	If objects or functions are declared multiple times their types should be compatible.	Required
M8.5.1	Rule 8.5	The body of function %name% should not be located in a header file.	Required

Table 3. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
E.8.50		Use the const qualification for variable %name% which is pointer and which is not used to change the pointed object	Required
E.8.51		The object %name% is never referenced	Required
M8.5.2	Rule 8.5	The memory storage (definition) for the variable %name% should not be in a header file.	Objects shall be defined at block scope if they are only accessed from within a single function. Required
M8.6	Rule 8.6	Functions should not be declared at block scope.	Required
M8.7	Rule 8.7	Global objects should not be declared if they are only used from within a single function.	Objects shall be defined at block scope if they are only accessed from within a single function Required
M8.8.2	Rule 8.8	Static function %name% should only be declared in a single file. Redundant declaration found at: %location%	Required
M8.8.3	Rule 8.8	Static object %name% should only be declared in a single file. Redundant declaration found at: %location%	Required
M8.8.4	Rule 8.8	Identifiers %name% that declare objects or functions with external linkage shall be declared once in one and only one file	Required
M8.8.5	Rule 8.8	Identifiers %name% that declare objects or functions with external linkage shall be unique	Required

Table 3. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
M8.9.1	Rule 8.9	The global object or function %name% should have exactly one external definition. Redundant definition found in %location%	An identifier with external linkage shall have exactly one external definition
M8.9.2	Rule 8.9	The global object or function %name% should have exactly one external definition. No definition found.	Required
M8.10.1	Rule 8.10	Global object %name% that are only used within the same file should be declared using the static storage-class specifier.	All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required. Required
M8.10.2	Rule 8.10	Global function %name% that are only used within the same file should be declared using the static storage-class specifier.	All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required Required
M8.11	Rule 8.11	Global objects or functions that are only used within the same file should be declared with using the static storage-class specifier.	The static storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage Required
M8.12	Rule 8.12	When a global array variable can be used from multiple files, its size should be defined at initialization time.	Required
E.8.14		Inline function %name% should be static	Required
Initialization		The restrict type qualifier shall not be used	Required

Table 3. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
M9.1	Rule 9.1	Variables with automatic storage duration should be initialized before being used.	Required
M9.2	Rule 9.2	Nested braces should be used to initialize nested multi-dimension arrays and nested structures.	Required
E9.2		Arrays shall not be partially initialized	Required
M9.3	Rule 9.3	Either all members or only the first member of an enumerator list should be initialized.	In an enumerator list, the "=" construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized Required
M9.3	Rule 9.3	Either all members or only the first member of an enumerator list should be initialized	Required
E9.3	Rule E9.3	Enumeration member %name% have a not unique implicitly specified value	Required
E9.4		The global variable %name% is not initialized	Required
Arithmetic type conversions			
E10.1		Constraint violation : can't use floating type as operand of '[]', '%', '<<', '>>', '~', '&', ' ', '^'	Required
M10.1.1	Rule 10.1	Implicit conversion of a complex integer expression to a smaller sized integer is not allowed.	The value of an expression of integer type shall not be implicitly converted to a different underlying type if:

Table 3. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
			<ul style="list-style-type: none"> • a) it is not a conversion to a wider integer type of the same signedness, or • b) the expression is complex, or • c) the expression is not constant and is a function argument, or • d) the expression is not constant and is a return expression. <p>Required</p>
M10.1.2	Rule 10.1	Implicit conversion of an integer expression to a different signedness is not allowed.	Required
M10.2	Rule 10.2	Conversion of a complex floating expression is not allowed. Only constant expressions can be implicitly converted and only to a wider floating type of the same signedness.	<p>The value of an expression of floating type shall not be implicitly converted to a different type if:</p> <ul style="list-style-type: none"> • a) it is not a conversion to a wider floating type, or • b) the expression is complex, or • c) the expression is a function argument, or • d) the expression is a return expression. <p>Required</p>
E10.2		Operand should be boolean.	Required
M10.3	Rule 10.3	Type cast of complex integer expressions is only allowed into a narrower type of the same signedness.	<p>The value of a complex expression of integer type may only be cast to a type that is narrower and of the same signedness as the underlying type of the expression</p> <p>Required</p>

Table 3. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
E10.3		Can't use a boolean as a numeric value	Required
M10.4	Rule 10.4	Type cast of complex floating expressions is only allowed into a narrower type of the same signedness.	The value of a complex expression of floating type may only be cast to a narrower floating type Required
E10.4		Can't use a char as a numeric value	Required
M10.5	Rule 10.5	When using operator '~' or '<<' on 'unsigned char' or 'unsigned int', you should always cast returned value	Required
E10.5	Rule E10.5	Can't use a not anonymous enum as a numeric value	Required
M10.6	Rule 10.6	Definitions of unsigned type constants should use the 'U' suffix.	A "U" suffix shall be applied to all constants of unsigned type Required
E10.6		Shift and bitwise operations should be performed on unsigned value	Required
E10.7		Right hand operand of shift operation should be an unsigned value	Required
E10.8		Unary minus operation should not be performed on unsigned value	Required
E10.9		Expressions of essentially character type shall not be used inappropriately in addition and subtraction operations	Required

Table 3. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
E10.10		The value of an expression shall not be assigned to an object with a narrower essential type	Required
E10.11		The value of an expression shall not be assigned to an object with a different essential type category	Required
E10.12		Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category	Required
E10.13		The value of an expression should not be cast to an inappropriate essential type	Required
E10.14		The value of a composite expression shall not be assigned to an object with wider essential type	Required
E10.15		If a composite expression is used as one operand of an operation in which the usual arithmetic conversions are performed then the other operand shall not have wider essential type	Required
E10.16		The value of a composite expression shall not be cast to a different essential type category or a wider essential type	Required
Pointer type conversions			
M11.1	Rule 11.1	A function pointer should not be converted to another type of pointer.	Conversions shall not be performed between a pointer to a function and any type other than an integral type

Table 3. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
			Required
E11.1		Conversions shall not be performed between a pointer to an incomplete type and any other type	Required
M11.2	Rule 11.2	An object pointer should not be converted to another type of pointer.	Conversions shall not be performed between a pointer to object and any type other than an integral type, another pointer to object type or a pointer to void Required
E11.2		A conversion should not be performed from pointer to void into pointer to object	Required
M11.3	Rule 11.3	Casting a pointer type to an integer type should not occur.	A cast should not be performed between a pointer type and an integral type Advisory
E11.3	E11.3	A cast shall not be performed between pointer to void and an arithmetic type	Required
E11.4		A cast shall not be performed between pointer to object and a non-integer arithmetic type	Required
M11.4.1	Rule 11.4	Casting an object pointer type to a different object pointer type should not occur.	A cast should not be performed between a pointer to object type and a different pointer to object type Advisory
M11.4.2	Rule 11.4	Casting an object pointer type to a different object pointer type should not occur, especially	Advisory

Table 3. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
		when object sizes are not the same.	
M11.5	Rule 11.5	Casting of pointers to a type that removes any const or volatile qualification on the pointed object should not occur.	A cast shall not be performed that removes any const or volatile qualification from the type addressed by a pointer. Required
Expressions			
M12.1	Rule 12.1	Implicit operator precedence may cause ambiguity. Use parenthesis to clarify this expression.	Limited dependence should be placed on C's operator precedence rules in expressions Advisory
E12.11		Implicit bitwise operator precedence may cause ambiguity. Use parenthesis to clarify this expression.	Advisory
M12.3	Rule 12.3	The sizeof operator should not be used on expressions that contain side effects.	Required
M12.4.1	Rule 12.4	An expression that contains a side effect should not be used in the right-hand operand of a logical && or operator.	The right-hand operand of a logical && or operator shall not contain side effects Required
M12.4.2	Rule 12.4	The function in the right-hand operand of a logical && or operator might cause side effects.	
M12.5	Rule 12.5	Parenthesis should be used around expressions that are operands of a logical && or .	Required
E12.51		Ternary expression ?: should not be used.	Advisory

Table 3. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
E12.54		Expressions should not cause a side effect assignment.	Advisory
M12.6	Rule 12.6	Only Boolean operands should be used with logical operators (&&, and !).	The operands of logical operators (&&, and !) should be effectively Boolean. Expressions that are effectively Boolean should not be used as operands to operators other than (&&, and !) Advisory
E12.61		The operator on a Boolean expression should be a logical operator (&&, or !).	Advisory
M12.7	Rule 12.7	Bitwise operators should only use unsigned operands.	Bitwise operators shall not be applied to operands whose underlying type is signed Required
M12.8	Rule 12.8	The right-hand operand of a shift operator should not be too big or negative.	The right-hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left-hand operand Required
M12.9	Rule 12.9	Only use unary minus operators with signed expressions.	The unary minus operator shall not be applied to an expression whose underlying type is unsigned Required
M12.10	Rule 12.10	Do not use the comma operator	Required
M12.12	Rule12.12		Advisory Parenthesis should be used around expression that is operand of 'sizeof' operator.
M12.13	Rule 12.13	The increment (++) or the decrement (--) operators should not be	Advisory

Table 3. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
		used with other operators in an expression.	
Control statement expressions			
E13.1		The result of an assignment operator should not be used in an expression	Required
M13.1.1	Rule 13.1	Boolean expressions should not contain assignment operators.	Assignment operators shall not be used in expressions that yield a Boolean value Required
M13.1.2	Rule 13.1	Boolean expressions should not contain side effect operators.	Required
M 13.2	Rule 13.2	Non-Boolean values that are tested against zero should have an explicit test	Tests of a value against zero should be made explicit, unless the operand is effectively Boolean Advisory
M13.3	Rule 13.3	The equal or not equal operator should not be used in floating-point expressions.	Floating-point expressions shall not be tested for equality or inequality Required
M13.4	Rule 13.4	Floating-point variables should not be used to control a for statement.	Required
M13.5.1	Rule 13.5	Only loop counter should be initialized in a loop initialization part.	The three expressions of a statement shall be concerned with loop control only. Required

Table 3. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
M13.5.2	Rule 13.5	In the 'update part' of a 'for statement', only 'loop counter' should be updated	Required
M13.5.3	Rule 13.5	There should be one and only one loop counter for loop statement.	Required
M13.6	Rule 13.6	Loop counter of a 'for statement' should not be modified within the body of the loop.	Required
M13.7	Rule 13.7	Invariant Boolean expressions should not be used.	Boolean operations whose results are invariant shall not be permitted Required
Control flow			
M14.1	Rule 14.1	Unreachable code.	Required
M14.2	Rule 14.2	A non-null statement should either have a side effect or change the control flow.	Required
M14.3	Rule 14.3	A null statement in original source code should be on a separate line and the semicolon should be followed by at least one white space and then a comment.	Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment provided that the first character following the null statement is a white-space character Required
M14.4	Rule 14.4	Do not use the goto statement.	Required
E14.4.1		The goto statement shall jump to a label declared later in the same function	Required
E14.4.2		Any label referenced by a goto statement shall be declared in	Required

Table 3. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
		the same block, or in any block enclosing the goto statement	
E14.4.3		There should be no more than one break or goto statement used to terminate any iteration statement	Required
M14.5	Rule 14.5	Do not use the continue statement.	Required
M14.6	Rule 14.6	Only one break statement should be used within a loop.	For any iteration statement there shall be at most one break statement used for loop termination Required
M14.7.1	Rule 14.7	Only one exit point should be defined in a function.	A function shall have a single point of exit at the end of the function Required
M14.7.2	Rule 14.7	The return keyword should not be used in a conditional block.	Required
M14.8.1	Rule 14.8	The switch statement should be followed by a compound statement {}.	The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement Required
M14.8.2	Rule 14.8	The while statement should be followed by a compound statement {}.	
M14.8.3	Rule 14.8	The do..while statement should contain a compound statement {}.	
M14.8.4	Rule 14.8	The for statement should be followed by a compound statement {}.	

Table 3. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
M14.9.1	Rule 14.9	The if (expression) construct should be followed by a compound statement {}.	An if (expression) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement Required
M14.9.2	Rule 14.9	The else keyword should be followed by either a compound statement or another if statement.	
M14.9.3	Rule 14.9	The else keyword should be followed by a compound statement	
M14.10	Rule 14.10	All if ... else if sequences should have an else block.	All if ... else if constructs shall be terminated with an else clause Required
Switch statements			
M15.0	Rule 15.0	A switch block should start with a case.	The MISRA C switch syntax shall be used Required
M15.1	Rule 15.1	A case or default statements should only be used directly within the compound block of a switch statement.	A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement Required
E15.10		The switch expression should not have side effects.	Required
M15.2	Rule 15.2	The break statement should only be used to terminate every non-empty switch block.	An unconditional break statement shall terminate every non-empty switch clause Required
M15.3.1	Rule 15.3	The switch statement should have a default clause.	Required

Table 3. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
M15.3.2	Rule 15.3	The default clause should be the last clause of the switch statement.	
M15.4.1	Rule 15.4	A Boolean should not be used as a switch expression.	A switch expression shall not represent a value that is effectively Boolean Required
M15.4.2	Rule 15.4	A constant should not be used as a switch expression.	Required
M15.5	Rule 15.5	At least one case should be defined in the switch.	Every switch statement shall have at least one case clause Required
Functions			
M16.1	Rule 16.1	The function %name% should not have a variable number of arguments.	Functions shall not be defined with a variable number of arguments Required
Rule M16.1.2	Rule 16.1	The library functions 'va_list, va_arg, va_start, va_end, va_copy' should not be used	Required
M16.2.1	Rule 16.2	Recursive functions are not allowed. The function %name% is directly recursive.	Functions shall not call themselves, either directly or indirectly Functions shall not call themselves, either directly or indirectly
M16.2.2	Rule 16.2	Recursive functions are not allowed. The function %name% is recursive when calling %name% .	Required
M16.3	Rule 16.3	The function prototype should name all its parameters.	Identifiers shall be given for all of the parameters in a function prototype declaration Required

Table 3. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
M16.4	Rule 16.4	The identifiers used in the prototype and definition should be the same.	Required
M16.5	Rule 16.5	Functions with no parameters should use the void type.	Required
E16.50		The function %name% is never referenced.	Required
M16.6	Rule 16.6	The number of arguments used in the call does not match the number declared in the prototype.	Required
M16.7	Rule 16.7	Use the const qualification for parameter %name% which is pointer and which is not used to change the pointed object.	A pointer parameter in a function prototype should be declared as pointer to const if the pointer is not used to modify the addressed object Required
M16.8	Rule 16.8	The return should always be defined with an expression for non-void functions.	All exit paths from a function with non-void return type shall have an explicit return statement with an expression Required
M16.9	Rule 16.9	Function identifiers should always use a parenthesis or a preceding &.	A function identifier shall only be used with either a preceding &, or with a parenthesized parameter list, which may be empty Required
M16.10	Rule 16.10	When a function returns a value, this value should be used.	If a function returns error information, then that error information shall be tested Required
Pointers and arrays			

Table 3. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
M17.4	Rule 17.4	Pointer arithmetic except array indexing should not be used.	Array indexing shall be the only allowed form of pointer arithmetic Required
M17.5	Rule 17.5	A declaration should not use more than two levels of pointer indirection.	Advisory
Structures and unions			
M18.1	Rule 18.1	Structure or union types should be finalized before the end of the compilation units.	Required
E18.1		Flexible arrays members shall not be declared	Required
18.2		Variable-length array types shall not be used	Required
E18.3		The declaration of an array parameter shall not contain the static keyword between the []	Required
M18.4	Rule 18.4	Do not use unions.	Required
Preprocessing directives			
M19.1	Rule 19.1	Only preprocessor directives or comments may occur before the #include statements.	#include statements in a file should only be preceded by other preprocessor directives or comments Advisory
M19.2	Rule 19.2	Do not use non-standard characters in included file names.	Advisory
M19.3	Rule 19.3	Filenames with the #include directive should always use the <filename> or "filename" syntax.	Required

Table 3. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
M19.4	Rule 19.4	A C macro should only be expanded to a constant, a braced initializer, a parenthesised expression, a storage class keyword, a type qualifier, or a do-while-zero block.	Required
M19.5	Rule 19.5	Macro definitions or #undef should not be located within a block.	Required
M19.6	Rule 19.6	Do not use the #undef directive.	Required
M19.7	Rule 19.7	Function should be used instead of macros when possible.	Advisory
M19.8	Rule 19.8	Missing argument when calling the macro.	A function-like macro shall not be invoked without all of its arguments. Required
M19.9	Rule 19.9	The preprocessing directive %name% should not be used as argument to the macro.	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives Required
M19.10	Rule 19.10	The parameter %name% in the macro should be enclosed in parentheses except when it is used as the operand of # or ##.	In the definition of a function-like macro each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of # or ## Required
M19.11	Rule 19.11	Undefined macro identifier in the preprocessor directive.	All macro identifiers in preprocessor directives shall be defined before use, except in #ifdef and #ifndef preprocessor directives and the defined() operator Required

Table 3. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
M19.12	Rule 19.12	The # or ## preprocessor operator should not be used more than once.	There shall be at most one occurrence of the # or # preprocessor operators in a single macro definition Required
M19.13	Rule 19.13	The # and ## preprocessor operator should be avoided.	Advisory
M19.14	Rule 19.14	Only use the 'defined' preprocessor operator with a single identifier.	The defined preprocessor operator shall only be used in one of the two standard forms Required
M19.15	Rule 19.15	Header file contents should be protected against multiple inclusions	Precautions shall be taken in order to prevent the contents of a header file being included twice Required
M19.16	Rule 19.16	Possible bad syntax in preprocessing directive.	Preprocessing directives shall be syntactically meaningful even when excluded by the preprocessor Required
M19.17	Rule 19.17	A #if, #ifdef, #else, #elif or #endif preprocessor directive has been found without its matching directive in the same file.	All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related Required
E19.18		The controlling expression of a #if or #elif preprocessing directive shall evaluate to 0 or 1	Required
E19.19		A macro parameter immediately following a # operator shall not	Required

Table 3. MISRA 2004 rules**(continued)**

Code review reference	MISRA-C: 2004 reference	Code review message	Description
		immediately be followed by a ## operator	
E19.20		Macro parameter %name% used as an operand to the # and ## operators shall not be used elsewhere in this macro	Required
Standard libraries			
M20.1	Rule 20.1	%name% should not be defined, redefined or undefined.	Reserved identifiers, macros and functions in the standard library, shall not be defined, redefined or undefined Required
E20.1		A macro shall not be defined with the same name as a keyword: %name%	Required
M20.2.1		#define and #undef shall not be used on a reserved identifier or reserved macro name: Identifier %name% already found in %name%	Required
M20.2.2	Rule 20.2	#define and #undef shall not be used on identifier beginning with an underscore or on 'defined' keyword: %name%	Required
M20.2.3	Rule 20.2	Declared identifier should not be a reserved identifier or reserved macro name: Identifier %name% already found in %name%	Required
M20.2.4	Rule 20.2	Declared identifier should not begin with an underscore or be 'defined' keyword: %name%	Required

Table 3. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
M20.4	Rule 20.4	Dynamic heap memory allocation shall not be used.	<p>This precludes the use of the functions <code>calloc</code>, <code>malloc</code>, <code>realloc</code>, <code>free</code> and <code>strdup</code>. There is a whole range of unspecified, undefined and implementation-defined behaviour associated with dynamic memory allocation, as well as a number of other potential pitfalls. Dynamic heap memory allocation may lead to memory leaks, data inconsistency, memory exhaustion, non-deterministic.</p> <p>Note that some implementations may use dynamic heap memory allocation to implement other functions (for example functions in the library <code>string.h</code>). If this is the case then these functions shall also be avoided.</p> <p>Required</p>
M20.5	Rule 20.5	The error indicator <code>errno</code> shall not be used.	<p><code>errno</code> is a facility of C, which in theory should be useful, but which in practice is poorly defined by the standard. A non zero value may or may not indicate that a problem has occurred; as a result it shall not be used. Even for those functions for which the behaviour of <code>errno</code> is well defined, it is preferable to check the values of inputs before calling the function rather than rely on using <code>errno</code> to trap errors (see Rule 16.10).</p> <p>Required</p>
M20.6	Rule 20.6	The macro <code>offsetof</code> , in library <code><stddef.h></code> , shall not be used.	<p>Use of this macro can lead to undefined behaviour when the types of the operands are incompatible or when bit fields are used.</p> <p>Required</p>

Table 3. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
M20.7	Rule 20.7	The setjmp macro and the longjmp function shall not be used.	<p>etjmp and longjmp allow the normal function call mechanisms to be bypassed, and shall not be used.</p> <p>Remark : sigsetjmp and siglongjmp (Gnu Library) are also detected</p> <p>Required</p>
E20.7		The standard header file <setjmp.h> shall not be used	Required
M20.8	Rule 20.8	The signal handling facilities of <signal.h> shall not be used.	<p>Signal handling contains implementation-defined and undefined behavior.</p> <p>Required</p>
M20.9	Rule 20.9	The input/output library <stdio.h> shall not be used in production code.	<p>This includes file and I/O functions fgetpos, fopen, ftell, gets, perror, remove, rename, and ungetc.</p> <p>Streams and file I/O have a large number of unspecified, undefined and implementation-defined behaviours associated with them. It is assumed within this document that they will not normally be needed in production code in embedded systems.</p> <p>If any of the features of stdio.h need to be used in production code, then the issues associated with the feature need to be understood.</p> <p>Required</p>
M20.10	Rule 20.10	The library functions atof, atoi and atol from library <stdlib.h> shall not be used.	These functions have undefined behavior associated with them when the string cannot be converted.

Table 3. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
			Required
M20.11	Rule 20.11	The library functions abort, exit, getenv and system from library <stdlib.h> shall not be used.	<p>These functions will not normally be required in an embedded system, which does not normally need to communicate with an environment</p> <p>Then, it is essential to check on the implementation-defined behavior of the function in the environment.</p> <p>Required</p>
E20.11		The library macro or function 'bsearch, qsort' should not be used	Required
M20.12	Rule 20.12	The time handling functions of library <time.h> shall not be used.	<p>Includes time, strftime. This library is associated with clock times. Various aspects are implementation dependent or unspecified, such as the format of times. If any of the facilities of time.h are used, then the exact implementation for the compiler being used must be determined, and a deviation being raised.</p> <p>Required</p>
E20.12		The input/output library <wchar.h> shall not be used in production code	Required
E20.13		The standard header file <tg-math.h> shall not be used	Required
E20.14		The library macro or function 'fclearexcept, fegetexceptflag, feraiseexcept, fesetexceptflag, fetestexcept, FE_INEXACT, FE_DIVBYZERO, FE_UNDERFLOW,	Required

Table 3. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
		FE_OVERFLOW, FE_INVALID, FE_-ALL_EXCEPT' should not be used	
U99.1		User defined rule	Required



Note: Applies to HCL OneTest™ Embedded Studio only:

The code review references in bold in this table are disabled when they are run from the code review link checker in test mode. To verify these rules, you must run the code review from the application node in HCL OneTest™ Embedded Studio. For more information, see [Running complete verification of MISRA rules from an application node on page 298](#).

Code review MISRA 2012 rules

The code review tool covers rules from the lists the rules that produced an error or a warning. Each rule can be individually disabled or assigned a Warning or Error severity by using the Rule configuration window. Some rules also have parameters that can be changed. Among other guidelines, the code review tool implements most rules from the MISRA-C:2012 standard, "Guidelines for the use of the C language in critical systems". These rules are referenced with an M prefix. In addition to the industry standard rules, HCL OneTest™ Embedded provides some additional coding guidelines, which are referenced with an E prefix.

Code Review - MISRA 2012 rules

D is set for Decidable, U for Undecidable.

Code review reference	Type	D/U	Description	Level
M1.1	Error	D	ANSI C error: %name %	Required
M1.1W	Error	D	ANSI C warning: %name%	Required
M1.2	Error	U	Use of #pragma %name% should always be encapsulated and documented	Advisory

Code review reference	Type	D/U	Description	Level
E1.1	Error	D	Function max number of line	Required
E.1.2	Error	D	Function max V(g)	Required
E1.3			Functions should have less than '%param%' lines, outside empty lines (current value: %name%).	
E1.4			Functions should have less than '%param%' lines, outside empty lines or comment lines (current value : %name%).	
E1.5			<p>Functions should have less than '%param%' lines, outside empty lines, comment lines or bracket lines (current value : %name%).</p> <p>Lines are not counted in the following cases:</p> <ul style="list-style-type: none"> • If they contain spaces (including \t, \r, \n), • If they contain only brackets (there might be several brackets on same line), • If they contain comments on- 	

Code review reference	Type	D/U	Description	Level
E1.6			ly, or if they contain brackets and comments only.	
E1.6			Optional Compilation units should define less than '%param%' functions (current value: %name%).	
E1.7			Default parameter value: 10.	
E1.7			Optional Compilation units should define less than '%param%' variables (current value: %name%).	
E1.8			Default parameter value: 10.	
E1.8			Optional Compilation units should have less than '%param%' lines (current value: %name%).	
E1.9			Default parameter value : 200.	
E1.9			Optional Compilation unit should have less than '%param%' lines,	

Code review reference	Type	D/U	Description	Level
E1.10			not counting empty lines (current value : %name%).	
			Empty lines (current value : %name%) are not counted.	
			Default parameter value : 200.	
			Optional	
			Compilation unit should have less than '%param%' lines not counting empty lines or comments (current value : %name%).	
			Empty lines or comments (current value : %name%) are not counted.	
			Default parameter value : 200.	
E1.11			Optional	
			Compilation unit should have less than '%param%' lines not counting empty lines, comments or brackets (current value: %name%).	
			Empty lines, comments or brackets (current value :	

Code review reference	Type	D/U	Description	Level
E1.12			%name%) are not counted. Default parameter value : 200. Functions should have less than '%param%' parameters (current value : %name%).	
M2.1	Error	U	a project shall not contain unreachable code	Required
M2.2.1	Error	U	A non-null statement should either have a side effect or change the control flow	Required
M2.2.2	Error	U	The function %name% is never referenced	Required
M2.2.3	Error	D	The object %name% is never referenced	Required
M2.3	Warning	D	Type %name% is never used	Advisory
M2.4	Warning	D	Tag %name% is never used	Advisory
M2.5	Warning	D	Macro %name% is never used	Advisory
M2.6	Warning	D	A function should not contain unused label declarations	Advisory
M2.7	Warning	D	There should be no unused parameters in functions	Advisory
M3.1.1	Error	D	The character sequence /* should not	Required

Code review reference	Type	D/U	Description	Level
M3.1.2	Error	D	be used within a comment The character sequence // should not be used within a 'C-style' comment	Required
M3.2	Error	D	Line-splicing shall not be used in // comments	Required
E3.1	Error	D	A null statement in original source code should be on a separate line and the semicolon should be followed by at least one white space and then a comment	Required
M4.1	Error	D	Octal and hexadecimal escape sequences shall be terminated	Required
M4.2	Warning	D	Trigraphs should not be used	Advisory
E4.1	Error	D	Only ISO C escape sequences are allowed	Advisory
E.4.2	Error	D	Only ISO C escape sequences are allowed(\v)	Advisory
M5.1.1	Error	D	External identifiers shall be distinct in the first 31 characters	Required
M5.1.2	Error	D	External identifiers shall be distinct in the first 6 characters ignoring case	Required

Code review reference	Type	D/U	Description	Level
M5.2	Error	D	Identifiers %name% declared in the same scope and name space shall be distinct. Identifier identical in the first %param% characters already found in %location%	Required
M5.3	Error	D	Identifier %name% declared in an inner scope shall not hide an identifier declared in an outer scope. Identifier identical in the first %param% characters already found in %location%	Required
M5.4.1	Error	D	Macros %name% and %name% are identical in the first %param% characters	Required
M5.4.2	Error	D	Macros %name% and %name% are identical in the first %param% characters ignoring case.	Required
M5.5.1	Error	D	Macro %name% and identifier %name% are identical in the first %param% characters.	Required
M5.5.2	Error	D	Macro %name% and identifier %name% are identical in the first %param% characters ignoring case.	Required

Code review reference	Type	D/U	Description	Level
M5.6	Error	D	Macro %name% and identifier %name% are identical in the first %name% %param% characters ignoring case. The typedef name %name% should not be reused except for its tag. Name already found in %location%	Required
M5.7.1	Error	D	The tag name %name% should not be reused	Required
M5.7.2	Error	D	A struct and union cannot use the same tag name	Required
M5.8	Error	D	Identifiers that define objects or functions with external linkage shall be unique	Required
M5.9	Error	D	Identifiers that define objects or functions with internal linkage should be unique	Advisory
E5.1	Error	D	External identifiers shall not be ambiguous because of possible character confusion.	Advisory
E5.2	Error	D	External identifiers shall not be ambiguous because of character repetition	Advisory
E5.3	Warning	D	The identifier<name> should not be reused.	Advisory

Code review reference	Type	D/U	Description	Level
E5.4	Error	D	Identifier already found in %location% Identifier %name% in an inner scope hides the same identifier in an outer scope : %location%	Advisory
E5.5	Error	D	The typedef name %name% should not be reused even for its tag. Name already found in %location%	Advisory
M6.1.1	Error	D	Bit fields should only be of type 'unsigned int' or 'signed int'	Required
M6.1.2	Error	D	Bit fields should not be of type 'enum'	Required
M6.1.3	Error	D	Bit fields should only be of explicitly signed or unsigned type	Required
M6.1.4	Error	D	Bit fields should not be of type 'bool' under c99	Required
M6.1.5	Error	D	Bit fields should not be of type 'boolean' outside c99	Required
M6.2	Error	D	Single-bit fields shall not be of a signed type	Required
E6.1	Warning	D	The C language numeric type %name% should not be used directly but instead used to define typedef	Required

Code review reference	Type	D/U	Description	Level
E6.2	Warning	D	The implicit 'int' type should not be used	Required
M7.1	Error	D	Octal constants shall not be used	Required
M7.2	Error	D	A "u" or "U" suffix shall be applied to all integer constants that are represented in an unsigned type"	Required
M7.3	Error	D	The lowercase character "l" shall not be used in a literal suffix"	Required
M7.4	Error	D	A string literal shall not be assigned to an object unless the object's type is pointer to a const-qualified char	Required
M8.1	Error	D	Types shall be explicitly specified	Required
M8.2.1	Error	D	The function prototype should name all its parameters	Required
M8.2.2	Error	D	Functions with no parameters should use the void type	Required
M8.2.3	Error	D	The type of parameter %name% should be explicitly stated	Required
M8.3.1	Error	D	Parameters and return types should use compatible type in the declaration and in the definition	Required

Code review reference	Type	D/U	Description	Level
M8.3.2	Error	D	The identifiers used in the prototype and definition should be the same	Required
M8.4.1	Error	D	A prototype for the global function %name% should be declared before defining the function	Required
M8.4.2	Error	D	A prototype for the global object %name% should be declared before defining the object	Required
M8.4.3	Error	D	If objects or functions are declared multiple times their types should be compatible	Required
M8.5	Error	D	Identifiers %name% that declare objects or functions with external linkage shall be declared once in one and only one file	Required
M8.6	Error	D	Identifiers %name% that declare objects or functions with external linkage shall be unique	Required
M8.7.1	Warning	D	Global object %name% that are only used within the same file should be declared using the static storage-class specifier.	Advisory
M8.7.12	Warning	D	Global function %name% that are on-	Advisory

Code review reference	Type	D/U	Description	Level
M8.8	Error	D	ly used within the same file should be declared using the static storage-class specifier. The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage	Required
M8.9	Warning	D	An object should be defined at block scope if its identifier only appears in a single function	Advisory
M8.10	Error	D	Inline function %name% should be static	Required
M8.11	Warning	D	When an array with external linkage is declared, its size should be explicitly specified	Advisory
M8.14	Error	D	The restrict type qualifier shall not be used	Required
E.8.1	Error	D	Parameters and return types should use exactly the same type names in the declaration and in the definition	Required
E.8.2	Error	D	A prototype for the static function %name% should be declared before defining the function	Required

Code review reference	Type	D/U	Description	Level
E.8.3	Error	D	Static function %name% should only be declared in a single file. Redundant declaration found at: %name%	Required
E.8.4	Error	D	Static object %name% should only be declared in a single file. Redundant declaration found at: %location%	Required
E.8.5	Error	D	Either all members or only the first member of an enumerator list should be initialized	Required
E.8.6	Error	D	The body of function %name% should not be located in a header file	Required
E.8.7	Error	D	The memory storage (definition) for the variable %name% should not be in a header file	Required
E.8.8	Error	D	Functions should not be declared at block scope	Required
E.8.9	Error	D	The global object or function '%name%' should have exactly one external definition. Redundant definition found in %location%	Required
E.8.10	Error	D	The global object or function %name%	Required

Code review reference	Type	D/U	Description	Level
E.8.11	Error	D	%name% should have exactly one external definition. No definition found	Required
M9.2	Error	D	Use the const qualification for variable %name% which is pointer and which is not used to change the pointed object	Required
M9.3	w	D	The initializer for an aggregate or union shall be enclosed in braces	Required Exception not covered
E9.1	Error	D	Arrays shall not be partially initialized	Required Exception not covered
E9.2	Error	D	Variables with automatic storage duration should be initialized before being used	Required
M10.1.1	Error	D	The global variable %name% is not initialized	Required
M10.1.2	Error	D	Constraint violation : can't use floating type as operand of "[], %, <<, >>, ~, &, , ^"	Required
M10.1.3	Error	D	Operand should be boolean	Required
M10.1.3	Error	D	Can't use a boolean as a numeric value	Required

Code review reference	Type	D/U	Description	Level
M10.1.4	Error	D	Can't use a char as a numeric value	Required
M10.1.5	Error	D	Can't use a not anonymous enum as a numeric value	Required
M10.1.6	Error	D	Shift and bitwise operations should be performed on unsigned value	Required
M10.1.7	Error	D	Right hand operand of shift operation should be performed on unsigned value	Required
M10.1.8	Error	D	Unary minus operation should not be performed on unsigned value	Required
M10.2	Error	D	Expressions of essentially character type shall not be used inappropriately in addition and subtraction operations	Required
M10.3.1	Error	D	The value of an expression shall not be assigned to an object with a narrower essential type	Required
M10.3.2	Error	D	The value of an expression shall not be assigned to an object with a different essential type category	Required
M10.4	Error	D	Both operands of an operator in which the usual arithmetic	Required

Code review reference	Type	D/U	Description	Level
M10.5	Warning	D	<p>conversions are performed shall have the same essential type category</p> <p>The value of an expression should not be cast to an inappropriate essential type</p>	Advisory
M10.6	Error	D	The value of a composite expression shall not be assigned to an object with wider essential type	Required
M10.7	Error	D	If a composite expression is used as one operand of an operation in which the usual arithmetic conversions are performed then the other operand shall not have wider essential type	Required
M10.8	Error	D	The value of a composite expression shall not be cast to a different essential type category or a wider essential type	Required
E10.1	Error	D	When using operator '~' or '<<it>>' on 'unsigned char' or 'unsigned int', you should always cast returned value	Required
M11.1	Error	D	A function pointer should not be con-	Required

Code review reference	Type	D/U	Description	Level
M11.2	Error		verted to another type of pointer Conversions shall not be performed between a pointer to an incomplete type and any other type	Required
M11.3.1	Error		Casting an object pointer type to a different object pointer type should not occur	Required
M11.3.2	Error		Casting an object pointer type to a different object pointer type should not occur, especially when object sizes are not the same	Required
M11.3.3	Error		An object pointer should not be converted to another type of pointer	Required
M11.4	Warning		Casting a pointer type to an integer type should not occur	Advisory
M11.5	Warning		A conversion should not be performed from pointer to void into pointer to object	Advisory
M11.6	Error		A cast shall not be performed between pointer to void and an arithmetic type	Required
M11.7	Error		A cast shall not be performed between	Required

Code review reference	Type	D/U	Description	Level
M11.8	Error		<p>pointer to object and a non-integer arithmetic type</p> <p>Casting of pointers to a type that removes any const or volatile qualification on the pointed object should not occur</p>	Required
M12.1.1	warning		<p>Implicit operator precedence may cause ambiguity. Use parenthesis to clarify this expression</p>	Advisory
M12.1.2	warning		<p>Implicit bitwise operator precedence may cause ambiguity. Use parenthesis to clarify this expression</p>	Advisory
M12.1.3	warning		<p>Parenthesis should be used around expressions that are operands of a logical && or </p>	Advisory
M12.1.4	warning		<p>Parenthesis should be used around expression that is operand of 'sizeof' operator.</p>	Advisory
M12.3	warning		<p>The comma operator should not be used.</p>	Advisory
E12.1	warning		<p>The operator on a Boolean expression should be a logical operator (&& or !)</p>	Advisory

Code review reference	Type	D/U	Description	Level
E12.2	warning		Ternary expression '?' should not be used	Advisory
E12.3	error		Expressions should not cause a side effect assignment	Advisory
E12.4	error		The equal or not equal operator should not be used in floating-point expressions	Advisory
M13.3	Warning		a full expression containing an increment (++) or decrement (-) operator should have no other potential side effects other than that caused by the increment or decrement operator	Advisory
M13.4.1	Warning		Boolean expressions should not contain assignment operators.	Advisory
M13.4.2	Warning		The result of an assignment operator should not be used in an expression	Advisory
M13.6	Error		The operand of the sizeof operator shall not contain any expression which has potential side effects	Required
E13.1	Error		Boolean expressions should not contain side effect operators	Required
E13.2	Error		An expression that contains a side ef-	Required

Code review reference	Type	D/U	Description	Level
E13.3	Error		fect should not be used in the right-hand operand of a logical && or operator	Required
M14.1.1	Error		The function in the right-hand operand of a logical && or operator might cause side effects	Required
M14.2.1	Error		Floating-point variables should not be used to control a for statement	Required
M14.2.2	Error		Only loop counter should be initialized in a for loop initialization part	Required
M14.2.3	Error		In the 'update part' of a 'for statement', only 'loop counter' should be updated	Required
M14.2.4	Error		There should be one and only one loop counter for loop statement	Required
M14.3.1	Error		Loop counter of a 'for statement' should not be modified within the body of the loop	Required
M14.4	Error		Invariant Boolean expressions should not be used	Required
			Non-Boolean values that are tested	Required

Code review reference	Type	D/U	Description	Level
M15.1	Warning		against zero should have an explicit test The goto statement should not be used	Advisory
M15.2	Error		The goto statement shall jump to a label declared later in the same function	Required
M15.3	Error		Any label referenced by a goto statement shall be declared in the same block, or in any block enclosing the goto statement	Required
M15.4	Warning		There should be no more than one break or goto statement used to terminate any iteration statement	Advisory
M15.5	Warning		A function should have a single point of exit at the end	Advisory
M15.6.1	Error		The switch statement should be followed by a compound statement	Required
M15.6.1	Error		The switch statement should be followed by a compound statement	Required
M15.6.2	Error		The while statement should be followed by a compound statement	Required
M15.6.3	Error		The do..while statement should contain	Required

Code review reference	Type	D/U	Description	Level
M15.6.4	Error		a compound statement The for statement should be followed by a compound statement	Required
M15.6.5	Error		The if (expression) construct should be followed by a compound statement	Required
M15.6.6	Error		The else keyword should be followed by a compound statement	Required
M15.7	Error		All if ... else constructs shall be terminated with an else statement	Required
E15.1	Error		Do not use the continue statement	Required
E15.2	Error		Only one break statement should be used within a loop	Required
E15.3	Error		The return keyword should not be used in a conditional block	Required
E15.4	Error		The else keyword should be followed by either a compound statement or another if statement.	Required
M16.1	Error		All switch statement should be well formed	Required
M16.2	Error		A switch label shall only be used when	Required

Code review reference	Type	D/U	Description	Level
M16.3	Error		the most closely-en-closing copound statement is the body of a switch statement	Required
M16.4	Error		An unconditional break statement shall terminate every switch-clause	Required
M16.5	Error		Every switch state-ment shall have a de-fault label	Required
M16.6	Error		A default label appear as either the first or the last switch label of a switch statement	Required
M16.7	Error		Every switch state-ment shall have at least two switch-clauses	Required
E16.1	Error		A switch expression shall not have essentially Boolean type	Required
E16.2	Error		Case char value is applicable only if the switch statement value is plain character variable	Required
E16.3	Error		A constant should not be used as a switch expression	Required
M17.1.1	Error		The switch expres-sion should not have side effects	Required
			The function '%name %' should not have a	Required

Code review reference	Type	D/U	Description	Level
M17.1.2	Error		variable number of arguments The va_list, va_arg, va_start, va_end and va_copy functions of <stdarg.h> shall not be used	Required
M17.2.1	Error		Recursive functions are not allowed. The function '%name%' is directly recursive	Required
M17.2.2	Error		Recursive functions are not allowed. The function '%name%' is recursive when calling '%name%'	Required
M17.3	Error		A function shall not be declared implicitly	Required
M17.4	Error		All exit paths from a function with non-void return type shall have an explicit return statement with an expression	Required
M17.6	Error		The declaration of an array parameter shall not contain the static keyword between the []	Advisory
M17.7	Error		The value returned by function having non-void return type shall be used	Required
E17.1	Error		The number of arguments used in the call does not match the	Advisory

Code review reference	Type	D/U	Description	Level
E17.2	Error		number declared in the prototype Use the const qualification for parameter '%name%' which is pointer and which is not used to change the pointed object	Advisory
E17.3	Error		Function identifiers should always use a parenthesis or a preceding &	Advisory
M18.4	Error		The +, -, += and -= operators should not be applied to an expression of pointer type	Advisory
M18.5	Error		Declarations should contain no more than two levels of pointer nesting	Advisory
M18.7	Error		Flexible arrays members shall not be declared	Required
M18.8	Error		Variable-length array types shall not be used	Required
M19.2	Warning		The union keyword should not be used	Advisory
E19.1	Error		Structure or union types '%name%' should be finalized before the end of the compilation units	Advisory
M20.1	Warning		#include directive should only precede	Advisory

Code review reference	Type	D/U	Description	Level
M20.2	Error		by preprocessor directives or comments The ', or \ character and the /* or // character sequences shall not occur in a header file name"	Required
M20.3	Error		The #include directive shall be followed by either a <file-name> or a filename" sequence"	Required
M20.4	Error		A macro shall not be defined with the same name as a keyword %name%	Required
M20.5	Warning		#undef should not be used	Advisory
M20.6	Error		Token that look like a preprocessing directive should not occur withing a macro argument	Required
M20.7	Error		Expressions resulting from the expansion of macro parameters shall be enclosed in parenthesis	Required
M20.8	Error		The controlling expression of a #if or #elif preprocessing directive shall evaluate to 0 or 1	Required
M20.9	Error		All identifiers used in the controlling expression of #if or	Required

Code review reference	Type	D/U	Description	Level
M20.10	Warning		#elif preprocessing directives shall be #define'd before evaluation The # and ## pre-processor operators should not be used	Advisory
M20.11	Error		A macro parameter immediately following a # operator shall not immediately be followed by a ## operator	Required
M20.12	Error		A macro parameter used as an operand to the # and ## operators shall only be used as an operand to these operators	Required
M20.13	Error		A line whose first token is # shall be a valid preprocessing directive	Required
M20.14	Error	Error	All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if, #ifdef or #ifndef directive to which they are related	Required
E20.1	Error		Header file contents should be protected against multiple inclusions	Required
E20.2	Error		The # or ## pre-processor operator	Required

Code review reference	Type	D/U	Description	Level
E20.3	Error		should not be used more than once Missing argument when calling the macro	Required
E20.4	Error		Only use the 'defined' preprocessor operator with a single identifier	Required
E20.5	Error		Macro definitions or '#undef' should not be located within a block	Required
E20.6	Error		A C macro should only be expanded to a constant, a braced initialiser, a parenthesised expression, a storage class keyword, a type qualifier, or a do-while-zero block	Required
M21.1.1	Error		#define and #undef shall not be used on a reserved identifier or reserved macro name: Identifier %name% already found in <%libname%>	Required
M21.1.2	Error		#define and #undef shall not be used on identifier beginning with an underscore or on 'defined' keyword %name%	Required

Code review reference	Type	D/U	Description	Level
M21.2.1	Error		Declared identifier should not be a reserved identifier or reserved macro name: Identifier %name% already found in <%lib-name%>	Required
M21.2.2	Error		Declared identifier should not begin with an underscore or be 'defined' keyword %name%	Required
M21.3	Error		The memory allocation and deallocation functions of <stdlib.h> shall not be used	Required
M21.4	Error		The standard header file <setjmp.h> shall not be used	Required
M21.5	Error		The standard header file <signal.h> shall not be used	Required
M21.6.1	Error		The input/output library <stdio.h> shall not be used in production code	Required
M21.6.2	Error		The input/output library <wchar.h> shall not be used in production code	Required
M21.7	Error		The library macro or functions atof, atoi, atol and atoll of <stdlib.h> shall not be used	Required

Code review reference	Type	D/U	Description	Level
M21.8	Error		The library macro or functions <code>abort</code> , <code>exit</code> , <code>getenv</code> and <code>system</code> of <code><stdlib.h></code> shall not be used	Required
M21.9	Error		The library macro or functions <code>bsearch</code> and <code>qsort</code> of <code><stdlib.h></code> shall not be used	Required
M21.10	Error		The standard library time and date functions shall not be used	Required
M21.11	Error		The standard header file <code><tgmath.h></code> shall not be used	Required
M21.12	Warning		The library macro or function 'fexcept', <code>fegetexceptflag</code> , <code>feraiseexcept</code> , <code>fesetexceptflag</code> , <code>fetestexceptflag</code> , <code>FE_INEXACT</code> , <code>FE_DIVBYZERO</code> , <code>FE_UNDERFLOW</code> , <code>FE_OVERFLOW</code> , <code>FE_INVALID</code> or <code>FE_ALL_EXCEPT</code> should not be used.	Advisory
E21.1	Error		The variable 'errno' should not be used	Required
E21.2	Error		The macro 'offsetof' should not be used	Required
E21.3	Error		The library macro or function 'setjmp', <code>longjmp</code> , <code>sigsetjmp</code>	Required

Code review reference	Type	D/U	Description	Level
Rule U99.1	Error		p,siglongjmp' should not be used	User custom rule

Customizing the code review report

The default code review report is generated in an HTML format from a template named **misrereport.template** as that you can modify to customize the code review reports.

The code review HTML reports are generated from a template named **misrereport.template** that you can find in the following folder as a text file:

- On Windows: <installation_directory>\lib\reports
- On Unix: <installation_directory>/lib/reports

The template file uses the following JavaScript libraries:

- Bootstrap
- JQuery
- Font Awesome
- VisJS
- Chart

These libraries are not provided. An internet connection is required to open the report. If you don't have any internet connection, download the libraries (.css and .js files), copy them in the folder in which the report is saved, and modify the template file as follows:

Replace the following block of lines:

```
<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css"
  integrity="sha384-MCw98/SFnGE8fJT3GXwEOngsV7Zt27NXFoaoApmYm81iuXoPkFOJwJ8ERdknLPMO"
  crossorigin="anonymous">
<link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.5.0/css/all.css"
  integrity="sha384-B4dIYHKNBt8Bc12p+WXckhzcICo0wtJAoU8YZTY5qE0Id1GSseTk6S+L3BlXeVIU"
  crossorigin="anonymous">
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/vis/4.21.0/vis.min.css">
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/Chart.js/2.8.0/Chart.min.css">
...
<script src="https://code.jquery.com/jquery-3.3.1.slim.min.js"
  integrity="sha384-q8i/X+965Dz00rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8abtTE1Pi6jizo"
  crossorigin="anonymous"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.3/umd/popper.min.js"
  integrity="sha384-ZMP7rVo3mIyKv+2+9J3UJ46jBk0WLaUAdn689aCwoqbBJiSnjAK/l8WvCWPIpM49"
  crossorigin="anonymous"></script>
<script src="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/js/bootstrap.min.js"
  integrity="sha384-ChfqqxuZUCnJSK3+MXmPNIyE6ZbWh2IMqE241rYiqJxyMiZ60W/JmZQ5stwEULTy"
  crossorigin="anonymous"></script>
```

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/vis/4.21.0/vis.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/Chart.js/2.8.0/Chart.min.js"></script>
```

With the following one:

```
<link rel="stylesheet" href="./bootstrap.min.css">
<link rel="stylesheet" href="./all.css">
<link rel="stylesheet" href="./vis.min.css">
<link rel="stylesheet" href="./Chart.min.css">
...
<script src="./jquery-3.3.1.slim.min.js"></script>
<script src="./popper.min.js"></script>
<script src="./bootstrap.min.js"></script>
<script src="./vis.js"></script>
<script src="./Chart.min.js"></script>
```

The following sections give the list of elements that you can use in the raw data and the JavaScript functions to customize your report file.

Data format

The **misrareport.template** template consists of two sections:

- The HTML section that is common to all reports,
- A JavaScript section that sets tables depending on two variables that are initialized dynamically when the report is created:

```
var data = {{json}};           // the raw data, in json format
var d = new Date({{date}})    // the generation date
```

Raw data contains the following information at the top level:

- **output** is the name of the json file that contains the raw data
- **title** is the internal title of the report (displayed in the "crc" file format)
- **configurationTitle** is the title of the used configuration file
- **systemLevel** is the C level norm used. The possible values are "C90", "C90 and Normative Addendum 1", "C99" or "C11"
- **configuration** is the configuration file used to generate this report
- **date** is the generation date of raw data
- **nbAnalyzedFiles** is the number of analyzed files
- **nbFilesKO** is the number of files containing errors
- **nbFilesOK** is the number of files without errors
- **nbError** is the total number of all errors in all analyzed files
- **nbWarning** is the total number of all warnings in all analyzed files
- **files** is the array of **file element** (each one represents a physical file) or array of **deactivated element**
- **statistics** is the array of **rule statistics element**



Example:



```
{
  "output": "../build/fullreport_1.crc.json",
  "title": "HCL OneTest (TM) Embedded MISRA C:2012 Report using C90",
  "configurationTitle": "MISRA C:2012",
  "systemLevel": "C90",
  "configuration": "C:\\Program
    Files\\HCL\\HCLOneTest\\Embedded\\plugins\\Common\\lib\\confrule_2012.xml",
  "date": "Mon Oct 19 15:52:07 2020",
  "nbAnalyzedFiles": 5,
  "nbFilesKO": 4,
  "nbFilesOK": 1,
  "nbError": 49,
  "nbWarning": 68,
  "files": [
  ],
  "statistics": [
  ]
}
```

Each **file element** represents an analyzed source file. It contains the following information at the top level:

- **source** is the physical location of source file
- **fileDate** is the date of last editing of this source
- **nbErrorOrWarning** is the total of error or warning in this file
- **content** is an array of **rule element** (if the rule is directly raised at file level) or **function element**. It is always available but it can be empty (file with no function and with no error or warning)

Each **function element** represents a function. It contains the following information at the top level:

- **function** is the name of the function
- **kind** is the analysis result of this function. The possible values are 'Failed' or 'Passed'
- **content** is an array of **rule element** (rules that are raised inside this function). It is always available but it can be empty (function with no error or no warning)



Examples:

file element

```
{
  "source": "C:\\workspace\\project\\src\\core.h",
  "fileDate": "Mon Sep 07 10:31:50 2020",
  "nbErrorOrWarning": 25,
  "content": [
  ]
}
```

function element:

```
{
  "function": "win",
  "kind": "Failed",
  "content": [
  ]
}
```



```
]
}
```

Each **rule element** represents a triggered rule, justified or not. It contains the following information at the top level:

- **rule** is the name of the rule, corresponding to its label defined in the configuration file
- **group** is the family of this rule, it corresponds to the label of the rule's group that is defined in the configuration file
- **kind** is the severity of the rule. The possible values are 'error', 'warning' or 'info', depending on the error level in the configuration file and on the possible justification (the justified rules have an 'info' type value)
- **line** is the line of the current file where the rule was triggered
- **column** is the column of the current file where the rule was triggered
- **text** is the rule description. It is related to the rule text in configuration file
- **justification** is the justification text for the rule. This field is optional, and is present only if the rule is justified



Example:

```
{
  "rule": "M21.6.1",
  "group": "21- Standard libraries",
  "kind": "info",
  "line": 21,
  "column": 10,
  "text": "The input/output library <stdio.h> shall not be used in production code.",
  "justification": "This rule does not apply to the following line"
}
```

Each deactivated element represents a group of rules that is deactivated for a specific reason. It contains the following information at the top level:

- **deactivated_rules_by_user** is used for all the rules that are deactivated when it is used in the configuration file with the error set to level 0. This field is optional, it can be empty, or you can enter an array of **deactivated rule element**



Example:

```
{
  "deactivated_rules_by_user": [
  ]
}
```

- **deactivated_rules_by_test_option** is used for all the rules that are deactivated by using the "-test" option. This field is optional, it can be empty, or you can enter an array of **deactivated rule element**



Example:



```
{
  "desactivated_rules_test_option": [
  ]
}
```

Each **desactivated rule element** represents a deactivated rule for any reason. It contains the following information at the top level:

- **rule** is the name of the rule, it corresponds to the rule label that is defined in the configuration file
- **text** is the rule description, it corresponds to the rule text in configuration file

**Example:**

```
{
  "rule": "E15.3",
  "text": "The return keyword should not be used in a conditional block."
}
```

Each **rule statistics element** represents global statistics for the rule raised during test. It contains the following information at the top level:

- **ruleStatistics** is the array of the **statistic rule element**

**Example:**

```
{
  "rulesStatistics": [
  ]
}
```

Each **statistic rule element** contains a rule that was raised one or several times. It contains the following information at the top level:

- **rule** is the name of the rule. It corresponds to the rule label that is defined in the configuration file
- **kind** is the severity of the rule. The possible values are 'error' or 'warning' that correspond to the error level in the configuration file
- **occurrences** is the number of times that the rule was raised

**Example:**

```
{
  "rule": "M17.7",
  "kind": "error",
  "text": "When a function returns a value, this value should be used.",
}
```



```
"occurrences": 4
}
```

Javascript functions

You can find in the **misrareport.template** template a set of JavaScript functions.

Some of the helper functions simplify access to "raw data":

- **isFct**(element) checks whether an element is a function or not
- **isFile**(element) checks whether an element is a file or not
- **isFileInError**(element) checks whether an element is a file that contains an error or a warning
- **isFctPassed**(element) checks whether an element is a passed function or not
- **isFctFailed**(element) checks whether an element is a failed function or not
- **isRuleError**(element) checks whether a rule level is error or not
- **isRuleWarning**(element) checks whether a rule level is warning or not
- **isRuleInfo**(element) checks whether a rule level is an information or not
- **isRuleJustified**(element) checks whether a rule is justified or not

Other functions are used to display each section of the report:

- **emptyLine**() displays an empty line (helper function)
- **startFile**(element) is called at start of a file element.
- **endFile**() is called at end of a file element.
- **startFileRules**() is called at the beginning of a group of rules that is relative to a file. Used to display array headers
- **endFileRules**() is called at end of a group of rules relative to a file.
- **startFileFunctions**() is called at the beginning of a function
- **rule**(element) is called to display details of a raised rule.

The last section is a set of functions that is used to display summaries:

- **displayDesactivatedbytest**(elem) displays all deactivated rules by using the '-test' option
- **displayDesactivatedbyuser**(elem) displays all deactivated rules that are used in the configuration file
- **displayrulesstatistics**(elem) displays statistics for all rules that are raised during the test

The main algorithm dispatches the function calls by parsing the raw data.

Coupling Analysis

Coupling Analysis consists of Control Coupling and Data Coupling.

Overview

Control Coupling is defined as “the manner or degree by which one software component influences the execution of another software component” in the [Clarification of Structural Coverage Analyzes of Data Coupling and Control Coupling](#) document edited by the **Certification Authorities Software Team (CAST)**. The purpose is 'to provide a measurement and assurance of the correctness of these modules/components' interactions and dependencies'. Control Coupling is used to verify that all the interactions between modules have been covered by at least one test.

HCL OneTest™ Embedded introduces a new coverage level called “Control Coupling” for C language that consists in verifying that all the interactions between modules have been covered by at least one test. This new coverage level is implemented in HCL OneTest™ Embedded in two ways:

- Modules are compilation units, in this case:
 - Control Couplings are calls between two functions that are in two different compilation units.
 - Control Coupling is not a simple interaction. It is a control flow in the calling module that ends with an interaction with another module.
 - Groups of compilation units can be defined as a single module. This will increase the number of calls between modules but also increase the number of control flows in the calling modules.
 - The report contains a button to display:
 - All the Control Couplings (default option).
 - Only the shortest Control Couplings (only the last calls between modules are taken into account)
 - Only the longest Control Couplings (the sub-control flows are ignored)
- Modules are Functions, in this case:
 - Control Couplings are considered as all the calls between two functions, in the same compilation unit or not.
 - Each Control Coupling is only a call, and not a control flow as previously defined.

So, to identify the Control Couplings, HCL OneTest™ Embedded analyzes all the external calls between modules (definition of the modules could be different depending on the option) and statically identifies all the possible paths in the calling module that end with each external call, excluding the one that starts with a static function (ex: a function that can't be called from another module). This constitutes the set of Control Coupling of the application.

For each of them, HCL OneTest™ Embedded provides the following information:

- The calling modules.
- The complete control flow (example: the set of successive calls, the last one is the external call). If the option "module as function" is set, each control flow has two functions only.
- In case of option module as "compilation unit":
 - Is it the longest one that leads to this external call (it is not the longest when there is another Control Coupling that includes the current one).
 - Is it the shortest one that leads to this external call (it is not the shortest when there is another Control Coupling that is included by the current one).
- It is covered or not.

- The list of test cases that each Control Coupling covered.
- The list of requirements that are related to the test cases.

How Control Coupling Works

When an application node or a test is executed, the source code is instrumented by the Instrumentor (attolcc4 for C language) that produces a static file with the extension **.tsf** containing information on the Control Couplings. The resulting source code is then compiled, linked and executed and the Control Coupling feature outputs a dynamic file with the extension **.tgf**.

These 2 types of files are the input of the report generator that produces a report in HTML format (and optionally the raw data can be generated in a Json file). A template is provided for this generator. You can provide your own template to modify the report.

If the Control Coupling feature is used with unit testing feature, the report generator can take the **.tdc** files as input files. This allows to have also in the report the test cases that covered each Control Coupling and the associated requirements declared in the **.ptu** file. If not, the test cases are identified by their execution date, and there is no requirement.



Note:

To visualize your report in HCL OneTest™ Embedded for Eclipse IDE, if you are using the default browser option, be sure that JavaScript is enabled. Otherwise, you can choose another browser that is compatible with your version of JavaScript by changing it in **Window > Preferences > General > Web Browser**.

Set Control Coupling options

You can set options for Control Coupling to build your project in HCL OneTest™ Embedded for Eclipse IDE. Control Coupling feature must be enabled to be selected in the build settings before running the build.

Enable Control Coupling

- In the **Project Explorer**, right-click on the project and click **Properties**.
- In the **Properties** window, click **C C++ Build > Settings**.
- In the **Build Settings** tab, click **Settings > General > Selective instrumentation**.
- In the right pane, click the **Value** field in **Build options** and click ... to open the **Build options** window.
- In the Build options list, click **Performance Profiling** to enable the feature.

Control Coupling

In the **Project Explorer**, right-click on the project and click **Properties**, then click **C C++ Build > Settings**. In the **Build Settings** tab, under the **Coupling** menu, select **Control Coupling**.

From this setting page, you can change the following choices:

- **Trace file name (.tgf)**: Sets the name of the trace file dedicated to control coupling, click the edit button to change the name. By default, this name is the base name of the test with the extension **.tgf**.
- **Exclude libraries**: Include (No) or exclude (Yes) the control couplings must be included or excluded. that end with a call to a function that is not part of the application .
- **Report Template**: changes the template of the report generator. By default, this template is **ccreport.template**.
- **Module as**: Select the choice that corresponds the best to your definition of a module. A module can be defined as a function or a compilation unit. HCL OneTest™ Embedded offers two ways to interpret Control Coupling, depending on how the "module" in CAST-19 is interpreted:
 - **Module as function**: Each call between each function is considered as Control Coupling.
 - **Module as compilation unit**: Only the calls between two functions in two different compilation units are considered as Control Coupling. Moreover, the different called stacks in the calling module are also considered as different Control Couplings. With the previous option set, the user can group two or more compilation units in a single module (called component) in order to ignore the calls between these compilation units.
- **Components List**: Select a file that contains a list of components. This option is used only when the option "module as compilation unit" is selected. This file is in a JSON format and contains a list of components with their associated compilation units as follows:

```
{
  "component_name" : [ "file1", "file2",...],
  ...
}
```

Set Control Coupling Options

You can set the options for Control Coupling to build your project in HCL OneTest™ Embedded Studio.

Execute a build with Control Coupling

- In HCL OneTest™ Embedded Studio, open the Settings of the project and click the **Configuration Properties > Build > Build options** menu.
- In the right panel, click on the **Build options** and edit the options by clicking on the ... button.
- In the dialog window that shows up on the right, you can select the different tools that can be used for the build. Select **Ctrl Coupling analysis** to enable the control coupling feature.

Control Coupling options

Options for Control Coupling can be updated in the following menu of the settings: **Configuration Properties > Runtime analysis > Control coupling**

From this setting page, you can change the following choices:

- **Trace file name (.tgf)**: sets the name of the trace file dedicated to control coupling. By default, this name is the base name of the test with the extension **.tgf**.
- **Exclude libraries**: Include or exclude the control couplings that end with a call to a function that is not part of the application (sets the **-noccext** option of the report generator if it is set to yes).

- **Report Template:** changes the template of the report generator. By default, this template is **ccreport.template**.
- **Module as:** Select the choice that corresponds the best to your definition of a module. A module can be defined as a function or a compilation unit. HCL OneTest™ Embedded offers two ways to interpret Control Coupling, depending on how the "module" in CAST-19 is interpreted:
 - **Module as function:** Each call between each function is considered as Control Coupling.
 - **Module as compilation unit:** Only the calls between two functions in two different compilation units are considered as Control Coupling. Moreover, the different called stacks in the calling module are also considered as different Control Couplings. With the previous option set, the user can group two or more compilation units in a single module (called component) in order to ignore the calls between these compilation units.

Control Coupling Report

Depending on the build settings, from HCL OneTest™ Embedded V8.2.0, you can get a Control Coupling report with module as compilation unit as module or a Control Coupling report with module as Function as module as a result to a project build.

The default Control Coupling report is in HTML format. It is generated from a template named **ccreport.template** (for the option module as compilation unit"), or **ccfreport.template** (for the option "module as function") provided as text file that you can modify to customize the report. It uses 4 online JavaScript libraries:

- Bootstrap,
- JQuery,
- Font Awesome,
- VisJS.

These libraries are not provided. You need an internet connectivity when you open the report. If not, download the libraries (.css and .js files), copy them in the same folder than your report, and modify the template file as follows:

Replace the following lines:

```
<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css"
  integrity="sha384-MCw98/SFnGE8fJT3GXwEOngsV7Zt27NXFoaoApmYm81iuXoPkFOJwJ8ERdknLPM0"
  crossorigin="anonymous">
<link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.5.0/css/all.css"
  integrity="sha384-B4dIYHKNBt8Bc12p+WXckhzcICo0wtJAoU8YZTY5qE0Id1GSseTk6S+L3B1XeVIU"
  crossorigin="anonymous">
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/vis/4.21.0/vis.min.css">
...
<script src="https://code.jquery.com/jquery-3.3.1.slim.min.js"
  integrity="sha384-q8i/X+965Dz00rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8abtTE1Pi6jizo"
  crossorigin="anonymous"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.3/umd/popper.min.js"
  integrity="sha384-ZMP7rVo3mIykV+2+9J3UJ46jBk0WLaUAdn689aCwoqbBJiSnjAK/l8WvCWIPI49"
  crossorigin="anonymous"></script>
<script src="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/js/bootstrap.min.js"
  integrity="sha384-ChfqqxuZUCnJSK3+MXmPNIyE6ZbWh2IMqE241rYiqJxyMiZ60W/JmZQ5stwEULTy"
  crossorigin="anonymous"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/vis/4.21.0/vis.js"></script>
```

With the following ones:

```
<link rel="stylesheet" href="./bootstrap.min.css">
<link rel="stylesheet" href="./all.css">
<link rel="stylesheet" href="./vis.min.css">
...
<script src="./jquery-3.3.1.slim.min.js"></script>
<script src="./popper.min.js"></script>
<script src="./bootstrap.min.js"></script>
<script src="./vis.js"></script>
```

If you set a module as a compilation unit in the control coupling properties, you get a control coupling report with compilation unit in output of your project build. If you set a module as a function, you get a control coupling report with function as module. For more details on the control coupling settings, see [Set Control Coupling options on page 163](#) for HCL OneTest™ Embedded for Eclipse IDE. In a report with function as module, the report shows all the function calls (internal and external).

The Control Coupling report is made of three parts.

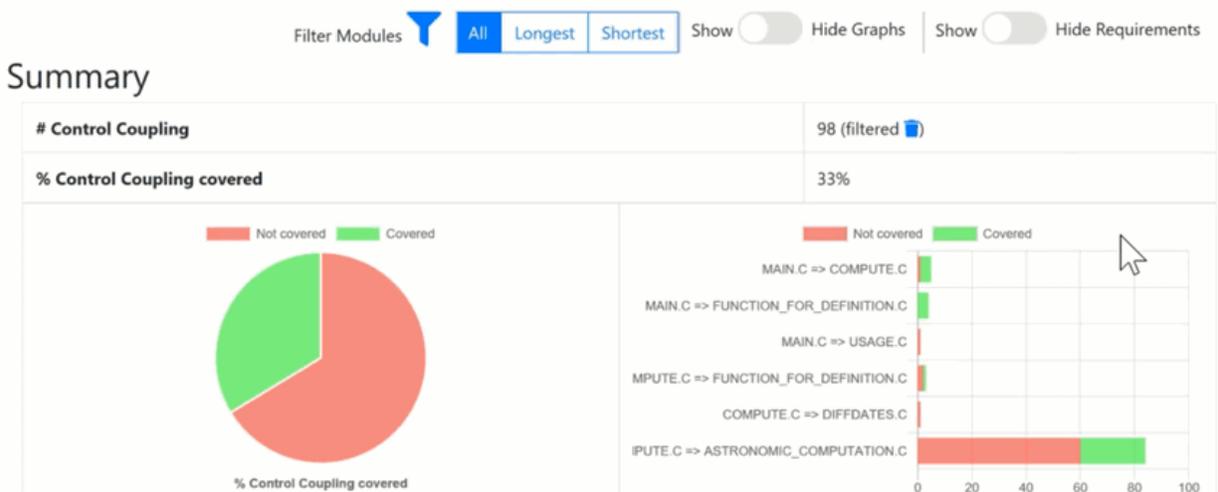
Summary

In the Summary section, you find the number of Control Couplings for your application that have been covered, given the information you provided and the percentage of Control Couplings that have been covered.

A graph displays the global percentage of covered and not covered control couplings for all the application.

The Summary table displays the following information:

- The percentage of Control Couplings of your application by pair of modules that have not been covered, given the information you provided.
- The percentage of Control Couplings that have been covered by pair of modules.



Details

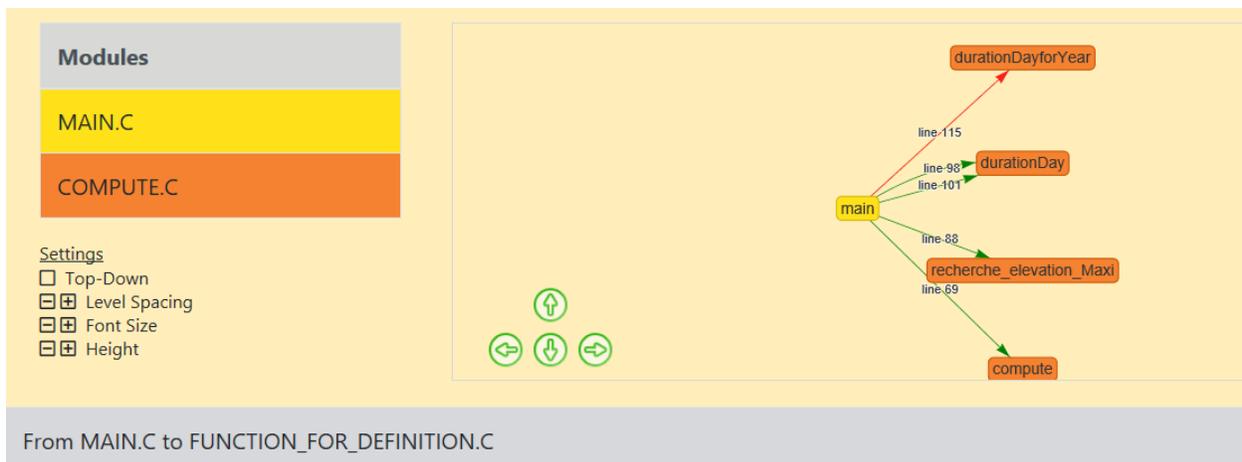
The **Details** table lists all the Control Couplings and displays for each of them the following information:

- The calling compilation unit.
- The control flow, for example: the successive calls in the module that end with the external call in the called module. Note that the called module is mentioned in the last function of the control flow. In case of option "module as function", this control flow contains only two functions.
- A check mark if it is a longest Control Flow (only if the option "module as compilation unit" is set).
- A check mark if it is a shortest Control Flow (only if the option "module as compilation unit" is set.)
- The list of test cases that covered this control flow. If the Control Coupling feature has been set with the unit testing feature, the test cases are the one in the **.ptu** files named as <service>/<test>.
- The associated requirements. If the Control Coupling feature has been set with the unit testing feature, the requirements are those that have been described in the **.ptu** files with the keyword REQUIREMENT for each test cases that covered this Control Coupling.
- A check mark if the control coupling has been covered.

Call Graph

For each compilation unit, a partial call graph displays all the functions as an interactive call graph from left to right or from top to bottom, depending on the selector button position on the top of the call graph.

You can select a control coupling in the table to highlight it in the call graph.



At the end of the report, a complete Call Graph displays all the functions calls.

Filters

Filters can be applied in the report with different buttons at the top:

- If the option "module as compilation unit" is set, you can choose first to display all Control Couplings, the longest (only the Control Couplings that have the longest control flow in the calling module) or the shortest (only the Control Couplings that have the shortest control flow in the calling module). The summary tables

and the details table are updated accordingly to your selection. This option applies to reports with compilation unit as module only.

- You can select the calling modules and the called modules. It filters the Control Couplings depending on their calling and called modules. The summary tables and the details table are updated accordingly to your selection.
- You can choose to display all the graphs or hide them in the report.
- You can show or hide the Requirements.

Customize Control Coupling Report

The Control Coupling report is created from a template called **ccreport.template** (if option "module as compilation unit" is set), or **ccfreport.template** (if option "module as function" is set) that you can find in the folder **<install>/lib/reports**.

This template is made of 2 parts:

- The HTML part that is the common part of all reports,
- A JavaScript part that sets the tables and call graph depending of 2 variables initialized dynamically when the report is creating:

```
var data = {{json}};    // the raw data
var d = new Date({{date}}) // the date of the generation
```

Raw data

Raw data is composed of 4 sections at the top level:

- A summary of the Control Coupling metrics:
 - **nbcc** is the number of Control Coupling found in the application,
 - **nbcovered** is the number of Control Coupling found in the application that have been covered by at least one test,
 - **nbccShortest** and **nbcoveredShortest** are the same for the shortest Control Coupling,
 - **nbccLongest** and **nbcoveredLongest** are the same for the longest Control Coupling,

- filtered is set to true if the report has been generated with a filter (shortest or longest),
- **filtered_longest** is set to true if the report has been generated with a filter longest (set only if filter is true).

```
"filtered": false,
"filtered_longest": false,
"nbcc": 112,
"nbcovered": 48,
"nbccShortest": 32,
"nbcoveredShortest": 25,
"nbccLongest": 58,
"nbcoveredLongest": 23
```

- The list of the modules, each of them has the following information:
 - **Name** is the short name of the C file,
 - **Fullname** is the name and path of the C file,
 - **uuid** is a unique identifier of the module,
 - **unknown** is set to true is the module is not part of the information you provided (there is only one unknown module that gathers all the call to functions that are not in the known modules),
 - **functions** is the list of the unique identifiers of functions of the module.

Modules are listed as hashmap with the uuid, as follows:

```
"modules": {
  "f5b5579edeaca82df478a6780c0c4c92": {
    "name": "USAGE.C",
    "fullname": "...",
    "uuid": "f5b5579edeaca82df478a6780c0c4c92",
    "unknown": false,
    "functions": [
      "ba9eb05ad703046fed306b4271b7ead7"
    ]
  }, ...
}
```

- The list of functions including following information:
 - **name** is the name of the C function,
 - **line** is the first line of the function in the module,
 - **id** is the number used in .tsf file to identify this function,
 - **stacksize** is the stack size computed during the execution if this option has been set (otherwise -1),
 - **uuid** is a unique identifier of the function,

- **module** is a unique identifier of the module in which the function is declared,
 - **calls** is the list of the calls in this function. Each of them have the following information:
 - **calling_uid** is the unique identifier of the calling function,
 - **called_uid** is the unique identifier of the called function,
 - **line** is the line number of the call in the module,
 - **col** is the column number of the call in the module,
 - **same_module** is set to true if the called function is in the same module that the calling function.
 - **level** is a number that represent the level of the function in the call graph, starting to 0.
 - **calledby** is the list of unique identifiers of functions that call this one.
- Functions are listed as hashmap with the uuid, as following:

```

"functions": {
  "ba9eb05ad703046fed306b4271b7ead7": {
    "name": "write_usage",
    "line": 9,
    "id": 1,
    "stacksize": -1,
    "uuid": "ba9eb05ad703046fed306b4271b7ead7",
    "module": "f5b5579edeaca82df478a6780c0c4c92",
    "calls": [
      {
        "calling_uid": "ba9eb05ad703046fed306b4271b7ead7",
        "called_uid": "7b6cd643b5b44e1e0510f30f62729eba",
        "line": 10,
        "col": 2,
        "same_module": false
      }
    ],
    "level": 1,
    "calledby": [
      "3fb6b20659c9b70fc6d01ba797abae1f"
    ]
  }, ...
}

```

- The list of the Control Couplings, each of them have the following information:
 - **calls** is the list of successive calls that composed this control coupling, each of them have the following information:
 - **calling_uid** is the unique identifier of the calling function.
 - **called_uid** is the unique identifier of the called function.
 - **isShortest** is set to true if the control coupling is a shortest one.
 - **isLongest** is set to true if the control coupling is a longest one.
 - **line** is the line number of the call in the module.

- **col** is the column number of the call in the module.
- **same_module** is set to true if the called function is in the same module that the calling function.
- **testcases** is the list of test cases that covered the control coupling, each of them have the following information:
 - **name** is the name of the test case.
 - **requirements** is the list of requirements that is covered by this test case.

Control couplings are listed as an array, as follows:

```
"controlcouplings": [
  {
    "isShortest": true,
    "isLongest": true,
    "calls": [
      {
        "calling_uuid": "3fb6b20659c9b70fc6d01ba797abae1f",
        "called_uuid": "0dd641fbc509e237cb0600f451d27d59",
        "line": 100,
        "col": 19,
        "same_module": false
      }
    ],
    "testcases": [
      {
        "name": "fct_8/1",
        "requirements": [
          {
            "name": "REQ_PTU_123"
          }
        ]
      }
    ]
  }
], ...
```

Overview

Data Coupling is defined as "the manner or degree by which one software component influences the execution of another software component" in the [Clarification of Structural Coverage Analyzes of Data Coupling and Control Coupling](#) document edited by the **Certification Authorities Software Team (CAST)**. The purpose is 'to provide a measurement and assurance of the correctness of these modules/components' interactions and dependencies'. Data Coupling is used to verify that all the global variables of the application under test have been consumed in read (also called *use*) and write (also called *def*) during the tests.

HCL OneTest™ Embedded introduces a new coverage level call "data coupling" for C language that consists to verify that all the global variables of the application under test has been consumed in read (also called *use*) and write (also called *def*) during the tests, as following:

- For each global variable, HCL OneTest™ Embedded identifies the *def* and *use*. Then it considers all the possible *def/use* pair as a data coupling.
- To cover a Data Coupling, i.e. a *def/use* pair, this *def* and this *use* must be executed from at least one test.

HCL OneTest™ Embedded provides a new interactive HTML-based report for Data Coupling.

To identify Data Coupling instances, HCL OneTest™ Embedded analyzes all the global variables of the application, where they are read and written. For one global variable, each pair of write and read constitutes an instance of Data Coupling.

For each data coupling, HCL OneTest™ Embedded provides the following information:

- The name of the global variable.
- The def position (file name, line, and column).
- The use position (file name, line, and column).
- The list of test cases that covered the Data Coupling.
- The list of requirements that are relative to these test cases.

How Data Coupling works

HCL OneTest™ Embedded identifies the position if the *def/use* using coverage information. When you select the Data Coupling option, some coverage options are set automatically: blocks, calls and conditions.

Coverage files (**.fdc** and **.tio**) are the input of the report generator that produces a report in HTML format (and optionally the raw data can be generated in a Json file). A template is provided for this generator. You can provide your own template to modify the report.

If the Data Coupling feature is used with unit testing feature, the report generator could take as input the **.tdc** files. This allows to have also in the report the test cases that covered each Control Coupling and the associated requirements declared in the **.ptu** file. If not, the test cases are identified by its execution date, and there is no requirement.

Set Data Coupling Options

You can set the options for Data Coupling to build your project in HCL OneTest™ Embedded for Eclipse IDE.

In the **Project Explorer**, right-click on the project and click **Properties**, then click **C C++ Build > Settings**. In the **Build Settings** tab, under the **Coupling** menu, select **Data Coupling**.

From this setting page, you can change the following choice:

- **Report Template:** You can change the template of the report generator. By default, this template is **ccreport.template**.

Data Coupling report

From HCL OneTest™ Embedded V8.2.0, you can get a HTML interactive Data Coupling report as a result to your project build.

The default Data Coupling report is in HTML format. It is generated from a template named **dcreport.template** provided as a text file that you can modify to customize the report. It uses four online JavaScript libraries:

- Bootstrap,
- JQuery,
- Font Awesome,
- VisJS.

These libraries are not provided. You need an Internet connection when you open the report. Otherwise, download the libraries (.css and .js files), copy them in the same folder as your report's, and modify the template file as follows:

Replace:

```
<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css"
  integrity="sha384-MCw98/SFnGE8fJT3GXwEOngsV7Zt27NXFoaoApmYm81iuXoPkF0JwJ8ERdknLPMO"
  crossorigin="anonymous">
<link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.5.0/css/all.css"
  integrity="sha384-B4dIYHKNBt8Bc12p+WXckhzcICo0wtJAoU8YZTY5qE0Id1GSseTk6S+L3B1XeVIU"
  crossorigin="anonymous">
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/vis/4.21.0/vis.min.css">
...
<script src="https://code.jquery.com/jquery-3.3.1.slim.min.js"
  integrity="sha384-q8i/X+965Dz00rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8abtTE1Pi6jizo"
  crossorigin="anonymous"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.3/umd/popper.min.js"
  integrity="sha384-ZMP7rVo3mIyKv+2+9J3UJ46jBk0WLaUAAdn689aCwoqBjBjSjAK/l8WvCWPIpM49"
  crossorigin="anonymous"></script>
<script src="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/js/bootstrap.min.js"
  integrity="sha384-ChfqxuxZUCnJSK3+MXmPNiY6ZbWh2IMqE241rYiqJxyMiZ6OW/JmZQ5stweULTy"
  crossorigin="anonymous"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/vis/4.21.0/vis.js"></script>
```

with

```
<link rel="stylesheet" href="./bootstrap.min.css">
<link rel="stylesheet" href="./all.css">
<link rel="stylesheet" href="./vis.min.css">
...
<script src="./jquery-3.3.1.slim.min.js"></script>
<script src="./popper.min.js"></script>
<script src="./bootstrap.min.js"></script>
<script src="./vis.js"></script>
```

The Report is made of three parts.

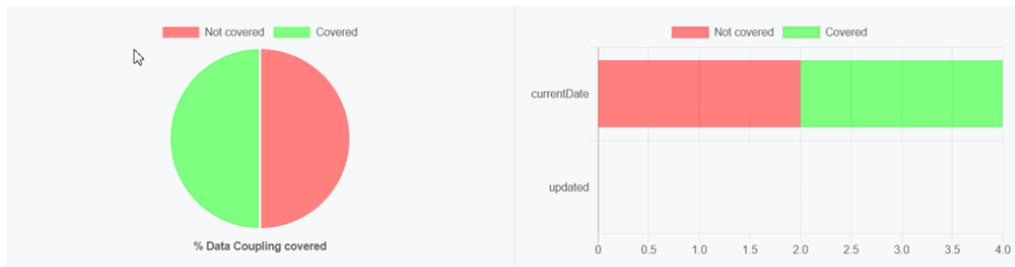
Summary

In the summary section, a table displays the following information:

- The number of global variables in your application.
- The number of Data Couplings in your application.
- The number and the list of global variables without Data Coupling. If you get this information, it means that HCL OneTest™ Embedded has identified global variables that are read but never written, or written but never read. This could be due to the fact that only a part of the application is analyzed.

Two charts display the following information:

- The percentage of Data Coupling in a pie graph.
- A two-colored horizontal graph that provides a number of covered and uncovered Data Couplings for each global variable.



Details

A table lists all the Data Couplings and displays the following information for each of them:

- **Variable:** The name of the global variable.
- **Def:** The Def position of the column: file name [line] and (column).
- **Use:** The Use position of the column: file name [line] and (column).
- **Test Cases:** The list of cases that covered the Data Coupling.
- **Requirements:** The list of requirements relative to these test cases.
- **Covered:** This option is checked if the Data Coupling has been covered.

They are grouped by global variables.

Details

Variable	Def	Use	Test Cases	Requirements	Covered
Global Variable ' currentDate '					
currentDate	main [MAIN.C] (33:2)	main [MAIN.C] (118:7)	• <i>no name #1</i> [Thu Nov 14 14:06:14 2019]		✓
currentDate	main [MAIN.C] (33:2)	main [MAIN.C] (123:2)	• <i>no name #1</i> [Thu Nov 14 14:06:14 2019]		✓
currentDate	DiffDays [DIFFDATES.C] (74:28)	main [MAIN.C] (118:7)			
currentDate	DiffDays [DIFFDATES.C] (74:28)	main [MAIN.C] (123:2)			
Global Variable ' updated '					
This variable ' updated ' is written but never read within the selected compilation units					

Call graph

The call graph displays all the global variables with their interactions with one or more functions of the application that read or/and write them.

- Incoming arrows are 'Def' (write).
- Outcoming arrows are 'Use' (read).

The arrows between them represent a 'Def' or a 'Use' (depending of the sense of the arrow). It is green if the corresponding 'Def' or 'Use' has been covered. These arrows are not representing Data Coupling. A Data Coupling instance is a couple of incoming and outcoming arrows that reach the same global variables.

Filters

Buttons can be used to filter different sections of the report.

- **Show/Hide Graph:** It is used to show or hide the call graph at the end of the report.
- **Show/Hide Requirements:** It is used to show or hide the **Requirements** column in the **Details** section of the report.

Customize Data Coupling Report

The Data Coupling report is based on a template called **ccreport.template** that you can find in the following folder:

Raw data

This template is made of 2 parts:

- The HTML part that is the common part of all reports,
- A JavaScript part that sets the tables and call graph depending of 2 variables initialized dynamically when the report is creating:

```
var data = {{json}};    // the raw data
var d = new Date({{date}}) // the date of the generation
```

Raw data is composed of 4 sections at the top level:

- A summary of the Data Coupling metrics:
 - **nbGlobalVariables** is the number of global variables found in the application.
 - **nbDefUses** is the number of Def/Use pairs found in the application.
 - **nbDefUsesCovered** Def/Use pairs found in the application that have been covered by at least one test.
 - **nbVariablesWithoutDefUse** is the number of global variables that have no Def/Use pairs in the application.
 - **variablesWithoutDefUse** is the list of global variables that have no Def/Use pairs in the application.

```
"nbGlobalVariables": 2,
"nbDefUses": 4,
"nbDefUsesCovered": 2,
"nbVariablesWithoutDefUse": 1,
"variablesWithoutDefUse": [
  "updated"
]
```

- The list of the modules, each of them has the following information:
 - **Name** is the short name of the C file,
 - **Fullname** is the name and path of the C file,
 - **uuid** is a unique identifier of the module,
 - **unknown** is set to true is the module is not part of the information you provided (there is only one unknown module that gathers all the call to functions that are not in the known modules),
 - **functions** is the list of the unique identifiers of functions of the module.

Modules are listed as hashmap with the uuid, as follows:

```
"modules": {
  "f5b5579edeaca82df478a6780c0c4c92": {
    "name": "USAGE.C",
    "fullname": "...",
    "uuid": "f5b5579edeaca82df478a6780c0c4c92",
    "unknown": false,
    "functions": [
      "ba9eb05ad703046fed306b4271b7ead7"
    ]
  },...
}
```

- The list of functions including following information:
 - **name** is the name of the C function,
 - **line** is the first line of the function in the module,
 - **id** is the number used in **.tsf** file to identify this function,
 - **stacksize** is the stack size computed during the execution if this option has been set (otherwise -1),
 - **uuid** is a unique identifier of the function,
 - **module** is a unique identifier of the module in which the function is declared,
 - **calls** is the list of the calls in this function. Each of them have the following information:
 - **calling_uuid** is the unique identifier of the calling function,
 - **called_uuid** is the unique identifier of the called function,
 - **line** is the line number of the call in the module,
 - **col** is the column number of the call in the module,
 - **same_module** is set to true if the called function is in the same module that the calling function.
 - **level** is a number that represent the level of the function in the call graph, starting to 0.
 - **calledby** is the list of unique identifiers of functions that call this one.
- Functions are listed as hashmap with the uuid, as following:

```

"functions": {
  "ba9eb05ad703046fed306b4271b7ead7": {
    "name": "write_usage",
    "line": 9,
    "id": 1,
    "stacksize": -1,
    "uuid": "ba9eb05ad703046fed306b4271b7ead7",
    "module": "f5b5579edeaca82df478a6780c0c4c92",
    "calls": [
      {
        "calling_uuid": "ba9eb05ad703046fed306b4271b7ead7",
        "called_uuid": "7b6cd643b5b44e1e0510f30f62729eba",
        "line": 10,
        "col": 2,
        "same_module": false
      }
    ],
    "level": 1,
    "calledby": [
      "3fb6b20659c9b70fc6d01ba797abae1f"
    ]
  },...
}

```

- The list of the control flows, each of them have the following information:
 - **stacksize** is the size computed for this control flow. This value is -1 if the tool was unable to compute.
 - **calls** is the list of successive calls that composed this Control Flow, each of them have the following information:
 - **calling_uuid** is the unique identifier of the calling function.
 - **called_uuid** is the unique identifier of the called function.
 - **line** is the line number of the call in the module.
 - **col** is the column number of the call in the module.

- **same_module** is set to true if the called function is in the same module that the calling function.
- **alternates** is a list of line and column if the function is called several times in this function
- **isRecursive** is set to true if a recursive call has been found in this control flow.
- **name** is the name of the test case.
- **missingFunctions** is the list of functions (name and unique identifier) in the control flow for which there is no stack size.

Control couplings are listed as an array, as follows:

```

"variables": [
  {
    "name": "currentDate",
    "line": 7,
    "moduleuuid": "e60218b872e86c7d154af4e306e9160a",
    "defs": [
      {
        "variablename": "currentDate",
        "linelocal": -1,
        "line": 33,
        "col": 2,
        "function": "main",
        "moduleuuid": "4306a1f82e1b1400a35d13ac6e2efce7",
        "isdef": true,
        "where": "bloc",
        "variabletype": "global",
        "covered": true
      },...
    ]
  },...
]
"uses": [
  {
    "variablename": "currentDate",
    "linelocal": -1,
    "line": 118,
    "col": 7,
    "function": "main",
    "moduleuuid": "4306a1f82e1b1400a35d13ac6e2efce7",
    "isdef": false,
    "where": "cond",
    "variabletype": "global",
    "covered": true
  },...
],...
"nbDefUses": 4,
"testcases": [
  {
    {
      "name": "fct_8/1",
      "requirements": [
        {
          "name": "REQ_PTU_123"
        }
      ]
    }
  },...
]
}

```

Application Profiling

Application Profiling is gathering the main features that provide profiling information at the application level: the Worst Stack Size feature and the Worst performance (coming soon) feature.

Worst Stack Size

HCL OneTest™ Embedded introduces the Worst Stack Size feature to compute an estimation of the maximum stack size of the application under test.

Overview

To implement this feature, HCL OneTest™ Embedded uses two mixed technologies:

- Static analysis that computes the call graph of the application (Example: all the calls between functions are analyzed and computed as a graph),
- Dynamic analysis that provides the stack size of each functions when executing them.

This information is used to provide an estimation of the worst stack size. This estimation is accurate under the following conditions:

- All the functions of the application should have been executed at least once in order to have the stack size for each of them.
- Your application should not have recursive calls, because the number of loops in the recursive calls being unpredictable, it is impossible to compute the stack size.
- If your application used libraries (Example: call functions for which we have not the source code), you should provide an additional file that gives an estimation of the stack size for each of them. These estimations do not need to be precise, but should be an upper bound of the exact stack size.
- If your compiler optimizes the Stack Size, you might have different Stack Sizes for the same function. In this case, the Worst Stack Size is computed with the maximum value found in the different runs.
- If your application is multi-threaded, you can provide the list of entry points so that HCL OneTest™ Embedded can calculate the worst total stack size and compare it to the maximum memory stack available on your target to produce a pass/failed verdict.

For the Worst Stack, HCL OneTest™ Embedded provides a brand-new interactive HTML-based report. This report identifies if one or more of these conditions are not met.

How Worst Stack Size Works

When an application node is executed, the source code is instrumented by the Instrumentor (atolcc4 for C language) that produces a static file with the **.tsf** extension that contains information on the functions (this file is common with Control Coupling feature). The resulting source code is then compiled, linked and executed and the Control Coupling feature outputs a dynamic file with the extension **.tzf**.

These 2 types of files are used in input of the report generator that produces a report in HTML format (and optionally the raw data can be generated in a Json file). A template is provided for this generator. You can provide your own template to modify the report. An addition file could be provided to this report generator in order to specify the stack size of the external functions.

**Note:**

To visualize your report in Eclipse, if you are using the default browser option, be sure that JavaScript is enabled. Otherwise, you can choose another browser that is compatible with your version of JavaScript by changing it in Window > Preferences > General > Web Browser.

Set Worst Stack Size Options

Enable Worst Stack Size

- In HCL OneTest™ Embedded Studio, open the settings of the project and click **Configuration Properties > Build > Build options**.
- Then, in the right panel, click on the value field of the **Build options** line and click the ... button to open the Build options editor.
- Then, a dialog window shows you on the right the different tools that you can select during the build. Select **Application profiling** to enable the Worst Stack Size feature.

Multi-thread option

The Multi-thread option for the Worst Stack Size feature can be configured in the following menu of the settings:

- Click **Configuration Properties > Runtime analysis > Multi-Threads**.
- In the right pane, click the ... in the value field of the **Entry points** option to open the **Entry points** editor.
- In the Entry points editor, enter the list of entry points for each thread and click **OK**.

Stack Size options

Options for the Worst Stack Size feature can be updated in the following menu of the settings: **Configuration Properties > Runtime analysis > Application Profiling > Stack size**.

In the setting page, you can change the following options:

- **Trace file name (.tzf)**: set the name of the trace file dedicated to worst stack size. By default this name is the base name of the test with the extension **.tzf**.
- **Report Template**: change the template of the report generator. By default this template is **wssreport.template**.
- **External functions stack size**: this is a file that contains the stack size of the external functions (generally functions that are in libraries and used by your application). The format of this file should be in Json, with the extension **.tzfe**, as follows:

```
[
  {"name":"printf", "stacksize":4},
  {"name":"sin", "stacksize":4},
  {"name":"cos", "stacksize":4},
  {"name":"tan", "stacksize":4}
]
```

- **Maximum Size:** Enter the maximum stack size in bytes that the application should not exceed.
- **Security:** Enter a percentage of available Stack Size for security.

If you provide the maximum Stack Size allowed and a percentage of available Stack Size for security, the report displays the total Stack Size and verify if this size does not go over the available Stack Size.

Set Worst Stack Size Options

Enable Worst Stack Size

- In the **Project Explorer**, right-click on the project and click **Properties**.
- In the **Properties** window, click **C C++ Build > Settings**.
- In the **Build Settings** tab, click **Settings > General > Selective instrumentation**.
- In the right pane, click the **Value** field in **Build options** and click ... to open the **Build options** window.
- In the Build options list, click **Application Profiling** to enable the Worst Stack Size feature.

Multi-thread option

- In the **Project Explorer**, right-click on the project and click **Properties**.
- In the **Properties** window, click **C C++ Build > Settings**.
- In the **Build Settings** tab, click **Settings > General > Multi-thread options**.
- In the right pane, click the ... in the value field of the **Entry points** option to open the **Entry points** editor.
- In the Entry points editor, enter the list of entry points for each thread and click **OK**.

Worst Stack Size options

In the **Project Explorer**, right-click on the project and click **Properties**, then click **C C++ Build > Settings**. In the **Build Settings** tab, under the **Application Profiling** menu, select **Stack Size**.

In the setting page, you can change the following options:

- **Trace file name (.tzf):** set the name of the trace file dedicated to worst stack size. By default this name is the base name of the test with the extension **.tzf**.
- **Report Template:** change the template of the report generator. By default this template is **wssreport.template**.
- **External functions stack size:** this is a file that contains the stack size of the external functions (generally functions that are in libraries and used by your application). The format of this file should be in Json, with the extension **.tzfe**, as follows:

```
[
  {"name": "printf", "stacksize": 4},
  {"name": "sin", "stacksize": 4},
  {"name": "cos", "stacksize": 4},
  {"name": "tan", "stacksize": 4}
]
```

- **Maximum Stack Size (byte):** Enter the maximum stack size in bytes that the application should not exceed.

- **Percentage of available Stack Size for security:** Enter a percentage of available Stack Size for security.

If you provide the maximum Stack Size allowed and a percentage of available Stack Size for security, the report displays the total Stack Size and verify if this size does not go over the available Stack Size.

Worst Stack Size Report

The default Worst Stack Size report is in HTML format. It is generated from a template named **wssreport.template** provided as a text file that you can modify to customize the report. It uses four online JavaScript libraries:

- Bootstrap,
- JQuery,
- Font Awesome,
- VisJS.

These libraries are not provided. You need an Internet connection when you open the report. Otherwise, you need to download the libraries (.css and .js files), copy them in the same folder as your report's, and modify the template file as follows:

Replace:

```
<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css"
  integrity="sha384-MCw98/SFnGE8fJT3GXwEOngsV7Zt27NXFoaoApmYm81iuXoPkFOJwJ8ERdknLPM0"
  crossorigin="anonymous">
<link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.5.0/css/all.css"
  integrity="sha384-B4dIYHKNBt8Bc12p+WXckhzcICo0wtJAoU8YZTY5qE0Id1GSseTk6S+L3BlXeVIU"
  crossorigin="anonymous">
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/vis/4.21.0/vis.min.css">
...
<script src="https://code.jquery.com/jquery-3.3.1.slim.min.js"
  integrity="sha384-q8i/X+965Dz00rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8abtTE1Pi6jizo"
  crossorigin="anonymous"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.3/umd/popper.min.js"
  integrity="sha384-ZMP7rVo3mIyKv+2+9J3UJ46jBk0WLaUAdn689aCwoqbBJiSnjAK/l8WvCWPIpM49"
  crossorigin="anonymous"></script>
<script src="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/js/bootstrap.min.js"
  integrity="sha384-ChfqqxuZUCnJSK3+MXmPNIyE6ZbWh2IMqE241rYiqJxyMiZ6OW/JmZQ5stwEULTy"
  crossorigin="anonymous"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/vis/4.21.0/vis.js"></script>
```

with

```
<link rel="stylesheet" href="./bootstrap.min.css">
<link rel="stylesheet" href="./all.css">
<link rel="stylesheet" href="./vis.min.css">
...
<script src="./jquery-3.3.1.slim.min.js"></script>
<script src="./popper.min.js"></script>
<script src="./bootstrap.min.js"></script>
<script src="./vis.js"></script>
```

The Worst Stack Size report is made of three parts.

Summary

Worst Stack Size per Entry Point table

Summary

	main	DiffDays
Worst Stack Size per Entry Point	1616 bytes	
# Control Flows	165	2
# Control Flows without Stack Size	111	2
# Recursive Computed Control Flows	0	0
# Functions	37	3
# Functions without Stack Size	14	3

The Summary section displays a table with the Worst Stack Size calculated by the tools, given the information you provided in the build settings. This number is provided in bytes.

The Worst Stack Size is given per entry point and per thread if you have entered the list of entry point threads of your application in the Build Settings. You can set the list of entry point threads of your application in the Build Settings.

The table displays the following information:

- The number of control flows found in your application. A control flow is a set of successive calls starting from an entry point (each function that is never called by another one is considered as an entry point) to a function with no call or to an external function.
- The number of control flows for which we have no estimation of the stack size. This happens when one of the functions in this control flow has not been executed or if it is an external function for which no estimation of the stack size is provided.

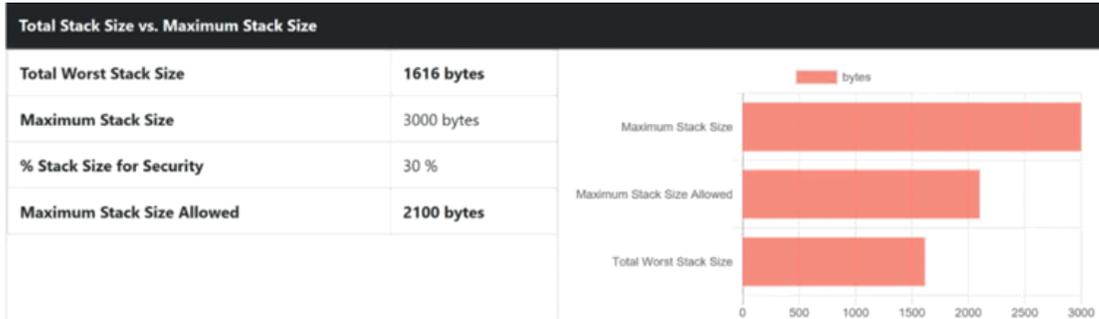
If this number is greater than 0, it is highlighted in red because there is no way to be sure that the worst stack size is really the worst regarding the missing information.

- The number of recursive control flows found in the application. If this number is greater than 0, it is highlighted in red because there is no way to be sure that the worst stack size is really the worst.
- The number of functions in your application.
- The number of functions without stack size estimation. These are the functions that have not been executed or the external functions for which we have not provided an estimation of the stack size. If this number is greater than 0, it is highlighted in red because we can't be sure that the worst stack size is really the worst.

The information is given for each entry thread.

If you don't provide the list of entry points in the build settings, the information is displayed only for the control flow and gives the Worst Stack Size.

Total Stack Size vs. Maximum Stack Size graph



If you provide in the Settings the list of entry points, optionally you can provide the maximum Stack Size allowed and a percentage of available Stack Size for security. In such case, the report displays the total Stack Size and verifies if this size does not go over the available Stack Size.

The **Maximum Stack Size** and **Percentage of available Stack Size for security** options can be set in the Build Settings.

In the report, you can compare the Stack Size or the sum of Stack Size with the maximum of Stack Size allowed and the percentage of available Stack Size for security if both options are provided in the settings.

In the toolbar that is under the graph, you can select the information to display or hide (all entry points, or for only one thread) and the number of control flows in the table. You can also show or hide the graph in the report from a button.

Details

The **Details** table lists by default the 10 first control flows with the biggest Stack Size and displays for each of them the following information:

- The control flow, for example, the successive functions starting from an entry point (any function that is never called by another one is considered as an entry point) to a function with no call, or to an external function. **Each function is identified by its name, its module (example: C file) between brackets, and by the line and column where this call to the next function calls appear in the code in parenthesis.**
- The estimation of the Stack Size. The information is blank if the tool has not been able to calculate the Stack Size for this control flow. In this case, the functions in the control flow that prevent us from computing the Stack Size are highlighted in red.

A drop down menu at the top of the table allows you to choose 10, 20, 30, 50, 100 or all the control flows to display.

Functions

The Functions table lists all the functions of your application, including external functions. The following information is provided for each function:

- The module name (i.e. the C file) where the function is saved.,
- The function name. This name is in red if there is no stack information for this function,
- The number of functions called in the current one.
- The Stack Size of the function in bytes.

Call Graph

The Call Graph part displays all the functions as an interactive call graph from left to right or from the top to the bottom, depending on the selector button position on the top of the call graph.

You can select a control flow in the table to highlight it in the call graph.

Customize the Worst Stack Size Report

The Worst Stack Size report is based on a template called **wssreport.template** that you can find in the folder **<install>/lib/reports**.

This template is made of 2 parts:

- The HTML part that is the common to all reports,
- A JavaScript part that sets the tables and call graph depending on 2 variables dynamically initialized when the report is created:

```
o var data = {{json}}; // the raw data

o var d = new Date({{date}}); // the date of the generation
```

Raw data

Raw data is made of four sections at the top level:

- A summary of the Worst Stack Size metrics:
 - **worstStackSize** is the worst stack size computed by the tools, depending on the information you provided. This number is provided in bytes.
 - **nbFlows** is the number of control flows found in your application. A control flow is a set of successive calls starting from an entry point (each function that is never called by another one is considered as an entry point) to a function without calls or to an external function.
 - **nbFlowsWithoutStack** is the number of control flows for which there is no estimation of the stack size. This happens when one of the functions in this control flow has not been executed, or if it is an external function for which we have not provided an estimation of the stack size.
 - **nbRecursiveFlows** is the number of recursive control flows found in the application.

- **nbFunctions** is the number of functions in your application.
- **nbFunctionsNoValue** is the number of functions without stack size estimation. These are the functions that have not been executed, or the external functions for which there is no estimation of the stack size provided.

```
"worstStackSize": 2139,
"nbFlows": 167,
"nbFlowsWithoutStack": 70,
"nbRecursiveFlows": 0,
"nbFunctions": 40,
"nbFunctionsNoValue": 10
```

The list of the modules, each of them has the following information:

- **name** is the short name of the C file,
- **fullname** is the name and path of the C file,
- **uuid** is a unique identifier of the module,
- **unknown** is set to true if the module is not part of the information you provided (there is only one unknown module that gathers all the function calls that are not in the known modules),
- **functions** is the list of the unique identifiers of functions of the module.

Modules are listed as Hashmap with the uuid, as following:

```
"modules": {
  "f5b5579edeaca82df478a6780c0c4c92": {
    "name": "USAGE.C",
    "fullname": "...",
    "uuid": "f5b5579edeaca82df478a6780c0c4c92",
    "unknown": false,
    "functions": [
      "ba9eb05ad703046fed306b4271b7ead7"
    ]
  },...
}
```

The list of functions, each of them have the following information:

- **name** is the name of the C function.
- **line** is the first line of the function in the module.
- **id** is the number used in **.tsf** file to identify this function.
- **stacksize** is the stack size computed during the execution if this option has been set (otherwise -1).
- **uuid** is a unique identifier of the function.
- **module** is a unique identifier of the module in which the function is declared.
- **calls** is the list of the calls in this function. Each of them have the following information:

- **calling_uid** is the unique identifier of the calling function.
- **called_uid** is the unique identifier of the called function.
- **line** is the line number of the call in the module.
- **col** is the column number of the call in the module.
- **same_module** is set to true if the called function is in the same module that the calling function.
- **level** is a number that represents the level of the function in the call graph, starting from 0.
- **calledby** is the list of unique identifiers of functions that call the function.

Functions are listed as hashmap with the uuid, as following:

```
"functions": {
  "ba9eb05ad703046fed306b4271b7ead7": {
    "name": "write_usage",
    "line": 9,
    "id": 1,
    "stacksize": -1,
    "uuid": "ba9eb05ad703046fed306b4271b7ead7",
    "module": "f5b5579edeaca82df478a6780c0c4c92",
    "calls": [
      {
        "calling_uid": "ba9eb05ad703046fed306b4271b7ead7",
        "called_uid": "7b6cd643b5b44e1e0510f30f62729eba",
        "line": 10,
        "col": 2,
        "same_module": false
      }
    ],
    "level": 1,
    "calledby": [
      "3fb6b20659c9b70fc6d01ba797abae1f"
    ]
  },
  ...
}
```

The list of the Control Flows, each of them have the following information:

- **stacksize** is the size of the stack computed for the control flow. This value is -1 if the tool was unable to compute it.
- **calls** is the list of successive calls that composed this control flow, each of them is including the following information:
 - **calling_uid** is the unique identifier of the calling function.
 - **called_uid** is the unique identifier of the called function.
 - **line** is the line number of the call in the module.
 - **col** is the column number of the call in the module.
 - **same_module** is set to true id. The called function is in the same module that the calling function.
 - **alternates** is a list of line & column in case of the calling function is called several times in this function.

- **isRecursive** is set to true if a recursive call has been found in this control flow.
- **missingFunctions** is the list of functions (name and unique identifier) in the control flow for which we have not the stack size.

Control flows are listed as an array, as follows:

```
"controlflows": [
  {
    "isRecursive": false,
    "stacksize": 2139,
    "calls": [
      {
        "calling_uuid": "3fb6b20659c9b70fc6d01ba797abae1f",
        "called_uuid": "0dd641fbc509e237cb0600f451d27d59",
        "line": 97,
        "col": 19,
        "same_module": false,
        "alternates": [
          {
            "line": 100,
            "col": 19
          }
        ]
      },...
    ],
    "missingfunctions": [],
  },...
],...
```

Testing software components

Component testing overview

Component testing provides a unique, fully automated, and proven solution for applications written in C/C ++, dramatically increasing test productivity.

Component testing in HCL OneTest™ Embedded supports C ++ ANSI C89 and C99.

A test case contains code blocks which call the methods under test and check blocks for variable checks, which verify that the values of a variable are within a specified set of requirements during the run. The test harness is the execution unit producing the executable. It contains the test cases, the source code under test and any files required to run the application, including libraries, stubs, and the runtime of the Target Deployment Port (TDP), which allows the test to run on a target platform. When you run the test harness, the code is compiled and tested. If any runtime analysis tools are engaged on the test harness, then the source code is also instrumented.

During the run, the test cases interact with the source code, producing test results, and if engaged, coverage and runtime analysis results.

After the run, you can open the test results in the test editor to check which test cases passed or failed, and to view the actual values obtained for each variable during the run.

A test suite is a list of test harnesses to run automatically. It generates an additional test suite report and a merged coverage report. The test suite can be executed in batch mode or interactively. Each test suite allows you to select one or two different configurations. When the two different configurations are selected, the tool generates the result report in comparison mode so that you can have the obtained values in both configurations.

Test assets overview

HCL OneTest™ Embedded several types of assets, which each describe different levels of the test environment.

These test assets include the following items:

- Test cases contain the verification actions for source code functions.
- Stubs are dummy components that allow you isolate the components under test or to replace components that do not exist.
- Test harnesses contain test cases and the associated source files and stubs required to run the test.
- Test suites contain multiple test harnesses that are run sequentially.

Test cases

A test case applies to a function and describes the checks that are performed against the variables contained in the component under test.

For each variable, array, or struct, you can specify an initialization value and an expected value. These values can be finite values, sets, or ranges, with multiple comparison types. When the test case is run, each check compares the expected value to the actual value and generates a *Passed* or *Failed* verdict.

The data used to specify initialization and expected values can be provided by native code, function calls, data pools or linked to a data dictionary. A data pool is a table, typically imported from a spreadsheet, containing multiple associated data sets. A data dictionary is a list of initialization and expected values for each variable type that can be reused by multiple test cases in the project.

You create a test case by selecting a function in the project explorer or the call graph. The test case is generated with the variables that are visible from outside the function. For each variable, a default check is added to the test case. You can use the test case editor to specify the initialization and expected values of each variable check.

Stubs

A stub is a dummy software component designed to replace a component that the component under test relies on, but cannot use in the test because it is not practical or available. A stub simulates the response of the stubbed component. Stubs can also be used to isolate the behavior of the component under test to provide more reliable test results or to simulate specific input values that cannot be practically simulated with the actual component. Stubs can be used in the following roles:

- Retrieving and storing input values to stubbed functions from a function under test.
- Assigning output values from the stubbed functions to a function under test.

Stubs generate passed or failed results based on the number of times that they are called.

You create a stub by selecting a function in the project explorer or the call graph. The stub is generated with the same interface as the stubbed function.

You can use the stub editor to specify the behavior of the stubbed function. You can also add additional blocks of code and conditions to structure the behavior of the test case.

Test configurations

The test configuration is an instance of a target deployment port (TDP) and its associated configuration settings. Configuration settings are the particular properties assigned to each test harness for a given test configuration.

For example, you can create a test configuration for each compiler involved in your project. If you are developing for an embedded platform, you can have one test configuration for native development on your Unix or Windows™ development platform and another test configuration for running and testing the same code on the target platform.

You can set up several test configurations based on the same TDP, but with different libraries, compilers or settings. The configuration settings allow you to customize test and runtime analysis options for each test asset in the project. You can reach the configuration settings for each test asset by right-clicking any node in the project explorer window and selecting **Properties > C/C++ Build > Settings** and **Build TDP** or **Build Instru.**

Test harnesses

The test harness contains all the test assets that are required to compile and run the test. These test assets include:

- Test cases
- Stubs
- Required source files, including:
 - Tested files: These are source files under test. The functions of these components are instrumented and integrated into the test harness.
 - Additional sources: These are dependency files that are added to test harness, but are not tested or instrumented. For example: resource files can be compiled inside a test harness by specifying them as additional files.
 - Linked files: These are source files that are linked with the test harness but are not tested or instrumented.
 - Libraries: These are libraries that are required for the link. For example: math libraries.

The test harness can also contain header code and global declarations that are required to run the test and instantiates the parameters of the test case.

You can use the test harness editor to add and remove test assets from the test harness and to graphically arrange the order in which the test cases are run. You can also add additional blocks of code and conditions to structure the behavior of the test harness.

To run a test harness, it must be associated with a test configuration. You can do this in a run configuration or in a test suite.

Test suites

A test suite contains multiple test harnesses that are run sequentially to provide global results for a project.

In the test suite, each test harness is associated with a test configuration (a TDP with associated configuration settings) and can be run a second time with another test configuration to provide comparison results. For example, this can be useful for certification purposes.

Creating test projects

In HCL® OneTest™ Embedded, projects are similar to C/C++ projects, but contain extra folders and a specific toolchain for component testing and runtime analysis.

About this task

HCL OneTest™ Embedded can only work with its own managed build toolchain. You can also import and convert existing Eclipse CDT projects to work with HCL OneTest™ Embedded.

To create a new project:

1. In the **C/C++** perspective, click **File > New > C Project/C ++ Project** . Or you can work in **Test RealTime** perspective, and click **File > New > Project**, and in the **New project** window, click **C/C++**, then **C Project** or **C ++ Project**.
2. In the **C Project** or **C ++ Project** wizard, type a **Project name**.
3. In **Project type**, select **Executable > Empty project** and in **Toolchains**, select HCL OneTest™ Embedded. Click **Next**.
4. On the **Select Configurations** page, ensure that the correct configuration is selected and click **Next**. You can select multiple configurations for the project.
5. On the **Target Deployment Port**, select a TDP that you want to use as the native target platform for your project.
6. Click **Finish**.

What to do next

After creating a project, you can import an existing C/C++ project into the product or use the Eclipse CDT tools to create a new project.

Related information

[Importing C projects on page 46](#)

[Creating test harnesses from the call graph on page 205](#)

[Creating a test case from the project explorer on page 195](#)

Test cases

Test case structure

The main objective of a test case is to define the variable checks that will compare the values obtained during the run with the expected values defined in the test case.

During the run, the test case performs a call to the C function using a set of initialization expressions and compares the return values with expected value expression. Each variable check is defined by:

- The name of the variable in the function.
- An initial expression: this is the expression of a value, or a set of values, that is submitted to the function during the test. You can express multiple initialization values, which causes multiple iterations of calls to the function under test.
- An expected expression: this is the expression of a value, or a set of values, that is compared to the actual value obtained during the test. Compliance with the expected expression produces either a *failed* or *passed* verdict for the test.

Activity diagram

The **Activity** diagram displays a flow chart describing the blocks that are required in the test case. If necessary, you can add and remove blocks, conditions and arrow lines to edit the activity diagram. The test case criteria are contained in one or several **Check** blocks.

The graphical flow chart allows you to add decision blocks and native C code to the test. For example, you can use decision blocks to run specific checks when a variable matches a specific value, or you can write a code block to define a counter and associate it with a decision block to create a loop.

Initialization and stubs

The **Init & Stubs** block summarizes the initialization values from all the check blocks and stub behaviors in the test case.

Code

By default, the **Code** block contains code that performs the call to the function under test.

The code block enables you to add native code to a test. This can be useful to run a specific portion of code in the middle of a test case. For example, you can change a hardware configuration before running a test or between two check blocks that verify the same function.

You can also write a code block to define a counter and associate it with a decision block to create a loop.

Variable checks

The main objective of editing a test case is to define the variable and structure checks. This is done in the **Checks** block by using the **Variable checks** table.

Variable initial expressions

The initial expressions are used to assign an initial value to a variable under test. The initial expression for each variable check is displayed in the test case.

Initial expressions can be among any of the following types:

- Numeric (integer or floating-point), character, or character string literal values, expressed using standard C syntax.
- Native constants, which can be numeric, characters, or character strings.
- **Series** of values, with a **From** and **To** value, and a **Step**.
- Global variables that are declared by the program under test.
- A null pointer.
- Arrays and structures, any of the above-mentioned expressions between braces (`{}`).
- C functions or expressions with one or more of the above elements combined using any operators and casting, with all required levels of parentheses.
- **Multiple** arbitrary values, which can be specified in the test case editor, randomly picked between a given range, or extracted from a datapool (read from a linked CSV file).
- **No Change**, which indicates that the test case does not set the value for the test.
- **No Dump** indicates that the variable initial value is not taken into account in the report, it is the same as 'unchanged'. This option is used so that the variable is not read during the initialization phase of the test case execution.

The data type of the variable defines what is a valid initial expression.

Initial expressions can be synchronized, which means that a list of multiple values for one variable will be synchronized with a matching number of values for another variable. See [Synchronizing multiple values on page 198](#) for more information.

Additional notes

The number of values inside an initialization expression is limited to 100 elements in a single variable.

If variables are used in the initialization expression, the test evaluates the initialization value with variable values from after the execution.

Related information

[Variable expected value expressions on page 193](#)

[Editing test cases on page 195](#)

Variable expected value expressions

The expected expressions are used to specify a test criteria by comparison with the value of a variable. The test receives a *passed* verdict when the actual obtained value matches the expected value expression.

The expected expressions can be among any of the following values:

- Numeric (integer or floating-point), character, or character string literal values. Strings can be delimited by single or double quotes.
- Native constants, which can be numeric, characters, or character strings.
- Ranges with lower and upper values and inclusive or exclusive bounds.
- Global variables that are declared by the program under test.
- A null pointer or a non-null pointer.
- Arrays and structures, any of the above-mentioned expressions between braces ('{}').
- C functions or expressions with one or more of the above elements combined using any operators and casting, with all required levels of parentheses. The + operator allows to concatenate character string variables.
- **No Check**, which specifies that no check is performed on that variable.
- **Same As Init**, which specifies that the expected variable equals the initialization expression.
- Data sets that are synchronized with a multiple initialization expression.

The data type of the variable defines the acceptable values for the expected value.

Numeric values can be associated with a comparison operator in the test case editor.

Expected expressions can be synchronized, which means that a list of multiple values for one variable will be synchronized with a matching number of values for another variable. See [Synchronizing multiple values on page 198](#) for more information.

Additional notes

Any integers contained in an expression must be written either in accordance with native lexical rules, or under the form:

- *hex_integer***H** for hexadecimal values. In this case, the integer must be preceded by 0 if it begins with a letter.
- *binary_integer***B** for hexadecimal values.

Ranges are not allowed for pointers.

The number of values inside an expected expression is limited to 100 elements in a single variable.

If variables are used in the expected expression, the test evaluates the initialization value with variable values from after the execution.

Euclidean divisions performed by the test case round to the inferior integer. Therefore, writing $-a/b$ returns a different result than $-(a/b)$, as in the following examples:

- $-(9/2)$ returns -4
- $-9/2$ returns -5

Related information

[Variable initial expressions on page 193](#)

[Editing test cases on page 195](#)

Creating a test case from the project explorer

You can create a test case from the project by simply selecting a source file or a function. Each test case focuses on a particular function.

To create a test case from the project explorer:

1. In the project explorer, right-click the project, source file, or a function, and click **New > Test Case**.

If you select a function, skip to step 3.

Result

The **Create Test Case** wizard opens.

2. On the **Select Test Assets** page, select the function or variable that you want to test and click **Next**.
You can choose to only display **Only functions**, **Only variables**, or you can filter the list by typing characters in **Filter**. Click **Clear** to clear the filter list.
3. On the Test Documentation page, you can edit the description of the test, and click **Next**.
The **Published description** contains information that you want to display in the test report. Use the **Internal notes** to add personal notes and comments that can be viewed and edited in the test editor.
4. On the **Test Case Location** page, select a folder and a type a file name for the test case and click **Finish**.
5. Choose whether you want to create a new test harness or use an existing one.

A test harness contains one or several test cases and is necessary to run the test.

Choose from:

- If you want to add the test case to an existing test harness, in the click **No**. You must edit the test harness to add the new test case.
- If not, click **Yes** and create a test harness with the **Create Test Harness** wizard.

Results

The test case and test harness are generated in the project explorer and the test case editor opens. [Editing test cases on page 195](#) for information about the test case editor.

Using the test case editor

Editing test cases

The test case editor enables you to visually design test cases associated with your source code and to create variable checks.

About this task

The test case editor is made of two panes:

- The **Activity** diagram displays a flow chart describing the blocks that are required in the test case. If necessary, you can add and remove blocks, conditions and arrow lines to edit the activity diagram. The test case criteria are contained in one or several **Check** blocks.
- The **Details** pane contains information about the selected block. For example, click the **Inits & Stubs** block to edit the initialization parameters, headers, and stubs required to run the test case.

 **Tip:** You can find where the edited file is located by clicking on the title of editor or on the header and selecting **Navigate > Show In > Project Explorer** . The explorer selects the current test case and expands automatically all parent nodes.

The main objective of editing a test case is to define the variable and structure checks in the **Check** block.

- In the project explorer, open a test case.
- In the **Activity** diagram, create the necessary blocks for the test case and connect them with connector lines. The default flow chart contains an **Init. & Stubs** block, followed by a **Code** block, followed by a **Check** block.
 - Click **Create Code Block**  or **Create Check Block**  buttons to create new blocks. Code blocks can be used to run portions of C code inside the test case. Check blocks contain the test criteria for the variables under test.
 - Click **Create Decision Block**  to make the execution of other blocks conditional. You can combine code blocks and decision blocks to create loops.
 - Click **Create Connector**  to connect new blocks in the diagram. Ensure that all blocks are properly connected.
- On the **Requirements** page, document requirements that are specific to your program or quality process. You can enter the name, a comment, and if a web page exists in your requirement management tools, enter the link to the web page that displays the requirement. You can also add requirements that come from a .cvs file.
 - To add a requirement, click  and enter a name for the requirement. You can modify the name. The table is editable, you can modify the name of the requirement, add a comment and add a link to a web page that is used as requirement in the table.
 - Click  to duplicate a requirement in the table.
 - Click  to view the requirement in a browser.
 - Click  to add a requirement from a list. This button is available only if you previously set the preferences to retrieve the requirements from a .cvs file. For more information, see Link Tests to Requirements. The requirements are filtered by name and comment. In the test reports, you can find the list of tests associated with the list of requirements.
- In the **Activity** diagram, select a check block. The **Checked Variables** table displays the variables and structures contained in the function under test.
- For each variable or structure, specify an initial value and an expected value. These values can be simple values, multiple values (ranges, series) or C structures.

- a. In the **Checked Variables** table, select a variable **Initial Expression** cell that you want to set and click the menu button (▼) to specify a single **Value**, **Multiple** values, a **Series**, whether to **Use Structure Fields**, or to apply **No Change** to the initial value. You can also choose **constructor** in the list, which means that you choose a constructor other than the default one. A constructor is a set of methods that has the same name as the class it belongs to. It is used to initialize the current instance and it is available only for a variable which is an instance of C++ class.
See [Variable initial expressions on page 193](#) for more information.
- b. To edit single values, multiple values, or series, type the values in the quick edition area line above the table. To specify structure values, edit the individual fields of the structure. To select a new constructor, click the menu button (▼) and select a value in the drop-down list.
The quick editor area adapts to the selected data type or entry mode.
- c. In the **Checked Variables** table, select a variable **Expected Expression** cell that you want to set and click the menu button ▼ to specify an expected value or range.
See [Variable expected value expressions on page 193](#) for more information.



Note: By default, the **Obtained Value** column displays the actual value for the variable obtained during the last run. Use the **Available Runs** list, located at the top of the test case editor, to display the actual values obtained during a specific run.

6. When you have finished editing the test case, click **Save** and close the test case.

Defining series value sets in initialization values

When a series is defined as the initial expression, the variable check generates one call to the function under test (or iteration) for each step in the series.

To create a series value set.

1. In the test editor, select a Check block to edit the variable checks.
2. In the **Initial Expression** column of one variable, click the menu button (▼) and select **Series**.

The quick edition area switches to series edition mode.

The image shows a series edition interface with the following elements:

- On the left, two small square icons: one with a blue circular arrow (undo) and one with a red 'X' (redo).
- The text "From" followed by an empty rectangular input box.
- The text "To" followed by an empty rectangular input box.
- The text "By Step" followed by an empty rectangular input box.

3. In the quick edition area, type the starting and end values of the series and the step.
The number of iterations is evaluated and displayed on the **Iterations** line of the test editor.
4. Press **ENTER** or click to apply the changes.

Specifying multiple value sets in initialization values

When a multiple initialization value is defined, the test generates one call to the function under test (or iteration) for each element in the set.

To create a multiple value set.

1. In the test editor, select a **Check** block to edit the variable checks.
2. In the **Initial Expression** column of one variable, click the menu button (☰) and select **Multiple**.
3. In the **Multiple Initial Expression** window, specify the number of values in the set, and click **OK**.

The number of iterations is evaluated and displayed at the top of the test editor and the quick edition area switches to multiple edition mode.

Enter initial and expected values for checked variables



4. In the quick edition area, type a value for each element in the set.
 - Press **TAB** to move to the next value in the set.
 - Click the Previous ◀ and Next ▶ buttons to scroll through the elements of the set.
 - You can increase and decrease the number of elements in the set.
 - Click the ... button to open the advanced editor window
5. Press **ENTER** or click  to apply the changes.

Synchronizing multiple values

In a variable check, when multiple values have been defined for a variable, you can create a synchronized set of values, with the same number of elements, which can be synchronized.

Before you begin

Synchronizing values requires that at least two sets of values (series, multiple, datapool) have been defined in the test case. Both value sets must have the same number of elements.

About this task

Without synchronization, each combination of all the values from all sets generates one call to the function under test, or *iteration*. The number of iterations is displayed in the test case editor. Using multiple sets can rapidly generate a large number of iterations, which can cause tests to run for long periods. For example, for the values in the following table, the test generates $5 \times 5 \times 2 = 50$ iterations.

Variable	Initialization value	Number of elements
a	[0.0, 1.0, 2.0, 3.0, 4.0]	5
b	[0.0, 0.1, 5.0, 10.0, 10.1]	5
c	[0, 1]	2

When two or more sets are synchronized, elements of each set are run together. In the previous example, if the initialization values for a and b are synchronized, a=0 is called with b=0.0, a=2 is called with 0.1, and so on. The test generates $5 \times 2 = 10$ iterations.

Synchronizing variables enables you to run two or more sets of values in parallel, such as linked curves or sets of coordinates.

To create a synchronized multiple value set.

1. In the test editor, select a check block to edit the variable checks.
2. In the **Initial Expression** column of one variable, click the **Menu** button and select **Multiple**.
3. In the **Multiple Initial Expression** window, select **Synchronized with** and select the variable which is initialized with another multiple set.

The number of iterations is evaluated and displayed on the **Iterations** line of the test editor and the quick edition area switches to multiple edition mode.

Enter initial and expected values for checked variables



4. In the quick edition area, type a value for each element in the set.
 - Press **TAB** to move to the next element in the set.
 - Click the **Previous** ◀ and **Next** ▶ buttons to scroll through the elements of the set.
 - Click the ... button to open the advanced editor window. The advanced editor provides an expanded table view of the values.
5. Press **ENTER** or click  to apply the changes.

Defining ranges in expected values

When a range expected expression is defined, the test checks that the obtained value is within the bounds of the range.

To create a range expected expression.

1. In the test editor, select a check block to edit the variable checks.
2. In the **Expected Expression** column of one variable, click the menu button (☰) button and select **Range > Native Expression**.

The quick edition area switches to range edition mode.



3. In the quick edition area, type the lower and upper bound values for the range and click the [and] buttons to set each bound as inclusive or exclusive.
4. Press **ENTER** or click  to apply the changes.

Defining a synchronized expected value

When a multiple initialization value is defined, you can specify a synchronized set of expected values. The test checks that for each initialization value element in the multiple set, the obtained result matches the corresponding element in the synchronized expected value set.

To create a multiple value set.

1. In the test editor, select a Check block to edit the variable checks.
2. In the **Expected Value** column of one variable, click the menu button (▼) and select **Synchronized**.
3. In the **Multiple Initialization Expression** window, specify the number of values in the set, and click **OK**.

The quick edition area switches to multiple edition mode.

Enter initial and expected values for checked variables



4. In the quick edition area, type an expected value for each element in the set.
The number of synchronized expected values matches the number of multiple initialization values.
 - Press **TAB** to move to the next value in the set.
 - Click the **Previous** ◀ and **Next** ▶ buttons to scroll through the elements of the set.
 - You can increase and decrease the number of elements in the set.
 - Click the ... button to open the advanced editor window
5. Press **ENTER** or click  to apply the changes.

Using values from a data pool

Data pools contain a series of values, or data patterns, that can be used as initialization or expected values for use in a test case or the data dictionary.

Before you begin

The values contained in the data pool must match the type of the variable that you want to initialize.

About this task

Data pools do not import the data contained in a CSV file. When a CSV file is updated externally, any tests that refer to the data pool will use the data contained in the updated CSV file.

To use values from a data pool:

1. In the test editor, select a **Check** block to edit the variable checks.
2. In the **Initial Expression** column of one variable, click the menu button (▼) and select **Data pool**.
3. In **Data pool**, select a data pool that is in the project.
Values number indicates the number of rows contained in the CSV table.
4. In **Column**, select the column number of the data set that you want to use to initialize the variables and click **OK**.

Results

The number of iterations displayed in the test case editor is updated to incorporate the number of values of the data pool (or rows in the CSV file).

Related information

[Creating data pools on page 202](#)

[Generating 2D and 3D chart data on page 924](#)

Using the data dictionary

Data dictionary overview

The data dictionary contains data sets, which are user-defined sets of values, multiples, ranges, series, or structures that can be applied to initialization and expected values.

The data dictionary enables you to create, modify and reuse data sets in variable checks of the same type throughout your project. You can also export data dictionaries, import them into other projects, or share them with a team.

For example, if your application frequently uses values representing the speed of a vehicle, you can predefine a data set **speed** in the data dictionary, which will use a range from 0 to 200 kilometers per hour with a step of 20. You can then apply this data set to any variable check in your project that represents speed.

The data dictionary maintains links between the data set and the variables that are linked to it. Variables that are linked to a data set in the data dictionary are highlighted in green in the test editor.

When you modify an initial or expected value that is linked to the data dictionary, the changes automatically affect the data set stored in the data dictionary and any other variables that are also linked to the data set.

The Data Dictionary view

The Data Dictionary view lists the data sets that you have created. Each data set has a name, a type and a set of initial and expected values.

You can edit data sets in the data dictionary. Any changes to the initial or expected values affect the variable checks in the same project that are linked to the data set.

If you delete a data set from the data dictionary, all variable checks that are linked to the data set retain the last known values, but the links are removed.

Adding data sets to the data dictionary

Data sets are user-defined values that can be used as initial values or expected values in variable checks.

About this task

Data sets in the data dictionary can be linked to variables or structures in the test case editor. Once a data set is created, it can be linked to a variable or structure. When you update the data set of a variable check that is linked to the data dictionary, all other variable checks linked to the same data set are updated.

To add and edit data sets:

1. In the variable check table of the test case editor, select a variable or a structure and specify its initial value and expected value.

For a structure, specify the initial values and expected values of its components.

2. Right-click the variable or structure and select **Add Initial Expression to Dictionary** or **Add Expected Expression to Dictionary**.

Alternatively, you can drag and drop the variable or structure into the **Data Dictionary** view. You can also choose to only add the initial value or the expected value.

3. Type a name for the data set and click **OK**.

By default, the name of the variable or structure is used.

4. The variable or structure is listed in the **Data Dictionary** view and the value that is linked to a data set is highlighted in green in the test case editor.

Choose from:

- To dissociate a highlighted value in the test case editor from its data set in the data dictionary, right-click the value and select **Remove Link from Data Dictionary**.
- To associate a data set to an existing variable of the corresponding type, drag and drop the data set from the data dictionary on to the variable check in the test case editor.
- To delete a data set, right-click the data set in the data dictionary and select **Delete**. All variable checks that are linked to the data set retain the last values, but the links are removed.

Creating data pools

Data pools are links to a CSV file that is either in the file system or in the workspace.

About this task

The data pool contains series of values, or data patterns, that can be used as initialization or expected values for use in a test case or the data dictionary. The data pattern can also be used to produce a 2D or 3D chart with the test results.

Data pools do not import the data contained in a CSV file. When a CSV file is updated externally, any tests that refer to the data pool will use the data contained in the updated CSV file.

To create a datapool link to a CSV file:

1. Click **File > New > Other > HCL OneTest™ Embedded Data Pool**.
2. In the **Create Data Pool** wizard, click **Browse** to locate the CSV file, click **Open**, and click **Next**.
3. Select a folder in the workspace, type a name for the new data pool, and click **Finish**.

Result

The data pool editor opens.

4. In the data pool editor, select the **Import** parameters and **Separator options**.

Ensure that the selected language matches the locale settings used to generate the CSV file.

5. When the **Preview** area displays the correct data, save the data pool and close the editor.

Related reference

[Data pool editor reference on page 957](#)

Related information

[Using values from a data pool on page 200](#)

[Generating 2D and 3D chart data on page 924](#)

Test harnesses

Test harness structure

Test harnesses contains all the information required to compile and run a test. This includes, test cases, source files under test, stubs, and Target Deployment Port (TDP) configuration settings.

These test assets include:

- Test cases
- Stubs
- Required source files, including:
 - Tested files: These are source files under test. The functions of these components are instrumented and integrated into the test harness.
 - Additional sources: These are dependency files that are added to test harness, but are not tested or instrumented. For example: resource files can be compiled inside a test harness by specifying them as additional files.
 - Linked files: These are source files that are linked with the test harness but are not tested or instrumented.
 - Libraries: These are libraries that are required for the link. For example: math libraries.

To run a test harness, you must associate it with a test configuration from a run configuration for a standalone run or from a test suite if you want to run multiple test harnesses in a step. For more information, see [Running a test harness on page 211](#) and [Running a test suite on page 212](#).

You can use the test harness editor to add and remove test assets from the test harness and to graphically arrange the order in which the test cases are run. You can also add additional blocks of code and conditions to structure the behavior of the test harness.

Activity flow chart

The **Activity** area is located on the left of the test harness editor and contains a flow chart, which describes the behavior of the test harness. You can use this flow chart to define the order in which each test case is run.

The Activity flow chart can contain blocks of native code, which can be run before or between test cases. This can be useful for setting parameters or changing hardware to a specific configuration before running the test case.

You can also add decision blocks, making the execution of paths in the flow chart conditional.

The calls of test cases in a test harness are all taken into account by default when a test harness is run but you can deactivate a test case from the activity flow chart so that it is not taken into account in the generation.

Test harness details

In addition to the behavior of the test, the test harness includes information that is required to run the test. The **Details** section contains the following pages:

- **Context Definitions:** This page lists the source code assets that are required to run the test.
 - **Tested files:** These are source files under test. The functions of these components are instrumented and integrated into the test harness.
 - **Additional sources:** These are dependency files that are added to test harness, but are not tested or instrumented. For example: resource files can be compiled inside a test harness by specifying them as additional files.
 - **Linked files:** These are source files that are linked with the test harness but are not tested or instrumented.
 - **Libraries:** These are libraries that are required for the link. For example: math libraries.
- **Build Instrumentation:** This page contains the configuration settings that are used to build the test. These settings override the default configuration settings of the project.
- **Stubs:** This page specifies any stub files that simulate functions that are required by the functions under test. Stubs can be used to replace functions that are under development or not practical to use for testing. They can also be used to inject specific values or conditions into the test.
- **Requirements:** This page allows you document the requirements for the test case.
- **Header Code:** This page contains code that is run before the test cases are executed.
- **Declarations:** This page specifies global and local variables that must be declared in the test harness.

Creating test harnesses

Use the **New Test Harness** wizard to create new test harnesses. A test harness contains one or several test cases and is required to run the test, it also includes source files under test, stubs, and Target Deployment Port (TDP) configuration settings.

To create a test harness from the project explorer:

1. In the project explorer, right-click the project and click **New > Test Harness**.
If you select a function, skip to step 3.
Result
The **Create Test Case** wizard opens.
2. In the Create Test Harness wizard, select one or several test cases that you want to run together and click **Next**.
If no test cases exist, you can click **New Test Case** to create a new one.
3. On the **Test Harness Location** page, select the folder and name for the test harness and click **Finish**.

Results

The test harness is created in the specified folder and opens in the test harness editor.

Creating test harnesses from the call graph

The call graph provides a visual diagram that helps you select the functions that require testing in your project. You can use this diagram to create a test harness that contains a test case, stubs, and other test assets required to run the test.

1. In the project explorer, right-click the project, source file, or a function, and click **Open Call Graph**.

Result

The call graph displays a diagram representing the function calls in the selected component.

2. In the call graph toolbar, click **Create Test Harness** .

Result

This opens the **Test Creation Activity** view, which details the steps to create the test harness.

3. Under **Test Asset Selection**, select a function to test and click **Next**.
You can take advantage of the call graph display to locate the functions that are critical to your application.
4. If some functions require stubbing, under **Stub Selection**, select a function to simulate, and click **Next**. If the test does not require stubs, click **Next**.
See [Stubbing overview on page 220](#) for more information about stubs.
5. Under **Test Case Creation**, select a folder or create a new one, type a file name for the test case, and click **Next**.
6. Under **Test Harness Creation**, select a folder or create a new one, type a file name for the test harness, and click **Finish**.

The test harness contains one or several test cases and is necessary to run the test.

Results

The test cases, stubs, and test harness are generated in the project explorer and the test harness editor opens.

[Editing test harnesses on page 205](#) for information about the test harness editor.

Editing test harnesses

Use the test harness editor to edit test harnesses.

About this task

The test harness editor is made of two panes:

- The **Activity** diagram displays a flow chart describing the blocks that are required in the test harness. If necessary, you can add and remove blocks, conditions and arrow lines to edit the activity diagram. The test case criteria are contained in one or several **Check** blocks. You can also activate or deactivate a test case call in a test harness. Click a test case block in the **Activity** diagram, and click the  icon in the test case block to deactivate a test case call, or click the  icon to activate a test case call.

- The **Details** pane contains information about the selected block. For example, click a code block to edit the C/C++ source code that you want to insert into the test harness or click the black initialization circle to define the properties of the test harness. If you click a test case block in the **Activity** diagram, the pane displays all functions/methods and variables used by the test harness.



Tip: You can find where the edited file is located by clicking on the title of editor or on the header and selecting **Navigate > Show In > Project Explorer** . The explorer selects the current test harness and expands automatically all parent nodes.

To edit a test harness:

1. In the project explorer, open a test harness.
2. In the **Activity** diagram, create the necessary blocks for the test harness and connect them with connector lines.
The default flow chart contains a test case.
 - a. Click **Insert Test Case**  to add an existing test case into the test harness.
 - b. Click the **Create Code Block**  to add a block containing native C code that can be run between test cases.
 - c. Click **Create Decision Block**  to make the execution of other blocks conditional.
You can combine code blocks and decision blocks to create loops.
 - d. Click **Create Connector**  to connect new blocks in the diagram.
Ensure that all blocks are properly connected.
3. On the **Context Definition** page, ensure that all the source files and libraries required to compile and run the test harness are properly defined.
 - Tested files: These are source files under test. The functions of these components are instrumented and integrated into the test harness.
 - Additional sources: These are dependency files that are added to test harness, but are not tested or instrumented. For example: resource files can be compiled inside a test harness by specifying them as additional files.
 - Linked files: These are source files that are linked with the test harness but are not tested or instrumented.
 - Libraries: These are libraries that are required for the link. For example: math libraries.
4. On the **Build Settings** page, you can override the project the build settings.
See [Build configuration settings on page 947](#) for information about each of these settings.
5. On the **Stubs** page, specify any stub behaviors that you want to replace a function with.
See [Stubbing overview on page 220](#) for information about stub simulation.

6. On the **Requirements** page, document requirements that are specific to your program or quality process. You can enter the name, a comment, and if a web page exists in your requirement management tools, enter the link to the web page that displays the requirement. You can also add requirements that come from a .cvs file.
 - a. To add a requirement, click  and enter a name for the requirement. You can modify the name. Table is editable, you can modify the name of the requirement, add a comment and add a link to a web page that is used as requirement directly in the table.
 - b. Click  to duplicate a requirement in the table.
 - c. Click  to view the requirement in a browser.
 - d. Click  to add a requirement from a list. This button is available only if you previously set the preferences to retrieve the requirements from a .cvs file. For more information, see Link Tests to Requirements. The requirements are filtered by name and comment. In the test reports, you can find the list of tests associated with the list of requirements.
7. On the **Header Code** page, add native C source code that might be required run as a header for the test harness. For example, you could add code to initialize or set the hardware to a specific state before running the test cases.
8. On the **Declarations** page, add any global or local variables that need to be set before running the test harness.
 - a. Click **Add application variable** () to initialize a variable in the test harness.
Select one of the variables that are declared in the application.
 - b. Click **Add application variable to simulate** () to simulate a variable in the test harness.
Select one of the variables that are declared in the application.
 - c. Click **Add local variable** () to create a local variable for the test harness.
Specify a name and a type for the new variable.
9. In the **Details** pane, select the icon corresponding to the feature that you want to add to the settings of your project: Code coverage, Memory profiling, Performance profiling, Application profiling, Control coupling, Data coupling, Runtime tracing, Static metrics, and Code review.
10. When you have finished editing, save the test harness.



Note: You can run the test harness from the editor. For details, see [Running a test harness on page 211](#).

Test suites

Creating test suites

A test suite contains multiple test harnesses that are run sequentially to provide global results for a project. When you create a test suite, you select the test harnesses that will be used in the test suite run. You can also select test scripts that can be run from a test suite.

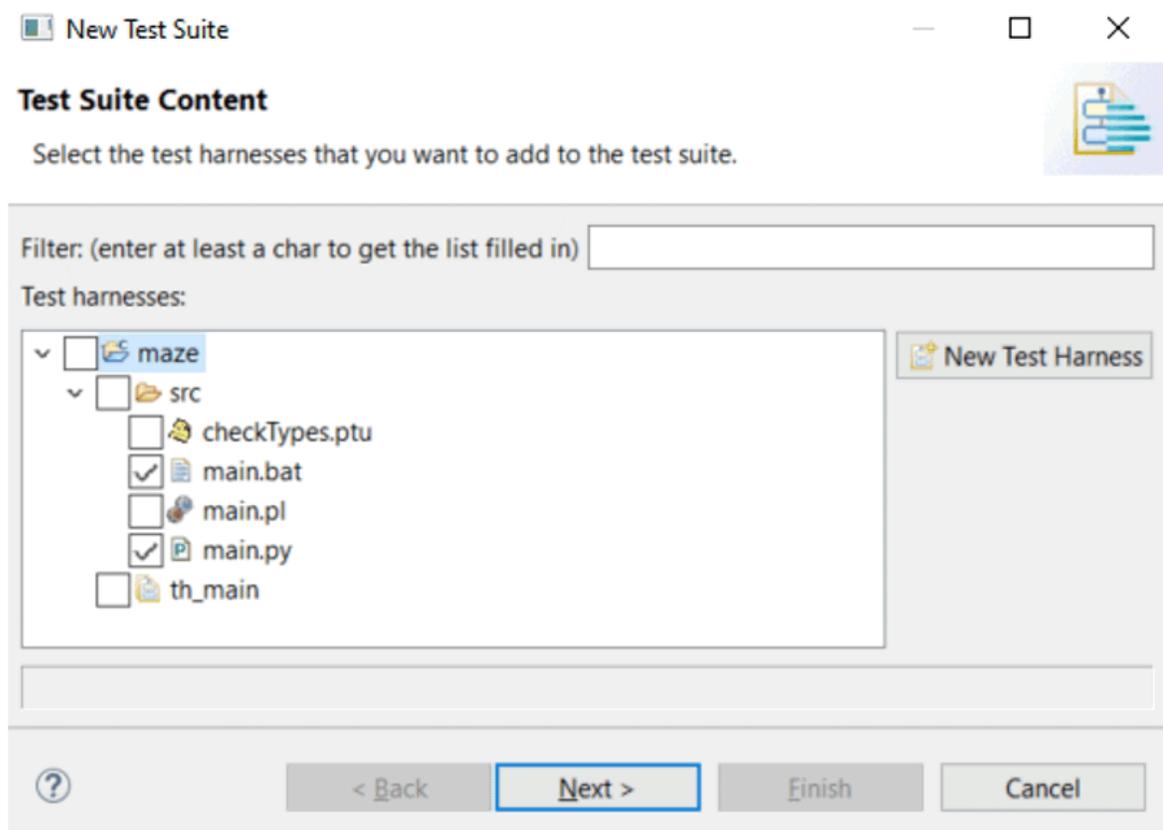
About this task

In the test suite, each test harness is associated with a test configuration (a TDP with associated configuration settings). In the test suite editor, you select the **main** test configuration, that is an instance of a target deployment port (TDP) and its associated configuration settings and usually carries the name of the TDP. A test harness can optionally be run a second time with another test configuration to provide comparison results. This can be useful for certification purposes or to compare the results of a test on two different hardware platforms.

From HCL OneTest™ Embedded for Eclipse IDE v8.3.0, the **Test Suite Content** wizard automatically displays all the files that are available in your project and that can be run with the selected test harnesses: **main** test configuration files for test harnesses, .ptu files for PTU test scripts, .otd files for OTD test scripts, .bat files for Windows scripts, .pl files for Perl scripts, .py files for Python scripts (.py files), or .shell for Linux scripts.

To create a test suite:

1. In the project explorer, right-click the project and select **New > Test Suite**.
2. In the **Create Test Suite** wizard, select the test harnesses that you want to run together and the test scripts located in your project. Then, click **Next**.



3. Select the **main** test configuration files for the test harnesses that are compatible with your test suite.
4. If you want to compare the test results with another test configuration, select **Compare with** and choose a secondary test configuration.

This will run the test suite twice, using both the main configuration and the secondary configuration. You can use this option to run the same test suite on a native platform and an embedded platform, to ensure that results are consistent.

5. Click **Next**.
6. Specify a location and file name for the test suite, and click **Finish**.

Results

The test suite is created in the selected location in the project, under the test suite folder in the **Project Explorer**.

Related information

[Running a test suite on page 212](#)

Configuring the Jenkins environment for running test suites

HCL OneTest™ Embedded for Eclipse IDE has command line interface that facilitates the integration of Jenkins in HCL OneTest™ Embedded. See [Running test suites from the command line on page 213](#)

About this task

First create a test suite in your project and add all the test harness that you want to execute.

To configure Jenkins:

1. On the Jenkins dashboard, click **Configure**.
2. Under **Build**, click **Add build step** where you want to insert your test execution.
3. Select **Execute Windows batch command** for Windows, or **Execute shell** for UNIX.
4. Setup your command as described here to execute your test suite: `rrteclipse -WORKSPACE= <your workspace> <your test suite>`.

Test configurations

Creating test configurations

Test configurations contain the settings required to apply a target deployment port (TDP) to your compiler, linker, debugger, and target deployment.

About this task

A test configuration can be understood as the base target deployment port settings, augmented with the various build and settings for the project.

To create a new test configuration:

1. In the project explorer, right-click the project and click **Properties**.
2. Expand **C/C++ Build**, select **Settings**, and click **Manage Configurations**.

Result

The **Manage Configurations** window for the project opens.

3. Click **New**.
4. Type a **Name** and **Description** for the new configuration.

Example

For example, use the name of the compiler or target platform.

5. Specify the source settings to use to create the new configuration.

Choose from:

- Select **Existing configuration** to base this configuration on one of the previously created configurations for this project.
 - Select **Default configuration** to base the configuration on the default configurations for the project.
 - Select **Import from projects** to copy the configuration from another project in the workspace.
 - Select **Import predefined** to copy the configuration from one of the predefined configurations provided with the product.
6. Click **OK**. If you want to use the new configuration, click **Set Active**.
 7. Click **OK** to close the **Manage Configurations** window.

What to do next

To make any changes to the test configuration, edit the **Build TDP** and **Build Settings** pages of the **Properties** window. See the Configuration Settings reference for more information.



Note: It is possible to rename test configurations. However, when the configuration is renamed, the previous directory of the configuration is not renamed and a new one is created. To build the new makefiles for the renamed configuration, you must edit the managed build to point to the source files that are in the new configuration directory.

Related information

[Switching test configurations on page 210](#)

Switching test configurations

Although a project can use multiple configurations, as well as multiple TDPs, there must always be at least one active configuration. You can switch from one configuration to another at any time, except during build activity.

About this task

The active configuration affects compiler and deployment options for each resource in the project.



Note: You can also run a test harness with two different test configurations by creating a test suite. See [Creating test suites on page 207](#).

To change the active test configuration:

1. In the project explorer, right-click the project and click **Properties**.
2. Expand **C/C++ Build**, select **Settings**, and click **Manage Configurations**.

Result

The **Manage Configurations** window for the project opens.

3. Select the configuration that you want to use to build and run the test and click **Set Active**.
4. Click **OK** to close the **Manage Configurations** window.

Related information

[Creating test configurations on page 209](#)

Running a test

Running a test harness

The test harness contains everything required to run the test.

About this task

The test harness associates the test cases with the source code and other required components to a test configuration. The test configuration is an instance of a target deployment port (TDP) with its association configuration settings.

To run a test harness:

1. In the project explorer, in the **Test Harness** folder, right-click the test harness and click **Run As > Run Test Harness**.
2. Alternatively, you can run a test harness from the test harness editor.



Note: In some environments, if you have installed the product in an existing Eclipse, the test result timestamps and verdicts are not properly displayed in the package explorer. To correct this, in the project explorer, click **View Menu > Customize View > Content** and ensure that only **Working Sets, Elements, CDT Elements, and Resources** are selected.



Note: To run multiple test harnesses in a step, you must create a test suite, select the test harnesses that will be run from the test suite and then run the test suite. For more information, see [Creating test suites on page 207](#) and [Running a test suite on page 212](#).

Results

The **Test Result** folder in the **Project Explorer** contains the test harness result file. To open the reports, right-click the Test result, select **Open with > HTML reports** and select the appropriate report.

Running a test suite

Test suites enable you to run multiple test harnesses or test scripts in a single step. You can update the list of test harnesses and test scripts to be run, and the build configuration from the **Test Editor** before running a test suite.

About this task

In the test suite, each test harness is associated with a test configuration (a TDP with associated configuration settings) and can be run a second time with another test configuration to provide comparison results. This can be useful for validation purposes.

To run a test suite:

1. In the project explorer, open the **Test Suite** folder and double-click the test suite to open the **Test Suite** editor.
2. In the **Test harness** section of the test editor window, to update the test harness list, you can:
 - a. Select or deselect test harnesses and test scripts (examples: .ptu, .otd,.py, .pl, .bat) that are available in your project
 - b. Add to the test suite other resources that are not displayed in the list by using one of the following procedures:
 - You can drag the test script files from the **Project Explorer** and drop them in the test harness list in the **Test harness** section of the test suite editor.
 - Click the **Add test harness** icon to select resources compatible with your project: supported scripts and test harnesses.
 - c. Click the **Delete** icon to remove a test harness or a test script file from the test suite.
 - d. Click the '**Up arrow**' and '**Down arrow**' icons to modify the order of resources in the list. The test harnesses and test scripts will be run in the order they are listed.
3. Save and click **Run**.



Result

After running the test suite, you can see the run result details in the **Run results for selected test harnesses** with the run status (success, failed, inconclusive) in the test suite editor.

The Test Result folder in the **Project Explorer** contains the test results for each test harness and for the generated test scripts contained in the test suite. You can produce a common, merged result file, containing the results of all the test harnesses of the test, see details in the Merging test suite results page.

4. To open reports, right-click a Test result, select **Open with > HTML reports** and select the appropriate report.
5. To order the test results, select **Sort result files by ascending date** in **Window > Preferences > > Navigator**



Note: In some environments, if you have installed the product in an existing Eclipse, the test result timestamps and verdicts are not properly displayed in the package explorer. To fix this issue, in the **project explorer**, click **View Menu > Customize View > Content** and ensure that only **Working Sets**, **HCL® OneTest™ Embedded Elements**, **CDT Elements**, and **Resources** are selected.

Running test suites from the command line

You can integrate test suites created with HCL OneTest™ Embedded for Eclipse IDE into your command line tool chain.

About this task

To run the test suite in command line mode, a Perl launcher script launches the Eclipse workbench silently. In this mode, the Eclipse workbench is not started and there is no user interaction. All information is output to the console.

The launcher script is located in the bin folder of the HCL OneTest™ Embedded installation directory. This folder directory is added to the *PATH* environment variable when the product is installed.

To run a test suite from the command line:

1. Close HCL OneTest™ Embedded for Eclipse IDE.
The Eclipse workspace must not be in use when you run the command line.
2. Type the following command line:

```
rtrteclipse [-WORKSPACE={workspace directory}] [testsuite_pathname [{testsuite_pathname}]]
[-BUILD_PROJECT={project_name | all}][-BINDIR={directory}][-TDPDIR={directory}]
[-REPORTDIR={directory}]
```

- *<workspace>* is the path to the workspace that contains the test suite. For example "C:\temp\workspace".
- *<test suite_pathname>* is the path and filename of the test suite in the workspace. You can run multiple test suites in the same workspace.
- *<bin directory>* optionally indicates the location of `eclipse.exe`. By default, the product uses:

```
"C:\Program Files\HCL\HCLOneTest\Embedded"
```

- *<tdp directory>* optionally indicates the location of the target deployment port directory. By default, the product uses:

```
"C:\Program Files\HCL\HCLOneTest\Embedded\targets"
```

- *<reportdir directory>* indicates the location where all the .xml report files are copied.

Example

For example:

```
° rtrteclipse -WORKSPACE={workspace} {testsuitePathFromWorkspace} [{testsuite}] [options]
```

```
° rtrteclipse {testsuiteWithAbsolutePath} [{testsuite}] [options] #. In this case, the workspace
```

and the directory where are located the test suites, are deducted from the first test suite path.

3. When the test is finished, start HCL OneTest™ Embedded for Eclipse IDE to view the results or open the directory reports with a web browser.

Test scripts files

Testing with PTU test scripts

You can add and configure PTU test scripts and execute them in a standalone mode or from a test suite in HCL OneTest™ Embedded for Eclipse IDE.

About this task

You must import a PTU file in a project to be able to execute the test script.

1. To import a PTU file, select **File > Import** and choose **General > File System** to select the file. You can import the file in any folder at any file structure level.
2. To configure a PTU test script file, see [Configuring .ptu or .otd test scripts on page 215](#).
3. To execute PTU file, use one of the following method:
 - a. To execute one PTU test script, right-click on a PTU file and choose **Run as > Script test file**.
 - b. Alternatively, select **Run Configurations** and **Test Script file** in the dialog box that opens. Right-click and select **New** to create a new launch configuration. Then proceed as follows:
 - Set a name to your launch configuration.
 - In the **Testing Script** tab, select your PTU file in **Select Application** panel.
 - Select your configuration in the **Configurations** panel and click **Run**.
 - c. To execute multiple PTU test scripts, create a test suite and select the PTU test scripts before running the test suite. For more details, see [Creating test suites on page 207](#) and [Running a test suite on page 212](#).

Result

A test report and runtime analysis reports are generated. The test result and the test script results are displayed in the **Test Result** folder. From these files, you can open the appropriate HTML reports.

4. To open the reports, select the report file corresponding to your last execution, right-click and select **Open With > Test Report**.

Testing with .otd test scripts

You can add, configure and execute .otd test scripts in a standalone mode or from a test suite in HCL OneTest™ Embedded for Eclipse IDE.

About this task

You must import the .otd file in a project and configure the test script to be able to execute it and see the results.

1. Select **File > Import** and choose **General > File System** and select the files that you want to import.



Note: You can import these files in any folder at any file structure level.

2. Follow the instructions that are described in the [Configuring .ptu or .otd test scripts on page 215](#) page to configure .otd test scripts.
3. To execute .otd files, use one of the following methods:
 - Right-click the .otd file and choose **Run as > Test script file**.
 - Alternatively, proceed as follows:
 - a. Select **Run Configurations**.
 - b. Select the .otd file under **Test Script file**, in the dialog that opens.
 - c. Right-click and select **New** to create a new launch configuration.
 - d. Set a name to your launch configuration.
 - e. Select your .otd file in **Select Application** panel, in the **Testing Script** tab.
 - f. Select your configuration in the **Configurations** panel and click **Run**. The reports are available into the **Test Result** folder.
 - To execute multiple .otd test scripts from a Test Suite, see [Running a test suite on page 212](#).

Result

A test report and runtime analysis reports are generated. The reports are available into the **Test Result** folder. From these files, you can open the appropriate HTML reports.

4. To open the reports, select the report file corresponding to your last execution, right-click and select **Open With > Test Report**.

Configuring .ptu or .otd test scripts

You can add additional options in a .ptu or .otd test script before executing.

About this task

A .ptu or an .otd file test script might need additional files and additional options before running that must be specified into the .ptu or the .otd file itself, as follows:

1. Enter instructions with specific lines starting with `--%f` and `--%o` located on top of the file, before the **HEADER** keyword setting.
2. In the line starting with `%o`, enter build options. Options format must follow the one used for **atolcc**.
3. In the line starting with `%f`, enter the list of additional source files that must be taken into account into the build.
4. Set relative paths to specify the test scripts location.
5. Set the PATH environment variable as follows to make the test scripts portable:
 - a. `${workspace_loc}/myproject/src/sub.c`
 - b. `$workspace_loc:/myproject/src/sub.c`
 - c. `$(project_loc)/src/sub.c`



Note: When the Path environment variable is configured and the test script run, the build automatically creates the three following environment variables:

- 'workspace_loc' corresponding to the workspace location
- 'project_loc' corresponding to the project location
- 'tstscript_loc' corresponding to the test script location

Testing with Python, Perl, Windows or Linux scripts

In HCL OneTest™ Embedded for Eclipse IDE, you can import and execute PTU and OTD test scripts but also other scripts such as Batch (Windows only), Shell (Linux only), Perl or Python.

Before you begin

To be able to run Python (.py files) scripts, you must install the PyDev plugin in Eclipse and configure Python Interpreter preferences. You can download the plug-in from this page <https://www.python.org/downloads/>.

About this task

This task applies to users who want to test with .bat files (Windows only), .pl files (Perl), .py files (Python), and .sh (Shell for Linux only). You must import the script files in a project to execute them.

1. To import a test script file, proceed as follows: select **File > Import** and choose **General > File System** to select the files. You can import these files in any folder at any file structure level.
2. To configure Python, Perl, Windows or Linux script files, see [Configuring Python, Perl, Windows or Linux scripts on page 217](#).
3. To execute the test script file, use one of the following methods:
 - a. To execute a test in a standalone mode in HCL OneTest™ Embedded for Eclipse IDE, right-click the script file and choose **Run as > Script test file**.
 - b. Alternatively, select **Run Configurations**. In the dialog that opens, under **Script test file**, select a script file. Right-click and select **New** to create a new launch configuration. Then proceed as follows:
 - Set a name to your launch configuration.
 - In the **Testing Script** tab, select your test script file in **Select Application** panel.
 - Select your configuration in the **Configurations** panel and click **Run**. The reports are available into the **Test Result** folder.
 - c. To execute test scripts from a Test Suite in HCL OneTest™ Embedded for Eclipse IDE, see [Running a test suite on page 212](#).

Result

A test report and runtime analysis reports are generated. The reports are available into the **Test Result** folder. From these files, you can open the appropriate HTML reports.

 **Tip:** The default execution timeout is set to 20 seconds but some scripts can take more time to execute. You can modify the script execution timeout from **Window > Preferences > > installation settings > installation settings**.

4. To open the reports, select the report file corresponding to your last execution, right-click and select **Open With > Test Report**.

Configuring Python, Perl, Windows or Linux scripts

To run Python, Perl, Windows or Linux scripts in a standalone mode, you must configure your scripts by using the java runtime that is delivered with HCL OneTest™ Embedded.

A java runtime named `ScriptReport-1.0.jar` is available in the `lib/java` folder when you install HCL OneTest™ Embedded. It is used by default to ensure that test script results are displayed directly in your test suite, and in your workspace after a manual refresh when the script is executed in a standalone mode. You must use some of the runtime commands in your `.py`, `.pl`, `.sh` or `.bat` files to customize your reports.



Note: Using runtime commands in scripts requires advanced user level.

initreport

```
initreport <logfile>
```

This command is used in a script when all result files are created by the script.

It initializes necessary resources needed to create a log file.

This log file will contain the list of all intermediate files that are needed to create a result file.

`<logfile>` parameter is the name of this log file.

By convention, the extension is **.xtp**.

This log file will be generated if this command is used in a script that is executed in a standalone mode.

If this command used in a script that is executed from a test suite, the name of the test suite is taken into account, and the parameter is ignored.

addreport

```
addreport <logfile> -path=<resource path> [-kind=<resource kind>]
```

`<resource path>` : Resource to be added to the report. The resource path can be a relative path that points to the script location, or an absolute path.

<resource kind>: Kind of resource (optional)

This command registers a resource to be added to a logfile.

If you add a folder as a resource, this folder will be the used as relative resource path.

For very advanced users: If you have a resource with an unusual extension, you can enter your own <resource kind> option.

The following table gives the list of file extensions that are recognized in HCL OneTest™ Embedded for Eclipse IDE and the corresponding <resource kind> options that must be entered in the script file.

File extension	<resource kind> option	Resource added to the log file
.ccf	CCF	ccf file
.crc	CRC	crc file for MISRA report
.crc.json	CRJ	crj file
.crx.html	CRX	MISRA code review report in html format
.dcp	DCP	dcp file
.dcp.json	DCJ	dcj file
.dcx.html	DCX	dcx file in html format for data coupling report
<executable without any extension>	EXE	
.exe	EXE	exe file
.fdc	FDC	fdc file for coverage report
<folder>	DIR	new reference for further relative path
.html	HTM	.html user file in html format
.log	LOG	log file
.met	MET	met file for metrics report
.o	OBJ object file	
.obj		
.req	REQ	req file
.rio	RIO	rio file

File extension	<resource kind> option	Resource added to the log file
.rtx	RTX	rtx file for charts report
.tdf	TDF	tdf file for trace report
.tgj.json	TGJ	tgj file
.tgx.html	TGX	tgx file for control coupling report in html format
.tio	TIO	tio file
.tpf	TPF	tpf file for memory profiling report
.tqf	TQF	tqf file
.tqf.html	TQX	tqx file for performance profiling report in html format
.tqf.json	TQJ	tqj file
.tsf	TSF	tsf file for trace
.tzf	TZF	tzf file
.tzf.json	TZJ	tzj file
.tzx.html	TZX	tzx file for stack profiling report
.xob	XOB	xob file
.xrd	XRD	xrd file for test report
.xtp	XTP	xtp file

genresult

```
genresult <logfile> [-path=<path>] [-name=<basename>]
```

<logfile> : Log file containing reports

<path> : Location where the result is generated (optional, the default value is <logfile folder>).

This command generates a result file from a logfile.

By default, it is the same location as the log file, with the same base name.

You can change this default behavior with optional parameters.

This command is supposed to be the last one, any resource added after this one will be ignored.

getConfig

```
getConfig <key> [<env key> <default value>]
```



Note: This command should be used by very advanced users only.

This command returns a key from the config file if it is executed from a test suite.

The command returns "`<not found>`" if the key is not found or executed in a standalone mode.

If the key is "`<not found>`", it returns an environment variable `<env key>`.

If the environment variable `<env key>` is not found, the command returns the `<default value>`.

This command is used to retrieve preferences from HCL OneTest™ Embedded for Eclipse IDE, when you call your script from a test suite.

If it is executed in a standalone mode, you can enter an environment variable as optional parameter or a default value if there is no environment variable.

For example, you can retrieve "`confrule`" file when you use Code Rule Checker.

You can retrieve the multiple keys that are existing in a generated file named "`envTestRTcc.pl`" that is located in the `%home%` folder.

You can find examples in the `ExamplesEclipse` folder under the product installation files. The folder contains a set of sources and three scripts (perl, bat and python). These scripts perform Code Rule Checker on sources and produce reports. They all use relative locations for sources and results so that they can be executed in a standalone mode or from a test suite, by using the runtime commands.

Stubbing functions

Stubbing overview

Stubs are simulations of actual functions, which can be used to isolate the function under test and to check that calls to the stubbed function are correctly formulated.

Stub simulation is based on the idea that certain functions are simulated and are replaced with stubs generated in the test harness. Stubs provide the same interface as the simulated functions, but the body of the functions is replaced with a basic behavior. From the point of view of other functions in the test harness, the stub looks identical to the actual function that it simulates.

Stubs can be used in the following roles:

- Retrieving and storing input values to stubbed functions from a function under test.
- Assigning output values from the stubbed functions to a function under test.

Stubs are described with the following elements:

- A variable array for the input parameters of the stub.
- A variable array for the output parameters of the stub.
- A body declaration for the stub behavior.

To create a stub, the source code of the stubbed function must be included in the project. HCL OneTest™ Embedded analysis the prototype of the stubbed function to generate a stub with the same interfaces. Once the stub is created, you use the stub editor to define the stub checks, which verify that each parameter in the call to the stubbed function matches an expected expression.

Stub checks are based on the sequential number of the call, which typically reflects the iteration of the calling function in the test case. The sequential call number is expressed as a range. For example a stub check for a parameter *a* can be set to match an expected expression *x* for the first 10 calls received by the stub (range 0 to 10), an expression *y* for the 11th to 20th calls (range 10 to 20), and an expression *z* for any following calls (range *Others*).

Stub expected value expressions

The expected expressions are used to specify a test criteria by comparison with the value of a call parameter received by a stub. The test receives a *passed* verdict when the actual obtained value matches the expected value expression.

The expected expressions for a stub can be among any of the following values:

- Numeric (integer or floating-point), character, or character string literal values. Strings can be delimited by single or double quotes.
- Native constants, which can be numeric, characters, or character strings.
- Ranges with lower and upper values and inclusive or exclusive bounds.
- Global variables that are declared by the program under test.
- A null pointer or a non-null pointer.
- Arrays and structures, any of the above-mentioned expressions between braces ('{}').
- C functions or expressions with one or more of the above elements combined using any operators and casting, with all required levels of parentheses. The + operator allows to concatenate character string variables.
- **No Check**, which specifies that no check is performed on that variable.
- **Same As Init**, which specifies that the expected variable equals the initialization expression.
- Data sets that are synchronized with a multiple initialization expression.

The data type of the variable defines the acceptable values for the expected value.

Numeric values can be associated with a comparison operator in the stub editor.

Stub return value

Return values are used for parameters and functions if a return value is defined in the signature of stubbed function. A special line named *return* in the parameter table is added to define the value for the return value of the function.

A return value can be defined for output parameters or input/output parameters. Change this setting in the **Mode** column. The return value is a C native expression as numeric, character, or string...

The function's return value can be replaced by a special user source code. In this case, write the appropriate C source code and add the return statement so that the function returns a value to the calling expression. To activate this feature, select the return line and click the **Use source code rather than return type** tool button. The user source code panel is activated and the C source code can be added.

Stub memory usage

For each STUB, the test harness allocates memory for the following tasks:

- Storing the expected expression of the input parameters during the test.
- Storing the obtained value of the input parameters during the test when an error is detected.
- Storing the values assigned to output parameters before the test.

A stub can be called several times during the execution of a test. The test harness allocates memory for expected and returned values in accordance to the maximum number of calls to the stub in the test harness.

You can reduce the stub memory allocation value to a lower value in the configuration settings when running tests on a target platform that is short on memory resources.

Creating stubs from the project explorer

You can create a stub from the project by simply selecting a source file or a function. Each stub simulates and replaces a particular function.

To create a stub from the project explorer:

1. In the project explorer, right-click the project, source file, or a function, and click **New > Stub Behavior**.
If you select a function, skip to step 3.
- Result**
The **Create Test Case** wizard opens.
2. On the **Stubbed function** page, enter the function name that you want to test in the **Filter** field. You can choose the functions displayed into the list.
3. On the **Stub Behavior** page, type a name for the stub behavior, an optional **Description**, and click **Next**.
The description contains information that can be viewed and edited in the test editor.
4. On the **Stub Location** page, select a folder and type a file name for the stub and click **Finish**.

Results

The stub is generated in the project explorer and the stub editor opens. See [Editing stubs on page 223](#) for information about the stub editor. To use the stub in a test, you must add it to a test case, and add the function in the stubbed function list of the test harness.

Editing stubs

The stub editor enables you to visually describe the stub behavior and to define input and output parameters for the stub.

About this task

The test case editor is made of three panes:

- The **Stub Behaviors** list displays one or several behaviors for the stub function. You can add new behaviors or duplicate existing behaviors.
- The **Calling Function** pane displays the names of components that call the stubbed function.
- The **Details** pane contains the input and output values for the selected behavior.
- The **User source code** pane contains the user code added to compute a return value for the stub.

 **Tip:** You can find where the edited file is located by clicking on the title of editor or on the header and selecting **Navigate > Show In > Project Explorer**. The explorer selects the current stub and expands automatically all parent nodes.

The main objective of editing a test case is to define the checks for each stub's call in the tested code.

1. In the project explorer, open a stub.
2. In the **Stub Behaviors** list, select the default behavior or create new one.
3. In the **Details** section, select a check block.

The **Checked Variables** table displays the variables and structures contained in the function under test.
4. For each variable or structure, specify an expected value and a return code.

These values can be simple values, multiple values (ranges, series) or C structures.

 - a. In the **Stub call** definition table, select a variable **Expected Value** cell that you want to set and click the menu button ▼ to specify an expected value or range.

See [Variable initial expressions on page 193](#) for more information.
 - b. To edit single values, multiple values, or series, type the values in the quick edition area line above the table. To specify structure values, edit the individual fields of the structure.

The quick editor area adapts to the selected data type or entry mode.
 - c. In the **Stub call** definition table, select a variable return cell that you want to set and click the menu button ▼ to specify a return value. This value can be a C native expression. If you want to replace a single value by a section of source code, click on the  button. The User source code is activated and you can enter your special source code for the stub. Don't forget the return statement to return a value for the calling expression.
5. When you have finished editing the test case, click **Save** and close the stub editor.

Testing with Studio

HCL OneTest™ Embedded Studio overview

HCL OneTest™ Embedded Studio is the classic user interface that supports C, C++, Ada test and analysis tools.

The HCL OneTest™ Embedded Studio test environment is not compatible with the HCL OneTest™ Embedded for Eclipse IDE environment that was introduced in version 8.0 of the product. The documentation in this section is intended for:

- Users who want to use existing projects with test scripts created in versions 7.5 and earlier of HCL OneTest™ Embedded Studio.
- Users who are testing programs written in Ada.

If you are creating new test projects in C, use HCL OneTest™ Embedded for Eclipse IDE. The Eclipse workbench provides many benefits, including visual test design, a more accessible user interface, and a higher level of compatibility with other software development environments.

Analyzing static source code

The static analysis feature set of HCL OneTest™ Embedded allows you to analyze your source code to measure complexity and compliance to standards. Each feature analyzes the source code without compiling and running it.

To learn about	See
How to perform static analysis on your source code	Static analysis overview on page 224
How to evaluate the complexity of your source code	Static metrics overview on page 225
Verifying compliance with industry-wide coding standards	Code review overview on page 233

Checking with static analysis

The static analysis features of HCL OneTest™ Embedded allow you to measure the complexity of your source code and to check the adherence to coding guidelines.

These tools are able analyze the source code providing without compiling or running the application.

- Static metrics provide statistic indicators of code complexity.
- Code review performs in-depth verification of the source code against a set of rules that implement best practices, coding guidelines, and standards.

These static analysis features can be used together with any of the automated testing features and runtime analysis features.

Here is a basic rundown of the main steps to using the runtime analysis feature set.

To use the static analysis features:

1. From the Start page, set up a new project. This can be done automatically with the New Project Wizard.
2. Follow the Activity Wizard to add your application source files to the workspace.
3. Select the source files under analysis in the wizard to create the application node.
4. Select the runtime analysis tools to be applied to the application in the Build options.
5. Use the Project Explorer to set up the test campaign and add any additional runtime analysis or test nodes.
6. Run the application node to build and execute the instrumented application.
7. View and analyze the generated analysis and profiling reports.

The runtime analysis options can be run within a test by simply adding the runtime analysis setting to an existing test node.

Runtime or static analysis tools do not run on System Testing nodes.

Related Topics

[Static analysis overview on page 224](#) | [Code review overview on page 233](#)

Static metrics

About Static Metrics

Static Metrics for C, C++ and Ada

Statistical measurement of source code properties is an extremely important matter when you are planning multiple tests or for project management purposes. HCL OneTest™ Embedded provides a Metrics Viewer, which displays detailed source code complexity data and statistics for your C, C++ and Ada source code.

Static Metrics supports the following languages:

- **Ada:** Ada 83 and Ada 95
- **C:** C89 and C99
- **C++:** ISO/IEC 14882:1998

How the static metrics tool works

Metrics are updated each time a file is modified. Static metrics can be computed each time a node is built, but can also be calculated without executing the application.

The metrics are stored in **.met** metrics files alongside the actual source files.

To learn about	See
Opening a Metrics Report	Viewing Static Metrics on page 226
V(g) or cyclomatic complexity metrics	V(g) or Cyclomatic Number on page 232
Halstead metrics	Halstead Metrics on page 231
Customizing metrics reports	Metrics Viewer Preferences on page 1017

Related Topics

[Runtime Analysis on page 305](#) | [About Code Coverage on page 54](#)

Viewing Static Metrics

Viewing static metrics

Static Metrics for C, C++ and Ada

Use the Metrics Viewer to view static testability measurements of the source files of your project. Source code metrics are created each time a source file is added to the project.

To compute static metrics without executing the application:

1. In the **Project Browser**, select a node.
2. From the **Build** menu, select **Options** or click the **Build Options** ▾ button in the toolbar.
3. Clear all build options. Select only **Source compilation** and **Static metrics**.
4. Click the **Build**  toolbar button.

To open the Metrics Viewer:

1. Right-click a node in the **Asset Browser** of the **Project Explorer**.
2. From the pop-up menu, select **View Metrics**.

To manually open a report file:

1. From the **File** menu, select **Open...** or click the **Open**  icon in the main toolbar.
2. In the **Type** box of the File Selector, select the **.met Metrics File** file type.
3. Locate and select the metrics files that you want to open.
4. Click **OK**.

Report Explorer

The Report Explorer displays the scope of the selected nodes, or selected **.met** metrics files. Select a node to switch the Metrics Window scope to that of the selected node.

Metrics Window

Depending on the language of the analyzed source code, different pages are available:

- **Root Page - File View:** contains generic data for the entire scope
- **Root Page - Object View:** contains object related generic data for C++ only
- **Component View:** displays detailed component-related metrics for each file, class, method, function, unit, procedure, etc...

The metrics window offer hyperlinks to the actual source code. Click the name of a source component to open the [Text Editor on page 693](#) at the corresponding line.

Related Topics

[Root Level File View on page 228](#) | [Root Level Object View on page 230](#) | [Static Metrics on page 227](#) | [Exporting reports on page 705](#)

Static metrics

Static Metrics for C, C++ and Ada

The Source Code Parsers provide static metrics for the analyzed C and C++ source code.

File Level Metrics

The scope of the metrics report depends on the selection made in the [Report Explorer on page 1024](#) window. This can be a file, one or several classes or any other set of source code components.

- **Comment only lines:** the number of comment lines that do not contain any source code
- **Comments:** the total number of comment lines
- **Empty lines:** the number of lines with no content
- **Source only lines:** the number of lines of code that do not contain any comments
- **Source and comment lines:** the number of lines containing both source code and comments

- **Lines:** the number of lines in the source file
- **Comment rate:** percentage of comment lines against the total number of lines
- **Source lines:** total number of lines of source code

File, Class or Package, and Root Level Metrics

These numbers are the sum of metrics measured for all the components of a given file, class or package.

- **Total statements:** total number of statement in child nodes
- **Maximum statements:** the maximum number of statements
- **Maximum level:** the maximum nesting level
- **Maximum V(g):** the highest encountered cyclomatic number
- **Mean V(g):** the average cyclomatic number
- **Standard deviation from V(g):** deviation from the average V(g)
- **Sum of V(g):** total V(g) for the scope.

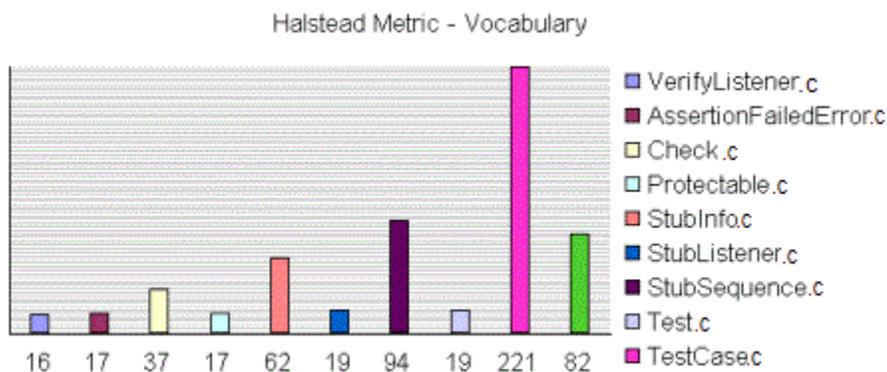
Root level file view

Static Metrics for C, C++, Ada

At the top of the Root page, the Metrics Viewer displays a graph based on Halstead data.

On the Root page, the scope of the Metrics Viewer is the entire set of nodes below the Root node.

Halstead Graph



The following display modes are available for the Halstead graph:

- VocabularySize
- Volume
- Difficulty
- Testing Effort
- Testing Errors
- Testing Time

See the [Halstead Metrics on page 231](#) section for more information.

Metrics Summary

The scope of the metrics report depends on the selection made in the Report Explorer window. This can be a file, one or several classes or any other set of source code components.

Below the Halstead graph, the Root page displays a metrics summary table, which lists for for the source code component of the selected scope:

- **V(g)**: provides a complexity estimate of the source code component
- **Statements**: shows the number of statements within the component
- **Nested Levels**: shows the highest nesting level reached in the component
- **Ext Comp Calls**: measures the number of calls to methods defined outside of the component class (C++)
- **Ext Var Use**: measures the number of uses of attributes defined outside of the component class (C++)

To select the File View:

1. Select **File View** in the View box of the Report Explorer.
2. Select the **Root** node in the Report Explorer to open the Root page.

Note With C and Ada source code, File View is the only available view for the Root page.

To change the Halstead Graph on the Root page:

1. From the **Metrics** menu, select **Halstead Graph for Root Page**.
2. Select another metric to display.

Related Topics

[Root Level Object View on page 230](#) | [Static Metrics on page 227](#) | [Viewing Static Metrics on page 226](#)

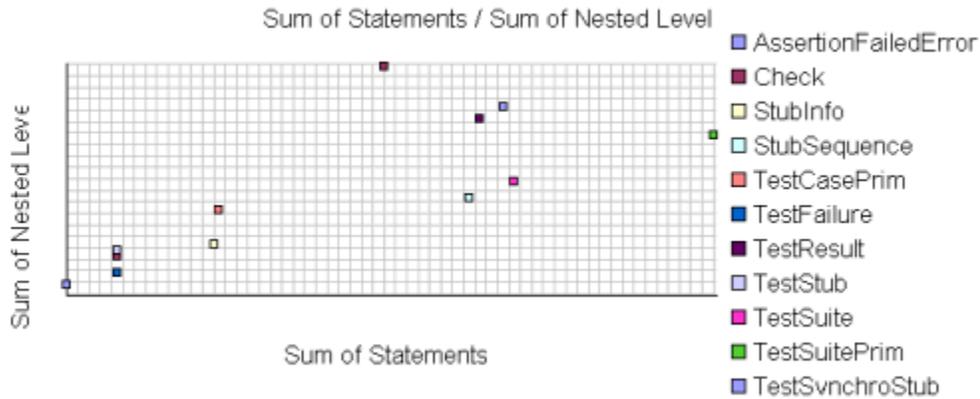
Object view

Static Metrics for C, C++ and Ada

Root Level Summary

At the top of the Root page, the Metrics Viewer displays a graph based on the sum of data.

On the Root page, the scope of the Metrics Viewer is the entire set of nodes below the Root node.



File View is the only available view with C or Ada source code. When viewing metrics for C++, an Object View is also available.

Two modes are available for the data graph:

- Vocabulary
- Size
- Volume
- Difficulty
- Testing Effort
- Testing Errors
- Testing Time

See the [Halstead Metrics on page 231](#) section for more information.

Metrics Summary

Below the Halstead graph, the Root page displays a metrics summary table, which lists for each source code component:

- **V(g)**: provides a complexity estimate of the source code component
- **Statements**: shows the total number of statements within the object
- **Nested Levels**: shows the highest statement nesting level reached in the object
- **Ext Comp Calls**: measures the number of calls to components defined outside of the object
- **Ext Var Use**: measures the number of uses of variables defined outside of the object

Note The result of the metrics for a given object is equal to the sum of the metrics for the methods it contains.

To select the Object View:

1. Select the **Root** node in the Report Explorer to open the Root page.
2. Select **Object View** in the **View** box of the Report Explorer.

To switch the object graph mode:

1. From the **Metrics** menu, select **Object Graph for Root Page**.
2. Select **ExtVarUse by ExtCompCall** or **Nested Level by Statement**.

Related Topics

[Root Level File View on page 228](#) | [Static Metrics on page 227](#) | [Viewing Static Metrics on page 226](#)

Halstead Metrics

Static Metrics for C, C++, Ada

Halstead complexity measurement was developed to measure a program module's complexity directly from source code, with emphasis on computational complexity. The measures were developed by the late Maurice Halstead as a means of determining a quantitative measure of complexity directly from the operators and operands in the module.

Halstead provides various indicators of the module's complexity

Halstead metrics allow you to evaluate the testing time of any C/C++ source code. These only make sense at the source file level and vary with the following parameters:

Parameter	Meaning
n1	Number of distinct operators
n2	Number of distinct operands

N1	Number of operator instances
N2	Number of operand instances

When a source file node is selected in the Metrics Viewer, the following results are displayed in the Metrics report:

Metric	Meaning	Formula
n	Vocabulary	$n_1 + n_2$
N	Size	$N_1 + N_2$
V	Volume	$N * \log_2 n$
D	Difficulty	$n^{1/2} * N_2 / n_2$
E	Effort	$V * D$
B	Errors	$V / 3000$
T	Testing time	E / k

In the above formulae, k is the *stroud* number, which has an arbitrary default value of 18. With experience, you can adjust the stroud number to adapt the calculation of the estimated testing time (T) to your own testing conditions: team background, criticality level, and so on.

When the Root node is selected, the Metrics Viewer displays the total testing time for all loaded source files.

Related Topics

[Viewing Static Metrics on page 226](#) | [V\(g\) or Cyclomatic Number on page 232](#)

V(g) or Cyclomatic Number

Static Metrics for C, C++ and Ada

The V(g) or cyclomatic number is a measure of the complexity of a function which is correlated with difficulty in testing. The standard value is between **1** and **10**.

A value of **1** means the code has no branching.

A function's cyclomatic complexity should not exceed **10**.

The Metrics Viewer presents V(g) of a function in the Metrics tab when the corresponding tree node is selected.

When the type of the selected node is a source file or a class, the sum of the V(g) of the contained function, the mean, the maximum and the standard deviation are calculated.

At the Root level, the same statistical treatment is provided for every function in any source file.

Related Topics

[Viewing Static Metrics on page 226](#) | [Halstead Metrics on page 231](#)

Code review

Code review overview

Code Review for C

Automated source code review is a method of analyzing code against a set of predefined rules to ensure that the source adheres to guidelines and standards that are part of any efficient quality control strategy. HCL OneTest™ Embedded provides an automated code review tool, which reports on adherence to guidelines for your C source code.

Among other guidelines, the code review tool implements rules from the MISRA-C:2004 standard, **Guidelines for the use of the C language in critical systems**.

Code Review supports C89 and C99.

How the code review tool works

When an application or test node is built, the source code is analyzed by the code review tool. The tool checks the source file against the predefined rules and produces a **.crc** report file that can be viewed and controlled from the HCL OneTest™ Embedded graphical user interface (GUI).

Code review can be performed each time a node is built, but can also be calculated without executing the application.

To learn about	See
Setting up the rules to used for reviewing code	Configuring code review rules on page 294
Performing a code review	Running a code review on page 297
Viewing and understanding the results of a code review	Viewing code review results on page 298
Interpreting code review reports	Understanding code review reports on page 299
Locally disabling a rule	Ignoring rules in a portion of code on page 94
The list of rules used by HCL OneTest™ Embedded code review	Code review rules on page 95

Code review MISRA 2004 rules

The code review tool covers rules from the lists the rules that produced and error or a warning. Each rule can be individually disabled or assigned a Warning or Error severity by using the Rule configuration window. Some rules also have parameters that can be changed. Among other guidelines, the code review tool implements most rules from the MISRA-C:2004 standard, "Guidelines for the use of the C language in critical systems". These rules are referenced with an M prefix. In addition to the industry standard rules, HCL OneTest™ Embedded provides some additional coding guidelines, which are referenced with an E prefix.

Code Review for C - MISRA 2004 rules

Table 4. MISRA 2004 rules

Code review reference	MISRA-C: 2004 reference	Code review message	Description
Code compliance			
M1.1	Rule 1.1	ANSI C error: <error>	All code shall conform to ISO 9899:1990 Required
M1.1w	Rule 1.1	ANSI C warning: <warning>	Required
Language extensions			Required
M2.2	Rule 2.2	Source code shall only use /* ... */ style comments	Source code shall only use /* ... */ style comments Required
M2.3	Rule 2.3	The character sequence /* shall not be used within a comment	The character sequence /* shall not be used within a comment Required
E2.3.1			The character sequence // should not be used within a 'C-style' comment Advisory
E2.3.2			Line-splicing shall not be used in // comments

Table 4. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
			Advisory
E2.6			A function should not contain unused label declarations Advisory
E2.7		There should be no unused parameters in functions	Advisory
E2.8		Macro %name% is never used	Advisory
E2.9		Type %name% is never used	Advisory
E2.10		Tag %name% is never used	Advisory
E2.50		Functions should have less than '100' lines. Note The number of lines can be specified.	Advisory
E2.51		Functions should have less than '15' V(g) complexity. Note: The complexity limit of lines can be specified.	Advisory
E2.52		Functions should have less than '%param%' lines, outside empty lines (current value: %name%).	
E2.53		Functions should have less than '%param%' lines, outside empty lines or comment lines (current value : %name%).	
E2.54		Functions should have less than '%param%' lines, outside empty lines, comment lines or bracket lines (current value : %name%).	Lines are not counted in the following cases:

Table 4. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
			<ul style="list-style-type: none"> • If they contain spaces (including \t, \r, \n), • If they contain only brackets (there might be several brackets on same line), • If they contain comments only, or if they contain brackets and comments only.
E2.55		Compilation units should define have less than '%param%' functions (current value : %name%).	<p>Optional</p> <p>Compilation unit max number of functions.</p> <p>Default parameter value: 10.</p>
E2.56		Compilation units should have less than '%param%' functions (current value: %name%).	<p>Optional</p> <p>Compilation unit max number of variables.</p> <p>Default parameter value: 10.</p>
E2.57		Compilation unit should have less than '%param%' lines (current value: %name%).	<p>Optional</p> <p>Compilation unit max number of lines.</p> <p>Default parameter value : 200.</p>
E2.58		Compilation unit should have less than '%param%' lines, not counting empty lines (current value : %name%).	<p>Optional</p> <p>Compilation unit max number of lines.</p> <p>Default parameter value : 200.</p>
E2.59		Compilation unit should have less than '%param%' lines, not counting empty lines or comments (current value: %name%).	<p>Optional</p> <p>Compilation unit max number of lines.</p> <p>Empty lines or comments (current value: %name%) are not counted.</p>

Table 4. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
			Default parameter value : 200.
E2.60		Compilation units should have less than '%param%' lines, not counting empty lines, comments or brackets (current value: %name%) are not counted.	Optional Compilation unit max number of lines. Empty lines, comments or brackets (current value : %name%) are not counted. Default parameter value : 200.
E2.61		Functions should have less than '%param%' parameters (current value: %name%).	
Documentation			
M3.4	Rule 3.4	All uses of the #pragma directive shall be documented and explained.	Required
Character sets			
M4.1.1	Rule 4.1	Only escape sequences that are defined in the ISO C standard shall be used	Only escape sequences that are defined in the ISO C standard shall be used Required
M4.1.2	Rule 4.1	Only ISO C escape sequences are allowed(\v)	Only ISO C escape sequences are allowed(\v) Required
M4.2	Rule 4.2	Trigraphs shall not be used	Trigraphs shall not be used Required
Identifiers			
M5.1	Rule 5.1	Identifiers %name% and %name % are identical in the first <value> characters. Note The number of characters can be specified.	Identifiers (internal and external) shall not rely on the significance of more than 31 characters

Table 4. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
			Required
E5.1.1		Identifiers %name% and %name% are ambiguous because of possible character confusion. Note that ambiguous characters can be parameterized.	Advisory
E5.1.2		Possible typing mistakes between the variables %name% or %name% because of repeating character.	Advisory
E5.1.3		Identifiers %name% and %name% are identical in the first %param% characters ignoring case	Advisory
E5.1.4		Macros %name% and %name% are identical in the first %param% characters	Advisory
E5.1.5		Macro %name% and identifier %name% are identical in the first %param% characters	Advisory
E5.1.6		Macros %name% and %name% are identical in the first %param% characters ignoring case	Advisory
E5.1.7		Macro %name% and identifier %name% are identical in the first %param% characters ignoring case	Advisory
M5.2	Rule 5.2	Identifier %name% in an inner scope hides the same identifier in an outer scope : %location%	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier Required

Table 4. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
E5.3		The tag name %name% should not be reused. Name already found in %location%	Advisory
M5.3.1	Rule 5.3		The typedef name %name% should not be reused except for its tag. Name already found in %location% Required
M5.3.2	Rule 5.3		The typedef name '%name%' should not be reused even for its tag. Name already found in %location% Required
M5.4	Rule 5.4	A struct and union cannot use the same tag name	A tag name shall be a unique identifier Required
M5.5	Rule 5.5	The static object or function %name% should not be reused. Static object or function already found in %location%.	No object or function identifier with static storage duration should be reused Advisory
M5.6	Rule 5.6	Avoid using the same identifier %name% in two different name spaces. Identifier already found in %location%	No identifier in one name space should have the same spelling as an identifier in another name space, with the exception of structure and union member names Advisory
M5.7	Rule 5.7	The identifier %name% should not be reused. Identifier already found in %location%.	Advisory
Types			
M6.1.1	Rule 6.1	The C language plain char type should only be used for character values.	The C language plain char type should only be used for character values. Required

Table 4. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
M6.1.2	Rule 6.1	Case char value is applicable only if the switch statement value is plain character variable	Required
M6.1.3	Rule 6.1	Avoid using comparison operators on plain char.	Required
M6.2	Rule 6.2	The C language signed char or unsigned char types should only be used for numeric values.	The C language signed char or unsigned char types should only be used for numeric values. Required
M6.3	Rule 6.3	The C language numeric type %name% should not be used directly but instead used to define typedef.	typedefs that indicate size and signedness should be used in place of the basic types Advisory
E6.3		The implicit 'int' type should not be used.	Required
M6.4.1	Rule 6.4	Bit fields should only be of type 'unsigned int' or 'signed int'.	Required
M6.4.2	Rule 6.4	Bit fields should not be of type 'enum'	Required
M6.4.3	Rule 6.4	Bit fields should only be of explicitly signed or unsigned type	Required
M6.4.4	Rule 6.4	Bit fields should not be of type 'bool' under c99	Required
M6.4.5	Rule 6.4	Bit fields should not be of type 'boolean' outside c99	Required
M6.5	Rule 6.5	Bit fields of type 'signed int' must be at least 2 bits long.	Required
Constants			
M7.1	Rule 7.1	Octal constants and escape sequences should not be used.	Octal constants (other than zero) and octal escape sequences shall not be used

Table 4. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
			Required
E7.1		Octal and hexadecimal escape sequences shall be terminated	Required
E7.2		The lowercase character 'l' shall not be used in a literal suffix	Required
E7.3		A string literal shall not be assigned to an object unless the object's type is pointer to a const-qualified char	Required
Declarations and definitions			
M8.1.1	Rule 8.1	A prototype for the function %name% should be declared before defining the function.	Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call Required
E8.1.1		A prototype for the global object %name% should be declared before defining the object	Required
M8.1.2	Rule 8.1	A prototype for the function %name% should be declared before calling the function.	Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call Required
M8.1.3	Rule 8.1	A prototype for the function %name% should be declared before calling the function	Required
M8.2.1	Rule 8.2	The type of %name% should be explicitly stated.	Whenever an object or function is declared or defined, its type shall be explicitly stated Required

Table 4. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
M8.2.2	Rule 8.2	The type of parameter %name% should be explicitly stated	Required
M8.3	Rule 8.3	Parameters and return types should use the same type names in the declaration and in the definition, even if basic types are the same.	For each function parameter the type given in the declaration and definition shall be identical, and the return types shall also be identical Required
E8.3		Parameters and return types should use compatible type in the declaration and in the definition	Required
M8.4	Rule 8.4	If objects or functions are declared multiple times their types should be compatible.	Required
M8.5.1	Rule 8.5	The body of function %name% should not be located in a header file.	Required
E.8.50		Use the const qualification for variable %name% which is pointer and which is not used to change the pointed object	Required
E.8.51		The object %name% is never referenced	Required
M8.5.2	Rule 8.5	The memory storage (definition) for the variable %name% should not be in a header file.	Objects shall be defined at block scope if they are only accessed from within a single function. Required
M8.6	Rule 8.6	Functions should not be declared at block scope.	Required

Table 4. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
M8.7	Rule 8.7	Global objects should not be declared if they are only used from within a single function.	Objects shall be defined at block scope if they are only accessed from within a single function Required
M8.8.2	Rule 8.8	Static function %name% should only be declared in a single file. Redundant declaration found at: %location%	Required
M8.8.3	Rule 8.8	Static object %name% should only be declared in a single file. Redundant declaration found at: %location%	Required
M8.8.4	Rule 8.8	Identifiers %name% that declare objects or functions with external linkage shall be declared once in one and only one file	Required
M8.8.5	Rule 8.8	Identifiers %name% that declare objects or functions with external linkage shall be unique	Required
M8.9.1	Rule 8.9	The global object or function %name% should have exactly one external definition. Redundant definition found in %location%	An identifier with external linkage shall have exactly one external definition
M8.9.2	Rule 8.9	The global object or function %name% should have exactly one external definition. No definition found.	Required
M8.10.1	Rule 8.10	Global object %name% that are only used within the same file should be declared using the static storage-class specifier.	All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required. Required

Table 4. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
M8.10.2	Rule 8.10	Global function %name% that are only used within the same file should be declared using the static storage-class specifier.	All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required Required
M8.11	Rule 8.11	Global objects or functions that are only used within the same file should be declared with using the static storage-class specifier.	The static storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage Required
M8.12	Rule 8.12	When a global array variable can be used from multiple files, its size should be defined at initialization time.	Required
E.8.14		Inline function %name% should be static	Required
Initialization		The restrict type qualifier shall not be used	Required
M9.1	Rule 9.1	Variables with automatic storage duration should be initialized before being used.	Required
M9.2	Rule 9.2	Nested braces should be used to initialize nested multi-dimension arrays and nested structures.	Required
E9.2		Arrays shall not be partially initialized	Required
M9.3	Rule 9.3	Either all members or only the first member of an enumerator list should be initialized.	In an enumerator list, the "=" construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized Required

Table 4. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
M9.3	Rule 9.3	Either all members or only the first member of an enumerator list should be initialized	Required
E9.3	Rule E9.3	Enumeration member %name% have a not unique implicitly specified value	Required
E9.4		The global variable %name% is not initialized	Required
Arithmetic type conversions			
E10.1		Constraint violation : can't use floating type as operand of '[]', '%', '<<', '>>', '~', '&', ' ', '^'	Required
M10.1.1	Rule 10.1	Implicit conversion of a complex integer expression to a smaller sized integer is not allowed.	<p>The value of an expression of integer type shall not be implicitly converted to a different underlying type if:</p> <ul style="list-style-type: none"> • a) it is not a conversion to a wider integer type of the same signedness, or • b) the expression is complex, or • c) the expression is not constant and is a function argument, or • d) the expression is not constant and is a return expression. <p>Required</p>
M10.1.2	Rule 10.1	Implicit conversion of an integer expression to a different signedness is not allowed.	Required
M10.2	Rule 10.2	Conversion of a complex floating expression is not allowed. Only constant expressions can be implicitly converted and only to a	The value of an expression of floating type shall not be implicitly converted to a different type if:

Table 4. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
		wider floating type of the same signedness.	<ul style="list-style-type: none"> • a) it is not a conversion to a wider floating type, or • b) the expression is complex, or • c) the expression is a function argument, or • d) the expression is a return expression. <p>Required</p>
E10.2		Operand should be boolean.	Required
M10.3	Rule 10.3	Type cast of complex integer expressions is only allowed into a narrower type of the same signedness.	The value of a complex expression of integer type may only be cast to a type that is narrower and of the same signedness as the underlying type of the expression Required
E10.3		Can't use a boolean as a numeric value	Required
M10.4	Rule 10.4	Type cast of complex floating expressions is only allowed into a narrower type of the same signedness.	The value of a complex expression of floating type may only be cast to a narrower floating type Required
E10.4		Can't use a char as a numeric value	Required
M10.5	Rule 10.5	When using operator '~' or '<<' on 'unsigned char' or 'unsigned int', you should always cast returned value	Required
E10.5	Rule E10.5	Can't use a not anonymous enum as a numeric value	Required
M10.6	Rule 10.6	Definitions of unsigned type constants should use the 'U' suffix.	A "U" suffix shall be applied to all constants of unsigned type

Table 4. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
			Required
E10.6		Shift and bitwise operations should be performed on unsigned value	Required
E10.7		Right hand operand of shift operation should be an unsigned value	Required
E10.8		Unary minus operation should not be performed on unsigned value	Required
E10.9		Expressions of essentially character type shall not be used inappropriately in addition and subtraction operations	Required
E10.10		The value of an expression shall not be assigned to an object with a narrower essential type	Required
E10.11		The value of an expression shall not be assigned to an object with a different essential type category	Required
E10.12		Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category	Required
E10.13		The value of an expression should not be cast to an inappropriate essential type	Required
E10.14		The value of a composite expression shall not be assigned to an object with wider essential type	Required

Table 4. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
E10.15		If a composite expression is used as one operand of an operation in which the usual arithmetic conversions are performed then the other operand shall not have wider essential type	Required
E10.16		The value of a composite expression shall not be cast to a different essential type category or a wider essential type	Required
Pointer type conversions			
M11.1	Rule 11.1	A function pointer should not be converted to another type of pointer.	Conversions shall not be performed between a pointer to a function and any type other than an integral type Required
E11.1		Conversions shall not be performed between a pointer to an incomplete type and any other type	Required
M11.2	Rule 11.2	An object pointer should not be converted to another type of pointer.	Conversions shall not be performed between a pointer to object and any type other than an integral type, another pointer to object type or a pointer to void Required
E11.2		A conversion should not be performed from pointer to void into pointer to object	Required
M11.3	Rule 11.3	Casting a pointer type to an integer type should not occur.	A cast should not be performed between a pointer type and an integral type Advisory

Table 4. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
E11.3	E11.3	A cast shall not be performed between pointer to void and an arithmetic type	Required
E11.4		A cast shall not be performed between pointer to object and a non-integer arithmetic type	Required
M11.4.1	Rule 11.4	Casting an object pointer type to a different object pointer type should not occur.	A cast should not be performed between a pointer to object type and a different pointer to object type Advisory
M11.4.2	Rule 11.4	Casting an object pointer type to a different object pointer type should not occur, especially when object sizes are not the same.	Advisory
M11.5	Rule 11.5	Casting of pointers to a type that removes any const or volatile qualification on the pointed object should not occur.	A cast shall not be performed that removes any const or volatile qualification from the type addressed by a pointer. Required
Expressions			
M12.1	Rule 12.1	Implicit operator precedence may cause ambiguity. Use parenthesis to clarify this expression.	Limited dependence should be placed on C's operator precedence rules in expressions Advisory
E12.11		Implicit bitwise operator precedence may cause ambiguity. Use parenthesis to clarify this expression.	Advisory
M12.3	Rule 12.3	The sizeof operator should not be used on expressions that contain side effects.	Required

Table 4. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
M12.4.1	Rule 12.4	An expression that contains a side effect should not be used in the right-hand operand of a logical && or operator.	The right-hand operand of a logical && or operator shall not contain side effects Required
M12.4.2	Rule 12.4	The function in the right-hand operand of a logical && or operator might cause side effects.	
M12.5	Rule 12.5	Parenthesis should be used around expressions that are operands of a logical && or .	Required
E12.51		Ternary expression ?: should not be used.	Advisory
E12.54		Expressions should not cause a side effect assignment.	Advisory
M12.6	Rule 12.6	Only Boolean operands should be used with logical operators (&&, and !).	The operands of logical operators (&&, and !) should be effectively Boolean. Expressions that are effectively Boolean should not be used as operands to operators other than (&&, and !) Advisory
E12.61		The operator on a Boolean expression should be a logical operator (&&, or !).	Advisory
M12.7	Rule 12.7	Bitwise operators should only use unsigned operands.	Bitwise operators shall not be applied to operands whose underlying type is signed Required
M12.8	Rule 12.8	The right-hand operand of a shift operator should not be too big or negative.	The right-hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left-hand operand Required

Table 4. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
M12.9	Rule 12.9	Only use unary minus operators with signed expressions.	The unary minus operator shall not be applied to an expression whose underlying type is unsigned Required
M12.10	Rule 12.10	Do not use the comma operator	Required
M12.12	Rule 12.12		Advisory Parenthesis should be used around expression that is operand of 'sizeof' operator.
M12.13	Rule 12.13	The increment (++) or the decrement (--) operators should not be used with other operators in an expression.	Advisory
Control statement expressions			
E13.1		The result of an assignment operator should not be used in an expression	Required
M13.1.1	Rule 13.1	Boolean expressions should not contain assignment operators.	Assignment operators shall not be used in expressions that yield a Boolean value Required
M13.1.2	Rule 13.1	Boolean expressions should not contain side effect operators.	Required
M 13.2	Rule 13.2	Non-Boolean values that are tested against zero should have an explicit test	Tests of a value against zero should be made explicit, unless the operand is effectively Boolean Advisory

Table 4. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
M13.3	Rule 13.3	The equal or not equal operator should not be used in floating-point expressions.	Floating-point expressions shall not be tested for equality or inequality Required
M13.4	Rule 13.4	Floating-point variables should not be used to control a for statement.	Required
M13.5.1	Rule 13.5	Only loop counter should be initialized in a loop initialization part.	The three expressions of a statement shall be concerned with loop control only. Required
M13.5.2	Rule 13.5	In the 'update part' of a 'for statement', only 'loop counter' should be updated	Required
M13.5.3	Rule 13.5	There should be one and only one loop counter for loop statement.	Required
M13.6	Rule 13.6	Loop counter of a 'for statement' should not be modified within the body of the loop.	Required
M13.7	Rule 13.7	Invariant Boolean expressions should not be used.	Boolean operations whose results are invariant shall not be permitted Required
Control flow			
M14.1	Rule 14.1	Unreachable code.	Required
M14.2	Rule 14.2	A non-null statement should either have a side effect or change the control flow.	Required
M14.3	Rule 14.3	A null statement in original source code should be on a separate line and the semicolon	Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment provided that the

Table 4. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
		should be followed by at least one white space and then a comment.	first character following the null statement is a white-space character Required
M14.4	Rule 14.4	Do not use the goto statement.	Required
E14.4.1		The goto statement shall jump to a label declared later in the same function	Required
E14.4.2		Any label referenced by a goto statement shall be declared in the same block, or in any block enclosing the goto statement	Required
E14.4.3		There should be no more than one break or goto statement used to terminate any iteration statement	Required
M14.5	Rule 14.5	Do not use the continue statement.	Required
M14.6	Rule 14.6	Only one break statement should be used within a loop.	For any iteration statement there shall be at most one break statement used for loop termination Required
M14.7.1	Rule 14.7	Only one exit point should be defined in a function.	A function shall have a single point of exit at the end of the function Required
M14.7.2	Rule 14.7	The return keyword should not be used in a conditional block.	Required
M14.8.1	Rule 14.8	The switch statement should be followed by a compound statement {}.	The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement Required

Table 4. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
M14.8.2	Rule 14.8	The while statement should be followed by a compound statement {}.	
M14.8.3	Rule 14.8	The do..while statement should contain a compound statement {}.	
M14.8.4	Rule 14.8	The for statement should be followed by a compound statement {}.	
M14.9.1	Rule 14.9	The if (expression) construct should be followed by a compound statement {}.	An if (expression) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement
M14.9.2	Rule 14.9	The else keyword should be followed by either a compound statement or another if statement.	Required
M14.9.3	Rule 14.9	The else keyword should be followed by a compound statement	
M14.10	Rule 14.10	All if ... else if sequences should have an else block.	All if ... else if constructs shall be terminated with an else clause Required
Switch statements			
M15.0	Rule 15.0	A switch block should start with a case.	The MISRA C switch syntax shall be used Required
M15.1	Rule 15.1	A case or default statements should only be used directly within the compound block of a switch statement.	A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement Required

Table 4. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
E15.10		The switch expression should not have side effects.	Required
M15.2	Rule 15.2	The break statement should only be used to terminate every non-empty switch block.	An unconditional break statement shall terminate every non-empty switch clause Required
M15.3.1	Rule 15.3	The switch statement should have a default clause.	Required
M15.3.2	Rule 15.3	The default clause should be the last clause of the switch statement.	
M15.4.1	Rule 15.4	A Boolean should not be used as a switch expression.	A switch expression shall not represent a value that is effectively Boolean Required
M15.4.2	Rule 15.4	A constant should not be used as a switch expression.	Required
M15.5	Rule 15.5	At least one case should be defined in the switch.	Every switch statement shall have at least one case clause Required
Functions			
M16.1	Rule 16.1	The function %name% should not have a variable number of arguments.	Functions shall not be defined with a variable number of arguments Required
Rule M16.1.2	Rule 16.1	The library functions 'va_list, va_arg, va_start, va_end, va_copy' should not be used	Required
M16.2.1	Rule 16.2	Recursive functions are not allowed. The function %name% is directly recursive.	Functions shall not call themselves, either directly or indirectly Functions shall not call themselves, either directly or indirectly

Table 4. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
M16.2.2	Rule 16.2	Recursive functions are not allowed. The function %name% is recursive when calling %name% .	Required
M16.3	Rule 16.3	The function prototype should name all its parameters.	Identifiers shall be given for all of the parameters in a function prototype declaration Required
M16.4	Rule 16.4	The identifiers used in the prototype and definition should be the same.	Required
M16.5	Rule 16.5	Functions with no parameters should use the void type.	Required
E16.50		The function %name% is never referenced.	Required
M16.6	Rule 16.6	The number of arguments used in the call does not match the number declared in the prototype.	Required
M16.7	Rule 16.7	Use the const qualification for parameter %name% which is pointer and which is not used to change the pointed object.	A pointer parameter in a function prototype should be declared as pointer to const if the pointer is not used to modify the addressed object Required
M16.8	Rule 16.8	The return should always be defined with an expression for non-void functions.	All exit paths from a function with non-void return type shall have an explicit return statement with an expression Required
M16.9	Rule 16.9	Function identifiers should always use a parenthesis or a preceding &.	A function identifier shall only be used with either a preceding &, or with a parenthesized parameter list, which may be empty Required

Table 4. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
M16.10	Rule 16.10	When a function returns a value, this value should be used.	If a function returns error information, then that error information shall be tested Required
Pointers and arrays			
M17.4	Rule 17.4	Pointer arithmetic except array indexing should not be used.	Array indexing shall be the only allowed form of pointer arithmetic Required
M17.5	Rule 17.5	A declaration should not use more than two levels of pointer indirection.	Advisory
Structures and unions			
M18.1	Rule 18.1	Structure or union types should be finalized before the end of the compilation units.	Required
E18.1		Flexible arrays members shall not be declared	Required
18.2		Variable-length array types shall not be used	Required
E18.3		The declaration of an array parameter shall not contain the static keyword between the []	Required
M18.4	Rule 18.4	Do not use unions.	Required
Preprocessing directives			
M19.1	Rule 19.1	Only preprocessor directives or comments may occur before the #include statements.	#include statements in a file should only be preceded by other preprocessor directives or comments

Table 4. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
			Advisory
M19.2	Rule 19.2	Do not use non-standard characters in included file names.	Advisory
M19.3	Rule 19.3	Filenames with the #include directive should always use the <filename> or "filename" syntax.	Required
M19.4	Rule 19.4	A C macro should only be expanded to a constant, a braced initializer, a parenthesised expression, a storage class keyword, a type qualifier, or a do-while-zero block.	Required
M19.5	Rule 19.5	Macro definitions or #undef should not be located within a block.	Required
M19.6	Rule 19.6	Do not use the #undef directive.	Required
M19.7	Rule 19.7	Function should be used instead of macros when possible.	Advisory
M19.8	Rule 19.8	Missing argument when calling the macro.	A function-like macro shall not be invoked without all of its arguments. Required
M19.9	Rule 19.9	The preprocessing directive %name% should not be used as argument to the macro.	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives Required
M19.10	Rule 19.10	The parameter %name% in the macro should be enclosed in parentheses except when it is used as the operand of # or ##.	In the definition of a function-like macro each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of # or ##

Table 4. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
			Required
M19.11	Rule 19.11	Undefined macro identifier in the preprocessor directive.	All macro identifiers in preprocessor directives shall be defined before use, except in <code>#ifdef</code> and <code>#ifndef</code> preprocessor directives and the <code>defined()</code> operator Required
M19.12	Rule 19.12	The <code>#</code> or <code>##</code> preprocessor operator should not be used more than once.	There shall be at most one occurrence of the <code>#</code> or <code>#</code> preprocessor operators in a single macro definition Required
M19.13	Rule 19.13	The <code>#</code> and <code>##</code> preprocessor operator should be avoided.	Advisory
M19.14	Rule 19.14	Only use the 'defined' preprocessor operator with a single identifier.	The defined preprocessor operator shall only be used in one of the two standard forms Required
M19.15	Rule 19.15	Header file contents should be protected against multiple inclusions	Precautions shall be taken in order to prevent the contents of a header file being included twice Required
M19.16	Rule 19.16	Possible bad syntax in preprocessing directive.	Preprocessing directives shall be syntactically meaningful even when excluded by the preprocessor Required
M19.17	Rule 19.17	A <code>#if</code> , <code>#ifdef</code> , <code>#else</code> , <code>#elif</code> or <code>#endif</code> preprocessor directive has been found without its matching directive in the same file.	All <code>#else</code> , <code>#elif</code> and <code>#endif</code> preprocessor directives shall reside in the same file as the <code>#if</code> or <code>#ifdef</code> directive to which they are related

Table 4. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
			Required
E19.18		The controlling expression of a #if or #elif preprocessing directive shall evaluate to 0 or 1	Required
E19.19		A macro parameter immediately following a # operator shall not immediately be followed by a ## operator	Required
E19.20		Macro parameter %name% used as an operand to the # and ## operators shall not be used elsewhere in this macro	Required
Standard libraries			
M20.1	Rule 20.1	%name% should not be defined, redefined or undefined.	Reserved identifiers, macros and functions in the standard library, shall not be defined, redefined or undefined Required
E20.1		A macro shall not be defined with the same name as a keyword: %name%	Required
M20.2.1		#define and #undef shall not be used on a reserved identifier or reserved macro name: Identifier %name% already found in %name%	Required
M20.2.2	Rule 20.2	#define and #undef shall not be used on identifier beginning with an underscore or on 'defined' keyword: %name%	Required

Table 4. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
M20.2.3	Rule 20.2	Declared identifier should not be a reserved identifier or reserved macro name: Identifier %name% already found in %name%	Required
M20.2.4	Rule 20.2	Declared identifier should not begin with an underscore or be 'defined' keyword: %name%	Required
M20.4	Rule 20.4	Dynamic heap memory allocation shall not be used.	<p>This precludes the use of the functions calloc, malloc, realloc free and strdup. There is a whole range of unspecified, undefined and implementation-defined behaviour associated with dynamic memory allocation, as well as a number of other potential pitfalls. Dynamic heap memory allocation may lead to memory leaks, data inconsistency, memory exhaustion, non-deterministic.</p> <p>Note that some implementations may use dynamic heap memory allocation to implement other functions (for example functions in the library string.h). If this is the case then these functions shall also be avoided.</p> <p>Required</p>
M20.5	Rule 20.5	The error indicator errno shall not be used.	<p>errno is a facility of C, which in theory should be useful, but which in practice is poorly defined by the standard. A non zero value may or may not indicate that a problem has occurred; as a result it shall not be used. Even for those functions for which the behaviour of errno is well defined, it is preferable to check the values of inputs before calling the function rather than rely on using errno to trap errors (see Rule 16.10).</p>

Table 4. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
			Required
M20.6	Rule 20.6	The macro offsetof, in library <stddef.h>, shall not be used.	Use of this macro can lead to undefined behaviour when the types of the operands are incompatible or when bit fields are used. Required
M20.7	Rule 20.7	The setjmp macro and the longjmp function shall not be used.	etjmp and longjmp allow the normal function call mechanisms to be bypassed, and shall not be used. Remark : sigsetjmp and siglongjmp (Gnu Library) are also detected Required
E20.7		The standard header file <setjmp.h> shall not be used	Required
M20.8	Rule 20.8	The signal handling facilities of <signal.h> shall not be used.	Signal handling contains implementation-defined and undefined behavior. Required
M20.9	Rule 20.9	The input/output library <stdio.h> shall not be used in production code.	This includes file and I/O functions fgetpos, fopen, ftell, gets, perror, remove, rename, and ungetc. Streams and file I/O have a large number of unspecified, undefined and implementation-defined behaviours associated with them. It is assumed within this document that they will not normally be needed in production code in embedded systems. If any of the features of stdio.h need to be used in production code, then the issues

Table 4. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
			associated with the feature need to be understood. Required
M20.10	Rule 20.10	The library functions atof, atoi and atol from library <stdlib.h> shall not be used.	These functions have undefined behavior associated with them when the string cannot be converted. Required
M20.11	Rule 20.11	The library functions abort, exit, getenv and system from library <stdlib.h> shall not be used.	These functions will not normally be required in an embedded system, which does not normally need to communicate with an environment Then, it is essential to check on the implementation-defined behavior of the function in the environment. Required
E20.11		The library macro or function 'bsearch, qsort' should not be used	Required
M20.12	Rule 20.12	The time handling functions of library <time.h> shall not be used.	Includes time, strftime. This library is associated with clock times. Various aspects are implementation dependent or unspecified, such as the format of times. If any of the facilities of time.h are used, then the exact implementation for the compiler being used must be determined, and a deviation being raised. Required

Table 4. MISRA 2004 rules
(continued)

Code review reference	MISRA-C: 2004 reference	Code review message	Description
E20.12		The input/output library <wchar.h> shall not be used in production code	Required
E20.13		The standard header file <tg-math.h> shall not be used	Required
E20.14		The library macro or function 'fexcept_t', 'feclearexcept', 'fegetexceptflag', 'feraiseexcept', 'fesetexceptflag', 'fetestexcept', 'FE_INEXACT', 'FE_DIVBYZERO', 'FE_UNDERFLOW', 'FE_OVERFLOW', 'FE_INVALID', 'FE_ALL_EXCEPT' should not be used	Required
U99.1		User defined rule	Required



Note: Applies to HCL OneTest™ Embedded Studio only:

The code review references in bold in this table are disabled when they are run from the code review link checker in test mode. To verify these rules, you must run the code review from the application node in HCL OneTest™ Embedded Studio. For more information, see [Running complete verification of MISRA rules from an application node on page 298](#).

Code review MISRA 2012 rules

The code review tool covers rules from the lists the rules that produced an error or a warning. Each rule can be individually disabled or assigned a Warning or Error severity by using the Rule configuration window. Some rules also have parameters that can be changed. Among other guidelines, the code review tool implements most rules from the MISRA-C:2012 standard, "Guidelines for the use of the C language in critical systems". These rules are referenced with an M prefix. In addition to the industry standard rules, HCL OneTest™ Embedded provides some additional coding guidelines, which are referenced with an E prefix.

Code Review - MISRA 2012 rules

D is set for Decidable, U for Undecidable.

Code review reference	Type	D/U	Description	Level
M1.1	Error	D	ANSI C error: %name %	Required
M1.1W	Error	D	ANSI C warning: %name%	Required
M1.2	Error	U	Use of #pragma %name% should always be encapsulated and documented	Advisory
E1.1	Error	D	Function max number of line	Required
E.1.2	Error	D	Function max V(g)	Required
E1.3			Functions should have less than '%param%' lines, outside empty lines (current value: %name%).	
E1.4			Functions should have less than '%param%' lines, outside empty lines or comment lines (current value : %name%).	
E1.5			Functions should have less than '%param%' lines, outside empty lines, comment lines or bracket lines (current value : %name%). Lines are not counted in the following cases:	

Code review reference	Type	D/U	Description	Level
E1.6			<ul style="list-style-type: none"> • If they contain spaces (including \t, \r, \n), • If they contain only brackets (there might be several brackets on same line), • If they contain comments only, or if they contain brackets and comments only. 	
E1.7			<p>Optional</p> <p>Compilation units should define less than '%param%' functions (current value: %name%).</p> <p>Default parameter value: 10.</p>	
E1.8			<p>Optional</p> <p>Compilation units should define less than '%param%' variables (current value: %name%).</p> <p>Default parameter value: 10.</p>	

Code review reference	Type	D/U	Description	Level
E1.9			<p>Compilation units should have less than '%param%' lines (current value: %name%).</p> <p>Default parameter value : 200.</p>	
			<p>Optional</p> <p>Compilation unit should have less than '%param%' lines, not counting empty lines (current value : %name%).</p> <p>Empty lines (current value : %name%) are not counted.</p>	
E1.10			<p>Default parameter value : 200.</p> <p>Optional</p> <p>Compilation unit should have less than '%param%' lines not counting empty lines or comments (current value : %name%).</p> <p>Empty lines or comments (current value : %name%) are not counted.</p> <p>Default parameter value : 200.</p>	

Code review reference	Type	D/U	Description	Level
E1.11			<p>Optional</p> <p>Compilation unit should have less than '%param%' lines not counting empty lines, comments or brackets (current value: %name%).</p> <p>Empty lines, comments or brackets (current value : %name%) are not counted.</p> <p>Default parameter value : 200.</p>	
E1.12			<p>Functions should have less than '%param%' parameters (current value : %name%).</p>	
M2.1	Error	U	a project shall not contain unreachable code	Required
M2.2.1	Error	U	A non-null statement should either have a side effect or change the control flow	Required
M2.2.2	Error	U	The function %name% is never referenced	Required
M2.2.3	Error	D	The object %name% is never referenced	Required
M2.3	Warning	D	Type %name% is never used	Advisory

Code review reference	Type	D/U	Description	Level
M2.4	Warning	D	Tag %name% is never used	Advisory
M2.5	Warning	D	Macro %name% is never used	Advisory
M2.6	Warning	D	A function should not contain unused label declarations	Advisory
M2.7	Warning	D	There should be no unused parameters in functions	Advisory
M3.1.1	Error	D	The character sequence /* should not be used within a comment	Required
M3.1.2	Error	D	The character sequence // should not be used within a 'C-style' comment	Required
M3.2	Error	D	Line-splicing shall not be used in // comments	Required
E3.1	Error	D	A null statement in original source code should be on a separate line and the semicolon should be followed by at least one white space and then a comment	Required
M4.1	Error	D	Octal and hexadecimal escape sequences shall be terminated	Required
M4.2	Warning	D	Trigraphs should not be used	Advisory

Code review reference	Type	D/U	Description	Level
E4.1	Error	D	Only ISO C escape sequences are allowed	Advisory
E.4.2	Error	D	Only ISO C escape sequences are allowed(\v)	Advisory
M5.1.1	Error	D	External identifiers shall be distinct in the first 31 characters	Required
M5.1.2	Error	D	External identifiers shall be distinct in the first 6 characters ignoring case	Required
M5.2	Error	D	Identifiers %name% declared in the same scope and name space shall be distinct. Identifier identical in the first %param% characters already found in %location%	Required
M5.3	Error	D	Identifier %name% declared in an inner scope shall not hide an identifier declared in an outer scope. Identifier identical in the first %param% characters already found in %location%	Required
M5.4.1	Error	D	Macros %name% and %name% are identical in the first %param% characters	Required
M5.4.2	Error	D	Macros %name% and %name% are identical in the first %param%	Required

Code review reference	Type	D/U	Description	Level
M5.5.1	Error	D	Macro %name% and identifier %name% are identical in the first %param% characters.	Required
M5.5.2	Error	D	Macro %name% and identifier %name% are identical in the first %param% characters ignoring case.	Required
M5.6	Error	D	Macro %name% and identifier %name% are identical in the first %name% %param% characters ignoring case. The typedef name %name% should not be reused except for its tag. Name already found in %location%	Required
M5.7.1	Error	D	The tag name %name% should not be reused	Required
M5.7.2	Error	D	A struct and union cannot use the same tag name	Required
M5.8	Error	D	Identifiers that define objects or functions with external linkage shall be unique	Required
M5.9	Error	D	Identifiers that define objects or functions with internal linkage should be unique	Advisory

Code review reference	Type	D/U	Description	Level
E5.1	Error	D	External identifiers shall not be ambiguous because of possible character confusion.	Advisory
E5.2	Error	D	External identifiers shall not be ambiguous because of character repetition	Advisory
E5.3	Warning	D	The identifier<name> should not be reused. Identifier already found in %location%	Advisory
E5.4	Error	D	Identifier %name% in an inner scope hides the same identifier in an outer scope : %location%	Advisory
E5.5	Error	D	The typedef name %name% should not be reused even for its tag. Name already found in %location%	Advisory
M6.1.1	Error	D	Bit fields should only be of type 'unsigned int' or 'signed int'	Required
M6.1.2	Error	D	Bit fields should not be of type 'enum'	Required
M6.1.3	Error	D	Bit fields should only be of explicitly signed or unsigned type	Required
M6.1.4	Error	D	Bit fields should not be of type 'bool' under c99	Required

Code review reference	Type	D/U	Description	Level
M6.1.5	Error	D	Bit fields should not be of type 'boolean' outside c99	Required
M6.2	Error	D	Single-bit fields shall not be of a signed type	Required
E6.1	Warning	D	The C language numeric type %name% should not be used directly but instead used to define typedef	Required
E6.2	Warning	D	The implicit 'int' type should not be used	Required
M7.1	Error	D	Octal constants shall not be used	Required
M7.2	Error	D	A "u" or "U" suffix shall be applied to all integer constants that are represented in an unsigned type"	Required
M7.3	Error	D	The lowercase character "l" shall not be used in a literal suffix"	Required
M7.4	Error	D	A string literal shall not be assigned to an object unless the object's type is pointer to a const-qualified char	Required
M8.1	Error	D	Types shall be explicitly specified	Required
M8.2.1	Error	D	The function prototype should name all its parameters	Required

Code review reference	Type	D/U	Description	Level
M8.2.2	Error	D	Functions with no parameters should use the void type	Required
M8.2.3	Error	D	The type of parameter %name% should be explicitly stated	Required
M8.3.1	Error	D	Parameters and return types should use compatible type in the declaration and in the definition	Required
M8.3.2	Error	D	The identifiers used in the prototype and definition should be the same	Required
M8.4.1	Error	D	A prototype for the global function %name% should be declared before defining the function	Required
M8.4.2	Error	D	A prototype for the global object %name% should be declared before defining the object	Required
M8.4.3	Error	D	If objects or functions are declared multiple times their types should be compatible	Required
M8.5	Error	D	Identifiers %name% that declare objects or functions with external linkage shall be declared once in one and only one file	Required

Code review reference	Type	D/U	Description	Level
M8.6	Error	D	Identifiers %name% that declare objects or functions with external linkage shall be unique	Required
M8.7.1	Warning	D	Global object %name% that are only used within the same file should be declared using the static storage-class specifier.	Advisory
M8.7.12	Warning	D	Global function %name% that are only used within the same file should be declared using the static storage-class specifier.	Advisory
M8.8	Error	D	The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage	Required
M8.9	Warning	D	An object should be defined at block scope if its identifier only appears in a single function	Advisory
M8.10	Error	D	Inline function %name% should be static	Required
M8.11	Warning	D	When an array with external linkage is declared, its size should be explicitly specified	Advisory

Code review reference	Type	D/U	Description	Level
M8.14	Error	D	The restrict type qualifier shall not be used	Required
E.8.1	Error	D	Parameters and return types should use exactly the same type names in the declaration and in the definition	Required
E.8.2	Error	D	A prototype for the static function %name% should be declared before defining the function	Required
E.8.3	Error	D	Static function %name% should only be declared in a single file. Redundant declaration found at: %name%	Required
E.8.4	Error	D	Static object %name% should only be declared in a single file. Redundant declaration found at: %location%	Required
E.8.5	Error	D	Either all members or only the first member of an enumerator list should be initialized	Required
E.8.6	Error	D	The body of function %name% should not be located in a header file	Required
E.8.7	Error	D	The memory storage (definition) for the variable %name	Required

Code review reference	Type	D/U	Description	Level
E.8.8	Error	D	% should not be in a header file Functions should not be declared at block scope	Required
E.8.9	Error	D	The global object or function '%name%' should have exactly one external definition. Redundant definition found in %location%	Required
E.8.10	Error	D	The global object or function %name% %name% should have exactly one external definition. No definition found	Required
E.8.11	Error	D	Use the const qualification for variable %name% which is pointer and which is not used to change the pointed object	Required
M9.2	Error	D	The initializer for an aggregate or union shall be enclosed in braces	Required Exception not covered
M9.3	w	D	Arrays shall not be partially initialized	Required Exception not covered
E9.1	Error	D	Variables with automatic storage duration should be initialized before being used	Required

Code review reference	Type	D/U	Description	Level
E9.2	Error	D	The global variable %name% is not initialized	Required
M10.1.1	Error	D	Constraint violation : can't use floating type as operand of "[], %, <<, >>, ~, &, , ^"	Required
M10.1.2	Error	D	Operand should be boolean	Required
M10.1.3	Error	D	Can't use a boolean as a numeric value	Required
M10.1.4	Error	D	Can't use a char as a numeric value	Required
M10.1.5	Error	D	Can't use a not anonymous enum as a numeric value	Required
M10.1.6	Error	D	Shift and bitwise operations should be performed on unsigned value	Required
M10.1.7	Error	D	Right hand operand of shift operation should be performed on unsigned value	Required
M10.1.8	Error	D	Unary minus operation should not be performed on unsigned value	Required
M10.2	Error	D	Expressions of essentially character type shall not be used inappropriately in addition and subtraction operations	Required

Code review reference	Type	D/U	Description	Level
M10.3.1	Error	D	The value of an expression shall not be assigned to an object with a narrower essential type	Required
M10.3.2	Error	D	The value of an expression shall not be assigned to an object with a different essential type category	Required
M10.4	Error	D	Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category	Required
M10.5	Warning	D	The value of an expression should not be cast to an inappropriate essential type	Advisory
M10.6	Error	D	The value of a composite expression shall not be assigned to an object with wider essential type	Required
M10.7	Error	D	If a composite expression is used as one operand of an operation in which the usual arithmetic conversions are performed then the other operand shall not have wider essential type	Required

Code review reference	Type	D/U	Description	Level
M10.8	Error	D	The value of a composite expression shall not be cast to a different essential type category or a wider essential type	Required
E10.1	Error	D	When using operator '~' or '<<' on 'unsigned char' or 'unsigned int', you should always cast returned value	Required
M11.1	Error	D	A function pointer should not be converted to another type of pointer	Required
M11.2	Error		Conversions shall not be performed between a pointer to an incomplete type and any other type	Required
M11.3.1	Error		Casting an object pointer type to a different object pointer type should not occur	Required
M11.3.2	Error		Casting an object pointer type to a different object pointer type should not occur, especially when object sizes are not the same	Required
M11.3.3	Error		An object pointer should not be converted to another type of pointer	Required

Code review reference	Type	D/U	Description	Level
M11.4	Warning		Casting a pointer type to an integer type should not occur	Advisory
M11.5	Warning		A conversion should not be performed from pointer to void into pointer to object	Advisory
M11.6	Error		A cast shall not be performed between pointer to void and and an arithmetic type	Required
M11.7	Error		A cast shall not be performed between pointer to object and a non-integer arithmetic type	Required
M11.8	Error		Casting of pointers to a type that removes any const or volatile qualification on the pointed object should not occur	Required
M12.1.1	warning		Implicit operator precedence may cause ambiguity. Use parenthesis to clarify this expression	Advisory
M12.1.2	warning		Implicit bitwise operator precedence may cause ambiguity. Use parenthesis to clarify this expression	Advisory
M12.1.3	warning		Parenthesis should be used around expressions that are operands of a logi-	Advisory

Code review reference	Type	D/U	Description	Level
M12.1.4	warning		cal &&&am- p;&am- p; or Parenthesis should be used around expression that is operand of 'sizeof' operator.	Advisory
M12.3	warning		The comma operator should not be used.	Advisory
E12.1	warning		The operator on a Boolean expression should be a logical operator (&&&am- p;, or !)	Advisory
E12.2	warning		Ternary expression '?' should not be used	Advisory
E12.3	error		Expressions should not cause a side effect assignment	Advisory
E12.4	error		The equal or not equal operator should not be used in floating-point expressions	Advisory
M13.3	Warning		a full expression containing an increment (++) or decrement (-) operator should have no other potential side effects other than that caused by the increment or decrement operator	Advisory
M13.4.1	Warning		Boolean expressions should not contain assignment operators.	Advisory

Code review reference	Type	D/U	Description	Level
M13.4.2	Warning		The result of an assignment operator should not be used in an expression	Advisory
M13.6	Error		The operand of the sizeof operator shall not contain any expression which has potential side effects	Required
E13.1	Error		Boolean expressions should not contain side effect operators	Required
E13.2	Error		An expression that contains a side effect should not be used in the right-hand operand of a logical && or operator	Required
E13.3	Error		The function in the right-hand operand of a logical && or operator might cause side effects	Required
M14.1.1	Error		Floating-point variables should not be used to control a for statement	Required
M14.2.1	Error		Only loop counter should be initialized in a for loop initialization part	Required
M14.2.2	Error		In the 'update part' of a 'for statement', only 'loop counter' should be updated	Required

Code review reference	Type	D/U	Description	Level
M14.2.3	Error		There should be one and only one loop counter for loop statement	Required
M14.2.4	Error		Loop counter of a 'for statement' should not be modified within the body of the loop	Required
M14.3.1	Error		Invariant Boolean expressions should not be used	Required
M14.4	Error		Non-Boolean values that are tested against zero should have an explicit test	Required
M15.1	Warning		The goto statement should not be used	Advisory
M15.2	Error		The goto statement shall jump to a label declared later in the same function	Required
M15.3	Error		Any label referenced by a goto statement shall be declared in the same block, or in any block enclosing the goto statement	Required
M15.4	Warning		There should be no more than one break or goto statement used to terminate any iteration statement	Advisory
M15.5	Warning		A function should have a single point of exit at the end	Advisory

Code review reference	Type	D/U	Description	Level
M15.6.1	Error		The switch statement should be followed by a compound statement	Required
M15.6.1	Error		The switch statement should be followed by a compound statement	Required
M15.6.2	Error		The while statement should be followed by a compound statement	Required
M15.6.3	Error		The do..while statement should contain a compound statement	Required
M15.6.4	Error		The for statement should be followed by a compound statement	Required
M15.6.5	Error		The if (expression) construct should be followed by a compound statement	Required
M15.6.6	Error		The else keyword should be followed by a compound statement	Required
M15.7	Error		All if ... else constructs shall be terminated with an else statement	Required
E15.1	Error		Do not use the continue statement	Required

Code review reference	Type	D/U	Description	Level
E15.2	Error		Only one break statement should be used within a loop	Required
E15.3	Error		The return keyword should not be used in a conditional block	Required
E15.4	Error		The else keyword should be followed by either a compound statement or another if statement.	Required
M16.1	Error		All switch statement should be well formed	Required
M16.2	Error		A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement	Required
M16.3	Error		An unconditional break statement shall terminate every switch-clause	Required
M16.4	Error		Every switch statement shall have a default label	Required
M16.5	Error		A default label appear as either the first or the last switch label of a switch statement	Required
M16.6	Error		Every switch statement shall have at least two switch-clauses	Required

Code review reference	Type	D/U	Description	Level
M16.7	Error		A switch expression shall not have essentially Boolean type	Required
E16.1	Error		Case char value is applicable only if the switch statement value is plain character variable	Required
E16.2	Error		A constant should not be used as a switch expression	Required
E16.3	Error		The switch expression should not have side effects	Required
M17.1.1	Error		The function '%name%' should not have a variable number of arguments	Required
M17.1.2	Error		The va_list, va_arg, va_start, va_end and va_copy functions of <stdarg.h> shall not be used	Required
M17.2.1	Error		Recursive functions are not allowed. The function '%name%' is directly recursive	Required
M17.2.2	Error		Recursive functions are not allowed. The function '%name%' is recursive when calling '%name%'	Required
M17.3	Error		A function shall not be declared implicitly	Required
M17.4	Error		All exit paths from a function with non-	Required

Code review reference	Type	D/U	Description	Level
M17.6	Error		void return type shall have an explicit return statement with an expression	Advisory
M17.7	Error		The declaration of an array parameter shall not contain the static keyword between the []	Advisory
E17.1	Error		The value returned by function having non-void return type shall be used	Required
E17.2	Error		The number of arguments used in the call does not match the number declared in the prototype	Advisory
E17.3	Error		Use the const qualification for parameter '%name%' which is pointer and which is not used to change the pointed object	Advisory
M18.4	Error		Function identifiers should always use a parenthesis or a preceding &	Advisory
M18.5	Error		The +, -, += and -= operators should not be applied to an expression of pointer type	Advisory
M18.5	Error		Declarations should contain no more than two levels of pointer nesting	Advisory

Code review reference	Type	D/U	Description	Level
M18.7	Error		Flexible arrays members shall not be declared	Required
M18.8	Error		Variable-length array types shall not be used	Required
M19.2	Warning		The union keyword should not be used	Advisory
E19.1	Error		Structure or union types '%name%' should be finalized before the end of the compilation units	Advisory
M20.1	Warning		#include directive should only precede by preprocessor directives or comments	Advisory
M20.2	Error		The ', or \ character and the /* or // character sequences shall not occur in a header file name"	Required
M20.3	Error		The #include directive shall be followed by either a <file-name> or a filename" sequence"	Required
M20.4	Error		A macro shall not be defined with the same name as a keyword %name%	Required
M20.5	Warning		#undef should not be used	Advisory
M20.6	Error		Token that look like a preprocessing directive should not occur	Required

Code review reference	Type	D/U	Description	Level
M20.7	Error		withing a macro argument Expressions resulting from the expansion of macro parameters shall be enclosed in parenthesis	Required
M20.8	Error		The controlling expression of a #if or #elif preprocessing directive shall evaluate to 0 or 1	Required
M20.9	Error		All identifiers used in the controlling expression of #if or #elif preprocessing directives shall be #define'd before evaluation	Required
M20.10	Warning		The # and ## preprocessor operators should not be used	Advisory
M20.11	Error		A macro parameter immediately following a # operator shall not immediately be followed by a ## operator	Required
M20.12	Error		A macro parameter used as an operand to the # and ## operators shall only be used as an operand to these operators	Required
M20.13	Error		A line whose first token is # shall be a	Required

Code review reference	Type	D/U	Description	Level
M20.14	Error	Error	valid preprocessing directive All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if, #ifdef or #ifndef directive to which they are related	Required
E20.1	Error		Header file contents should be protected against multiple inclusions	Required
E20.2	Error		The # or ## preprocessor operator should not be used more than once	Required
E20.3	Error		Missing argument when calling the macro	Required
E20.4	Error		Only use the 'defined' preprocessor operator with a single identifier	Required
E20.5	Error		Macro definitions or '#undef' should not be located within a block	Required
E20.6	Error		A C macro should only be expanded to a constant, a braced initialiser, a parenthesised expression, a storage class keyword, a type qualifier, or a do-while-zero block	Required

Code review reference	Type	D/U	Description	Level
M21.1.1	Error		#define and #undef shall not be used on a reserved identifier or reserved macro name: Identifier %name% already found in <libname%>	Required
M21.1.2	Error		#define and #undef shall not be used on identifier beginning with an underscore or on 'defined' keyword %name%	Required
M21.2.1	Error		Declared identifier should not be a reserved identifier or reserved macro name: Identifier %name% already found in <libname%>	Required
M21.2.2	Error		Declared identifier should not begin with an underscore or be 'defined' keyword %name%	Required
M21.3	Error		The memory allocation and deallocation functions of <stdlib.h> shall not be used	Required
M21.4	Error		The standard header file <setjmp.h> shall not be used	Required
M21.5	Error		The standard header file <signal.h> shall not be used	Required

Code review reference	Type	D/U	Description	Level
M21.6.1	Error		The input/output library <stdio.h> shall not be used in production code	Required
M21.6.2	Error		The input/output library <wchar.h> shall not be used in production code	Required
M21.7	Error		The library macro or functions atof, atoi, atol and atoll of <stdlib.h> shall not be used	Required
M21.8	Error		The library macro or functions abort, exit, getenv and system of <stdlib.h> shall not be used	Required
M21.9	Error		The library macro or functions bsearch and qsort of <stdlib.h> shall not be used	Required
M21.10	Error		The standard library time and date functions shall not be used	Required
M21.11	Error		The standard header file <tgmath.h> shall not be used	Required
M21.12	Warning		The library macro or function 'fexcept_t', fexcept_t, fexcept_t, fexcept_t, fexcept_t, fexcept_t, FE_INEXACT, FE_DIVBYZERO, FE_-	Advisory

Code review reference	Type	D/U	Description	Level
			UNDERFLOW, FE_-OVERFLOW, FE_IN-VALID or FE_ALL_EX-CEPT' should not be used.	
E21.1	Error		The variable 'errno' should not be used	Required
E21.2	Error		The macro 'offsetof' should not be used	Required
E21.3	Error		The library macro or function 'setjmp, longjmp, sigsetjmp, siglongjmp' should not be used	Required
Rule U99.1	Error		User custom rule	

Configuring code review rules

Code Review for C

The code review tool uses a set of predefined rules. You can select the default rule configuration file for the code review tool. MISRA 2004 and MISRA 2012 from HCL OneTest™ Embedded Studio V8.2.0 are the default installed rule configuration files. You can either disable or set the severity level to Warning or Error.

By default all rules are enabled and produce either an error or a warning in the code review report. You can save multiple customized rule policies.

The default rule policy files are located in the *<installation directory> /plugins/Common/lib/confrule.xml* file for MISRA 2004 and in *<installation directory> /plugins/Common/lib/confrule_2012.xml* for MISRA 2012.

Note All new projects use the default rule configuration file that you have selected in the configuration settings. Do not modify the default rule configuration files. The only change that can be done in the default rule configuration files is to change or disable the severity level of the rule from the settings.

To select the configuration file and disable or set the severity level of code review rules:

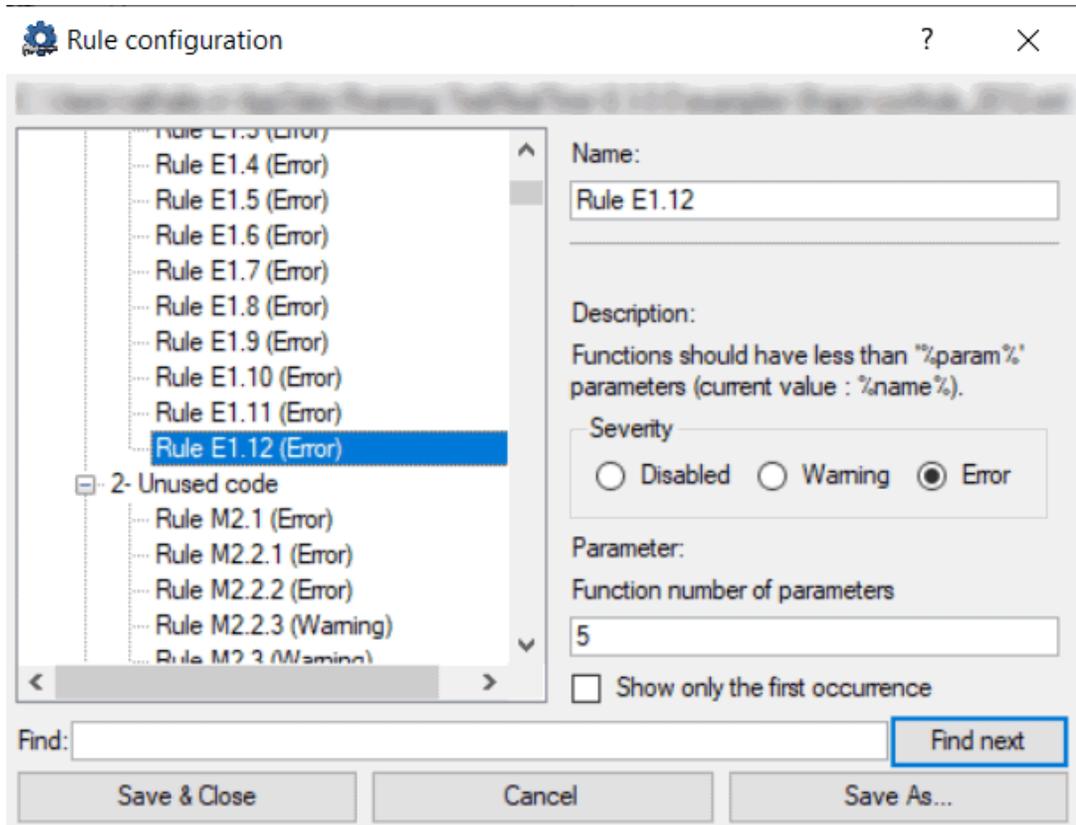
1. Select a node in the **Project Explorer** view and click the **Settings**  button.
2. In the Configuration Settings list, select **Code Review**.

In **Default configuration**, select the MISRA rules to apply to your project: MISRA 2004 or MISRA 2012.

3. To modify the default set of rules, in **Rule configuration**, click ...and select the rule file that you want to configure.
4. In **Rule configuration** click **Edit** . This opens the **Rule Configuration** window where rules are grouped into categories.



Note: You can filter the rules by labels from the **Find** field. Search is not case sensitive. When a rule is selected, its description is displayed on the right panel with the parameter description and value if they are defined in the selected rule.



5. Select the severity level:
 - **Disabled:** The selected rule is ignored. The list of disabled rules is displayed at the end of the report.
 - **Warning:** When any non-compliance instance is found, a warning is displayed in the code review report.
 - **Error:** When any non-compliance instance is found, an error is displayed in the code review report.
6. Select **Show only the first occurrence** to only show the first occurrence of a non-compliance in a file.
7. Select **Save and Close** to save the current configuration or **Save As** to create a new rule configuration file.

If your application is multi-threaded, you can provide the list of entry points to avoid that the rules about 'non-used functions' are raised.

To configure the **Multi_thread** option, follow these steps:

1. Click **Configuration Properties > Runtime Analysis > Multi-threads**.
2. In the right pane, click ... in the value field of the **Entry points** option to open the editor.
3. In the editor, enter the list of entry points for each thread and click **OK**

The Entry point option applies to rule E16.50 (MISRA_2004) and M2.2.2 (MISRA 2012).

Related Topics

[Code review overview on page 233](#) | [Code review settings on page 1008](#) | [Code review MISRA 2004 rules on page 95](#) | [Understanding code review reports on page 299](#)

Code review deviations

In some cases, it can be useful to temporarily ignore a rule non-compliance on a short portion of source code, while documenting the reason why you are allowing this deviation.

About this task

You can justify why you are allowing the deviation in a text. The text is added to the non-compliance in the source code. You can declare a deviation in the source code, for a specified number of lines and for the first or all occurrences of the error, by adding pragma lines to your source code.

1. Open the source file in the Text editor and find the lines of code that you want the rule to ignore.
2. Before the section of code for which compliance to the rule should be ignored, add one of the following lines:

- To justify non-compliance of a rule to the following pragma statement in the first occurrence:

```
#pragma attol crc_justify (<rule>[,<lines>], "<text>")
```

- To justify non-compliance of a rule to the following pragma statement in all occurrences:

```
#pragma attol crc_justify_all (<rule>,<lines>,"<text>")
```

- To justify the first occurrence of non-compliance of a rule in all the files of the current project, including in traps located before the pragma statement:

```
#pragma attol crc_justify_everywhere (<rule>,"<text>")
```

For all the pragma statements: <rule>

- <rule> is the name of the code review rule (for example: 'Rule M8.5').
- <lines> is the number of lines.
- <text> is the reason why the rule is ignored.

The recommended usage for `crc_justify_everywhere` is to create a specific source file containing only the list of pragma statements and add this file to the project.

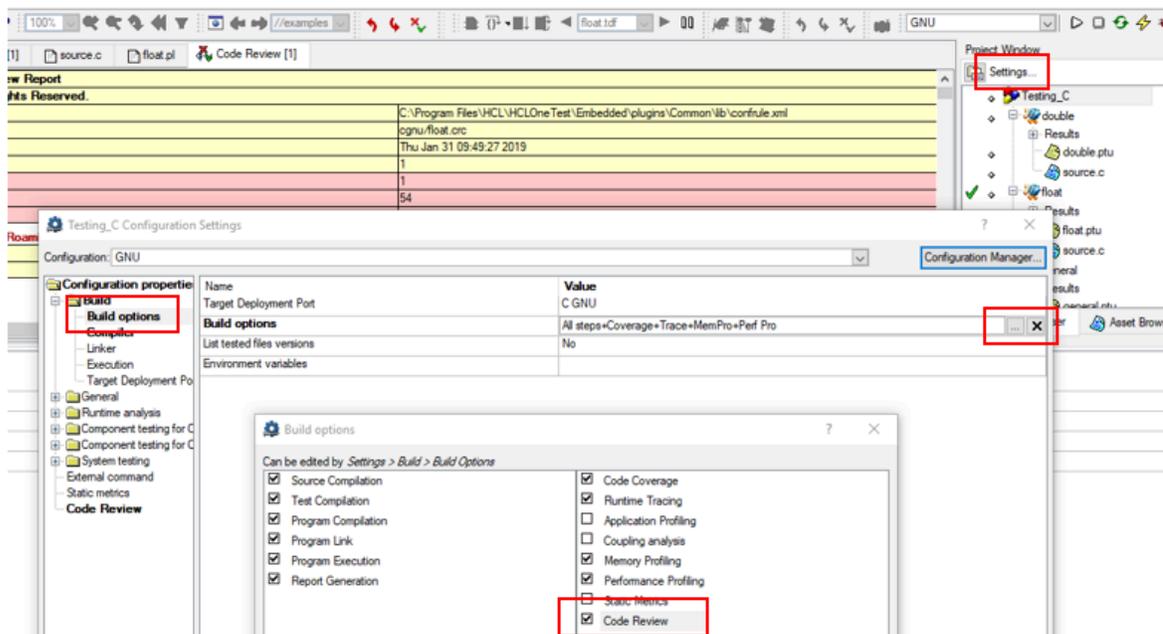
Running a code review

Code Review for C

You can use the code review tool on any test or application node or a single source file. The code review tool is run on the source code whenever you build the file.

To enable the code review tool on a source file, test or application node:

1. In **Project Explorer**, select the node that you want to review, right-click and select **Build > Build options**.
Alternatively, click the **Settings**  button and select **Build options**.
2. Select **Code Review**.



To perform a code review without compiling and executing the application:

1. In the **Project Browser**, select the node that you want to check.
2. Select **Build > Options** from the menu or click the **Settings**  button and select **Build > Build options**.
3. Clear all build options and build steps (in the left) except **Code Review**.
4. Click the **Build**  toolbar button.
5. Double-click the results in the **Project Browser** to open the report. If the report is already open, close the report and reopen it again.

Related Topics

[Working with projects on page 674](#) | [Building and running a node on page 698](#)

Executing the code review from a script

You can execute the build from HCL OneTest™ Embedded Studio graphical interface or for advanced users, from the command line interface.



Note: The following procedure is for advanced-users.

- When **crccc** has been used, use **crclid** as follows:

```
crclid -xref="<model_file>.pl" "<crccc_result_file>.xob" -RULE="<confrule_file>.xml" -TEST
```

- **<model_file>.pl** file will be generated, it contains data needed to perform custom namecheck rule.
- A file named **<model_file>.R99.1.xob** will be generated, it will be used on a final step when executing **crclid**.
- Use this **xob** file for the next call of **crclid**:

```
crclid -crc="<crc_file_name>.crc" "<all other xob file name>.xob" "<model_file>.R99.1.xob" -RULE="<confrule_file>.xml" -TEST
```

Running complete verification of MISRA rules from an application node

To get a complete verification of MISRA rules, you must run the code review from an application node.

About this task

When running a code review from the code review link checker in test mode from HCL OneTest™ Embedded Studio, the option '-TEST' is set by default on all test nodes and, as a consequence some rules are filtered out. To see the list of the rules that are filtered out, see the [C Code Review Linker - crclid](#) page in the Studio Reference category of the help, under 'Runtime Analysis command line interface reference page'.

To enable these rules, you must run a full check of MISRA rules from an application node in HCL OneTest™ Embedded Studio.

Create a project if not already done.

1. Create a project.
2. create an 'application' node.
3. Add all your sources under this node.
4. Select the application node, right-click and select **Settings > Build > Build options**.
5. In the right panel, deselect all options except the **Code Review** option.
Edit `compiler / user include` directories to point to your header files.
6. Click **Apply** and then **OK**.
7. To run the build, select the application node, right-click and select **Build**.
8. To see the report, select the application node, right-click and select **Open Report > Code Review**.

Viewing code review results

Code Review for C

The GUI displays code review results in the Report Viewer.

Reloading a Report

If you open the report in the report viewer and the report has been updated in the meantime, you can use the Reload command to refresh the display:

To reload a report:

1. From the View Toolbar, click the **Reload** button.

Exporting a Report to HTML

Code review results can be exported to an HTML file.

To export results to an HTML file:

1. From the **File** menu, select **Export** and **Export Project Report in HTML files format**.
2. In the **HTML Export Configuration** window, select **Code Review**.
3. Specify an output directory and click **Export**.

Related Topics

[Understanding code review reports on page 299](#) | [Code review MISRA 2004 rules on page 95](#)

Understanding code review reports

Code Review for C

The Code Review report lists the rules that produced an error or a warning.

Report explorer

The Report Explorer window displays a list of rules that were broken for each source file and function. You can use these elements in this view to navigate through the report.

Report summary

At the top of the Code Review report a summary provides information about the general configuration, the date and the number of analyzed files.

It also lists the number of errors and warnings that were encountered.

Code review details

The code review report lists the rules for which errors or warnings were detected. It also provides information about the location of the error. You can click the title to go directly to the corresponding line in the source code.

Related Topics

[Using the code review viewer on page 298](#) | [Viewing reports on page 704](#) | [Understanding reports on page 706](#) | [Code review MISRA 2004 rules on page 95](#)

Customizing the code review report

The default code review report is generated in an HTML format from a template named **misrareport.template** as that you can modify to customize the code review reports.

The code review HTML reports are generated from a template named **misrareport.template** that you can find in the following folder as a text file:

- On Windows: <installation_directory>\lib\reports
- On Unix: <installation_directory>/lib/reports

The template file uses the following JavaScript libraries:

- Bootstrap
- JQuery
- Font Awesome
- VisJS
- Chart

These libraries are not provided. An internet connection is required to open the report. If you don't have any internet connection, download the libraries (.css and .js files), copy them in the folder in which the report is saved, and modify the template file as follows:

Replace the following block of lines:

```
<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css"
  integrity="sha384-MCw98/SFnGE8fJT3GXwEOngsV7Zt27NXFoaoApmYm81iuXoPkFOJwJ8ERdknLPMO"
  crossorigin="anonymous">
<link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.5.0/css/all.css"
  integrity="sha384-B4dIYHKNBt8Bc12p+WXckhzcICo0wtJAoU8YZTY5qE0Id1GSseTk6S+L3B1XeVIU"
  crossorigin="anonymous">
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/vis/4.21.0/vis.min.css">
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/Chart.js/2.8.0/Chart.min.css">
...
<script src="https://code.jquery.com/jquery-3.3.1.slim.min.js"
  integrity="sha384-q8i/X+965Dz00rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8abtTE1Pi6jizo"
  crossorigin="anonymous"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.3/umd/popper.min.js"
  integrity="sha384-ZMP7rVo3mIyKv+2+9J3UJ46jBk0WLaUAdn689aCwoqbBJiSnjAK/l8WvCWPiPm49"
  crossorigin="anonymous"></script>
<script src="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/js/bootstrap.min.js"
  integrity="sha384-ChfqxuZUCnJSK3+MXmPNIyE6ZbWh2IMqE241rYiqJxyMiZ6OW/JmZQ5stwEULTy"
  crossorigin="anonymous"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/vis/4.21.0/vis.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/Chart.js/2.8.0/Chart.min.js"></script>
```

With the following one:

```

<link rel="stylesheet" href="./bootstrap.min.css">
<link rel="stylesheet" href="./all.css">
<link rel="stylesheet" href="./vis.min.css">
<link rel="stylesheet" href="./Chart.min.css">
...
<script src="./jquery-3.3.1.slim.min.js"></script>
<script src="./popper.min.js"></script>
<script src="./bootstrap.min.js"></script>
<script src="./vis.js"></script>
<script src="./Chart.min.js"></script>

```

The following sections give the list of elements that you can use in the raw data and the JavaScript functions to customize your report file.

Data format

The **misrareport.template** template consists of two sections:

- The HTML section that is common to all reports,
- A JavaScript section that sets tables depending on two variables that are initialized dynamically when the report is created:

```

var data = {{json}};           // the raw data, in json format
var d = new Date({{date}})    // the generation date

```

Raw data contains the following information at the top level:

- **output** is the name of the json file that contains the raw data
- **title** is the internal title of the report (displayed in the "crc" file format)
- **configurationTitle** is the title of the used configuration file
- **systemLevel** is the C level norm used. The possible values are "C90", "C90 and Normative Addendum 1", "C99" or "C11"
- **configuration** is the configuration file used to generate this report
- **date** is the generation date of raw data
- **nbAnalyzedFiles** is the number of analyzed files
- **nbFilesKO** is the number of files containing errors
- **nbFilesOK** is the number of files without errors
- **nbError** is the total number of all errors in all analyzed files
- **nbWarning** is the total number of all warnings in all analyzed files
- **files** is the array of **file element** (each one represents a physical file) or array of **deactivated element**
- **statistics** is the array of **rule statistics element**



Example:

```

{
  "output": "../build/fullreport_1.crc.json",
  "title": "HCL OneTest (TM) Embedded MISRA C:2012 Report using C90",
  "configurationTitle": "MISRA C:2012",
  "systemLevel": "C90",

```



```
"configuration": "C:\\Program
Files\\HCL\\HCLOneTest\\Embedded\\plugins\\Common\\lib\\confrule_2012.xml",
"date": "Mon Oct 19 15:52:07 2020",
"nbAnalyzedFiles": 5,
"nbFilesK0": 4,
"nbFilesOK": 1,
"nbError": 49,
"nbWarning": 68,
"files": [
],
"statistics": [
]
}
```

Each **file element** represents an analyzed source file. It contains the following information at the top level:

- **source** is the physical location of source file
- **fileDate** is the date of last editing of this source
- **nbErrorOrWarning** is the total of error or warning in this file
- **content** is an array of **rule element** (if the rule is directly raised at file level) or **function element**. It is always available but it can be empty (file with no function and with no error or warning)

Each **function element** represents a function. It contains the following information at the top level:

- **function** is the name of the function
- **kind** is the analysis result of this function. The possible values are 'Failed' or 'Passed'
- **content** is an array of **rule element** (rules that are raised inside this function). It is always available but it can be empty (function with no error or no warning)



Examples:

file element

```
{
"source": "C:\\workspace\\project\\src\\core.h",
"fileDate": "Mon Sep 07 10:31:50 2020",
"nbErrorOrWarning": 25,
"content": [
]
}
```

function element:

```
{
"function": "win",
"kind": "Failed",
"content": [
]
}
```

Each **rule element** represents a triggered rule, justified or not. It contains the following information at the top level:

- **rule** is the name of the rule, corresponding to its label defined in the configuration file
- **group** is the family of this rule, it corresponds to the label of the rule's group that is defined in the configuration file
- **kind** is the severity of the rule. The possible values are 'error', 'warning' or 'info', depending on the error level in the configuration file and on the possible justification (the justified rules have an 'info' type value)
- **line** is the line of the current file where the rule was triggered
- **column** is the column of the current file where the rule was triggered
- **text** is the rule description. It is related to the rule text in configuration file
- **justification** is the justification text for the rule. This field is optional, and is present only if the rule is justified

**Example:**

```
{
  "rule": "M21.6.1",
  "group": "21- Standard libraries",
  "kind": "info",
  "line": 21,
  "column": 10,
  "text": "The input/output library <stdio.h> shall not be used in production code.",
  "justification": "This rule does not apply to the following line"
}
```

Each deactivated element represents a group of rules that is deactivated for a specific reason. It contains the following information at the top level:

- **desactivated_rules_by_user** is used for all the rules that are deactivated when it is used in the configuration file with the error set to level 0. This field is optional, it can be empty, or you can enter an array of **desactivated rule element**

**Example:**

```
{
  "desactivated_rules_by_user": [
  ]
}
```

- **desactivated_rules_by_test_option** is used for all the rules that are deactivated by using the "-test" option. This field is optional, it can be empty, or you can enter an array of **desactivated rule element**

**Example:**

```
{
  "desactivated_rules_test_option": [
```



```
]
}
```

Each **deactivated rule element** represents a deactivated rule for any reason. It contains the following information at the top level:

- **rule** is the name of the rule, it corresponds to the rule label that is defined in the configuration file
- **text** is the rule description, it corresponds to the rule text in configuration file

**Example:**

```
{
  "rule": "E15.3",
  "text": "The return keyword should not be used in a conditional block."
}
```

Each **rule statistics element** represents global statistics for the rule raised during test. It contains the following information at the top level:

- **ruleStatistics** is the array of the **statistic rule element**

**Example:**

```
{
  "rulesStatistics": [
  ]
}
```

Each **statistic rule element** contains a rule that was raised one or several times. It contains the following information at the top level:

- **rule** is the name of the rule. It corresponds to the rule label that is defined in the configuration file
- **kind** is the severity of the rule. The possible values are 'error' or 'warning' that correspond to the error level in the configuration file
- **occurrences** is the number of times that the rule was raised

**Example:**

```
{
  "rule": "M17.7",
  "kind": "error",
  "text": "When a function returns a value, this value should be used.",
}
```



```
"occurrences": 4
}
```

Javascript functions

You can find in the **misrareport.template** template a set of JavaScript functions.

Some of the helper functions simplify access to "raw data":

- **isFct**(element) checks whether an element is a function or not
- **isFile**(element) checks whether an element is a file or not
- **isFileInError**(element) checks whether an element is a file that contains an error or a warning
- **isFctPassed**(element) checks whether an element is a passed function or not
- **isFctFailed**(element) checks whether an element is a failed function or not
- **isRuleError**(element) checks whether a rule level is error or not
- **isRuleWarning**(element) checks whether a rule level is warning or not
- **isRuleInfo**(element) checks whether a rule level is an information or not
- **isRuleJustified**(element) checks whether a rule is justified or not

Other functions are used to display each section of the report:

- **emptyLine**() displays an empty line (helper function)
- **startFile**(element) is called at start of a file element.
- **endFile**() is called at end of a file element.
- **startFileRules**() is called at the beginning of a group of rules that is relative to a file. Used to display array headers
- **endFileRules**() is called at end of a group of rules relative to a file.
- **startFileFunctions**() is called at the beginning of a function
- **rule**(element) is called to display details of a raised rule.

The last section is a set of functions that is used to display summaries:

- **displayDesactivatedbytest**(elem) displays all deactivated rules by using the '-test' option
- **displayDesactivatedbyuser**(elem) displays all deactivated rules that are used in the configuration file
- **displayrulesstatistics**(elem) displays statistics for all rules that are raised during the test

The main algorithm dispatches the function calls by parsing the raw data.

Analyzing running applications

The runtime analysis feature set of HCL OneTest™ Embedded allows you to closely monitor the behavior of your application for debugging and validation purposes. Each feature *instruments* the source code providing real-time analysis of the application while it is running, either on a native or embedded target platform.

To learn about	See
How to perform runtime analysis on your source code	Using Runtime Analysis Features on page 306
Detecting memory leaks in C and C++ source code	About Memory Profiling on page 363
Measuring software performance with Performance Profiling	About Performance Profiling on page 384
Performing code and test coverage with Code Coverage	About Code Coverage on page 54
Obtaining real-time UML sequence diagram traces from your software with Runtime Tracing	About Runtime Tracing on page 394

Runtime analysis overview

The runtime analysis tools of HCL OneTest™ Embedded allow you to closely monitor the behavior of your application for debugging and validation purposes.

These features use source code insertion to instrument the source code providing real-time analysis of the application while it is running, either on a native or embedded target platform.

- [Memory Profiling on page 363](#) analyzes memory usage and detects memory leaks.
- [Performance Profiling on page 384](#) provides metrics on execution time for each procedure/function/method of the application. For C language, it also provides an estimation of Worst Case Estimation Time.
- [Code Coverage on page 54](#) performs code coverage analysis.
- [Control Coupling on page 162](#) provides coverage information on Control Coupling that represent the interactions between modules (C language only).
- [Data Coupling on page 171](#) provides coverage information on def/use pairs identified in the application(C language only).
- [Worst Stack Size on page 179](#) computes an estimation of the maximum of the application stack size (C language only).
- [Runtime Tracing on page 394](#) draws a real-time UML Sequence Diagram of your application.
- [Contract Check on page 513](#) (for C++ only) verifies behavioral assertions during execution of the code and produces a [Contract Check sequence diagram. on page 525](#)

Each of these runtime analysis tools can be used together with any of the automated testing features providing, for example, test coverage information.

Note SCI instrumentation of the source code generates a certain amount of overhead, which can impact application size and performance. See [Source code instrumentation overview on page 11](#) for more information.

Here is a basic rundown of the main steps to using the runtime analysis feature set.

To use the runtime analysis tools:

1. From the **Start** page, set up a new project. This can be done automatically with the [New Project Wizard on page 664](#).
2. Follow the [Activity Wizard on page 663](#) to add your application source files to the workspace.
3. Select the source files under analysis in the wizard to create the application node.
4. Select the runtime analysis tools to be applied to the application in the Build options.
5. Use the [Project Explorer on page 1021](#) to set up the test campaign and add any additional runtime analysis or test nodes.
6. [Run the application node on page 698](#) to build and execute the instrumented application.
7. [View and analyze the generated analysis and profiling reports on page 683](#).

The runtime analysis tools can be run within a test by simply adding the runtime analysis setting to an existing test node.

The runtime analysis tools for C and C++ can also be used in an Eclipse development environment. Runtime or static analysis tools do not run on System Testing nodes.

Related Topics

[Reducing Instrumentation Overhead on page 54](#)

[About Memory Profiling on page 363](#)

[About Performance Profiling on page 384](#)

[About Code Coverage on page 54](#)

[About Runtime Tracing on page 394](#)

Profiling shared libraries

Runtime Analysis

In order to perform runtime analysis on a shared library, you must create an application node containing both a small program that uses the library, and a reference link to the library.

After the execution of the application node, the runtime analysis results are located in the application node.

To profile a shared library:

1. Add the library to your project as described in [Using library nodes on page 686](#).
2. Create an empty application node:
 - Right-click a group or project node and select **Add Child** and **Application** from the popup menu.
 - Enter the name of the application node
3. Inside the application node, create a source file containing a short program that uses the shared library.
4. Link the application node to the shared library:
 - Right-click the application or test node that will use the shared library and select **Add Child** and **Reference** from the popup menu.
 - Select the library node and click **OK**.
5. Select the application node, click the Settings button, and set the Build options to include the runtime analysis tools that you want to use.
6. Build and execute the application node.

Example

An example demonstrating how to use Runtime Analysis tools on shared libraries is provided in the **Shared Library** example project. See [Example projects on page 677](#) for more information.

Related Topics

[Using library nodes on page 686](#) | [Testing shared libraries on page 449](#) | [Selecting Build Options for a Node on page 699](#)

Code coverage

Code coverage overview

Source code coverage consists of identifying which portions of a program are executed or not during a given test case. Source code coverage is recognized as one of the most effective ways of assessing the efficiency of the test cases applied to a software application.

The code coverage tool can provide the coverage information for the following source code elements:

- Statement blocks, decisions, and loops.
- Function or procedure calls.
- Basic conditions, modified conditions/decisions (MC/DC), multiple condition, and forced condition.
- Procedure entries and exits.
- Terminal or potentially terminal statements
- Statements that are considered non-coverable in C.

See [Coverage levels on page 55](#) for more details about each coverage level.

Information modes

The information mode is the method used to code the trace output. This has a direct impact of the size of the trace file as well as on CPU overhead. You can change the information mode in the coverage type settings. See [Changing a code coverage settings on page 63](#).

There are three information modes:

- Default mode: Each branch generates one byte of memory. This offers the best compromise between code size and speed overhead.
- Compact mode: This is functionally equivalent to Pass mode, except that each branch needs only one bit of storage instead of one byte. This implies a smaller requirement for data storage in memory, but produces a noticeable increase in code size (shift/bits masks) and execution time.
- Hit Count mode: In this mode, instead of storing a Boolean value indicating coverage of the branch, a specific count is maintained of the number of times each branch is executed. This information is displayed in the code coverage report.

Count totals are given for each branch, for all trace files transferred to the report generator as parameters. In the code coverage report, branches that have never been executed are highlighted with an asterisk '*'. The maximum count in the report generator depends on the amount of memory available on the computer running the tests. If this maximum count is reached, the report signals it with a Maximum reached message.



Note: The last bracket (}) in a function after a return statement is always displayed in red in the coverage report, even if the function reports 100% coverage.

On-the-fly display

By default, code coverage generates a report when the execution ends. The *on-the-fly* mode generates code coverage results dynamically during the execution. This is useful for applications that never exit or to interact with the execution during the analysis, for example if you want to stop the code coverage when you reach a specified coverage rate threshold.

Information Modes

Code Coverage for Ada, C and C++

The Information Mode is the method used by Code Coverage to code the trace output. This has a direct impact of the size of the trace file as well as on CPU overhead.

You can change the information mode used by Code Coverage in the Coverage Type settings. There are three information modes:

- **Default** mode
- **Compact** mode
- **Hit Count** mode

Default Mode

When using **Default** or *Pass* mode, each branch generates one byte of memory. This offers the best compromise between code size and speed overhead.

Compact Mode

The **Compact** mode is functionally equivalent to *Pass* mode, except that each branch needs only one bit of storage instead of one byte. This implies a smaller requirement for data storage in memory, but produces a noticeable increase in code size (shift/bits masks) and execution time.

Hit Count Mode

In **Hit Count** mode, instead of storing a Boolean value indicating coverage of the branch, a specific count is maintained of the number of times each branch is executed. This information is displayed in the Code Coverage report.

Count totals are given for each branch, for all trace files transferred to the report generator as parameters.

In the Code Coverage report, branches that have never been executed are highlighted with asterisk '*' characters.

The maximum count in the report generator depends on the machine on which tests are executed. If this maximum count is reached, the report signals it with a **Maximum reached** message.

Related Topics

[About Code Coverage on page 54](#) | [Selecting Coverage Type on page 310](#) | [Estimating Instrumentation Overhead on page 51](#) | [Reducing Instrumentation Overhead on page 54](#)

Coverage types

Code Coverage for Ada, C and C++

The Code Coverage feature provides the capability of reporting of various source code units and branches, depending on the coverage type selected.

By default, Code Coverage implements full coverage analysis, meaning that all coverage types are instrumented by source code insertion (SCI). However, in some cases, you might want to reduce the scope of the Code Coverage report, such as to reduce the overhead generated by SCI for example.

Branches

When referring to the Code Coverage feature, a *branch* denotes a generic unit of enumeration. For each branch, you specify the coverage type. Code Coverage instruments each branch when you compile the source under test.

Coverage Levels

The following table provides details of each coverage type as used in each language supported by the product

Coverage Level	Languages		
Block coverage	C on page 325	Ada on page 312	C++ on page 338
Call coverage	C on page 329	Ada on page 315	
Condition coverage	C on page 330	Ada on page 316	C++ on page 343
ATC coverage		Ada on page 316	
Function, unit or method coverage	C on page 335	Ada on page 320	C++ on page 341
Link files		Ada on page 322	
Templates			C++ on page 348
Additional statements	C on page 337	Ada on page 324	C++ on page 349

To select a coverage level:

1. Right-click the application or test node concerned by the Code Coverage report.
2. From the pop-up menu, select **Settings**.
3. In the Configuration list, expand **Code Coverage** and select **Instrumentation Control**.
4. Select or clear the coverage levels as required.
5. Click **OK**.

Related Topics

[Source code instrumentation overview on page 11](#) | [Generating SCI Dumps on page 1051](#) | [Reducing Instrumentation Overhead on page 54](#)

Ada coverage

Block coverage

Code Coverage for Ada

When analyzing Ada source code, Code Coverage can provide the following block coverage types:

- Statement blocks
- Statement and decision blocks
- Statement, decision, and loop blocks
- Asynchronous transfer of control (ATC) blocks

Statement blocks (or simple blocks)

Simple blocks are the main blocks within units as well as blocks introduced by decisions, such as:

- **then** and **else (elsif)** of an **if**
- **loop...end loop** blocks of a **for...while**
- **exit when...end loop** or **exit when** blocks at the end of an instruction sequence
- **when** blocks of a **case**
- **when** blocks of exception processing blocks
- **do...end** block of the **accept** instruction
- **or** and **else** blocks of the **select** instruction
- **begin...exception** blocks of the **declare** block that contain an exceptions processing block.
- **select...then abort** blocks of an **ATC** statement
- sequence blocks: instructions found after a [potentially terminal statement on page 324](#).

A simple block constitutes one branch. Each unit contains at least one simple block corresponding to its body, except packages that do not contain an initialization block.

Decision coverage (implicit blocks)

An if statement without an else statement introduces an implicit block.

```

-- Function power_10
-- -block=decision or -block=implicit
function power_10 ( value, max : in integer) return integer is
ret, i : integer ;
begin
if ( value == 0 ) then
return 0;
-- implicit else block
end if ;
for i in 0..9
loop
if ( (max /10) < ret ) then
ret := ret *10 ;
else
ret := max ;
end if ;
end loop ;
return ret;
end ;

```

An implicit block constitutes one branch.

Implicit blocks refer to simple blocks to describe possible decisions. The Code Coverage report presents the sum of these decisions as an absolute value and a ratio.

Loop coverage (logical blocks)

A **for** or **while** loop constitutes three branches:

- The simple block contained in the loop is never executed: the exit condition is *true* immediately
- The simple block is run only once: the exit condition is *false*, and then *true* on the next iteration
- The simple block run at least twice: the exit condition is *false* at least twice, then finally *true*)

A **loop...end loop** block requires only two branches because the exit condition, if it exists, is tested within the loop:

- The simple block is played only once: the exit condition is *true* on the first iteration, if the condition exists
- The simple block is played at least twice: the exit condition *false* at least once and then finally *true*, if the condition exists

In the following example, you need to execute the function **try_five_times()** several times for 100 % coverage of the three logical blocks induced by this while loop.

-- Function try_five_times

function try_five_times return integer is

result, i : integer := 0 ;

begin

-- try is any function

while (i < 5) and then (result <= 0) loop

result := try ;

i := integer'succ(i);

end loop ;

return result;

end ; -- 3 logical blocks

Logical blocks are attached to the **loop** introduction keyword.

Asynchronous transfer of control (ATC) blocks

This coverage type is specific to the Ada 95 asynchronous transfer of control (ATC) block statement (see your Ada documentation).

The ATC block contains tree branches:

- **Control immediately transferred:** The sequence of control never passes through the block then abort /end select, but is immediately transferred to the block select/then abort.
- **Control transferred:** The sequence of control starts at the block then abort/end select, but never reaches the end of this block. Because of trigger event appearance, the sequence is transferred to the block select/then abort.
- **Control never transferred:** Because the trigger event never appears, the sequence of control starts and reaches the end of the block then abort/end select, and was never transferred to the block select/then abort.

In the following example, you need to execute the **compute_done** function several times to obtain full coverage of the three ATC blocks induced by the select statement:

```
function compute_done return boolean is
result : boolean := true ;

begin
-- if computing is not done before 10s ...

select

delay 10.0;

result := false ;

then abort

compute;

end select;

return result;

end ; -- 3 logical blocks
```

Code Coverage blocks are attached to the **Select** keyword of the ATC statement.

Related Topics

[Selecting coverage types on page 310](#) | [Code Coverage settings on page 992](#)

Call coverage

Code Coverage for Ada

When analyzing Ada source code, Code Coverage can provide coverage of function, procedure, or entry calls.

Code Coverage defines as many branches as it encounters function, procedure, or entry calls.

This type of coverage ensures that all the call interfaces can be shown to have been exercised for each Ada unit (procedure, function, or entry). This is sometimes a pass/fail criterion in the software integration test phase.

Related Topics

[Selecting coverage types on page 310](#) | [Code Coverage settings on page 992](#)

Condition Coverage

Code Coverage for Ada

Basic Conditions

Basic conditions are operands of logical operators (standard or derived, but not overloaded) or, xor, and, not, or else, or and then, wherever they appear in ADA units. They are also the conditions of if, while, exit when, when of entry body, and when of select statement, even if these conditions do not contain logical operators. For each of these basic conditions, two branches are defined: the sub-condition is true and the sub-condition is false.

A basic condition is also defined for each when of a case statement, even each sub-expression of a compound when, that is when A | B: two branches.

-- power_of_10 function -- -cond

Function power_of_10(value, max : in integer)

is

result : integer ;

Begin

if value = 0 then

return 0;

end if ;

result := value ;

for i in 0..9 loop

if (max > 0) and then ((max / value) < result) then

result := result * value;

else

result := max ;

end if ;

```
end loop;
```

```
return result ;
```

```
end ; -- there are 3 basic conditions (and 6 branches).
```

```
-- Near_Color function
```

```
Function Near_Color ( color : in ColorType ) return ColorType
```

```
is
```

```
Begin
```

```
case color is
```

```
when WHITE | LIGHT_GRAY => return WHITE ;
```

```
when RED | LIGHT_RED .. PURPLE => return RED ;
```

```
end case ;
```

```
End ; -- there are 4 basic conditions (and 4 branches).
```

Two branches are enumerated for each boolean basic condition, and one per case basic condition.

Forced Conditions

A forced condition is a multiple condition in which any occurrence of the or else operator is replaced with the or operator, and the and then operator is replaced with the and operator. This modification forces the evaluation of the second member of these operators. You can use this coverage type after modified conditions have been reached to ensure that all the contained basic conditions have been evaluated. With this coverage type, you can be sure that only the considered basic condition value changes between both condition vectors.

```
-- Original source : -- -cond=forceevaluation
```

```
if ( a and then b ) or else c then
```

```
-- Modified source :
```

```
if ( a and b ) or c then
```

Note This replacement modifies the code semantics. You need to verify that using this coverage type does not modify the behavior of the software.

Example

```
procedure P ( A : in tAccess ) is
```

```
begin
```

if A /= NULL and then A.value > 0 -- the evaluation of A.value will raise an

-- exception when using forced conditions

-- if the A pointer is nul

then

A.value := A.value - 1;

end if;

end P;

Modified Conditions

A modified condition is defined for each basic condition enclosed in a composition of logical operators (standard or derived, but not overloaded). It aims to prove that this condition affects the result of the enclosing composition. To do that, find a subset of values affected by the other conditions, for example, if the value of this condition changes, the result of the entire expression changes.

Because compound conditions list all possible cases, you must find the two cases that can result in changes to the entire expression. The modified condition is covered only if the two compound conditions are covered.

-- State_Control state -- -cond=modified

Function State_Condtol return integer

is

Begin

if ((flag_running and then (process_count > 10))

or else flag_stopped)

then

return VALID_STATE ;

else

return INVALID_STATE ;

end if ;

End ;

-- There are 3 basic conditions, 5 compound conditions

- and 3 modified conditions :
- flag_running : TTX=T and FXF=F
- process_count > 10 : TTX=T and TFF=F
- flag_stopped : TFT=T and TFF=F, or FXT=T and FXF=F
- 4 test cases are enough to cover all the modified conditions :
- TTX=T
- FXF=F
- TFF=F
- FTF=F or FXT=T

Note You can associate a modified condition with more than one case, as shown in this example for **flag_stopped**. In this example, the modified condition is covered if the two compound conditions of at least one of these cases are covered.

Code Coverage calculates cases for each modified condition.

The same number of modified conditions as boolean basic conditions appear in a composition of logical operators (standard or derived, but not overloaded).

Multiple Conditions

A multiple condition is one of all the available cases of logical operators (standard or derived, but not overloaded) wherever it appears in an ADA unit. Multiple conditions are defined by the concurrent values of the enclosed basic boolean conditions.

A multiple condition is noted with a set of T, F, or X letters, which means that the corresponding basic condition evaluates to true or false, or it was not evaluated, respectively. Such a set of letters is called a condition vector. The right operand of or else or and then logical operators is not evaluated if the evaluation of the left operand determines the result of the entire expression.

```
-- State_Control Function -- -cond=compound
Function State_Control return integer
is
Begin
if ( ( flag_running and then ( process_count > 10 ) )
or else flag_stopped
```

```
then  
return VALID_STATE ;  
else  
return INVALIDE_STATE ;  
end if ;  
End ;
```

-- There are 3 basic conditions

-- and 5 compound conditions :

-- $TTX=T \Leftrightarrow ((T \text{ and then } T) \text{ or else } X) = T$

-- $TFT=T$

-- $TFF=F$

-- $FXT=T$

-- $FXF=F$

Code Coverage calculates the computation of every available case for each composition.

The number of enumerated branches is the number of distinct available cases for each composition of logical operators (standard or derived, but not overloaded).

Related Topics

[Selecting coverage types on page 310](#) | [Code Coverage settings on page 992](#)

Unit coverage

Code Coverage for Ada

Unit Entries

Unit entries determine which units are executed and/or evaluated.

-- Function factorial

-- -proc

function factorial (a : in integer) return integer is

begin

```

if ( a > 0 ) then
return a * factorial ( a - 1 );
else
return 1;
end if;
end factorial ;

```

One branch is defined for each defined and instrumented unit. In the case of a package, the unit entry only exists if the package body contains the begin/end instruction block.

For Protected units, no unit entry is defined because this kind of unit does not have any statements blocks.

Unit Exits and Returns

These are the standard exit (if it is coverable), each return instruction (from a procedure or function), and each exception-processing block in the unit.

```

-- Function factorial
-- -proc=ret
function factorial ( a : in integer ) return integer is
begin
if ( a > 0 ) then
return a * factorial ( a - 1 );
else
return 1;
end if ;
end factorial ; -- the standard exit is not coverable

-- Procedure divide
procedure divide ( a,b : in integer; c : out integer ) is
begin
if ( b == 0 ) then
text_io.put_line("Division by zero" );

```

```
raise CONSTRAINT_ERROR;

end if ;

if ( b == 1 ) then

c := a;

return;

end if ;

c := a / b;

exception

when PROGRAM_ERROR => null ;

end divide ;
```

For Protected units, no exit is defined because this kind of unit does not have any statements blocks.

In general, at least two branches per unit are defined; however, in some cases the coding may be such that:

- There are no unit entries or exits (a package without an instruction block (begin/end), protected units case).
- There is only a unit entry (an infinite loop in which the exit from the task cannot be covered and therefore the exit from the unit is not defined).

The entry is always numbered if it exists. The exit is also numbered if it is coverable. If it is not coverable, it is preceded by a terminal instruction containing return or raise instructions; otherwise, it is preceded by an infinite loop.

A raise is considered to be terminal for a unit if no processing block for this exception was found in the unit.

Related Topics

[Selecting coverage types on page 310](#) | [Code Coverage settings on page 992](#)

Link files

Code Coverage for Ada

Link files are the library management system used for Ada Coverage. These libraries contain the entire Ada compilation units contained by compiler sources, the predefined Ada environment and the source files of your projects. You must use link files when using Code Coverage in Ada for the Ada Coverage analyzer to correctly analyze your source code.

You can include a link file within another link file, which is an easy way to manage your source code.

Link File Syntax

Link files have a line-by-line syntax. Comments start with a double hyphen (--), and end at the end of the line. Lines can be empty.

There are two types of configuration lines:

- **Link file inclusion:** The link filename can be relative to the link file that contains this line or absolute.

<link filename> LINK

- **Compilation unit description:** The source filename is the file containing the described compilation unit (absolute or relative to the link filename). The full unit name is the Ada full unit name (beware of separated units, or child units).

<source filename> <full unit name> <type> [ada83]

The <type> is one of the following flags:

- - **SPEC** for specification
 - **BODY** for a body
 - **PROC** for procedure or function

Use the optional **ada83** flag if the source file cannot be compiled in Ada 95 mode, and must be analyzed in Ada 83 mode.

Generating a Link File

The link file can be generated either manually or automatically with the Ada Link File Generator (**attolink**) tool. See the **Studio Reference** section of the help for more information about command line tools.

Sending the Link File to the Instrumentor

The loading order of link files is important. If the same unit name is found twice or more in one (or more) loaded link files, the Instrumentor issues a warning and uses the last encountered unit.

Included link files are analyzed when the file including the link file is loaded.

In Ada, Code Coverage loads the link files in the following order:

- By default, either **adalib83.alk** or **adalib95.alk** is loaded. These files are part of the Target Deployment Port.
- If you use the **-STDLINK** command line option, the specified standard link file is loaded first. See the **Studio Reference section of the help** for more information

- The link file specified by the **ATTOLCOV_ADALINK** environment variable is loaded.
- The link files specified by the **-Link** option is loaded.

Now, you can start analyzing the file instrument.

Loading A Permanent Link File

You can ask Code Coverage to load the link file at each execution. To do that, set the environment variable **ATTOLCOV_ADALINK** with the link filename separated by ':' on a UNIX system, or ';' in Windows. For example:

```
ATTOLCOV_ADALINK="compiler.alk/projects/myproject/myproject.alk"
```

A Link file specified on the command line is loaded after the link file specified by this environment variable.

Related Topics

[Selecting coverage types on page 310](#) | [Code Coverage settings on page 992](#)

Additional Statements

Code Coverage for Ada

Terminal Statements

An Ada statement is terminal if it transfers control of the program anywhere other than to a sequence (*return, goto, raise, exit*).

By extension, a decision statement (*if, case*) is also terminal if all its branches are terminal (i.e., *if, then* and *else* blocks and non-empty *when* blocks contain a terminal instruction). An *if* statement without an *else* statement is never terminal, since one of the blocks is empty and therefore transfers control in sequence.

Potentially Terminal Statements

An Ada statement is potentially terminal if it contains a decision choice that transfers control of the program anywhere other than after it (*return, goto, raise, exit*).

Non-coverable Statements

An Ada statement is detected as being not coverable if it is not a *goto* label and if it is in a terminal statement sequence. Statements that are not coverable are detected by the feature during the instrumentation. A warning is generated to signal each one, which specifies its location source file and line. This is the only action Code Coverage takes for statements that cannot be covered.

Note Ada units whose purpose is to terminate execution unconditionally are not evaluated. This means that Code Coverage does not check that procedures or functions terminate or return.

Similarly, exit conditions for loops are not analyzed statistically to determine whether the loop is infinite. As a result, a for, while or loop/exit when loop is always considered non-terminal (i.e., able to transfer control in its sequence). This is not applicable to loop/end loop loops without an exit statement (with or without condition), which are terminal.

Related Topics

[Selecting coverage types on page 310](#) | [Code Coverage on page 992](#)

C coverage

Block coverage

Code Coverage for C

When running the Code Coverage feature on C source code, HCL OneTest™ Embedded can provide the following coverage types for code blocks:

- Statement Blocks
- Statement Blocks and Decisions
- Statement Blocks, Decisions, and Loops

Statement Blocks (or Simple Blocks)

Simple blocks are the C function main blocks, blocks introduced by decision instructions:

- **THEN** and **ELSE FOR IF**
- **FOR, WHILE** and **DO ... WHILE** blocks
- non-empty blocks introduced by switch case or default statements
- true and false outcomes of ternary expressions (*<expr> ? <expr> : <expr>*)
- blocks following a potentially terminal statement.

```
/* Power_of_10 Function */ /* -block */
int power_of_10 ( int value, int max )
{
int retval = value, i;
if ( value == 0 ) return 0; /* potentially terminal statement */
for ( i = 0; i < 10; i++ ) /* start of a sequence block */
{
```

```

retval = ( max / 10 ) < retval ? retval * 10 : max;
}
return retval;
} /* The power_of_10 function has 6 blocks */

/* Near_color function */
ColorType near_color ( ColorType color )
{
switch ( color )
{
case WHITE :
case LIGHT_GRAY :
return WHITE;
case RED :
case PINK :
case BURGUNDY :
return RED;
/* etc ... */
}
} /* The near_color function has at least 3 simple blocks */

```

Each simple block is a branch. Every C function contains at least one simple block corresponding to its main body.

Decisions (Implicit Blocks)

Implicit blocks are introduced by an **IF** statement without an **ELSE** or a **SWITCH** statement without a **DEFAULT**.

```

/* Power_of_10 function */
/* -block=decision */
int power_of_10 ( int value, int max )
{

```

```

int retval = value, i;

if ( value == 0 ) return 0; else ;

for (i =0;i <10;i++)

{

retval = ( max / 10 ) < retval ? retval * 10 : max;

}

return retval;

}

/* Near_color function */

ColorType near_color ( ColorType color )

{

switch ( color )

{

case WHITE :

case LIGHT_GRAY :

return WHITE;

case RED :

case PINK :

case BURGUNDY :

return RED;

/* etc ... with no default */

default ;;

}

}

```

Each implicit block represents a branch.

Because the sum of all possible decision paths includes implicit blocks as well as statement blocks, reports provide the total number of simple and implicit blocks as a figure and as a percentage. Code Coverage places this information in the **Decisions** report.

Loops (Logical Blocks)

A typical **FOR** or **WHILE** loop can reach three different conditions:

- The statement block contained within the loop is executed zero times, therefore the output condition is *True* from the start
- The statement block is executed exactly once, the output condition is *False*, then *True* the next time
- The statement block is executed at least twice. (The output condition is *False* at least twice, and becomes *True* at the end)

In a **DO...WHILE** loop, because the output condition is tested after the block has been executed, two further branches are created:

- The statement block is executed exactly once. The output is condition *True* the first time.
- The statement block is executed at least twice. (The output condition is *False* at least once, then true at the end)

In this example, the function **try_five_times ()** must run several times to completely cover the three logical blocks included in the **WHILE** loop:

```
/* Try_five_times function */
/* -block=logical */
int try_five_times ( void )
{
int result,i =0;

/*try ()is afunction whose return value depends
on the availability of a system resource, for example */
while ( ( ( result = try ())!=0 )&&
(++i <5 ));
return result;
} /* 3 logical blocks */
```

Related Topics

[Selecting Coverage Types on page 310](#) | [About Code Coverage on page 54](#) | [Code Coverage settings on page 992](#)

Call coverage

Code Coverage for C

When analyzing C source code, Code Coverage can provide coverage of function or procedure calls.

Code Coverage defines as many branches as it encounters function calls.

Procedure calls are made during program execution.

This type of coverage ensures that all the call interfaces can be shown to have been exercised for each C function. This may be a pass or failure criterion in software integration test phases.

You can use the **-EXCALL** option to select C functions whose calls you do not want to instrument, such as C library functions for example.

Example

```
/* Evaluate function */

/* -call */

int evaluate ( NodeTypeP node )
{
if ( node == (NodeTypeP)0 ) return 0;

switch ( node->Type )
{

int tmp;

case NUMBER :

return node->Value;

case IDENTIFIER :

return current value ( node->Name );

case ASSIGN :
```

```
set ( node->Child->Name,  
tmp = evaluate ( node->Child->Sibling ) );  
return tmp;  
case ADD :  
return evaluate ( node->Child ) +  
evaluate ( node->Child->Sibling );  
case SUBTRACT :  
return evaluate ( node->Child ) -  
evaluate ( node->Child->Sibling );  
case MULTIPLY :  
return evaluate ( node->Child ) *  
evaluate ( node->Child->Sibling );  
case DIVIDE :  
tmp = evaluate ( node->Child->Sibling );  
if ( tmp == 0 ) fatal error ( "Division by zero" );  
else return evaluate ( node->Child ) / tmp;  
}  
} /* There are twelve calls in the evaluate function */
```

Related Topics

[C Block Coverage on page 325](#) | [C Condition Coverage on page 330](#) | [C Function Coverage on page 335](#) | [C Additional Statements on page 337](#) | [Code Coverage settings on page 992](#)

Condition coverage

Code Coverage for C

When analyzing C source code, HCL OneTest™ Embedded can provide coverage for:

- Basic condition coverage
- Modified condition/decisioncoverage(MC/DC)

- Multiple condition coverage
- Forced condition coverage

Basic Conditions

Conditions are operands of either `||` or `&&` operators wherever they appear in the body of a C function. They are also **if** and ternary expressions, tests for **for**, **while**, and **do/while** statements even if these expressions do not contain `||` or `&&` operators. Two branches are involved in each condition: the sub-condition can be true or false.

Basic conditions enable different cases or a default (which could be implicit) in a switch. These are distinguished even when they invoke the same simple block. One basic condition is associated with every case and default, whether implicit or not.

In the following example, there are 4*2 basic conditions:

```
/* Power_of_10 function */
/* -cond */
int power_of_10 ( int value, int max )
{
    int result = value, i;
    if ( value == 0 ) return 0;
    for ( i = 0; i < 10; i++ )
    {
        result = max > 0 && ( max / value ) < result ?
        result * value :
        max;
    }
    return result ;
}
```

In the following example, there are 5 basic conditions:

```
/* Near_color function */
ColorType near_color ( ColorType color )
```

```

{
switch ( color )
{
case WHITE :

case LIGHT_GRAY :

return WHITE;

case RED :

case PINK :

case BURGUNDY :

return RED;

/* etc ... */
}
}

```

Two branches are enumerated for each condition, and one per case or default.

Modified Conditions

A modified condition (MC) is defined for each basic condition enclosed in a composition of || or && operators, proving that the condition affects the result of the enclosing composition. For example, in a subset of values affected by the other conditions, if the value of this condition changes, the result of the entire expression changes.

Because compound conditions list all possible cases, you must find the two cases that can result in changes to the entire expression. The modified condition is covered only if the two compound conditions are covered.

In this following example, there are 6 basic conditions (FALSE and TRUE of each), 5 compound conditions, and 3 modified conditions :

```

/* state_control function */

int state_control ( void )

{

if ( ( ( flag & 0x01 ) &&

( instances_number > 10 ) ) ||

( flag & 0x04 ) )

```

```

return VALID_STATE;

else

return INVALID_STATE;

}

```

The conditions can be described as True (T), False (F), or Not evaluated (X), as in the following example:

- flag & 0x01 : TTX=T and FXF=F
- nb_instances > 10 : TTX=T and TFF=F
- flag & 0x04 : TFT=T and TFF=F, or FXT=T and FXF=F

Therefore the 4 following test cases are enough to cover all those modified conditions :

- TTX=T
- FXF=F
- TFF=F
- TFT=T or FXT=T

Note You can associate a modified condition with more than one case, as shown in this example for flag & 0x04. In the example, the modified condition is covered if the two compound conditions of at least one of these cases are covered.

Code Coverage calculates matching cases for each modified condition.

The number of modified conditions matches the number of Boolean basic conditions in a composition of || and && operators.

Multiple Conditions

A multiple (or compound) condition is one of all the available cases for the || and && logical operator's composition, whenever it appears in a C function. It is defined by the simultaneous values of the enclosed Boolean basic conditions.

Remember that the right operand of a || or && logical operator is not evaluated if the evaluation of the left operand determines the result of the entire expression.

In the following example, there are 3 basic conditions and 5 compound conditions:

```

/* state_control function */

/* -cond=compound */

```

```

int state_control ( void )
{
if ( ( ( flag & 0x01 ) &&
( instances_number > 10 ) ) ||
( flag & 0x04 ) )
return VALID_STATE;
else
return INVALID_STATE;
}

```

The conditions can be described as True (T), False (F), or Not evaluated (X), as in the following example:

- $TTX=T \Leftrightarrow ((T \ \&\& \ T) \ || \ X) = T$
- $TFT=T$
- $TFF=F$
- $FXT=T$
- $FXF=F$

Code Coverage calculates every available case for each composition.

The number of enumerated branches is the number of distinct available cases for each composition of || or && operators.

Forced Conditions

Forced conditions are multiple conditions in which the Instrumentor replaces any occurrence of the || and && operators in the code, with | and & binary operators. You can use this coverage type, after evaluating all modified conditions, to be sure that every basic condition has been evaluated. With this forced condition coverage, you can ensure that only the basic condition has changed between two tests.

```
/* User source code */ /* -cond=forceevaluation */
```

```
if ( ( a && b ) || c ) ...
```

```
/* Replaced with the Code Coverage feature with : */
```

```
if ( ( a & b ) | c ) ...
```

```
/* Note : Operands evaluation results are enforced to one if different from 0 */
```

Note This replacement modifies the code semantics. Before running the test, you need to verify that this coverage type does not modify the behavior of the software.

```
int f ( MyStruct *A )
{
if (A && A->value > 0 ) /* the evaluation of A->value will cause a program error using
forced conditions if A pointer
is null */
{
A->value -= 1;
}
}
```

Related Topics

[Selecting coverage types on page 310](#) | [Code Coverage settings on page 992](#)

Function coverage

Code Coverage for C

When analyzing C source code, HCL OneTest™ Embedded can provide the following function coverage:

- Procedure Entries
- Procedure Entries and Exits

Procedure Entries

Inputs identify the C functions that are executed.

```
/* Factorial function */
/* -proc */
int factorial ( int a )
{
if ( a > 0 ) return a * factorial ( a - 1 );
```

```
else return 1;  
}
```

One branch is defined per C function.

Procedure Entries and Exits (Returns and Terminal Statements)

These include the standard output (if coverable), and all return instructions, exits, and other terminal instructions that are instrumented, as well as the input.

```
/* Factorial function */  
  
/* -proc=ret */  
  
int factorial ( int a )  
{  
if ( a > 0 ) return a * factorial ( a - 1 );  
else return 1;  
} /* standard output cannot be covered */  
  
/* Divide function */  
  
void divide ( int a, int b, int *c )  
{  
if ( b == 0 )  
{  
fprintf ( stderr, "Division by zero\n" );  
exit ( 1 );  
};  
if ( b == 1 )  
{  
*c = a;  
return;  
};
```

```
*c = a / b;
}
```

At least two branches are defined per C function.

The input is always enumerated, as is the output if it can be covered. If it cannot, it is preceded by a terminal instruction involving returns or an exit.

In addition to the terminal instructions provided in the standard definition file, you can define other terminal instructions using the pragma **attol exit_instr**.



Note: The last bracket '}' in a function after a return statement is always displayed in red in the coverage report, even if the function reports 100% coverage.

Related Topics

[Selecting coverage types on page 310](#) | [Code Coverage settings on page 992](#)

Additional statements

Code Coverage for C

Terminal Statements

A C statement is *terminal* if it transfers program control out of sequence (**RETURN, GOTO, BREAK, CONTINUE**), or stops the execution (**EXIT**).

By extension, a decision statement (**IF** or **SWITCH**) is terminal if all branches are terminal; that is if the non-empty **THEN ... ELSE, CASE, and DEFAULT** blocks all contain terminal statements. An **IF** statement without an **ELSE** and a **SWITCH** statement without a **DEFAULT** are never terminal, because their empty blocks necessarily continue program control in sequence.

Potentially Terminal Statements

The following decision statements are potentially terminal if they contain at least one statement that transfers program control out of their sequence (**RETURN, GOTO, BREAK, CONTINUE**), or that terminates the execution (**EXIT**):

- **IF** without an **ELSE**
- **SWITCH**
- **FOR**
- **WHILE** or **DO ... WHILE**

Non-coverable Statements in C

Some C statements are considered *non-coverable* if they follow a terminal instruction, a **CONTINUE**, or a **BREAK**, and are not a **GOTO** label. Code Coverage detects non-coverable statements during instrumentation and produces a warning message that specifies the source file and line location of each non-coverable statement.

Note User functions whose purpose is to terminate execution unconditionally are not evaluated. Furthermore, Code Coverage does not statically analyze exit conditions for loops to check whether they are infinite. As a result, **FOR ... WHILE** and **DO ... WHILE** loops are always assumed to be *non-terminal*, able to resume program control in sequence.

Related Topics

[Selecting coverage types on page 310](#) | [Code Coverage settings on page 992](#)

C++ coverage

Block coverage

Code Coverage for C++

When analyzing C++ source code, Code Coverage can provide the following block coverage types:

- Statement Blocks
- Statement Blocks and Decisions
- Statement Blocks, Decisions, and Loops

Statement Blocks

Statement blocks are the C++ function or method main blocks, blocks introduced by decision instructions:

- **THEN** and **ELSE FOR IF, WHILE** and **DO ... WHILE** blocks
- non-empty blocks introduced by **SWITCH CASE** or **DEFAULT** statements
- true and false outcomes of ternary expressions (`<expr> ? <expr> : <expr>`)
- **TRY** blocks and any associated catch handler
- blocks following a potentially terminal statement.

```
int main () /* -BLOCK */
{
try {
if ( 0 )
```

```

{
func ( "Hello" );
}

else

{

throw UnLucky ( );

}

}

catch ( Overflow & o ) {

cout << o.String << '\n';

}

catch ( UnLucky & u ) {

throw u;

} /* potentially terminal statement */

return 0; /* sequence block */

}

```

Each simple block is a branch. Every C++ function and method contains at least one simple block corresponding to its main body.

Decisions (Implicit Blocks)

Implicit blocks are introduced by **IF** statements without an **ELSE** statement, and a **SWITCH** statements without a **DEFAULT** statement.

```

/* Power_of_10 function */

/* -BLOCK=DECISION or -BLOCK=IMPLICIT */

int power_of_10 ( int value, int max )

{

int retval = value, i;

if ( value == 0 ) return 0; else ;

```

```
for ( i = 0; i < 10; i++ )
{
retval = ( max / 10 ) < retval ? retval * 10 : max;
}

return retval;
}

/* Near_color function */

ColorType near_color ( ColorType color )
{
switch ( color )
{
case WHITE :
case LIGHT_GRAY :
return WHITE;

case RED :
case PINK :
case BURGUNDY :
return RED;

/* etc ... with no default */

default ;;
}
}
```

Each implicit block represents a branch.

Since the sum of all possible decision paths includes implicit blocks as well as simple blocks, reports provide the total number of simple and implicit blocks as a figure and a percentage after the term decisions.

Loops (Logical Blocks)

Three branches are created in a for or while loop:

- The first branch is the simple block contained within the loop, and that is executed zero times (the entry condition is false from the start).
- The second branch is the simple block executed exactly once (entry condition true, then false the next time).
- The third branch is the simple block executed at least twice (entry condition true at least twice, and false at the end).

Two branches are created in a **DO/WHILE** loop, as the output condition is tested after the block has been executed:

- The first branch is the simple block executed exactly once (output condition true the first time).
- The second branch is the simple block executed at least twice (output condition false at least once, then true at the end).

```
/* myClass::tryFiveTimes method */ /* -BLOCK=LOGICAL */
int myClass::tryFiveTimes ()
{
int result, i = 0;

/* lets go ( ) is a function whose return value depends
on the availability of a system resource, for example */
while ( ( ( result = lets go ( ) ) != 0 ) &&
( ++i < 5 ) );
return result;
} /* 3 logical blocks */
```

You need to execute the method **tryFiveTimes ()** several times to completely cover the three logical blocks included in the while loop.

Related Topics

[Selecting coverage types on page 310](#) | [Code Coverage settings on page 992](#)

Method coverage

Code Coverage for C++

Inputs to Procedures

Inputs identify the C++ methods executed.

```

/* Vector::getCoord() method */ /* -PROC
*/
int Vector::getCoord ( int index )
{
if ( index >= 0 && index < size ) return Values[index];
else return -1;
}

```

One branch per C++ method is defined.

Procedure Inputs, Outputs and Returns, and Terminal Instructions

These include the standard output (if coverable), all return instructions, and calls to `exit()`, `abort()`, or `terminate()`, as well as the input.

```

/* Vector::getCoord() method */ /* -PROC=RET */
int Vector::getCoord ( int index )
{
if ( index >= 0 && index < size ) return Values[index];
else return -1;
}

/* Divide function */
void divide ( int a, int b, int *c )
{
if ( b ==0 )
{
fprintf ( stderr, "Division by zero\n" );
exit ( 1 );
};
if ( b ==1 )

```

```

{
*c =a;
return;
};
*c =a /b;
}

```

At least two branches per C++ method are defined. The input is always enumerated, as is the output if it can be covered. If it cannot, it is preceded by a terminal instruction involving returns or by a call to `exit()`, `abort()`, or `terminate()`.

Potentially Terminal Statements

The following decision statements are potentially terminal if they contain at least one statement that transfers program control out of its sequence (**RETURN**, **THROW**, **GOTO**, **BREAK**, **CONTINUE**) or that terminates the execution (**EXIT**).

- **IF** without an **ELSE**
- **SWITCH**, **FOR**
- **WHILE** or **DO...WHILE**

Related Topics

[Selecting coverage types on page 310](#) | [Code Coverage settings on page 992](#)

Condition coverage

Code Coverage for C++

When analyzing C++ source code, HCL OneTest™ Embedded can provide the following condition coverage:

- Basic Coverage
- Forced Coverage

Basic Conditions

Conditions are operands of either `||` or `&&` operators wherever they appear in the body of a C++ function. They are also if and ternary expressions, tests for `for`, `while`, and `do/while` statements even if these expressions do not contain `||` or

&& operators. Two branches are involved in each condition: the sub-condition being true and the sub-condition being false.

Basic conditions also enable different case or default (which could be implicit) in a switch to be distinguished even when they invoke the same simple block. A basic condition is associated with every case and default (written or not).

There are 4*2 basic conditions in the following example:

```
/* Power_of_10 function */
/* -cond */
int power_of_10 ( int value, int max )
{
int result = value, i;
if ( value == 0 ) return 0;
for ( i = 0; i < 10; i++ )
{
result = max > 0 && ( max / value ) < result ?
result * value :
max;
}
return result ;
}
```

There are at least 5 basic conditions in this example:

```
/* Near_color function */
ColorType near_color ( ColorType color )
{
switch ( color )
{
case WHITE :
case LIGHT_GRAY :
```

```

return WHITE;

case RED :

case PINK :

case BURGUNDY :

return RED;

/* etc ... */

}

}

```

Two branches are enumerated for each condition, and one per case or default.

Forced Conditions

Forced conditions are multiple conditions in which any occurrence of the || and && operators has been replaced in the code with | and & binary operators. Such a replacement done by the Instrumentor enforces the evaluation of the right operands. You can use this coverage type after modified conditions have been reached to be sure that every basic condition has been evaluated. With this coverage type, you can be sure that only the considered basic condition changed between the two tests.

```
/* User source code */ /* -cond=forceevaluation */
```

```
if ( ( a && b ) || c ) ...
```

```
/* Replaced with the Code Coverage feature with : */
```

```
if ( ( a & b ) | c ) ...
```

```
/* Note : Operands evaluation results are enforced to one if different from 0 */
```

Note This replacement modifies the code semantics. You need to verify that using this coverage type does not modify the behavior of the software.

```
int f ( MyStruct *A )
```

```
{
```

```
if ( A && A->value > 0 ) /* the evaluation of A->value will cause a program error using
```

```
forced conditions if A pointer
```

```
is null */
```

```

{
A->value -= 1;
}
}

```

Modified Conditions

A modified condition is defined for each basic condition enclosed in a composition of || or && operators. It aims to prove that this condition affects the result of the enclosing composition. To do that, find a subset of values affected by the other conditions, for example, if the value of this condition changes, the result of the entire expression changes.

Because compound conditions list all possible cases, you must find the two cases that can result in changes to the entire expression. The modified condition is covered only if the two compound conditions are covered.

```

/* state_control function */
int state_control ( void )
{
if ( ( ( flag & 0x01 ) &&
( instances_number > 10 ) ) ||
( flag & 0x04 ) )
return VALID_STATE;
else
return INVALID_STATE;
}

```

In this example, there are 6 basic conditions (FALSE and TRUE of each), 5 compound conditions, and 3 modified conditions :

- flag & 0x01 : TTX=T and FXF=F
- nb_instances > 10 : TTX=T and TFF=F
- flag & 0x04 : TFT=T and TFF=F, or FXT=T and FXF=F

Therefore the 4 following test cases are enough to cover all those modified conditions :

- TTX=T
- FXF=F
- TFF=F
- TFT=T or FXT=T

Note You can associate a modified condition with more than one case, as shown in this example for flag & 0x04. In this example, the modified condition is covered if the two compound conditions of at least one of these cases are covered.

Code Coverage calculates matching cases for each modified condition.

The same number of modified conditions as Boolean basic conditions appears in a composition of || and && operators.

Multiple Conditions

A multiple (or compound) condition is one of all the available cases for the || and && logical operator's composition, whenever it appears in a C++ class. It is defined by the simultaneous values of the enclosed Boolean basic conditions.

A multiple condition is noted with a set of T, F, or X letters. These mean that the corresponding basic condition evaluated to true, false, or was not evaluated, respectively. Remember that the right operand of a || or && logical operator is not evaluated if the evaluation of the left operand determines the result of the entire expression.

```

/* state_control function */

/* -cond=compound */

int state_control ( void )

{

if ( ( ( flag & 0x01 ) &&

( instances_number > 10 ) ) ||

( flag & 0x04 ) )

return VALID_STATE;

else

return INVALID_STATE;

}

```

In this example, there are 3 basic conditions and 5 compound conditions :

- $TTX=T \Leftrightarrow ((T \ \&\& \ T) \ || \ X) = T$
- $TFT=T$
- $TFF=F$
- $FXT=T$
- $FXF=F$

Code Coverage calculates every available case for each composition.

The number of enumerated branches is the number of distinct available cases for each composition of `||` or `&&` operators.

Related Topics

[Selecting coverage types on page 310](#) | [Code Coverage settings on page 992](#)

Template instrumentation

Code Coverage for C++

Code Coverage performs the instrumentation of templates, functions, and methods of template classes, considering that all instances share their branches. The number of branches computed by the feature is independent of the number of instances for this template. All instances will cover the same once-defined branches in the template code.

Files containing template definitions implicitly included by the compiler (no specific compilation command is required for such source files) are also instrumented by the Code Coverage feature and present in the instrumented files where they are needed.

For some compilers, you must specifically take care of certain templates (for example, static or external linkage). You must verify if your Code Coverage Runtime installation contains a file named **templates.txt** and, if it does, read that file carefully.

- To instrument an application based upon Rogue Wave libraries , you must use the **-DRW_COMPILE_INSTANTIATE** compilation flag that suppresses the implicit include mechanism in the header files. (Corresponding source files are so included by pre-processing.)
- To instrument an application based upon ObjectSpace C++ Component Series , you must use the **-DOS_NO_AUTO_INSTANTIATE** compilation flag that suppresses the implicit include mechanism in the header files. (Corresponding source files are so included by pre-processing.)
- Any method (even unused ones) of an instantiated template class is analyzed and instrumented by the Instrumentor. Some compilers do not try to analyze such unused methods. It is possible that some of these

methods are not fully compliant with C++ standards. For example, a template class with a formal class template argument named T can contain a compare method that uses the == operator of the T class. If the C class used for T at instantiation time does not define an == operator, and if the compare method is never used, compilation succeeds but instrumentation fails. In such a situation, you can declare an == operator for the C class or use the **-instantiationmode=used** Instrumentor option.

Related Topics

[Selecting coverage types on page 310](#) | [Code Coverage settings on page 992](#)

Additional Statements

Code Coverage for C++

Non-coverable Statements

A C++ statement is *non-coverable* if the statement can never possibly be executed. Code Coverage detects non-coverable statements during instrumentation and produces a warning message that specifies the source file and line location of each non-coverable statement.

Related Topics

[Selecting coverage types on page 310](#) | [Code Coverage settings on page 992](#)

Using the Code Coverage Viewer to view reports

Code Coverage for Ada, C and C++

The Code Coverage Viewer allows you to view code coverage reports generated by the Code Coverage feature. Select a tab at the top of the Code Coverage Viewer window to select the type of report:

- A [Source Report on page 351](#), showing the source code under analysis, highlighted with the actual coverage information.
- A [Rates Report on page 354](#), providing detailed coverage rates for each activated coverage type.

You can use the Report Explorer to navigate through the report. Click a source code component in the Report Explorer to go to the corresponding line in the Report Viewer.

You can jump directly to the next or previous Failed test in the report by using the **Next Failed Test** or **Previous Failed Test** buttons from the Code Coverage toolbar.

You can jump directly to the next or previous Uncovered line in the Source report by using the **Next Uncovered Line** or **Previous Uncovered Line** buttons in the Code Coverage feature bar.

When viewing a Source coverage report, the Code Coverage Viewer provides several additional viewing features for refined code coverage analysis.

To open a Code Coverage report, follow these steps:

1. Right-click a previously executed test or application node
2. If a Code Coverage report was generated during execution of the node, select **View Report** and then **Code Coverage**.

Coverage types

Depending on the language selected, the Code Coverage feature offers (see [Coverage Types on page 310](#) for more information):

- **Function or Method code coverage:** select between function **Entries**, **Entries and exits**, or **None**.
- **Call code coverage:** select **Yes** or **No** to toggle call coverage for Ada and C.
- **Block code coverage:** select the desired block coverage method.
- **Condition code coverage:** select condition coverage for Ada and C.

Please refer to the related topics for details on using each coverage type with each language.

Any of the Code Coverage types selected for instrumentation can be filtered out in the Code Coverage report stage if necessary.

To filter coverage types from the report, proceed as follows:

1. From the **Code Coverage** menu, select **Code Coverage Type**.
2. Toggle each coverage type in the menu.

For example, to filter out multiple conditions (MC) from the report, select **Code Coverage > Code Coverage Type**, and clear **Multiple conditions**.

Alternatively, you can filter out coverage types from the Code Coverage toolbar by toggling the Code Coverage type filter buttons.

Test by test analysis mode

The *test by test* analysis mode allows you to refine the coverage analysis by individually selecting the various tests that were generated during executions of the test or application node. In Test-by-Test mode, a **Tests** node is available in the Report Explorer.

When *test by test* analysis is disabled, the Code Coverage Viewer displays all traces as one global test.

To toggle Test-by-Test mode, follow these steps:

1. In the **Code Coverage Viewer** window, select the **Source** tab.
2. From the **Code Coverage** menu, select **Test-by-Test**.

To select the Tests to display in Test-by-Test mode, follow these steps:

1. Expand the Tests node at the top of the **Report Explorer**.
2. Select one or several tests. The **Code Coverage Viewer** provides code coverage information for the selected tests.

Reloading a report

If a Code Coverage report has been updated since the moment you have opened it in the **Code Coverage Viewer**, you can use the **Reload** command to refresh the display:

To reload a report, select **Reload** from the **Code Coverage** menu, select **Reload**.

Resetting a report

When you run a test or application node several times, the Code Coverage results are appended to the existing report. The **Reset** command clears previous Code Coverage results and starts a new report.

To reset a report, select **Reset** from the **Code Coverage** menu.

Related information

[Code Coverage viewer preferences on page 1014](#)

[Coverage types on page 310](#)

[Exporting reports to HTML on page 705](#)

Coverage source report

Code Coverage applies to Ada, C and C++ languages.

You can use the standards keys (arrow keys, home, end, etc.) to move about and to select the source code. The Code Coverage source report displays covered and uncovered lines of code colors. You can change these colors in the Code Coverage report preferences.

Note: In C source files, the last bracket '}' in a function after a return statement is always displayed as uncovered in the coverage report, even if the function reports 100% coverage.

Code colors

The covered and uncovered lines are displayed with the following colors by default:

- Green for covered lines of code.
- Red for uncovered lines of code.

- Orange for partially covered lines of code.
- Blue for justified lines of code.
- Blue with the + icon for justified lines of code, which means that they should not be justified.
- Red with - icon for unreachable code.

```

int count(int x) {
    if (x >= 0) {
        return x - 1;
    }  else {
#pragma attol cov_justify (0,block,, "block not reachable")
        return 0;
    }
}

int main(void) {
    puts("!! basic boolean conditions:");
#pragma attol cov_justify (1,cond,"cond not reachable")
    if (true) {
#pragma attol cov_justify (1,cond,"cond not reachable")
        if (a >= 2) {
#pragma attol cov_justify (block,"block not reachable")
            int b = a + 1;
#pragma attol cov_justify (return,"return not reachable")
            return 1;
        }
#pragma attol cov_justify (2,cond,"a!=2:true","cond not reachable")
#pragma attol cov_justify (0,mcde,"TF;TT;FX","mcde not reachable")
        int b = a != 2 && (a < 2) == 1;
        return count(b);
        int  c = a;
    }
}

```

For uncovered line of codes that are justified, click on the blue attributes value to see more details about the justification text.

```

int count(int x) {
    if (x_≥_0) {
        return x -1;
    }  else {
#pragma attol cov_justify (0,block,, "block not reachable")
        return 0;
    }
}

int main(void) {
    puts("!!!Hello World!!!");
#pragma attol cov_justify (2,cond,:true, "cond not reachable")
#pragma attol cov_justify (implicit, "else not reachable")
    if (a_≥_0) {
#pragma attol cov_justify (block, "block not reachable")
        int b = a+1;
#pragma attol cov_just: Justification: return not reachable not reachable")
        return 1;
    }
#pragma attol cov_justify (2,cond, "a!=2:true", "cond not reachable")
#pragma attol cov_justify (0,mdc, "TF;TT;FX", "mdc not reachable")
    int b = a_!=_2 && (a_≠_2)_==_1;
    return count(b);
    int  c = a;
}

```

You can change the default colors in the code coverage report preferences. In the main menu toolbar, click **Edit > Preferences > Code Coverage Viewer > Styles**, you can modify the text color for the covered lines, covered lines with justify, justified lines, partially covered lines, and uncovered lines.

Hypertext Links

The Source report provides hypertext navigation throughout the source code:

- Click a plain underlined function call to jump to the definition of the function.
- Click a dashed underlined text to view additional coverage information in a pop-up window.
- Right-click any line of code and select **Edit Source** to open the source file in the **Text Editor** at the selected line of code.

Macro Expansion

Certain macro-calls are preceded with a magnifying glass icon.

Click the magnifying glass icon to expand the macro in a pop-up window with the usual Code Coverage color codes.

Hit Count

The Hit Count tool-tip is a special capability that displays the number of times that a selected branch was covered.

Hit Count is only available when Test-by-Test analysis is disabled and when the Hit Count option has been enabled for the selected [Configuration on page 658](#).

To activate the Hit Count tool-tip:

1. In the **Code Coverage Viewer** window, select the **Source** tab.
2. From the **Code Coverage** menu select **Hit**. The mouse cursor changes shape.
3. In the **Code Coverage Viewer** window, click a portion of covered source code to display the Hit Count tool-tip.

Cross Reference

The Cross Reference tool-tip displays the name of tests that executed a selected branch.

Cross Reference is only available in Test-by-Test mode.

To activate the Cross Reference tool-tip:

1. In the **Code Coverage Viewer** window, select the **Source** tab.
2. From the **Code Coverage** menu select **Cross Reference**. The mouse cursor changes shape.
3. In the **Code Coverage Viewer** window, click a portion of covered source code to display the **Cross Reference** tooltip.

Comment

You can add a short comment to the generated Code Coverage report by using the Comment option in the **Misc. Options Settings** for Code Coverage. This can be useful to distinguish different reports generated with different Configurations.

Comments are displayed as a magnifying glass symbol at the top of the source code report. Click the magnifying glass icon to display the comment.

Coverage rates report

Code Coverage for Ada, C and C++

From the Code Coverage Viewer window, select the **Rates** tab to view the coverage rate report.

To view the coverage rate and type for a particular source code component, select the component in the Report Explorer. Select the Root node to view coverage rates for all current files.

To change the displayed format between absolute values, percentages, or both, click on the **Display** line located just above the table.

To sort the table by one of the values, click the column title.

Code Coverage rates are updated dynamically as you navigate through the **Report Explorer** and as you select various coverage types.

Related Topics

[About the Code Coverage Viewer on page 349](#) | [Source Report on page 351](#) | [Selecting Coverage Types on page 310](#)

Bitwise MC/DC coverage

Put your short description here; used for first paragraph and abstract.

Type **your** text here.

- an interesting point
- another interesting point

Subheading

Here's a little section in a concept.

Exemple

Example

Here's a little example section in a concept.

On-the-fly code coverage

Code Coverage for C and C++

By default code coverage generates a report when the execution ends. The On-the-fly mode generates code coverage results dynamically during the execution. This is useful for applications that never exit or to interact with the execution during the analysis, for example if you want to stop the code coverage when you reach at a given coverage rate threshold.

To enable the On-the-fly mode in Code Coverage:

1. In the **Project Explorer**, click the **Settings**  button.
2. Select a node in the **Project Explorer** pane.
3. In the Configuration Settings list, expand **Runtime Analysis** and select **Coverage > Advanced Options > On-the-fly frequency dump**.
4. Specify the number of function calls after which the coverage results are updated during execution. 0 means that there is no on-the-fly updating and that results are only generated at the end of the execution.
5. When you have finished, click **OK** to validate the changes.

Related Topics

[Code Coverage settings on page 992](#)

Code Coverage Dump Driver

Code Coverage for C and C++

In C and C++, you can dump coverage trace data without using standard I/O functions by using the Code Coverage Dump Driver API contained in the **atcapi.h** file, which is part of the Target Deployment Port

To customize the Code Coverage Dump Driver, open the Target Deployment Port directory and edit the **atcapi.h**. Follow the instructions and comments included in the source code.

Related Topics

[Generating SCI Dumps on page 1051](#)

Cleaning code coverage report files

Code Coverage for C and C++

Code Coverage produces reports on each execution of the application under test. After many executions, the .tio coverage report files can become quite large and take up a lot of disk space.

You can use the **-CLEAN** option with the **attolcov** command to remove unused and obsolete traces and to regain some space without losing your execution history.

You can use the **-MERGETESTS** command line option to merge all the specified .tio coverage report files together.

To clean the .tio coverage report files:

1. Run the following command line:

```
attolcov <oldfiles.tio> -clean=<newfile.tio> -mergetests
```

where *<oldfiles.tio>* is a list of old .tio coverage report files and *<newfile.tio>* is the new .tio coverage report file.

Related Topics

[About code coverage on page 54](#) | [File types on page 1032](#)

Justification of non-covered lines of code

You can enter justification statements in uncovered branches of a program so that they are considered as exceptions to the coverage rules. Thus, you identify in the source code the branches that are not covered and explain why they

are not covered. The justification text must be declared in the **attol cov_justify** pragma line of the uncovered branch with one or multiple attributes.



Note: This feature applies to C and C++ programming languages only.

SYNTAX:

The justification pragma syntax is the following one:

```
#pragama attol cov_justify (<lineOffset>, <type>, <what>, <justification text> ) [ (...) [ (...) ...]]
```

#pragama attol cov_justify is the pragma, and **<lineOffset>**, **<type>**, **<what>**, and **<justification text>** are the attributes.

Multiple statements can be specified in the same pragma line, with four attributes for each.

Each justification statement in a pragma line can cover only one branch of the code starting from a specified line of the source code.

Double-quotes can be added if the attribute includes commas.

The attributes are the following ones:

- **<type>** attribute:

This attribute is mandatory. It is used to determine what kind of code must be covered and how it must be covered.

You can use the following **<type>** values:

- **proc**: to justify that a function or a method is not covered.
- **return**: to justify that a return statement is not covered.
- **branch** or **block**: to justify that a block of code is not covered.
- **implicit**: to justify that an implicit else statement is not covered.
- **logical**, or **for**, or **while**: to justify that a loop is not covered (the loop number is given in the **<what>** attribute).
- **case**: to justify that a case statement in a switch is not covered.
- **call**: to justify that a function call or method call is not covered.
- **cond**: to justify that a simple condition is not covered (the value true or false of the condition is given in the **<what>** attribute).
- **mcdc**: to justify that a MC/DC is not covered (the description of the MD/DC is given in the **<what>** attribute).

- **<lineOffset>** attribute:

This attribute is optional.

<lineOffset> attribute represents the number of lines between the pragma and the branch that must be justified. The value can be '+' if the branch is located after the pragma, or '-' if the branch is located before the pragma.

If the **<lineOffset>** attribute is omitted **<lineOffset>** is considered as "0". It means that the justification applies to the closest type of branch (attribute **<type>**) from the pragma. If there are multiple branches at a same distance of the pragma, **<lineOffset>** helps distinguish between branches which branch should be justified.

The **<lineOffset>** attribute specifies the line where the branch or condition to justify starts, it is relative to the pragma line (+/-), allowing to write this pragma line anywhere in the source file.



Note: For the **<block>** value, the target block of lines is the block where the pragma is declared.

• **<what>** attribute:

The **<what>** values are used to help specify some of the branches to be justified. It depends on the attribute **<type>** values being used:

- For **block** or **branch** **<type>**: The value is a string that describes the logical position of the block in the function, like '/then/else/seq'.
- For **logical**, **for** and **while** **<type>**: The value is a list of '0' (the loop is not executed), '1' (the loop is executed only once) or '2+' (the loop is executed more that once) separated by ','; each of them could be prefixed with the block description string.
- For **cond** **<type>**: The value is **<expression>**:**<value>**, **<value>** is *true* or *false* and **<expression>**.
- For **mcadc** **<type>**: The value is a list of impossible combinations of the conditions separated by ','; each value of the conditions are set with 'T' for true, 'F' for false, or 'X' if the condition is not evaluated. For example, if the MC/DC consists of 3 conditions, the **<what>** value could be the following one: "TFX;FXX".

• **<justification text>** is the reason why this part of code can't be covered by a test.

<justification> is mandatory. It is presented as a free text in the coverage report that justifies a uncovered branch. It explains why it is not covered.

Example:

```
#pragma attol cov_justify (call, "my justification") (block, "myjustification") (cond, ":true",
"my justification") (for, "0;1" ,"my justification")
```

The following table lists the parameters that can be entered in the "**<lineOffset>**" and **<what>** attributes depending on the parameters indicated in the **<type>** attribute.

<type> attribute	<lineOffset> attribute	<what> attribute
proc	For the <type>=proc , the pragma line declared above or inside the body, or just after the end of the body justifies the function/procedure entry.	

<type> attribute	<lineOffset> attribute	<what> attribute
return	For <type>=return , the pragma line, must be just before or just after the return line.	
branch/block	For <type>=branch or block . It starts on the first { of that block or on the line of the unique statement.	<p>The "branch to cover" attribute is used for a branch=<type> The branch string format is a list of :</p> <p><i>/then/else/seq OR /</i></p> <p>It is used to indicate which branch to cover when there are multiple branches on the same line.</p> <p>It can be empty is there are no ambiguity with the line number.</p>
implicit	For <type>=implicit , the pragma line must be just before the decision, or at the else place.	<p>The "branch to cover" attribute is used for <type>= <branch>.</p> <p>The branch string format is a list of: / <i>then/else/seq OR /</i></p> <p>It is used to indicate which branch to cover when there are multiple branches on the same line.</p> <p>It can be empty is there are no ambiguity with the line number.</p>
logical/for/do/while	For <type>=logical/for/while , the pragma line must be just before the 'for' or 'while', or 'do' keyword.	<p>This attribute is used for <type>=logical, the branch string format looks like 'branch depth/instruction type/value' with :</p> <p>instruction type is:</p> <p><i>/for OR /while OR /do</i></p> <p>The value is <i>/0</i> or <i>/1</i> or <i>/2+</i> to specify which part must be covered and justified. It is mandatory and can speci-</p>

<type> attribute	<lineOffset> attribute	<what> attribute
		<p>fy multiple parts if separated by ; e.g. <code>"/0;/1"</code></p> <p>Branch depth is a suite of strings like <code>/then /else /for /while</code> etc. clarifying the code depth of the branch where the loop has been found.</p> <p>e.g. <code>"/else/then/for/while/1"</code> or <code>"/while/1"</code> or <code>"/1"</code></p>
case	<p>For <type>=case, the pragma line must be just before or just after the case line.</p> <p>A case is both a block and a condition. If you enter a justification for a case <type>, it is the condition that is justified. You need to declare another pragma to justify the corresponding block.</p>	<p>Used for <type>=case, the string details the case expression (between 'case' and the ':') to cover into the switch block.</p> <p>It can be empty is there are no ambiguity with the line number.</p>
call	<p>For <type>=call, the pragma line must be just before or just after the call line,</p>	<p>Used for type=call, the string details the called method name to cover.</p> <p>It can be empty is there are no ambiguity with the line number.</p>
cond	<p>For <type>=cond, the pragma line must be just before or just after the condition line.</p>	<p>"condition expression:value" is used for "<type>=cond".The string indicates the condition expression to cover into the decision with the value to cover, true or false. The "condition expression" can be empty if there is no ambiguity with the line number but the value must always be specified after a colon at the end of the string.</p> <p>Example: <code>"var>5:false"</code> or <code>":true"</code>.</p>

<type> attribute	<lineOffset> attribute	<what> attribute
		Multiple values can be justified, separated by ';' such as <code>":true ; :false"</code>
mcdc	For <type>=mcdc , the pragma line must be just before or after the first condition line, or just before or after the last condition line.	"combinations" is used for type=mcdc . It is a series of patterns separated by a semi-colon ";". It cannot be empty.

Once the source code is built, you can see the results of the non-coverage justification statements in the Code Coverage report, on the Source page.

For more information about the code coverage reports, see [About coverage reports on page 928](#).

Code coverage for assembler source files

From HCL OneTest™ Embedded V8.3.0, you can collect coverage metrics for assembler source files. Coverage information is displayed in the coverage report.



Note:

This feature supports only ARM in 32 bits mode. It is an extension of the C language mode.

You must have configured HCL OneTest™ Embedded to recognize the .asm file extension used for assembler files. For more details, see [Using assembler source files on page 684](#).

Coverage Assembler language for ARM processor is fully compatible with C/C++ and Ada code coverage.

Assembler source files are taken into account in the build as C/C++ source files. Optionally, C/C++ source files could be instrumented at the assembler level and not at the C/C++ level. To launch code coverage for assembler files from a command, see [Command line to launch code coverage for assembler files on page 1072](#).

The supported coverage levels are:

- Functions
- Functions and exits



- Statement blocks
- Calls

CONFIGURATION

Code coverage for assembler source files requires the use of an appropriate TDP. You can use the `clinCrossRaspiRemote.xdp` and `cwinCrossRaspiRemote.xdp` that are delivered, for example.

There two use case scenarios:

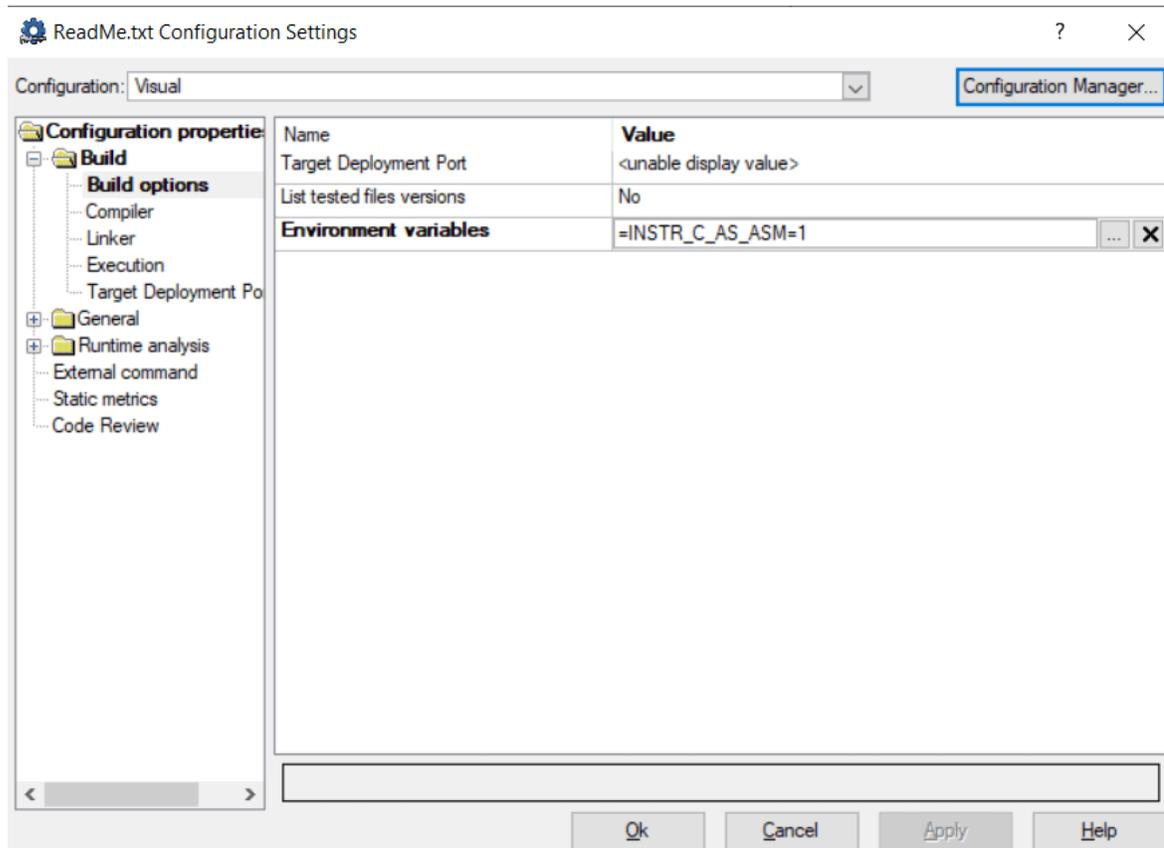
- For a project that uses both C and assembler source files, you only have to add the assembler `.asm` source files to the list of sources to be compiled. The `.asm` files are then instrumented, built, and linked with the other C sources to produce an executable file.
- For C code source files that are instrumented in assembler mode, the C source files are converted into assembler files by using the `gcc -S` command. Then, they are instrumented in assembler mode, they are converted into assembly language, and linked.

To implement this use case scenario, you must set the `INSTR_C_AS_ASM=1` environment variable.

To add this environment variable in Studio, proceed as follows:

- Click **Settings** in the **Project window**.
- Select **Build options > Environment**.
- Click the **Value** field on the **Environment variable** line, and click
- Click the Add icon, give a name to the variable, and enter `INSTR_C_AS_ASM=1` in **Value**.
- Apply and close the window.

See the following example:



Note: In some case, when the assembler code increased due to the code coverage level, it might be necessary to re-organize the assembler code (Example: you can move data pools), or to decrease the level of code coverage (Example: you can delete code coverage for some functions calls in libraries).

Memory profiling for C and C++

About Memory Profiling for C and C++

Memory Profiling for C and C++

Run-time memory errors and leaks are among the most difficult errors to locate and the most important to correct. The symptoms of incorrect memory use are unpredictable and typically appear far from the cause of the error. The errors often remain undetected until triggered by a random event, so that a program can seem to work correctly when in fact it's only working by accident.

That's where the Memory Profiling feature can help you.

- You associate Memory Profiling with an existing test node or application code.
- You compile and run your application.
- The application with the Memory Profiling feature, then directs output to the Memory Profiling Viewer, which provides a detailed report of memory issues.

Memory Profiling uses Source Code Insertion Technology for C and C++.

Because of the different technologies involved, [Memory Profiling for Java on page 379](#) is covered in a separate section.

Memory Profiling for C and C++ supports the following languages:

- **C**: ANSI 89, ANSI 99, or K&R C
- **C++**: ISO/IEC 14882:1998

How Memory Profiling for C and C++ Works

When an application node is executed, the source code is instrumented by the C or C++ Instrumentor (**attolc** or **attolcc1**). The resulting source code is then executed and the Memory Profiling feature outputs a static **.tsf** file for each instrumented source file and a dynamic **.tpf** file.

These files can be viewed and controlled from the HCL OneTest™ Embedded GUI. Both the **.tsf** and **.tpf** files need to be opened simultaneously to view the report.

Of course, these steps are mostly transparent to the user when the test or application node is executed in the HCL OneTest™ Embedded GUI or Eclipse (for C and C++).

To learn about	See
Performing Memory Profiling on C and C++ source code	Using Runtime Analysis Features on page 306
How Memory Profiling for C and C++ works	Memory Profiling User in C and C++ on page 372
Source Code Insertion technology	Source code instrumentation overview on page 11
JVMPI technology for Java memory analysis	JVMPI Technology on page 383
Understanding Memory Profiling Reports	Memory Profiling Results on page 365
Using the Memory Profiler Viewer	Using the Memory Profiling Viewer on page 377

Customizing the Memory Profiling Viewer

[Memory Profiling Viewer Preferences](#)
on page 1019

Related Topics

[Memory Profiling Settings on page 995](#) | [Runtime Analysis on page 305](#) | [Memory Profiling for Java on page 379](#)

Memory Profiling Results for C and C++

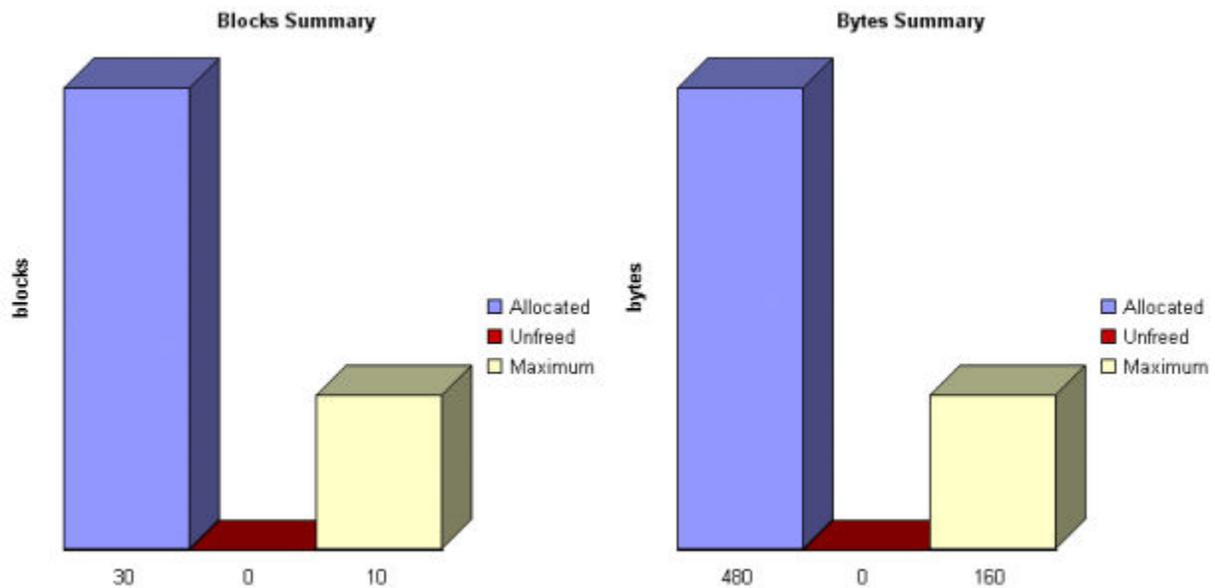
Memory Profiling for C and C++

After execution of an instrumented application, the Memory Profiling report provides a summary diagram and a detailed report for both byte and memory block usage.

A memory block is a number of bytes allocated with a single *malloc* instruction. The number of bytes contained in each block is the actual amount of memory allocated by the corresponding allocation instruction.

Summary diagrams

The summary diagrams give you a quick overview of memory usage in blocks and bytes.



Where:

- **Allocated** is the total memory allocated during the execution of the application
- **Unfreed** is the memory that remains allocated after the application was terminated
- **Maximum** is the highest memory usage encountered during execution

Detailed Report

The detailed section of the report lists memory usage events, including the following errors and warnings:

- [Error messages on page 366](#)
- [Warning messages on page 369](#)

Related Topics

[Using the Memory Profiling Viewer on page 377](#) | [Memory Profiling Settings on page 995](#)

Memory Profiling Error Messages

Memory Profiling Error Messages

Memory Profiling for C and C++

Error messages indicate invalid program behavior. These are serious issues you should address before you check in code.

List of Memory Profiling Error Messages

- [Free on page 366](#) ing Freed Memory (FFM) on page 367
- [Freeing Unallocated Memory \(FUM\) on page 367](#)
- [Freeing Invalid Memory \(FIM\) on page 367](#)
- [Late Detect Array Bounds Write \(ABWL\) on page 367](#)
- [Late Detect Free Memory Write \(FMWL\) on page 368](#)
- [Memory Allocation Failure \(MAF\) on page 369](#)
- [Core Dump \(COR\) on page 369](#)

Related Topics

[Memory Profiling Results on page 365](#) | [Memory Profiling Settings on page 995](#) | [Warning Messages on page 369](#)

Freeing Freed Memory (FFM)

Memory Profiling for C and C++

An FFM message indicates that the program is trying to free memory that has previously been freed.

This message can occur when one function frees the memory, but a data structure retains a pointer to that memory and later a different function tries to free the same memory. This message can also occur if the heap is corrupted.

Memory Profiling maintains a *free queue*, whose role is to actually delay memory free calls in order to compare with upcoming free calls. The length of the delay depends on the **Free queue length** and **Free queue threshold** Memory Profiling Settings. A large deferred free queue length and threshold increases the chances of catching FFM errors long after the block has been freed. A smaller deferred free queue length and threshold limits the amount of memory on the deferred free queue, taking up less memory at run time but providing a lower level of error detection.

Related Topics

[Error Messages on page 366](#) | [Memory Profiling Results on page 365](#) | [Memory Profiling Settings on page 995](#)

Freeing Unallocated Memory (FUM)

Memory Profiling for C and C++

An FUM message indicates that the program is trying to free unallocated memory.

This message can occur when the memory is not yours to free. In addition, trying to free the following types of memory causes a FUM error:

- Memory on the stack
- Program code and data sections

Related Topics

[Error Messages on page 366](#) | [Memory Profiling Results on page 365](#) | [Memory Profiling Settings on page 995](#)

Freeing Invalid Memory (FIM)

Memory Profiling for C and C++

An FIM message indicates that the program is trying to free allocated memory with the wrong instruction.

This message can occur when the memory free instruction mismatches the memory allocation instruction.

For example, a FIM occurs when memory is freed with a **free** instruction when it was allocated with a **new** instruction.

Related Topics

[Error Messages on page 366](#) | [Memory Profiling Results on page 365](#) | [Memory Profiling Settings on page 995](#)

Late Detect Array Bounds Write (ABWL)

Memory Profiling for C and C++

An ABWL message indicates that the program wrote a value before the beginning or after the end of an allocated block of memory.

Memory Profiling checks for ABWL errors whenever `free()` or `dump()` routines are called, or whenever the free queue is actually flushed.

This message can occur when you:

- Make an array too small. For example, you fail to account for the terminating NULL in a string.
- Forget to multiply by `sizeof(type)` when you allocate an array of objects.
- Use an array index that is too large or is negative.
- Fail to NULL terminate a string.
- Are off by one when you copy elements up or down an array.

Memory Profiling actually allocates a larger block by adding a Red Zone at the beginning and end of each allocated block of memory in the program. Memory Profiling monitors these Red Zones to detect ABWL errors.

Increasing the size of the Red Zone helps HCL OneTest™ Embedded catch bounds errors before or beyond the block at the expense of increased memory usage. You can change the Red Zone size in the Memory Profiling Settings.

The ABWL error does not apply to local arrays allocated on the stack.

Note Unlike PurifyPlus, the ABWL error in the HCL OneTest™ Embedded Memory Profiling tool only applies to heap memory zones and not to global or local tables.

Related Topics

[Error Messages on page 366](#) | [Memory Profiling Results on page 365](#) | [Memory Profiling Settings on page 995](#)

Late Detect Free Memory Write (FMWL)

Memory Profiling for C and C++

An FMWL message indicates that the program wrote to memory that was freed.

This message can occur when you:

- Have a dangling pointer to a block of memory that has already been freed (caused by retaining the pointer too long or freeing the memory too soon)
- Index far off the end of a valid block
- Use a completely random pointer which happens to fall within a freed block of memory

Memory Profiling maintains a *free queue*, whose role is to actually delay memory free calls in order to compare with upcoming free calls. The length of the delay depends on the **Free queue length** and **Free queue threshold** Memory Profiling Settings. A large deferred free queue length and threshold increases the chances of catching FMWL errors. A smaller deferred free queue length and threshold limits the amount of memory on the deferred free queue, taking up less memory at run time but providing a lower level of error detection.

Related Topics

[Error Messages on page 366](#) | [Memory Profiling Results on page 365](#) | [Memory Profiling Settings on page 995](#)

Memory Allocation Failure (MAF)

Memory Profiling for C and C++

An MAF message indicates that a memory allocation call failed. This message typically indicates that the program ran out of paging file space for a heap to grow. This message can also occur when a non-spreadable heap is saturated.

After Memory Profiling displays the MAF message, a memory allocation call returns *NULL* in the normal manner. Ideally, programs should handle allocation failures.

Related Topics

[Error Messages on page 366](#) | [Memory Profiling Results on page 365](#) | [Memory Profiling Settings on page 995](#)

Core Dump (COR)

Memory Profiling for C and C++

A COR message indicates that the program generated a UNIX core dump. This message can only occur when the program is running on a UNIX target platform.

Related Topics

[Error Messages on page 366](#) | [Memory Profiling Results on page 365](#) | [Memory Profiling Settings on page 995](#)

Memory Profiling Warning Messages

Memory Profiling Warning Messages

Memory Profiling for C and C++

Warning messages indicate a situation in which the program might not fail immediately, but might later fail sporadically, often without any apparent reason and with unexpected results. Warning messages often pinpoint serious issues you should investigate before you check in code.

List of Memory Profiling Warning Messages

- [Memory in Use \(MIU\) on page 370](#)
- [Memory Leak \(MLK\) on page 370](#)
- [Potential Memory Leak \(MPK\) on page 371](#)
- [File in Use \(FIU\) on page 371](#)
- [Signal Handled \(SIG\) on page 372](#)

Related Topics

[Memory Profiling Results on page 365](#) | [Memory Profiling Settings on page 995](#) | [Error Messages on page 366](#)

Memory in Use (MIU)

Memory Profiling for C and C++

An MIU message indicates heap allocations to which the program has a pointer.

Note At exit, small amounts of memory in use in programs that run for a short time are not significant. However, you should fix large amounts of memory in use in long running programs to avoid out-of-memory problems.

Memory Profiling generates a list of memory blocks in use when you activate the **MIU Memory In Use** option in the Memory Profiling Settings.

Related Topics

[Memory Profiling Results on page 365](#) | [Warning Messages on page 369](#) | [Memory Profiling Settings on page 995](#)

Memory Leak (MLK)

Memory Profiling for C and C++

An **MLK** message describes leaked heap memory. There are no pointers to this block, or to anywhere within this block.

Memory Profiling generates a list of leaked memory blocks when you activate the **MLK Memory Leak** option in the Memory Profiling Settings.

This message can occur when you allocate memory locally in some function and exit the function without first freeing the memory. This message can also occur when the last pointer referencing a block of memory is cleared, changed, or goes out of scope. If the section of the program where the memory is allocated and leaked is executed repeatedly,

you might eventually run out of swap space, causing slow downs and crashes. This is a serious problem for long-running, interactive programs.

To track memory leaks, examine the allocation location call stack where the memory was allocated and determine where it should have been freed.

You can ignore memory leaks that do not have a call stack, for memory allocations that occur before the application starts by changing the configuration **Runtime Analysis > Memory Profiling > Instrumentation control > Only show memory leaks with call stack.**

Related Topics

[Memory Profiling Results on page 365](#) | [Warning Messages on page 369](#) | [Memory Profiling Settings on page 995](#)

Memory Potential Leak (MPK)

Memory Profiling for C and C++

An **MPK** message describes heap memory that might have been leaked. There are no pointers to the start of the block, but there appear to be pointers pointing somewhere within the block. In order to free this memory, the program must subtract an offset from the pointer to the interior of the block. In general, you should consider a potential leak to be an actual leak until you can prove that it is not by identifying the code that performs this subtraction.

Memory in use can appear as an **MPK** if the pointer returned by some allocation function is offset. This message can also occur when you reference a substring within a large string. Another example occurs when a pointer to a C++ object is cast to the second or later base class of a multiple-inherited object and it is offset past the other base class objects.

Alternatively, leaked memory might appear as an **MPK** if some non-pointer integer within the program space, when interpreted as a pointer, points within an otherwise leaked block of memory. However, this condition is rare.

Inspection of the code should easily differentiate between different causes of **MPK** messages.

Memory Profiling generates a list of potentially leaked memory blocks when you activate the **MPK Memory Potential Leak** option in the Memory Profiling Settings.

Related Topics

[Memory Profiling Results on page 365](#) | [Warning Messages on page 369](#) | [Memory Profiling Settings on page 995](#)

File in Use (FIU)

Memory Profiling for C and C++

An FIU message indicates a file that was opened, but never closed. An FIU message can indicate that the program has a resource leak.

Memory Profiling generates a list of files in use when you activate the **FIU Files In Use** option in the Memory Profiling Settings.

Related Topics

[Memory Profiling Results on page 365](#) | [Warning Messages on page 369](#) | [Memory Profiling Settings on page 995](#)

Signal Handled (SIG)

Memory Profiling for C and C++

A **SIG** message indicates that a system signal has been received.

Memory Profiling generates a list of received signals when you activate the **SIG Signal Handled** option in the Memory Profiling Settings.

Related Topics

[Memory Profiling Results on page 365](#) | [Warning Messages on page 369](#) | [Memory Profiling Settings on page 995](#)

Memory Profiling User Heap in C and C++

Memory Profiling for C and C++

When using Memory Profiling on embedded or real-time target platforms, you might encounter one of the following situations:

- **Situation 1:** There are no provisions for **malloc**, **calloc**, **realloc** or **free** statements on the target platform.

Your application uses custom heap management routines that may use a user API. Such routines could, for example, be based on a static buffer that performs allocation and free actions.

In this case, you need to customize the memory heap parameters **RTRT_DO_MALLOC** and **RTRT_DO_FREE** in the TDP to use the custom **malloc** and **free** functions.

In this case, you can access the custom API functions.

- **Situation 2:** There are partial implementations of **malloc**, **calloc**, **realloc** or **free** on the target, but other functions provide methods of allocating or freeing heap memory.

In this case, you do not have access to any custom API. This requires customization of the Target Deployment Port. Please refer to the **Target Deployment Guide** provided with the [Opening the Target Deployment Port Editor on page 24](#).

In both of the above situations, Memory Profiling can use the heap management routines to detect memory leaks, array bounds and other memory-related defects.

Note Application pointers and block sizes can be modified by Memory Profiling in order to detect ABWL errors (Late Detect Array Bounds Write). Actual-pointer and actual-size refer to the memory data handled by Memory Profiling, whereas user pointer and user-size refer to the memory handled natively by the application-under-analysis. This distinction is important for the Memory Profiling ABWL and Red zone settings.

Target Deployment Port API

The Target Deployment Port library provides the following API for Memory Profiling:

```
void * _PurifyLTHeapAction ( _PurifyLT_API_ACTION, void *, RTRT_U_INT32, RTRT_U_INT8 );
```

In the function **_PurifyLTHeapAction** the first parameter is the type of action that will be or has been performed on the memory block pointed by the second parameter. The following actions can be used:

```
typedef enum {
    _PurifyLT_API_ALLOC,
    _PurifyLT_API_BEFORE_REALLOC,
    _PurifyLT_API_FREE
} _PurifyLT_API_ACTION;
```

The third parameter is the size of the block. The fourth parameter is either of the following constants:

```
#define _PurifyLT_NO_DELAYED_FREE 0

#define _PurifyLT_DELAYED_FREE 1
```

If an allocation or free has a size of 0 this fourth parameter indicates a delayed free in order to detect FWML (Late Detect Free Memory Write) and FFM (Freeing Freed Memory) errors. See the section on Memory Profiling Configuration Settings for Detect FFM, Detect FMWL, Free Queue Length and Free Queue Size.

A freed delay can only be performed if the block can be freed with **RTRT_DO_FREE** (situation 1) or ANSI **free** (situation 2). For example, if a function requires more parameters than the pointer to de-allocate, then the FMWL and FFM error detection cannot be supported and FFM errors will be indicated by an FUM (Freeing Unallocated Memory) error instead.

The following function returns the size of an allocated block, or 0 if the block was not declared to Memory Profiling. This allows you to implement a library function similar to the `msize` from Microsoft Visual 6.0.

```
RTRT_SIZE_T _PurifyLTHeapPtrSize ( void * );
```

The following function returns the actual-size of a memory block, depending on the size requested. Call this function before the actual allocation to find out the quantity of memory that is available for the block and the contiguous red zones that are to be monitored by Memory Profiling.

```
RTRT_SIZE_T _PurifyLTHeapActualSize ( RTRT_SIZE_T );
```

Examples

In the following examples, **my_malloc**, **my_realloc**, **my_free** and **my_msize** demonstrate the four supported memory heap behaviors.

The following routine declares an allocation:

```
void *my_malloc ( int partId, size_t size )
{
    void *ret;

    size_t actual_size = _PurifyLTHeapActualSize(size);

    /* Here is any user code making ret a pointer to a heap or
    simulated heap memory block of actual_size bytes */

    ...

    /* After comes Memory Profiling action */

    return _PurifyLTHeapAction ( _PurifyLT_API_ALLOC, ret, size, 0 );

    /* The user-pointer is returned */
}
```

In situation 2, where you have access to a custom memory heap API, replace the "..." with the actual *malloc* API function.

For a **my_calloc**(*size_t nelem*, *size_t elsize*) , pass on *nelem*elsize* as the third parameter of the **_PurifyLTHeapAction** function. In this case, you might need to replace this operation with a function that takes into account the alignments of elements.

To declare a reallocation, two operations are required:

```
void *my_realloc ( int partId, void * ptr, size_t size )
```

```

{
void *ret;

size_t actual_size = _PurifyLHeapActualSize(size);

/* Before comes first Memory Profiling action */

ret = _PurifyLHeapAction ( _PurifyLT_API_BEFORE_REALLOC, ptr, size, 0 );

/* ret now contains the actual-pointer */

/* Here is any user code making ret a reallocated pointer to a heap or
simulated heap memory block of actual_size bytes */

...

/* After comes second Memory Profiling action */

return _PurifyLHeapAction ( _PurifyLT_API_ALLOC, ret, size, 0 );

/* The user-pointer is returned */
}

```

To free memory without using the delay:

```

void my_free ( int partId, void * ptr )
{
/* Memory Profiling action comes first */

void *ret = _PurifyLHeapAction ( _PurifyLT_API_FREE, ptr, 0, 0 );

/* Any code insuring actual deallocation of ret */

}

```

To free memory using a delay:

```

void my_free ( int partId, void * ptr )
{
/* Memory Profiling action comes first */

void *ret = _PurifyLHeapAction ( _PurifyLT_API_FREE, ptr, 0, 1 );

/* Nothing to do here */
}

```

```
}

```

To obtain the user size of a block:

```
size_t my_msize ( int partId, void * ptr )
{
return _PurifyLTHeapPtrSize ( ptr );
}
```

Use the following macros to save customization time when dealing with functions that have the same prototypes as the standard ANSI functions:

```
#define _PurifyLT_MALLOC_LIKE(func) \
void *RTRT_CONCAT_MACRO(usr_func) ( RTRT_SIZE_T size ) \
{ \
void *ret; \
ret = func ( _PurifyLTHeapActualSize ( size ) ); \
return _PurifyLTHeapAction ( _PurifyLT_API_ALLOC, ret, size, 0 ); \
}

#define _PurifyLT_CALLOC_LIKE(func) \
void *RTRT_CONCAT_MACRO(usr_func) ( RTRT_SIZE_T nelem, RTRT_SIZE_T elsize ) \
{ \
void *ret; \
ret = func ( _PurifyLTHeapActualSize ( nelem * elsize ) ); \
return _PurifyLTHeapAction ( _PurifyLT_API_ALLOC, ret, nelem * elsize, 0 ); \
}

#define _PurifyLT_REALLOC_LIKE(func,delayed_free) \
void *RTRT_CONCAT_MACRO(usr_func) ( void *ptr, RTRT_SIZE_T size ) \
{ \
void *ret; \
```

```

ret = func ( _PurifyLHeapAction ( _PurifyLT_API_BEFORE_REALLOC, \
ptr, size, delayed_free ), \
_PurifyLHeapActualSize ( size ) ); \
return _PurifyLHeapAction ( _PurifyLT_API_ALLOC, ret, size, 0 ); \
}

#define _PurifyLT_FREE_LIKE(func,delayed_free) \

void RTRT_CONCAT_MACRO(usr_,func) ( void *ptr ) \

{ \

if ( delayed_free ) \

{ \

_PurifyLHeapAction ( _PurifyLT_API_FREE, ptr, 0, delayed_free ); \

} \

else \

{ \

func ( _PurifyLHeapAction ( _PurifyLT_API_FREE, ptr, 0, delayed_free ) ); \

} \

}

```

Related Topics

[About Memory Profiling on page 363](#) | [Error Messages on page 366](#) | [Opening the Target Deployment Port Editor on page 24](#)

Using the Memory Profiling Viewer

Memory Profiling for C and C++

Memory Profiling results for C and C++ are displayed in the Memory Profiling Viewer.

Error and Warning Filter

The Memory Profiling Viewer for C and C++ allows you to filter out any particular type of [Error on page 366](#) or [Warning on page 369](#) message from the report.

To filter out error or warning messages:

1. Select an active **Memory Profiling Viewer** window.
2. From the **Memory Profiling** menu, select **Errors and Warnings**.
3. Select or clear the type of message that you want to show or hide.

For example, you can disable **MLK (Memory Leak) with empty stack trace** to hide from the report memory allocations that occurred before the application started.

Reloading a Report

If a Memory Profiling report has been updated since the moment you have opened it in the Memory Profiling Viewer, you can use the Reload command to refresh the display:

To reload a report:

1. From the View Toolbar, click the **Reload** button.

Resetting a Report

When you run a test or application node several times, the Memory Profiling results are appended to the existing report. The **Reset** command clears previous Memory Profiling results and starts a new report.

To reset a report:

1. From the View Toolbar, click the **Reset** button.

Related Topics

[Memory Profiling results on page 365](#) | [Opening a report on page 683](#) | [Report Explorer on page 1024](#) | [Using the report viewer on page 705](#) | [Exporting reports on page 705](#)

Checking for ABWL and FMWL errors

By default, Memory Profiling checks for ABWL and FMWL errors whenever the routines are called, or whenever the free queue is actually flushed.

In some cases, it might be desirable to manually specify when to check for ABWL and FMWL errors, and on which functions.

By using the **ABWL and FMWL check frequency** setting you can order a check on:

- Each time the memory is dumped (by default).
- Each time a manual check macro is encountered in the code.
- Each function return.

The checks can be performed either on all memory blocks or only a selection of memory blocks.

Specifying a manual check

To indicate where you want an ABWL or FMWL check to occur in your source code, you insert an `_ATP_CHECK()` macro at the corresponding location. The syntax for the macro is:

```
#pragma attol insert _ATP_CHECK(@RELFINE)
```

Each time this macro is encountered during execution, Memory Profiling checks for ABWL and FMWL errors on the specified blocks. The `@RELFINE` parameter allows navigation from the Memory Profiling report to the corresponding line in the source code.

Selecting blocks to check

To create a selection of blocks that you specifically want to verify, you create a list in your source code using the `_ATP_TRACK()` macro variable. The syntax for this macro is:

```
#pragma attol insert _ATP_TRACK(<pointer>)
```

Example

A sample demonstrating how to use this feature is provided in the **ABWL Check Frequency** example project. See [Example projects on page 677](#) for more information.

Related Topics

[Memory Profiling Settings on page 995](#) | [Late Detect Free Memory Write \(FMWL\) on page 368](#) | [Late Detect Array Bounds Write \(ABWL\) on page 367](#)

Memory Profiling for Java

Run-time memory problems are among the most difficult errors to locate and the most important to correct. The symptoms of incorrect memory use are unpredictable and typically appear far from the cause of the error. The issue often remain undetected until triggered by a random event, so that a program can seem to work correctly when in fact it's only working by accident.

That's where the Memory Profiling feature can help you get ahead.

- You associate Memory Profiling with an existing test node or Application code.
- You compile and run your application.
- The application with the Memory Profiling feature, then directs output to the Memory Profiling Viewer, which provides a detailed report of memory issues.

The Java version of Memory Profiling differs from other programming languages, among other aspects, by the way memory is managed by the Java Virtual Machine (JVM). The technique used is the JVMPI Agent technology for Java.

Memory Profiling for Java supports JDK 1.3.x and JDK 1.4.x

- **C++**: ISO/IEC 14882:1998

How Memory Profiling for Java Works

When an application node is executed, the source code is executed. The Memory Profiling for Java feature uses the JVMPI mechanism to monitor the application. JVMPI outputs a dynamic **.jpt** file.

The **.jpt** file is split into a **.tpf** file and a **.txf** file, which can be viewed and controlled from the HCL OneTest™ Embedded GUI. Both the **.tpf** and **.txf** files need to be opened simultaneously to view the report.

Of course, these steps are mostly transparent to the user when the test or application node is executed in the HCL OneTest™ Embedded GUI.

To learn about	See
Performing Memory Profiling on C, C++ and Java source code	Using Runtime Analysis Features on page 306
JVMPI technology for Java memory analysis	JVMPI Technology on page 383
Understanding Memory Profiling Reports	Memory Profiling Results for Java on page 381
Viewing the Memory Profiling reports for Java	Using the Report Viewer on page 705
Customizing the Memory Profiling Viewer	Memory Profiling Viewer Preferences on page 1019

Related Topics

[Memory Profiling Settings on page 995](#) | [JVMPI Technology on page 383](#) | [About Memory Profiling for C and C++ on page 363](#)

Memory Profiling Results for Java

Memory Profiling for Java

After execution of an instrumented application, the Memory Profiling report displays:

- In the **Report Explorer** window: a list of available snapshots
- In the **Memory Profiling** window: the contents of the selected Memory Profiling snapshot

Report Explorer

The Report Explorer window displays a **Test** for each execution of the application node or for a test node when using Component Testing for Java. Inside each test, a **Snapshot** report is created for each Memory Profiling snapshot.

Method Snapshots

The Memory Profiling report displays snapshot data for each method that has performed an allocation. If the Java CLASSPATH is correctly set, you can click blue method names to open the corresponding source code in the Text Editor. System methods are displayed in black and cannot be clicked.

Method data is reset after each snapshot.

For each method, the report lists:

- **Method:** The method name. Blue method names are hyperlinks to the source code under analysis
- **Allocated Objects:** The number of objects allocated since the previous snapshot
- **Allocated Bytes:** The total number of bytes used by the objects allocated by the method since the previous snapshot
- **Local + D Allocated Objects:** The number of objects allocated by the method since the previous snapshot as well as any descendants called by the method
- **Local + D Allocated Bytes:** The total number of bytes used by the objects allocated by the method since the previous snapshot and its descendants

Referenced Objects

If you selected the **With objects** filter option in the JVMPI Settings dialog box, the report can display, for each method, a list of objects created by the method and object-related data.

From the **Memory Profiling** menu, select **Hide/Show Referenced Objects**.

For each object, the report lists:

- **Reference Object Class:**The name of the object class. Blue class names are hyperlinks to the source code under analysis.
- **Referenced Objects:** The number of objects that exist at the moment the snapshot was taken
- **Referenced Bytes:** The total number of bytes used by the referenced objects

Differential Reports

The Memory Profile report can display differential data between two snapshots within the same Test. This allows you to compare the referenced objects. There are two *diff* modes:

- Automatic differential report with the previous snapshot
- User differential report

Differential reports add the following columns to the current Memory Profiling snapshot report:

- **Referenced Objects Diff AUTO:** Shows the difference in the number of referenced objects for the same method in the current snapshot as compared to the previous snapshot
- **Referenced Bytes Diff AUTO :** Shows the difference in the memory used by the referenced objects for the same method in the current snapshot as compared to the previous snapshot
- **Referenced Objects Diff USER:** Shows the difference in the number of referenced objects for the same method in the current snapshot as compared to the user-selected snapshot
- **Referenced Bytes Diff USER:** Shows the difference in the memory used by the referenced objects for the same method in the current snapshot as compared to the user-selected snapshot

To add or remove data to the report:

1. From the **Memory Profiling** menu, select **Hide/Show Data**.
2. Toggle the data that you want to hide or display

To sort the report:

1. In the **Memory Profiling** window, click a column label to sort the table on that value.

To obtain a differential report:

1. From the **Memory Profiling** menu, select **Diff with Previous Referenced Objects**.

To obtain a user differential report:

1. In the **Report Explorer**, select the current snapshot
2. Right-click another snapshot in the same Test node and select **Diff Report**.

Related Topics

[Using the Memory Profiling Viewer on page 377](#) | [Memory Profiling Settings on page 995](#)

JVMPI Technology

Memory Profiling for Java

Memory Profiling for Java uses a special dynamic library, known as the Memory Profiling Agent, to provide advanced reports on Java Virtual Machine (JVM) memory usage.

Garbage Collection

JVMs implement a heap that stores all objects created by the Java code. Memory for new objects is dynamically allocated on the heap. The JVM automatically frees objects that are no longer referenced by the program, preventing many potential memory issues that exist in other languages. This process is called *garbage collection*.

In addition to freeing unreferenced objects, a garbage collector may also reduce heap fragmentation, which occurs through the course of normal program execution. On a virtual memory system, the extra paging required to service an ever growing heap can degrade the performance of the executing program.

JVMPI Agent

Because of the memory handling features included in the JVM, Memory Profiling for Java is quite different from the feature provided for other languages. Instead of Source Code Insertion technology, the Java implementation uses a JVM Profiler Interface (JVMPI) Agent whose task is to monitor JVM memory usage and to provide a memory dump upon request.

The JVMPI Agent analyzes the following internal events of the JVM:

- Method entries and exits
- Object and primitive type allocations

The JVMPI Agent is a dynamic library —**DLL** or **lib.so** depending on the platform used— that is loaded as an option on the command line that launches the Java program.

During execution, when the agent receives a snapshot trigger request, it can either an instantaneous JVMPI dump of the JVM memory, or wait for the next garbage collection to be performed.

Note Information provided by the instantaneous dump includes actual memory use as well as intermediate and unreferenced objects that are normally freed by the garbage collection. In some cases, such information may be difficult to interpret correctly.

The actual trigger event can be implemented with any of the following methods:

- A specified method entry or exit used in the Java code
- A message sent from the **Snapshot** button or menu item in the graphical user interface
- Every garbage collection

The JVMPI Agent requires that the Java code is compiled in debug mode, and cannot be used with Java in just-in-time (JIT) mode.

Related Topics

[Source code instrumentation overview on page 11](#) | [About Memory Profiling on page 363](#) | [Memory Profiling for Java on page 379](#)

Performance profiling

Performance Profiling

Performance Profiling applies to C and C++

The Performance Profiling feature puts successful performance engineering within your grasp. It provides complete, accurate performance data in an understandable and usable format so that you can see exactly where your code is least efficient. Using Performance Profiling, you can make virtually any program run faster. And you can measure the results.

Performance Profiling measures performance for every component in C and C++ source code, in real-time, and on both native or embedded target platforms. Performance Profiling instruments the C and C++ source code of your application. To test an application with the performance profiling feature: .

- Associate Performance Profiling with an existing test or application code.
- Build and execute your code in HCL OneTest™ Embedded.
- The application under test is instrumented with the Performance Profiling feature and provides a detailed report with metrics on execution time for each procedure/function/method of the application. For C language, it also provides an estimation of Worst Case Estimation Time.

Performance Profiling supports the following languages:

- **C**: ANSI 89, ANSI 99, or K&R C
- **C++**: ISO/IEC 14882:1998

Related Topics

[Source code instrumentation overview on page 11](#)

Performance profiling settings

You can configure the performance profiling settings before running your application in HCL OneTest™ Embedded for Studio.

Configuration Settings

All the following options must be set from the Configuration Settings window. To open this window:

- In the Project Window, right-click on the project and select Settings.

Enable the Performance Profiling

- In the Configuration Settings window, in the left panel, click Configuration properties > Build > Build options.
- In the right pane, click the Value field in Build options and click ... to open the Build options window.
- In the Build options list, click Performance Profiling to enable the feature.

Generate a trace file

- In the Configuration Settings window, in the left panel, click Configuration properties > Runtime analysis > Performance Profiling.
- In the right panel, click in the value field of the Trace file name (.tqf) line option, and click In the editor window that opens, specify a filename for the generated .tqf trace file for performance profiling.

To get an evaluation of the Worst Case Execution Time in the report, you must set the WCET option.

Select the Worst Case Execution Time and/or the maximum execution time for each function and descendants:

- In the Configuration Settings window, in the left panel, click Configuration properties > Runtime analysis > Performance Profiling.
- In the right pane, click Compute F max and F+D max time and select a value depending on the execution time that you want to be calculated for your project:
 - No: Does not calculate the maximum execution time for each function and descendants.
 - Yes: Calculate the maximum execution time for each function and its descendants.
 - Yes + WCET: Calculate the maximum execution time for each function and descendants, and the Worst Case Execution Time. With this option selected, the report indicates the number of time a function is called.

To get the performance profiling per entry point, you must enter the list of entry point threads of your application.

Entry points

To get the performance profiling per entry point, you must enter the list of entry points for each thread of your application.

- In the Configuration Settings window, in the left panel, click Configuration properties > Runtime analysis > General > Multi-thread options.
- Click in the value field and click ... to open the editor and enter the list of entry points for each thread of your application. Use commas to separate the thread names.

Then, run the application and see the Performance report.

Performance Profiling Results

The Performance Profiling report provides function profiling data for your program and its components so that you can see exactly where your program spends most of its time. When the configuration settings are set and the test application is run, you can see the Performance Profiling report.

The default Performance report is in HTML format. It is generated from a template named **wcetreport.template** provided as text file that you can modify to customize the report. It uses four online JavaScript libraries:

- Bootstrap,
- JQuery,
- Font Awesome,
- VisJS.

These libraries are not provided. You need an internet connectivity when you open the report. If not, download the libraries (.css and .js files), copy them in the same folder than your report, and modify the template file as follows:

Replace the following lines:

```
<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css"
  integrity="sha384-MCw98/SFnGE8fJT3GXwEOngsV7Zt27NXFoaoApmYm81iuXoPkFOJwJ8ERdknLPMO"
  crossorigin="anonymous">
<link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.5.0/css/all.css"
  integrity="sha384-B4dIYHKNBt8Bc12p+WXckhzcICo0wtJAoU8YZTY5qE0Id1GSseTk6S+L3BlXeVUIU"
  crossorigin="anonymous">
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/vis/4.21.0/vis.min.css">
...
<script src="https://code.jquery.com/jquery-3.3.1.slim.min.js"
  integrity="sha384-q8i/X+965Dz00rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8abtTE1Pi6jizo"
  crossorigin="anonymous"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.3/umd/popper.min.js"
  integrity="sha384-ZMP7rVo3mIyKv+2+9J3UJ46jBk0WLaUAdn689aCwoqbBJiSnjAK/l8WvCWPiPm49"
  crossorigin="anonymous"></script>
<script src="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/js/bootstrap.min.js"
  integrity="sha384-ChfqqxuZUCnJSK3+MXmPNIyE6ZbWh2IMqE241rYiqJxyMiZ6OW/JmZQ5stwEULTy"
  crossorigin="anonymous"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/vis/4.21.0/vis.js"></script>
```

With the following ones:

```

<link rel="stylesheet" href="./bootstrap.min.css">
<link rel="stylesheet" href="./all.css">
<link rel="stylesheet" href="./vis.min.css">
...
<script src="./jquery-3.3.1.slim.min.js"></script>
<script src="./popper.min.js"></script>
<script src="./bootstrap.min.js"></script>
<script src="./vis.js"></script>

```

The Performance profiling report is made of Summary, Functions and the Call Graph parts.

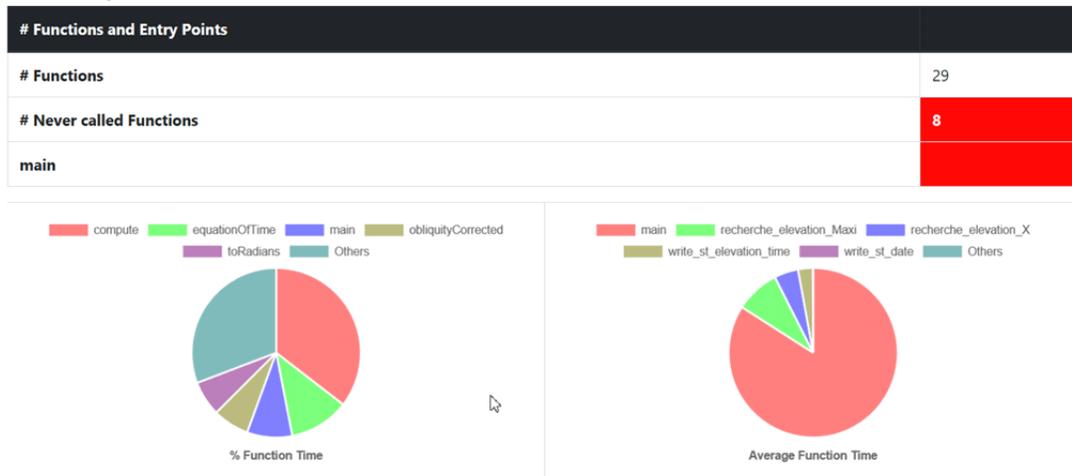
SUMMARY

Summary table

The Summary table displays the total number of functions and the number of functions that have never been executed and for which we have no data. If the instrumentation has been done with the WCET option (Worst Case Execution Time), then the table contains the list of the entry points with an evaluation of the WCET for each of them. This information can be empty (and the cell is red) if the WCET could not be computed. This can occur when one of the called functions in the call graph starting with this entry point has never been executed.

The WCET is given for each entry point if you have entered the list of entry point of your application in the Settings. For more details, see [Performance profiling settings on page 77](#).

Summary



Function time graphs

The Summary is followed by two graphs that provide a high level view of the largest time consumers detected by Performance Profiling in your application.

- **% Function Time:** It gives the five top functions with the greatest percentage of Function Time.
- **Average Function Time:** It gives the five top functions with the greatest Average Function Time.

FUNCTIONS

The Functions section of the report displays a table with the instrumented functions, procedures or methods (collectively referred to as functions) found in the application with the following information:

- **Functions:** Name of the function (in red if the function has never been executed).

If you have selected the WCET option, the chevron in front of the name allows the user to see how many times this function calls other functions. This can help to understand how the WCET is calculated.

- **EP:** Indicate if this function is an entry point or not. You can provide the list of the entry points, or, if not, they are deduced from the call graph (all the functions that are never called).
- **# Calls:** Number of times the function is called. If this value is 0, there is no more information for this function in the table because it has never been executed.
- **Function Time:** Total time spent for executing the function, excluding its descendants.
- **Function + Descendants Time:** Total time spent for executing the function, including its descendants.
- **% Function Time:** Percentage of time spent in this function against the total execution time.
- **% Function + Descendants Time:** Percentage of time spent for executing the function and its descendants against the total execution time.
- **Average Function Time:** Average time spent for executing this function, excluding its descendants.
- **Max Function Time:** Only if you set the option **Compute F max and F + D max**. Indicates the maximum time spent in a call while executing this function, excluding its descendants.
- **Max Function + Descendants Time:** Only if you set the option **Compute F max and F + D max time**, see [Performance profiling settings on page 77](#). This is the maximum time spent in a call while executing this function, including its descendants.
- **WCET:** Only if you set the option WCET, see [Performance profiling settings on page 77](#). It gives an evaluation of the Worst Case Execution Time. This information can be empty if the WCET could not be calculated during the execution. It is the case when one of the function and its descendants has never been executed. Click the chevron icon to deploy the list of functions that are not called.

Functions

Functions	EP	# Calls	Function Time	Function + Desc. Time	% Function Time	% Function + Desc. Time	Average Function Time	Max Function Time	Max Function + Desc. Time	WCET
> main	✓	1	13310us	153967us	8.64%	100%	13310us	13310us	153967us	
write_st_elevation_time		1	445us	445us	0.29%	0.29%	445us	445us	445us	445us

Call Graph

The Call Graph part displays all the functions in an interactive call graph that can be moved from left to right or from top to bottom. If the option WCET has been set, a tooltip on each function (node of the graph) gives the WCET. For more information, see [Performance profiling settings on page 77](#).

Customize the Performance Report

You can customize a Performance report.

The Performance report is based on a template called **wcetreport.template** that you can find in the following folder:

- In Windows:

```
<installation_directory>\IBM\TestRealTime\lib\reports
```

- In Unix:

```
<installation_directory>/IBM/TestRealTime/lib/reports
```

Raw data

This template is made of three sections:

- The HTML section that is the common part of all reports,
- A JavaScript section that sets the tables and call graph depending of 2 variables dynamically initialized while the report is creating:

```
var data = {{json}};    // the raw data
var d = new Date({{date}}) // the date of the generation
```

Raw data is composed of three sections at the top level:

- The list of the modules, each of them has the following information:
 - **Name** is the short name of the C file,
 - **Fullname** is the name and path of the C file,
 - **uuid** is a unique identifier of the module,
 - **unknown** is set to true is the module is not part of the information you provided (there is only one unknown module that gathers all the function calls that are not in the known modules),
 - **functions** is the list of the unique identifiers of functions of the module.

Modules are listed as hashmap with the uuid, as follows:

```
"modules": {
  "f5b5579edeaca82df478a6780c0c4c92": {
    "name": "USAGE.C",
    "fullname": "...",
    "uuid": "f5b5579edeaca82df478a6780c0c4c92",
    "unknown": false,
    "functions": [
      "ba9eb05ad703046fed306b4271b7ead7"
    ]
  },...
```

- The list of functions including following information:
 - **name** is the name of the C function,
 - **line** is the first line of the function in the module,
 - **id** is the number used in **.tsf** file to identify this function,
 - **stacksize** is the stack size computed during the execution if this option has been set (otherwise -1),
 - **uuid** is a unique identifier of the function,
 - **module** is a unique identifier of the module in which the function is declared,
 - **calls** is the list of the calls in this function. Each of them have the following information:
 - **calling_uuid** is the unique identifier of the calling function,
 - **called_uuid** is the unique identifier of the called function,
 - **line** is the line number of the call in the module,
 - **col** is the column number of the call in the module,
 - **same_module** is set to true id the called function is in the same module that the calling function.
 - **level** is a number that represent the level of the function in the call graph, starting to 0.
 - **calledby** is the list of unique identifiers of functions that call this one.
 - **maxLocal** is the maximum time spent in the function, excluding its descendants.
 - **maxTotal** is the maximum time spent in the function, including its descendants.
 - **sumLocal** is the time spent in the function, excluding its descendants.
 - **sumTotal** is the time spent in the function, including its descendants.
 - **wcet** is the Worst Case Execution Time of the function (this value is negative if it has not been calculated).
- Functions are listed as hashmap with the uuid, as following:

```

"functions": {
  "ba9eb05ad703046fed306b4271b7ead7": {
    "name": "write_usage",
    "line": 9,
    "id": 1,
    "stacksize": -1,
    "uuid": "ba9eb05ad703046fed306b4271b7ead7",
    "module": "f5b5579edeaca82df478a6780c0c4c92",
    "calls": [
      {
        "calling_uuid": "ba9eb05ad703046fed306b4271b7ead7",
        "called_uuid": "7b6cd643b5b44e1e0510f30f62729eba",
        "line": 10,
        "col": 2,
        "same_module": false
      }
    ]
  },
  "level": 1,
  "calledby": [
    "3fb6b20659c9b70fc6d01ba797abae1f"
  ],
  "maxLocal": 27,
  "maxTotal": 28,
  "sumLocal": 3190,
  "sumTotal": 3853,
  "averageLocal": 0,
  "wcet": 60
},...

```

- The final section contains the following information:
 - **entrypoints** is the list of entry points of the application; each of them contains:
 - **name** is the name of the entry points.
 - **module** is the uuid of the module where is the entry point.
 - **wcet** is the Worst Case Execution Time of the entry points (this value is negative if it has not been calculated).
 - **timeunit** is the unit of time used in the report (**us** is for micro-second, **ms** for millisecond, **s** for second).
 - **level** is the setting for performance (**0** when there is no "compute F max + D max time", **1** when this option is not set to yes, **2** when it is set to yes + WCET).

An example of this section:

```
"entrypoints": [
  {
    "name": "main",
    "module": "57f1afe89e0a74b786aab75cd448db9b",
    "wcet": -10
  }
],
"timeunit": "us",
"level": 2
```

Performance Profiling SCI Dump Driver

Performance Profiling for C and C++

In C and C++, you can dump profiling trace data without using standard I/O functions by using the Performance Profiling Dump Driver API contained in the **atqapi.h** file, which is part of the Target Deployment Port

To customize the Performance Profiling Dump Driver, open the Target Deployment Port directory and edit the **atqapi.h**. Follow the instructions and comments included in the source code.

Related Topics

[Generating SCI Dumps on page 1051](#)

Using the Performance Profiling Viewer

Performance Profiling for C and C++

The product GUI displays Performance Profiling results in the Performance Profiling Viewer.

Reloading a Report

If a Performance Profiling report has been updated since the moment you have opened it in the Performance Profiling Viewer, you can use the Reload command to refresh the display:

To reload a report:

1. From the View Toolbar, click the **Reload** button.

Resetting a Report

When you run a test or application node several times, the Performance Profiling results are appended to the existing report. The Reset command clears previous Performance Profiling results and starts a new report.

To reset a report:

1. From the View Toolbar, click the **Reset** button.

Exporting a Report to HTML

Performance Profiling results can be exported to an HTML file.

To export results to an HTML file:

1. From the File menu, select Export

Related Topics

[Performance Profiling Results on page 78](#) | [Applying Performance Profile Filters on page 392](#) | [Opening a Report on page 683](#) | [Report Explorer on page 1024](#)

Applying Performance Profile Filters

Performance Profiling for C and C++

Filters allow you to streamline a performance profile report by filtering out specific events. Use the **Filter List** dialog box to specify how events are to be detected and filtered.

The export and import facilities are useful if you want to share and re-use filters between Projects and users.

To access the Filter List:

1. From the **Performance Profile Viewer** menu, select **Filters** or click the **Filter** button in the Performance Profile Viewer toolbar.

To create a new filter:

1. Click the **New** button
2. Create the new filter with the [Event Editor on page 407](#).

To modify an existing filter:

1. Select the filter that you want to change.
2. Click the **Edit** button.
3. Modify the filter with the Event Editor.

To import one or several filters:

1. Click the **Import** button.
2. Locate and select the **.xlf** file(s) that you want to import.
3. Click **OK**.

To export a filter event:

1. Select the filter that you want to export.
2. Click the **Export** button.
3. Select the location and name of the exported **.xlf** file.
4. Click **OK**.

Related Topics

[Editing Performance Profile Filters on page 393](#) | [Performance Profiling Results on page 78](#) | [Using the Performance Profiling Viewer on page 391](#)

Editing Performance Profile Filters

Performance Profiling for C and C++

Use the Filter Editor to create or modify filters that allow you to hide or show routines in the performance profile report, based on specified filter criteria.

By default, routines that match the filter criteria are hidden in the report. Use the **Invert filter** option to invert this behaviour: only routines that match the filter criteria are displayed.

Routine filters can be defined with one or more of the following criteria:

- **Name:** Specifies the name of a routine as the filter criteria.
- **Calls >** and **Calls <:** The number times the function was called is greater or lower than the specified value.
- **F Time >** and **F Time <:** Function time greater or lower than the specified value.
- **F+D Time >** and **F+D Time <:** Function and descendant time greater or lower than the specified value.
- **F Time (%) >** and **F Time (%) <:** Function time, expressed in percentage, greater or lower than the specified value.
- **F+D Time (%) >** and **F+D Time (%) <:** Function and descendant time, expressed in percentage, greater or lower than the specified value.
- **Average >** and **Average <:** The average time spent executing the function greater or lower than the specified value.

To define a routine filter:

1. In the **Name** box, specify a name for the filter.
2. Click **More** or **Fewer** to add or remove a criteria.
3. From the drop-down criteria box, select a criteria for the filter, and an argument.

Arguments must reflect an exact match for the criteria. Pay particular attention when referring to labels that appear in the sequence diagram since they may be truncated.

You can use wildcards (*) or regular expressions by selecting the corresponding option.

4. Add or remove a criteria by clicking the **More** or **Fewer** buttons.
5. Click **Ok**.

Related Topics

[Applying Performance Profile Filters on page 392](#) | [Performance Profiling Results on page 78](#) | [Using the Performance Profiling Viewer on page 391](#)

Runtime tracing

Runtime Tracing

Runtime Tracing for C, C++

Runtime Tracing is a feature for monitoring real-time dynamic interaction analysis of your C, C++ source code.

Runtime Tracing uses exclusive Source Code Insertion (SCI) instrumentation technology to generate trace data, which is turned into UML sequence diagrams within the HCL OneTest™ Embedded GUI.

In HCL OneTest™ Embedded, Runtime Tracing can run either as a standalone product, or in conjunction with a Component Testing or System Testing test node.

- You associate Performance Profiling with an existing test or application code.
- You build and execute your code in HCL OneTest™ Embedded.
- The application under test, instrumented with the Runtime Tracing feature, then directs output to the [UML/SD Viewer on page 402](#), which provides a real-time UML Sequence Diagram of your application's behavior.

Runtime Tracing supports the following languages:

- **C**: ANSI 89, ANSI 99, or K&R C
- **C++**: ISO/IEC 14882:1998

How Runtime Tracing Works

When an application node is executed, the source code is instrumented by the C, C++ Instrumentor (**attolcc1**, **attolccp** or **attolcc4**). The resulting source code is then executed and the Runtime Tracing feature outputs a static **.tsf** file for each instrumented source file as well as a dynamic **.tdf** file.

These files can be viewed and controlled from the HCL OneTest™ Embedded GUI. Both the **.tsf** and **.tdf** files need to be opened simultaneously to view the report.

Of course, these steps are mostly transparent to the user when the test or application node is executed in the HCL OneTest™ Embedded GUI or Eclipse (for C and C++).

UML sequence diagram overview

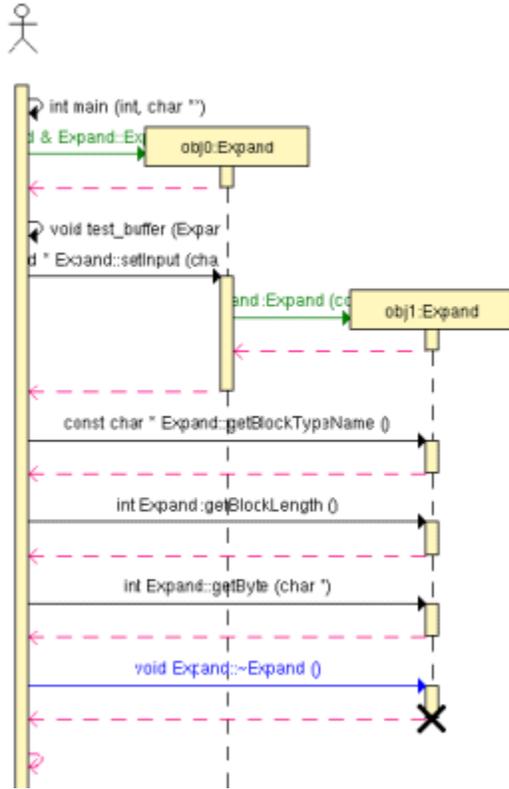
Runtime Tracing for C, C++./span>

The lifeline of an object is represented in the UML/SD Viewer as shown below.

The instance creation box displays the name of the instance.

Example

Below is an example of object lifelines generated by Runtime Tracing from a C++ application.



In this C++ example:

- Functions and static methods are attached to the World instance.
- Objects are labelled with **obj <number> : <classname>**
- The black cross represents the destruction of the instance.
- Constructors are displayed as green arrow actions.
- Destructors are the blue arrows.
- Return messages are dotted red lines.
- Other functions and methods are black.
- Themain() is a function of the World instance called by the same World instance.

You can perform the following tasks from the sequence diagram:

- To jump to the corresponding portion of source code, double-click an element of the object lifeline to open the Text Editor at the corresponding line in the source code

- To jump to the beginning or to the end of an instance:
 - Right-click an element of the object lifeline to jump to the beginning or to the end of an instance.
 - Select **Go to Head** or **Go to Destruction** in the pop-up menu.
- To filter an instance out of the UML sequence diagram:
 - Right-click an element of the object lifeline.
 - Select **Filter instance** in the pop-up menu.

UML Sequence Diagrams

A sequence diagram is a Unified Modeling Language (UML) diagram that provides a view of the chronological sequence of messages between instances (objects or classifier roles) that work together in an interaction or interaction instance. A sequence diagram consists of a group of instances (represented by lifelines) and the messages that they exchange during the interaction. You line up instances participating in the interaction in any order from left to right, and then you position the messages that they exchange in sequential order from top to bottom. Activations sometimes appear on the lifelines.

A sequence diagram belongs to an interaction in a collaboration or an interaction instance in a collaboration instance.

Related Topics

[About the UML/SD Viewer on page 402](#)

Tracing a test node

Runtime Tracing for C and C++

When Runtime Tracing is activated with a Component Testing or System Testing test node, monitoring a UML sequence diagram of the execution from Runtime Tracing is a matter of including Runtime Tracing in the Build options of an existing test node.

If however you are using Runtime Tracing on its own, you need to create an application node in the Project Explorer, and associate it with the source files that you want to monitor.

To enable the runtime tracing option:

1. From the Build toolbar, click the **Options** button.
2. In the options list, select **Runtime Tracing**.
3. Click anywhere outside the options list to close it.

Next time you run a **Make** command on the selected test node, a Runtime Tracing UML sequence diagram will be produced simultaneously with the standard test output.

To view runtime tracing output:

1. Runtime Tracing output is displayed, with the UML/SD Viewer, in the same UML sequence diagram as the standard test's graphical output.

Related Topics

[C and Ada Component Testing UML Sequence Diagrams on page 509](#) | [C++ Component Testing UML Sequence Diagrams on page 524](#) | [System Testing UML Sequence Diagrams on page 640](#)

Step-by-step tracing

Runtime Tracing for C and C++

When tracing large applications, it may be useful to slow down the display of the UML sequence diagram. You can do this by using the Step-by-Step mode.

To activate Step-by-Step mode:

1. From the **UML/SD Viewer** menu, select **Display Mode** and **Step-by-Step**.

To select the type of graphical element to skip over:

1. In the UML/SD Viewer toolbar, click the ▼ button.
2. Select the graphical elements that will stop the **Step** command. Clear the elements that are to be ignored.

To step to the next selected element:

1. Click the **Step** button in the UML/SD Viewer toolbar or press **F10**.

To skip to the end of execution:

1. Click the **Continue** button in the UML/SD Viewer toolbar. This will immediately display the rest of the UML sequence diagram.

To restart the Step-by-Step display:

1. Click the **Restart** button in the UML/SD toolbar.

To de-activate Step-by-Step mode

1. From the **UML/SD Viewer** menu, select **Display Mode** and **All**.

Related Topics

[UML/SD Viewer Preferences on page 1016](#) | [Runtime Tracing Control Settings on page 997](#) | [UML/SD Viewer Toolbar on page 1028](#)

Using sequence diagram triggers

Runtime Tracing for C and C++

Sequence Diagram triggers allow you to predefine automatic start and stop parameters for the UML/SD Viewer. The trigger capability is useful if you only want to trace a specific portion of an instrumented application.

Triggers can be inactive, time-dependent, or event-dependent.

To access the Trigger dialog box:

1. From the **UML/SD Viewer** menu, select **Triggers** or click the **Trigger** button in the UML/SD Viewer toolbar.

Start and End of Runtime Tracing:

The Runtime Tracing start is defined on the **Start** tab:

- **At the beginning:** Runtime Tracing starts when the application starts.
- **On time:** Runtime Tracing starts after a specified number of microseconds.
- **On event:** Runtime Tracing starts when a specified event is detected. One or several events must be specified with the Event Editor.

The Runtime Tracing end is defined on the **Stop** tab:

- **Never:** Runtime Tracing ends when the application exits.
- **On time:** Runtime Tracing ends after a specified number of seconds.
- **On event:** Runtime Tracing ends when a specified event is detected. One or several events must be specified with the Event Editor.

To create a new trigger event:

1. Click the **New** button
2. Create the new trigger event with the **Event Editor**.

To modify an existing trigger event:

1. Select the trigger event that you want to change.
2. Click the **Edit** button.
3. Modify the trigger event with the **Event Editor**.

To import one or several trigger events:

The import facility is useful if you want to reuse trigger events created in another Project.

1. Click the **Import** button.
2. Locate and select the file(s) that you want to import.
3. Click **OK**.

To export a trigger event:

The export facility allows you to transfer trigger events.

1. Select the trigger event that you want to export.
2. Click the **Export** button.
3. Select the location and name of the exported **.tft** file.
4. Click **OK**.

Related Topics

[Editing Trigger or Filter Events on page 407](#) | [Applying Filters on page 400](#)

Applying Sequence Diagram Filters

Runtime Tracing for C, C++.

Filters allow you to streamline a sequence diagram by filtering out specific event types. Use the Viewer's **Filter List** dialog box to specify how events are to be detected and filtered.

The export and import facilities are useful if you want to share and re-use filters between Projects and users.

To access the Filter List:

1. From the **UML/SD Viewer** menu, select **Filters** or click the **Filter** button in the UML/SD Viewer toolbar.

To create a new filter:

1. Click the **New** button
2. Create the new filter with the [Event Editor on page 407](#).

To modify an existing filter:

1. Select the filter that you want to change.
2. Click the **Edit** button.
3. Modify the filter with the Event Editor.

To import one or several filters:

1. Click the **Import** button.
2. Locate and select the **.tft** file(s) that you want to import.
3. Click **OK**.

To export a filter event:

1. Select the filter that you want to export.
2. Click the **Export** button.
3. Select the location and name of the exported **.tft** file.
4. Click **OK**.

Related Topics

[Editing Trigger or Filter Events on page 407](#)

Adding UML notes to source code

Runtime Tracing for C and C++

You can manually add your own notes inside your source code in order to make them display in the UML sequence diagram when runtime tracing is enabled. To do this, you must insert the following line, called an instrumentation pragma, in your C or C++ source code:

```
#pragma attol att_insert_ATT_USER_NOTE("Text")
```

This can be done automatically with the text editor.

To manually set the syntax coloring mode:

1. In a C or C++ source file, place your cursor at the line where you want a UML note to be displayed in the UML sequence diagram.
2. In the toolbar, click Add Note . This inserts the instrumentation pragma line in the source code:
3. Replace "**Text**" with a meaningful string that will be displayed in the note.

Related Topics

[Runtime tracing on page 394](#) | [Editing code and test scripts on page 693](#) | [UML sequence diagrams on page 397](#) | [Notes on page 650](#) | [Instrumentation pragmas on page 1046](#)

Viewing UML sequence diagrams

Runtime Tracing for C and C++

The UML/SD Viewer renders sequence diagram reports as specified by the UML standard.

UML sequence diagram can be produced directly via the execution of the SCI-instruction application when using the Runtime Tracing feature.

The UML/SD Viewer can also display UML sequence diagram results for Component and System Testing features.

To learn about	See
The meaning of UML sequence diagrams produced by the Runtime Tracing feature	Runtime Tracing sequence diagram representations
The meaning of UML sequence diagrams produced by the <i>Component Testing for C and Ada</i> feature	Component Testing for C and Ada sequence diagram representations on page 509
The meaning of UML sequence diagrams produced by the <i>Component Testing for C++</i> feature	Component Testing for C++ sequence diagram representations
The meaning of UML sequence diagrams produced by the <i>System Testing for C</i> feature	System Testing sequence diagram representations on page 640
Moving around in a UML sequence diagram	Navigating through UML/SD Viewer reports
Filtering out specific events from the UML sequence diagram	Applying filters
Setting start and stop triggers on specific events in the UML sequence diagram	Sequence diagram triggers
How to find particular items within a UML sequence diagram	Finding a text string in a UML Sequence Diagram
Using the zoom setting	Setting a zoom level
Customizing the UML/SD Viewer	UML/SD Viewer preferences

Related Topics

[About Runtime Tracing on page 394](#)

Navigating through UML Sequence Diagrams

Runtime Tracing for C and C++

There are several ways of moving around the UML sequence diagrams displayed by the UML/SD Viewer:

- **Navigation Panel:** Click and drag the **Navigation** button in the lower right corner of the **UML/SD Viewer** window to scroll through a miniature navigation pane representing the entire UML sequence diagram.
- **Free scroll:** Press the **Control** key and the left mouse button simultaneously. This displays a compass icon, allowing you to scroll the UML sequence diagram in all direction by the moving the mouse.
- **Report Explorer:** The Report Explorer is automatically activated when the UML/SD Viewer is activated. The Report Explorer offers a hierarchical view of instances. Click an item in the **Report Explorer** to locate and select the corresponding UML representation in the main **UML/SD Viewer** window.

Some elements in the sequence diagram provide links to the corresponding line in the source code. For example, if you click a message in a sequence diagram, the text editor opens the corresponding source file in the text editor.

Note If the source file is already open, it is not brought forward.

Related Topics

[Report Explorer on page 1024](#) | [Finding Text in a UML Sequence Diagram on page 409](#) | [Applying Filters on page 400](#) | [Sequence Diagram Triggers on page 399](#) | [UML/SD Viewer Preferences on page 1016](#) | [UML/SD Viewer Toolbar on page 1028](#)

Time Stamping

Runtime Tracing for C and C++

The UML/SD Viewer displays time stamping information on the left of the UML sequence diagram. Time stamps are based on the execution time of the application on the target.

You can change the display format of time stamp information in the UML/SD Viewer Preferences.

The following time format codes are available:

- **%n** - nanoseconds
- **%u** - microseconds
- **%m** - milliseconds
- **%s** - seconds
- **%M** - minutes
- **%H** - hours

These codes are replaced by the actual number. For example, if the time elapsed is 12ms, then the format **%mms** would result in the printed value **12ms**. If the number 0 follows the % symbol but precedes the format code, then 0

values are printed to the viewer - otherwise, 0 values are not printed. For example, if the time elapsed is 10ns, and the selected format code is **%0mms %nns**, then the time stamp would read **0ms 10ns** .

Note To change the format code you must press the Enter key immediately after selecting/entering the new code. Simply pressing the **OK** button on the **Preferences** window will not update the time stamp format code.

Related Topics

[UML/SD Viewer Preferences on page 1016](#) | [About the UML/SD Viewer on page 402](#)

Coverage Bar

Runtime Tracing for C and C++

In C and C++, the coverage bar provides an estimation of code coverage.

Note The coverage bar is unrelated to the Code Coverage feature. For detailed code coverage reports, use the dedicated Code Coverage feature.

When using the Runtime Tracing feature, the UML/SD Viewer can display an extra column on the left of the UML/SD Viewer window to indicate code coverage simultaneously with UML sequence diagram messages.

The UML/SD Viewer code coverage bar is merely an indication of the ratio of *encountered* versus *declared* function or method entries and potential exceptions since the beginning of the sequence diagram.

If new declarations occur during the execution the graph is recalculated, therefore the coverage bar always displays an increasing coverage rate.

To hide the coverage bar:

1. Right-click inside the **UML/SD Viewer** window.
2. From the pop-up menu, select **Hide Coverage**.

To show the coverage bar:

1. Right-click inside the **UML/SD Viewer** window.
2. From the pop-up menu, select **Show Coverage**.

Related Topics

[About Code Coverage on page 54](#) | [Memory Usage Bar on page 404](#) | [Time Stamping on page 403](#) | [UML/SD Viewer Preferences on page 1016](#) | [Memory Profiling Settings on page 995](#)

Memory Usage Bar

Runtime Tracing for C and C++.

When using the Runtime Tracing feature on a Java application, the UML/SD Viewer can display an extra bar on the left of the UML/SD Viewer window to indicate total memory usage for each sequence diagram message event.

The memory usage bar indicates how much memory has been allocated by the application and is still in use or not garbage collected.

In parallel to the UML sequence diagram, the graph bar represents the allocated memory against the highest amount of memory allocated during the execution of the application.

This ratio is calculated by subtracting the amount of free memory from the total amount of memory used by the application. The total amount of memory is subject to change during the execution and therefore the graph is recalculated whenever the largest amount of allocated memory increases.

A tooltip displays the actual memory usage in bytes.

To activate or disable coverage tracing with a Java application:

1. Before building the node-under-analysis, open the **Memory Profiling** settings box.
2. Set **Coverage Tracing** to **Yes** or **No** to respectively activate or disable coverage tracing for the selected node.
3. Click **OK** to override the default settings of the node

To hide the memory usage bar:

1. Right-click inside the **UML/SD Viewer** window.
2. From the pop-up menu, select **Hide Memory Usage**.

To show the memory usage bar:

1. Right-click inside the **UML/SD Viewer** window.
2. From the pop-up menu, select **Show Memory Usage**.

Related Topics

[Thread Bar on page 405](#) | [Time Stamping on page 403](#) | [UML/SD Viewer Preferences on page 1016](#) | [Memory Profiling Settings on page 995](#)

Thread Bar

Runtime Tracing for C and C++

When using the Runtime Tracing feature on C and C++ code, the UML/SD Viewer can display an extra column on the left of its window to indicate the active thread during each UML sequence diagram event.

Each thread is displayed as a different colored zone. A tooltip displays the name of the thread.

Click the thread bar to open the **Thread Properties** window.

To hide the thread bar:

1. Right-click inside the **UML/SD Viewer** window.
2. From the pop-up menu, select **Hide Thread Bar**.

To show the thread bar:

1. Right-click inside the **UML/SD Viewer** window.
2. From the pop-up menu, select **Show Thread Bar**.

Related Topics

[Thread Properties on page 406](#) | [Memory Usage Bar on page 404](#) | [Time Stamping on page 403](#) | [UML/SD Viewer Preferences on page 1016](#)

Thread properties

Runtime Tracing for C and C++

The **Thread Properties** window displays a list of all threads that are created during execution of the application.

Threads are listed with the following properties:

- **Colour tab:** As displayed in the Thread Bar.
- **Thread ID:** A sequential number corresponding to the order in which each thread was created.
- **Name:** The name of the thread.
- **State:** Either **Sleeping** or **Running** state.
- **Priority:** The current priority of the thread.
- **Since:** The timestamp of the moment the thread entered the current state.

Click the title of each column to sort the list by the corresponding property

Thread Properties Filter

By default, the Thread Properties window displays the entire list of thread states during execution of the program.

To switch the Thread Properties Filter:

1. Click **Filter** to display reduce the display to the list of threads created by the application.
2. Click **Unfilter** to return the full list of thread states.

Related Topics

[Thread Bar on page 405](#)

Filtering sequence diagram events

Runtime Tracing for C and C++

Use the Event Editor to create or modify event triggers or filters for UML sequence diagrams:

- **Filters:** Specified events are hidden or shown in the UML sequence diagram.
- **Start triggers:** The UML/SD Viewer starts displaying the sequence diagram when a specified event is encountered. If no event matches the output of the application, the diagram will appear blank.
- **Stop triggers:** The UML/SD Viewer stops displaying the sequence diagram when a specified event is encountered.

Events can be related to messages, instances, notes, synchronizations, actions or loops.

To define an event or filter:

1. Specify a name for the event.
2. Select the type of UML element you want to define for the event and select **Activate**. Several types of elements can be activated for a single filter or trigger event.
3. Click **More** or **Fewer** to add or remove line to the event criteria.
4. From the drop-down criteria box, select a criteria for the filter, and an argument.
5. Arguments must reflect an exact match for the criteria. Pay particular attention when referring to labels that appear in the sequence diagram since they may be truncated.
6. You can use wildcards (*) or regular expressions by selecting the corresponding option.
7. Click the  button to enable or disable case sensitivity in the criteria.
8. You can add or remove a criteria by clicking the **More** or **Fewer** buttons.
9. Click **Ok**.

Message Criteria

- **Name:** Specifies a message name as the filter criteria.
- **Internal message:** Considers all messages other than constructor calls coming from any internal source, as opposed to those messages coming from the World instance.

- **From Instance:** Considers all messages other than constructor calls prior to the first message sent from the specified object
- **To Instance:** Considers out all messages other than constructor calls if any message is sent to the specified object
- **From World:** Considers all messages received from the World instance
- **To World:** Considers all messages sent to the World instance

Instance Criteria

- **Name:** Specifies an instance name as the filter criteria
- **Instance child of:** Specifies a child instance of the specified class.

Note Criteria

- **All:** Considers all notes
- **Name:** Specifies a note name
- **All message notes:** Considers any note attached to a message
- **All instance notes:** Considers any note attached to an instance
- **Instance child of:** Specifies a note attached to an instance of the specified class
- **Note on message named:** Considers a note attached to a specified message
- **With style named:** Considers a note with the specified style attributes

Synchronization Criteria

- **All:** Considers all synchronization events
- **Name:** Specifies a synchronization name

Action Criteria

- **All:** Considers all actions
- **Name:** Specifies an action name
- **From Instance:** Considers an action performed by the specified object
- **From World:** Considers all actions performed by the World instance

- **Instance child of:** Specifies an action performed by an instance of the specified class
- **With style named:** Considers an action with the specified style attributes

Loop Criteria

- **All:** Considers all loops
- **Name:** Specifies a loop name

Boolean Operators

- **All Except** expresses a NOT operation on the criteria
- **Match All** performs an AND operation on the series of criteria
- **Match Any** performs an OR operation on the series of criteria

Related Topics

[Applying Filters on page 400](#) | [Sequence Diagram Triggers on page 399](#) | [Understanding UML Sequence Diagrams on page 395](#)

Finding text in a sequence diagram

Runtime Tracing for C and C++

The UML/SD Viewer has an extensive search facility that allows users to locate specific UML sequence diagram elements by searching for a text string.

To search for a text string inside the UML/SD Viewer:

1. Click inside a **UML/SD Viewer** window to activate it.
2. From the **Edit** menu, select **Find** menu item. The **Find** dialog box opens.
3. Type your search criteria in the **Find** dialog box.
4. Click the **Find Next** button.
5. If a string corresponding to the search criteria is found in the UML/SD Viewer, the string is highlighted and the following message is displayed: **Runtime Tracing has finished searching the document.**
6. Click **OK**.

Search Options

- **Forward** and **Backward** specifies the direction of the search.
- The **Search into** option allows you to specify type of object in which you expect to find the search string.
- The **Find** dialog box accepts either UNIX regular expressions or DOS-like wildcards ('?' or '*'). Select either **wildcard** or **reg. exp.** in the *Find* dialog box to select the corresponding mode.

Related Topics

[About the UML/SD Viewer on page 402](#) | [Navigating through UML/SD Viewer reports on page 402](#)

Exporting a sequence diagram to a text file (.csv)

The UML/SD Viewer can generate sequence diagram results in a **.csv** text file. A **.csv** file is a text file presented as a table. You can import these results into a text editor, a spreadsheet application or use them to operate a file *diff* comparison for non-regression evaluation.

You can specify the format used to generate the **.csv** text file in the Data table preferences.

To generate a **.csv** text file from a sequence diagram:

1. After running an application or test node with Runtime Tracing, open a sequence diagram.
2. From the **Runtime Trace** menu, select **Generate CSV**.
3. In the **Generate CSV** window, specify the name of the text file.
4. Select **Generate columns header** to insert a line with column titles at the top of the file.
5. In the **Columns** list, select the sequence diagram elements that you want to export to the text file. Use the **Up** and **Down** buttons to change the order.
6. In the **Additional Filters** list, select any sequence diagram elements that you want to filter out of the report.
7. Click **Preview** to see how the table will appear in a spreadsheet application. The **CSV Preview** window is limited to the first 100 lines. Click **Close** to exit the preview.
8. Click **OK**.

Related Topics

[Data table preferences on page 1011](#) | [Exporting reports to CSV](#) | [Exporting reports to HTML on page 705](#)

Advanced runtime tracing

Multi-thread support

Runtime Tracing for C, C++

Runtime Tracing can be configured for use in a multi-threaded environment such as Windows.

Multi-thread mode protects Target Deployment Port global variables against concurrent access. This causes a significant increase in Target Deployment Port size as well as an impact on performance. Therefore, select this option only when necessary.

Multi thread settings:

These settings are ignored if you are not using a multi-threaded environment. To change these settings, use the **Build Settings > Target Deployment Port build** dialog box.

- **Maximum number of threads:** This value sets the size of the thread management table inside the Target Deployment Port. Lower values save memory on the target platform. Higher values allow more simultaneous threads.
- **Record and display thread info:** When selected, the UML Sequence Diagram displays a note each time a new thread is created and each time a thread's schedule is changed.

To access the multi-thread build settings:

1. In the **Project Explorer**, click the **Open Settings...** button.
2. Select one or several nodes in the Configuration pane.
3. Select **Build > Target Deployment Port build**.
4. Set the **Multi-threaded application** and **Maximum number of threads** settings.
5. Select **Runtime Analysis > Runtime Tracing > Runtime options**.
6. Set **Record and display thread info** to **Yes** or **No**.
7. When you have finished, click **OK** to validate the changes.

Related Topics

[Runtime Tracing Control Settings on page 997](#) | [Build settings on page 983](#) | [About Configuration Settings on page 658](#)

Partial trace flush

Runtime Tracing for C, C++

When using this mode, the Target Deployment Port only sends messages related to instance creation and destruction, or user notes. All other events are ignored. This can be useful to reduce the output of trace.

When Partial Trace Flush mode is enabled, message dump can be toggled on and off during trace execution.

The Partial Trace Flush settings are located in the **Runtime Tracing Settings**.

To enable Partial Trace Flush from the Node Settings:

1. In the **Project Explorer**, click the **Open Settings...** button.
2. Select one or several nodes in the Configuration pane.
3. Select **Runtime Analysis > Runtime Tracing > Runtime options**.
4. Set the **Partial Runtime Tracing flush** setting to **Yes** or **No** to activate or disable the mode.
5. When you have finished, click **OK** to validate the changes.

To toggle message dump from within the source code:

1. To do this, use the Runtime Tracing pragma user directives:
 - `_ATT_START_DUMP`
 - `_ATT_STOP_DUMP`
 - `_ATT_TOGGLE_DUMP`
 - `_ATT_DUMP_STACK`

See the **Reference Manual** for more information about pragma directives.

To control message dump through a user signal (native UNIX only):

This capability is available only when using a native UNIX target platform.

Under UNIX, the kill command allows you to send a user signal to a process. Runtime Tracing can use this signal to toggle message dump on and off.

1. In the **Project Explorer**, click the **Open Settings...** button.
2. Select one or several nodes in the Configuration pane.
3. Select the **Runtime Analysis** node and the **Runtime Tracing** node.
4. Select **Runtime Tracing Control**.
5. Set the **Partial Runtime Tracing flush** setting to **Yes** or **No** to activate or disable the mode.
6. When you have finished, click **OK** to validate the changes.

Note By default, the expected signal is **SIGUSR1**, but you can change this by setting the **ATT_SIGNAL_DUMP** environment variable to the desired signal number. See the **Reference Manual** for more information about environment variables.

Related Topics

[Runtime Tracing Control Settings on page 997](#)

Trace item buffer

Runtime Tracing for C and C++

Buffering allows you to reduce formatting and I/O processing at time-critical steps by telling the Target Deployment Port to only output trace information when its buffer is full or at user-controlled points.

This can prove useful when using Runtime Tracing on real-time applications, as you can control buffer flush from within the source-under-trace.

To activate or de-activate trace item buffering:

1. In the **Project Explorer**, click the **Open Settings...** button.
2. Select one or several nodes in the Configuration pane.
3. Select the **Runtime Analysis** node and the **Runtime Tracing** node.
4. Select **Runtime Tracing Control**.
5. Set the **Buffer trace items** setting to **Yes** or **No** to activate or disable the mode.
6. Set the size of the buffer in the **Items buffer size** box.
7. When you have finished, click **OK** to validate the changes.

A smaller buffer optimizes memory usage on the target platform, whereas a larger buffer improves performance of the real-time trace. The default value is 64.

Flushing the Trace Buffer through a User Directive

It can be useful to flush the buffer before entering a time-critical part of the application-under-trace. You can do this by adding the **_ATT_FLUSH_ITEMS** user directive to the source-under-trace.

Note See Runtime Tracing pragma directives in the **Reference Manual** to control Target Deployment Port buffering from within the source code.

Splitting trace files

Runtime Tracing for C and C++

During execution, Runtime Tracing generates a **.tdf** dynamic file. When a large application is instrumented, the size of the **.tdf** file can impact performance of UML/SD Viewer.

Splitting trace files allows you to split the **.tdf** trace file into smaller files, resulting in faster display of the UML Sequence Diagram and to optimize memory usage. However, split trace files cannot be used simultaneously with On-the-Fly tracing.

When displaying split **.tdf** files, Runtime Tracing adds Synchronization elements to the UML sequence diagram to ensure that all instance lifelines are synchronized.

To set Split Trace mode:

1. In the **Project Explorer**, select the highest level node from which you want to activate split trace mode.
2. Click the **Open Settings...** button.
3. Select **Runtime Analysis** and the **Runtime Tracing** settings.

Select **Trace Control**.

Set the **Size (Kb)** of each split **.tdf**. The default size is 5000 Kb.

Specify a **File Name Prefix** for the split **.tdf** filenames. The prefix is followed by a 4-digit number that identifies each file.

1. Click **OK**.

Note The total size of split **.tdf** files is slightly larger than the size of a single **.tdf** file, because each file contains additional context information.

Trace Probes for C

Trace Probes for C

Trace Probes for C

The Trace Probes feature of HCL OneTest™ Embedded allows you to manually add special probe C macros at specific points in the source code under test, in order to trace messages.

Upon execution of the instrumented binary, the probes record information on the exchange of specified messages, including message content and a time stamp. Probe trace results can then be processed and displayed in the **UML/SD Viewer**.

The use of C macros offers extreme flexibility. For example, when delivering the final application, you can leave the macros in the final source and simply provide an empty definition.

Trace Probes supports ANSI 89, ANSI 99, or K&R C.

How Trace Probes work

The first step is to manually add specific macros to your C source code.

When the test or application node is executed, the Probe Processor produces an instrumented source file. which is functionally identical to the original, but which generates extra message tracing results.

The resulting source code is then executed and the Trace Probe feature outputs a **.rio** output file for each probe instance.

A **.tsf** static trace file is generated during instrumentation, and the **.rio** output file is processed and transformed into a **.tdf** file. These files can be viewed and controlled from the HCL OneTest™ Embedded GUI. Both the **.tsf** and **.tdf** files need to be opened simultaneously to view the UML sequence diagram report.

Of course, these steps are mostly transparent to the user when the test or application node is executed in the HCL OneTest™ Embedded GUI.

Related Topics

[Probe Control Settings on page 1006](#) | [Probe Output Modes on page 416](#) | [Circular Trace Buffer on page 655](#) | [About the UML/SD Viewer on page 402](#)

Using Probe Macros

Trace Probes for C

Before adding probe macros to your source code, add the following **#include** statement to each source file that is to contain a probe:

```
#include "atlprobe.h"
```

The **atl_start_trace()** macro must be called before any probe activity can occur; for example, it can be placed at the start of the application.

The **atl_end_trace()** macros must be called after all probe activity has ended; for example, when the application terminates.

Other macros must be placed inside the source code, at locations that are relevant for the messages that you want to trace.

The following probe macros are available:

- `atl_dump_trace()`
- `atl_end_trace()`
- `atl_rcv_trace()`
- `atl_select_trace()`
- `atl_send_trace()`
- `atl_start_trace()`
- `atl_format_trace()`

Please refer to the section on Probe Macros in Reference for a complete definition of each probe macro.

To activate the Trace Probe feature:

1. In the Project Browser, select the application or System Testing node on which you want to use the feature.
2. Click Settings and open the **Probe control** box.
3. Set **Probe enable** to **Yes**, select the correct output mode in **Probe Settings** and click **OK**.
4. Edit the source code under test to add the trace probe macros, including the `#include` line.
5. Set up your trace probes within your application source files.

To read the trace probe output:

1. From the **File** menu, select **Open and File**.
2. In the file selector, select **Trace Files (*.tsf, *.tdf)** and select the **.tsf** and **.tdf** files produced after the execution of the application under test.
3. Click **OK**.

Trace Probe output modes

Trace Probes for C

By default, the message traces are written to the **.rio** output file. However, in some cases, this may not be practical, therefore the Trace Probe feature can be configured to send trace information to a temporary buffer before writing to a file.

To change the way traces are stored, specify the trace mode as specified in the **Probe Control Settings**:

- **DEFAULT:** In this mode, the message traces are written directly to the **.rio** output file.
- **FIFO:** Binary format traces are directed to a temporary *first-in first-out* memory buffer before writing to the **.rio** file when the **atl_dump_trace** macro is encountered. This mode is intended for embedded or realtime applications which may not be able to access a filesystem when running.
- **FILE:** Binary format traces are written to a low footprint temporary file before writing to the **.rio** file when the **atl_dump_trace** macro is encountered. This mode is intended for embedded or realtime applications which may not have enough memory or processing power to continuously write to the **.rio** file. In this case for example, a second application could be set up to read the file and generate the **.rio** result file.

- **USER:** Uses methods, described in a user-defined **probecst.c** file to direct traces to a user-defined format before writing to the **.rio** file when the **atl_dump_trace** macro is encountered. See [Customizing the USER output mode on page 418](#) for more information.
- **IGNORE:** Use this setting to ignore trace probe macros on compilation. In this case, the binary is compiled without instrumentation.

When **FIFO**, **FILE** or **USER** are selected, the traces must be flushed to the **.rio** file with a specific **atl_dump_trace** macro placed in a source file.

Use the **DEFAULT** output mode whenever possible. In most other cases, the **FIFO** or **FILE** should be enough and can be optimized using parameters provided in the Reference section.

Only use **USER** mode if none of the other settings are practical for your application. Using the **USER** output mode requires that you rewrite your own **probecst.c** and **probecst.h** using the files provided with the product as a template. See [Customizing the USER output mode on page 418](#) for more information.

When using the **USER** mode, you must specify the location of the user-defined **probecst.c** and **probecst.h** files in the **USER** custom files directory setting. See [Probe control settings on page 1006](#) for details.

Related Topics

[Trace Probes on page 414](#) | [Circular Trace Buffer on page 655](#) | [Probe Control Settings on page 1006](#) | [Customizing the USER output mode on page 418](#)

Traces Probes and System Testing for C

Trace Probes for C

You can use Trace Probes to produce a System Testing **.pts** test script based on probe activity.

When a probed application is executed, the **.rio** result file is processed, which produces a **.pts** test script for System Testing for C.

The Script generation flags setting allows you to specify the command line arguments that will be used to generate the **.pts** test script. The available flags are:

-accuracy=<time>

-polling=<time>

These values express the desired accuracy and polling intervals to be used in the **.pts** test script, where **<time>** is expressed in milliseconds (ms).

You can edit and reuse this script in later tests to replay the exact same data exchanges in a System Testing for C test node.

Related Topics

[Trace Probes for C on page 414](#) | [About System Testing for C on page 586](#) | [Probe Control Settings on page 1006](#)

Customizing the USER output mode

Trace Probes for C

The **USER** output mode for Trace Probes requires that you rewrite user-defined **probecst.c** and **probecst.h** based on the files provided with the product.

Only use the **USER** mode if the **DEFAULT**, **FIFO** or **FILE** modes are not practical for your application.

To rewrite your own routines, make a copy of the **probecst.c** and **probecst.h** that are provided with the product and use them as a template. These files are located in the following directory located in the installation directory of the product:

```
/lib/probe/probecst/fifo
```

Note These are the files that are used for the **FIFO** output mode, therefore ensure that any changes that you make are performed on copies of these files.

The implementation delivered in the **FIFO** mechanism is based on a circular buffer. The instrumented application sends traces to the intermediate storage buffer, by using the **atl_write_probe** function. The traces can then be read by the **atl_read_probe** function.

You can modify this file to adapt the probe mechanism to your application and platform.

For example, when using **USER** mode, the main probed application may store messages in binary format in a shared memory or pipe, whereas a dedicated "dump application" can be written to read the shared memory or pipe and to generate the **.rio** result file.

By using this method, the probed application can still run with minimal overhead while another process generates the **.rio** result file either on the fly or after the execution of the probed application.

Whichever storage mechanism you use, it is important that the dump application runs within the same hardware architecture as the main application to avoid misalignment or little-big endian problems.

When using the **USER** mode, you must specify the location of the user-defined **probecst.c** and **probecst.h** files in the **USER** custom files directory setting. See [Probe control settings on page 1006](#) for details.

The **probecst.c** file contains definitions for the Trace Probe macro functions. These are detailed below. For the usage and syntax of the Trace Probe macros, please refer to the Reference section. For each function, the **probecst.c** file contains comments that should help you to rewrite each of these functions.

The following functions must be executed during the execution of the probed application:

- **atl_create_probe**
- **atl_end_probe**

- `atl_write_key`
- `atl_write_probe`

The following functions can be executed when the probed application ends or after the application has finished in a dedicated dump application:

- `atl_open_probe`
- `atl_close_probe`
- `atl_read_probe`

atl_start_trace

The **atl_start_trace** function executes **atl_create_probe**. It must be called before any other macros, once for each instance. Its role is to open, create and initialize the intermediate storage media used to keep messages in the intermediate binary format.

atl_end_trace

The **atl_start_trace** function executes **atl_end_probe**. It must be called at the end of the application, once for each instance. Its role is to close the intermediate storage media used to keep messages in the intermediate binary format.

atl_send_trace and atl_rcv_trace

The **atl_send_trace** and **atl_rcv_trace** functions execute **atl_write_probe** in order to dump the message to the intermediate storage media.

It is important that the `.rio` result file retains the message sequence. Therefore, ensure that data is recorded in the execution order.

atl_write_probe

The role of the **atl_write_probe** function is to record the following data:

- The complete message, the length of the message is provided to help.
- The date of the event.
- An internal code.
- The key format.

If your USER mechanism required the use of intermediate storage, the **atl_dump_trace** must be called after the **atl_end_trace** macro.

atl_dump_trace()

This macro can be either part of the probed application or part of a dedicated dump program that would be executed after the main application, depending on what is practical in your application.

The **atl_dump_trace()** macro executes, for each instance,

- **atl_open_probe**,
- **atl_read_probe** for each recorded message, and
- **atl_close_probe**.

atl_open_probe

The role of the **atl_open_probe** function is to reopen the intermediate storage and point to the first recorded message.

atl_close_probe

The role of the **atl_close_probe** function is to close, destroy or free the memory of the intermediate storage.

atl_read_probe

The role of the **atl_read_probe** function is to retrieve the following data from the intermediate storage:

- The message as it was recorded during the execution.
- A timestamp of the message.
- An internal code.
- The key format of the message.

atl_select_trace

The role of the **atl_select_trace** function is to execute **atl_write_key** in the API. The code of this function must not be customized. It must be copied from the original **probecst.c** without any change.

Related Topics

[Trace Probes for C on page 414](#) | [Trace Probe output modes on page 416](#) | [Probe Control Settings on page 1006](#)

Coupling Analysis

Coupling Analysis consists of Control Coupling and Data Coupling.

Overview

Control Coupling is defined as “the manner or degree by which one software component influences the execution of another software component” in the [Clarification of Structural Coverage Analyzes of Data Coupling and Control Coupling](#) document edited by the **Certification Authorities Software Team (CAST)**. The purpose is 'to provide a measurement and assurance of the correctness of these modules/components' interactions and dependencies'. Control Coupling is used to verify that all the interactions between modules have been covered by at least one test.

HCL OneTest™ Embedded introduces a new coverage level called “Control Coupling” for C language that consists in verifying that all the interactions between modules have been covered by at least one test. This new coverage level is implemented in HCL OneTest™ Embedded in two ways:

- Modules are compilation units, in this case:
 - Control Couplings are calls between two functions that are in two different compilation units.
 - Control Coupling is not a simple interaction. It is a control flow in the calling module that ends with an interaction with another module.
 - Groups of compilation units can be defined as a single module. This will increase the number of calls between modules but also increase the number of control flows in the calling modules.
 - The report contains a button to display:
 - All the Control Couplings (default option).
 - Only the shortest Control Couplings (only the last calls between modules are taken into account)
 - Only the longest Control Couplings (the sub-control flows are ignored)
- Modules are Functions, in this case:
 - Control Couplings are considered as all the calls between two functions, in the same compilation unit or not.
 - Each Control Coupling is only a call, and not a control flow as previously defined.

So, to identify the Control Couplings, HCL OneTest™ Embedded analyzes all the external calls between modules (definition of the modules could be different depending on the option) and statically identifies all the possible paths in the calling module that end with each external call, excluding the one that starts with a static function (ex: a function that can't be called from another module). This constitutes the set of Control Coupling of the application.

For each of them, HCL OneTest™ Embedded provides the following information:

- The calling modules.
- The complete control flow (example: the set of successive calls, the last one is the external call). If the option "module as function" is set, each control flow has two functions only.
- In case of option module as "compilation unit":
 - Is it the longest one that leads to this external call (it is not the longest when there is another Control Coupling that includes the current one).
 - Is it the shortest one that leads to this external call (it is not the shortest when there is another Control Coupling that is included by the current one).
- It is covered or not.

- The list of test cases that each Control Coupling covered.
- The list of requirements that are related to the test cases.

How Control Coupling Works

When an application node or a test is executed, the source code is instrumented by the Instrumentor (attolcc4 for C language) that produces a static file with the extension **.tsf** containing information on the Control Couplings. The resulting source code is then compiled, linked and executed and the Control Coupling feature outputs a dynamic file with the extension **.tgf**.

These 2 types of files are the input of the report generator that produces a report in HTML format (and optionally the raw data can be generated in a Json file). A template is provided for this generator. You can provide your own template to modify the report.

If the Control Coupling feature is used with unit testing feature, the report generator can take the **.tdc** files as input files. This allows to have also in the report the test cases that covered each Control Coupling and the associated requirements declared in the **.ptu** file. If not, the test cases are identified by their execution date, and there is no requirement.



Note:

To visualize your report in HCL OneTest™ Embedded for Eclipse IDE, if you are using the default browser option, be sure that JavaScript is enabled. Otherwise, you can choose another browser that is compatible with your version of JavaScript by changing it in **Window > Preferences > General > Web Browser**.

Set Control Coupling Options

You can set the options for Control Coupling to build your project in HCL OneTest™ Embedded Studio.

Execute a build with Control Coupling

- In HCL OneTest™ Embedded Studio, open the Settings of the project and click the **Configuration Properties > Build > Build options** menu.
- In the right panel, click on the **Build options** and edit the options by clicking on the ... button.
- In the dialog window that shows up on the right, you can select the different tools that can be used for the build. Select **Ctrl Coupling analysis** to enable the control coupling feature.

Control Coupling options

Options for Control Coupling can be updated in the following menu of the settings: **Configuration Properties > Runtime analysis > Control coupling**

From this setting page, you can change the following choices:

- **Trace file name (.tgf)**: sets the name of the trace file dedicated to control coupling. By default, this name is the base name of the test with the extension **.tgf**.
- **Exclude libraries**: Include or exclude the control couplings that end with a call to a function that is not part of the application (sets the **-noccext** option of the report generator if it is set to yes).
- **Report Template**: changes the template of the report generator. By default, this template is **ccreport.template**.
- **Module as**: Select the choice that corresponds the best to your definition of a module. A module can be defined as a function or a compilation unit. HCL OneTest™ Embedded offers two ways to interpret Control Coupling, depending on how the "module" in CAST-19 is interpreted:
 - **Module as function**: Each call between each function is considered as Control Coupling.
 - **Module as compilation unit**: Only the calls between two functions in two different compilation units are considered as Control Coupling. Moreover, the different called stacks in the calling module are also considered as different Control Couplings. With the previous option set, the user can group two or more compilation units in a single module (called component) in order to ignore the calls between these compilation units.

Control Coupling Report

Depending on the build settings, from HCL OneTest™ Embedded V8.2.0, you can get a Control Coupling report with module as compilation unit as module or a Control Coupling report with module as Function as module as a result to a project build.

The default Control Coupling report is in HTML format. It is generated from a template named **ccreport.template** (for the option module as compilation unit"), or **ccfreport.template** (for the option "module as function") provided as text file that you can modify to customize the report. It uses 4 online JavaScript libraries:

- Bootstrap,
- JQuery,
- Font Awesome,
- VisJS.

These libraries are not provided. You need an internet connectivity when you open the report. If not, download the libraries (.css and .js files), copy them in the same folder than your report, and modify the template file as follows:

Replace the following lines:

```
<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css"
  integrity="sha384-MCw98/SFnGE8fJT3GXwEOngsV7Zt27NXFoaAPmYm81iuXoPkFOJwJ8ERdknLPMO"
  crossorigin="anonymous">
<link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.5.0/css/all.css"
  integrity="sha384-B4dIYHKNBt8Bc12p+WXckhzcICo0wtJAoU8YZTY5qE0Id1GSseTk6S+L3BlXeVIU"
  crossorigin="anonymous">
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/vis/4.21.0/vis.min.css">
...
<script src="https://code.jquery.com/jquery-3.3.1.slim.min.js"
  integrity="sha384-q8i/X+965Dz00rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8abtTE1Pi6jizo"
  crossorigin="anonymous"></script>
```

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.3/umd/popper.min.js"
  integrity="sha384-ZMP7rVo3mIyKv+2+9J3UJ46jBk0WLaUAdn689aCwoqbBJiSnjAK/l8WvCWPIpM49"
  crossorigin="anonymous"></script>
<script src="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/js/bootstrap.min.js"
  integrity="sha384-ChfqqxuZUCnJSK3+MXmPNIyE6ZbWh2IMqE241rYiqJxyMiZ6OW/JmZQ5stwEULTy"
  crossorigin="anonymous"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/vis/4.21.0/vis.js"></script>
```

With the following ones:

```
<link rel="stylesheet" href="/bootstrap.min.css">
<link rel="stylesheet" href="/all.css">
<link rel="stylesheet" href="/vis.min.css">
...
<script src="/jquery-3.3.1.slim.min.js"></script>
<script src="/popper.min.js"></script>
<script src="/bootstrap.min.js"></script>
<script src="/vis.js"></script>
```

If you set a module as a compilation unit in the control coupling properties, you get a control coupling report with compilation unit in output of your project build. If you set a module as a function, you get a control coupling report with function as module. For more details on the control coupling settings, see [Set Control Coupling options on page 163](#) for HCL OneTest™ Embedded for Eclipse IDE. In a report with function as module, the report shows all the function calls (internal and external).

The Control Coupling report is made of three parts.

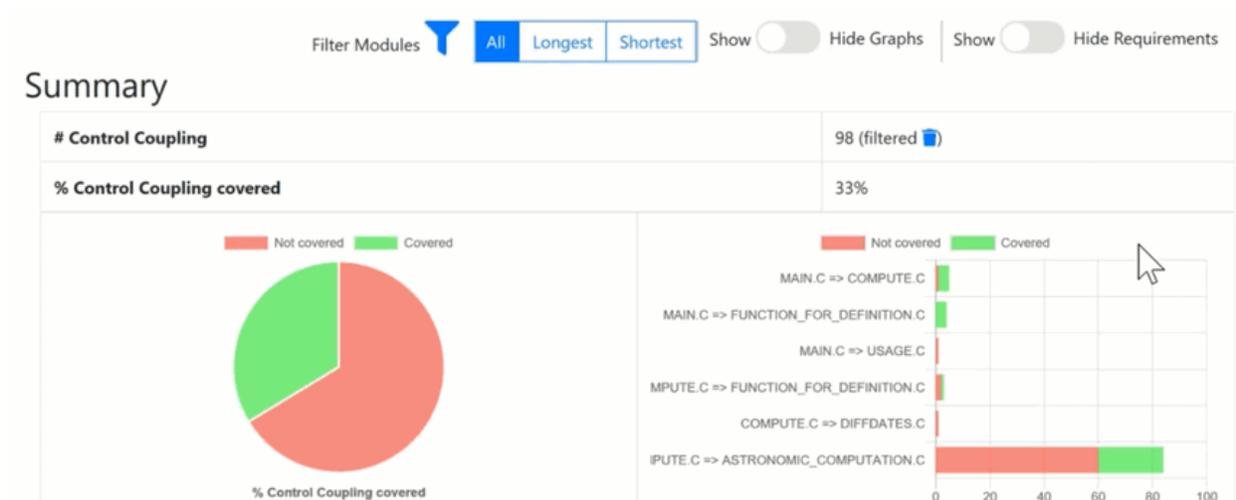
Summary

In the Summary section, you find the number of Control Couplings for your application that have been covered, given the information you provided and the percentage of Control Couplings that have been covered.

A graph displays the global percentage of covered and not covered control couplings for all the application.

The Summary table displays the following information:

- The percentage of Control Couplings of your application by pair of modules that have not been covered, given the information you provided.
- The percentage of Control Couplings that have been covered by pair of modules.



Details

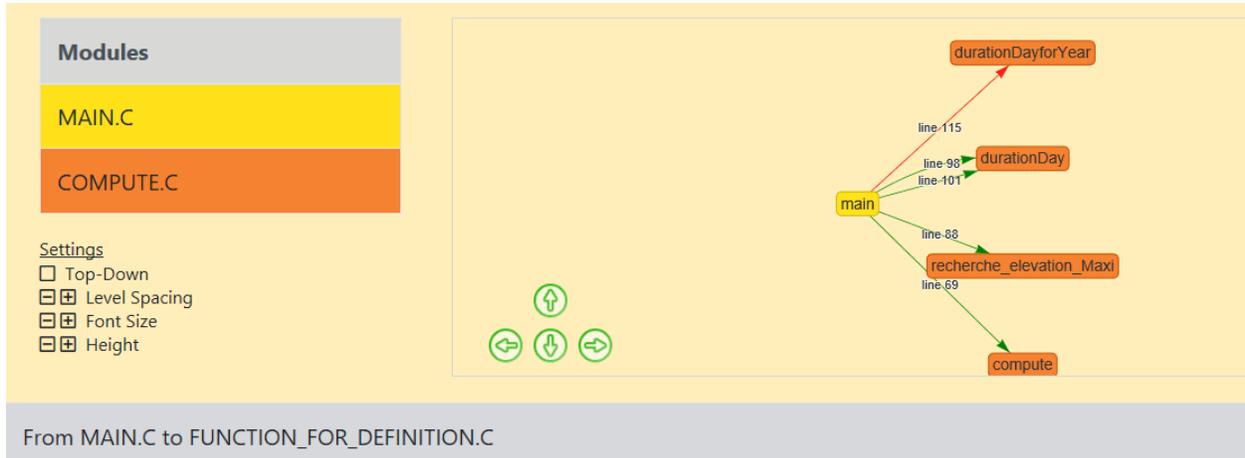
The **Details** table lists all the Control Couplings and displays for each of them the following information:

- The calling compilation unit.
- The control flow, for example: the successive calls in the module that end with the external call in the called module. Note that the called module is mentioned in the last function of the control flow. In case of option "module as function", this control flow contains only two functions.
- A check mark if it is a longest Control Flow (only if the option "module as compilation unit" is set).
- A check mark if it is a shortest Control Flow (only if the option "module as compilation unit" is set.)
- The list of test cases that covered this control flow. If the Control Coupling feature has been set with the unit testing feature, the test cases are the one in the **.ptu** files named as <service>/<test>.
- The associated requirements. If the Control Coupling feature has been set with the unit testing feature, the requirements are those that have been described in the **.ptu** files with the keyword REQUIREMENT for each test cases that covered this Control Coupling.
- A check mark if the control coupling has been covered.

Call Graph

For each compilation unit, a partial call graph displays all the functions as an interactive call graph from left to right or from top to bottom, depending on the selector button position on the top of the call graph.

You can select a control coupling in the table to highlight it in the call graph.



At the end of the report, a complete Call Graph displays all the functions calls.

Filters

Filters can be applied in the report with different buttons at the top:

- If the option “module as compilation unit” is set, you can choose first to display all Control Couplings, the longest (only the Control Couplings that have the longest control flow in the calling module) or the shortest (only the Control Couplings that have the shortest control flow in the calling module). The summary tables and the details table are updated accordingly to your selection. This option applies to reports with compilation unit as module only.
- You can select the calling modules and the called modules. It filters the Control Couplings depending on their calling and called modules. The summary tables and the details table are updated accordingly to your selection.
- You can choose to display all the graphs or hide them in the report.
- You can show or hide the Requirements.

Customize Control Coupling Report

The Control Coupling report is created from a template called **ccreport.template** (if option “module as compilation unit” is set), or **ccfreport.template** (if option “module as function” is set) that you can find in the folder **<install>/lib/reports**.

This template is made of 2 parts:

- The HTML part that is the common part of all reports,
- A JavaScript part that sets the tables and call graph depending of 2 variables initialized dynamically when the report is creating:

```
var data = {{{json}}}; // the raw data
```

```
var d = new Date({{date}}) // the date of the generation
```

Raw data

Raw data is composed of 4 sections at the top level:

- A summary of the Control Coupling metrics:
 - **nbcc** is the number of Control Coupling found in the application,
 - **nbcovered** is the number of Control Coupling found in the application that have been covered by at least one test,
 - **nbccShortest** and **nbcoveredShortest** are the same for the shortest Control Coupling,
 - **nbccLongest** and **nbcoveredLongest** are the same for the longest Control Coupling,
 - **filtered** is set to true if the report has been generated with a filter (shortest or longest),
 - **filtered_longest** is set to true if the report has been generated with a filter longest (set only if filter is true).

```
"filtered": false,
"filtered_longest": false,
"nbcc": 112,
"nbcovered": 48,
"nbccShortest": 32,
"nbcoveredShortest": 25,
"nbccLongest": 58,
"nbcoveredLongest": 23
```

- The list of the modules, each of them has the following information:
 - **Name** is the short name of the C file,
 - **Fullname** is the name and path of the C file,
 - **uuid** is a unique identifier of the module,
 - **unknown** is set to true is the module is not part of the information you provided (there is only one unknown module that gathers all the call to functions that are not in the known modules),
 - **functions** is the list of the unique identifiers of functions of the module.

Modules are listed as hashmap with the uuid, as follows:

```
"modules": {
  "f5b5579edeaca82df478a6780c0c4c92": {
    "name": "USAGE.C",
    "fullname": "...",
    "uuid": "f5b5579edeaca82df478a6780c0c4c92",
    "unknown": false,
    "functions": [
      "ba9eb05ad703046fed306b4271b7ead7"
    ]
  }, ...
}
```

- The list of functions including following information:
 - **name** is the name of the C function,
 - **line** is the first line of the function in the module,
 - **id** is the number used in **.tsf** file to identify this function,
 - **stacksize** is the stack size computed during the execution if this option has been set (otherwise -1),
 - **uuid** is a unique identifier of the function,
 - **module** is a unique identifier of the module in which the function is declared,
 - **calls** is the list of the calls in this function. Each of them have the following information:
 - **calling_uuid** is the unique identifier of the calling function,
 - **called_uuid** is the unique identifier of the called function,
 - **line** is the line number of the call in the module,
 - **col** is the column number of the call in the module,
 - **same_module** is set to true id the called function is in the same module that the calling function.
 - **level** is a number that represent the level of the function in the call graph, starting to 0.
 - **calledby** is the list of unique identifiers of functions that call this one.

- Functions are listed as hashmap with the uuid, as following:

```

"functions": {
  "ba9eb05ad703046fed306b4271b7ead7": {
    "name": "write_usage",
    "line": 9,
    "id": 1,
    "stacksize": -1,
    "uuid": "ba9eb05ad703046fed306b4271b7ead7",
    "module": "f5b5579edeaca82df478a6780c0c4c92",
    "calls": [
      {
        "calling_uuid": "ba9eb05ad703046fed306b4271b7ead7",
        "called_uuid": "7b6cd643b5b44e1e0510f30f62729eba",
        "line": 10,
        "col": 2,
        "same_module": false
      }
    ],
    "level": 1,
    "calledby": [
      "3fb6b20659c9b70fc6d01ba797abae1f"
    ]
  }, ...
}

```

- The list of the Control Couplings, each of them have the following information:
 - **calls** is the list of successive calls that composed this control coupling, each of them have the following information:
 - **calling_uuid** is the unique identifier of the calling function.
 - **called_uuid** is the unique identifier of the called function.
 - **isShortest** is set to true if the control coupling is a shortest one.
 - **isLongest** is set to true if the control coupling is a longest one.
 - **line** is the line number of the call in the module.
 - **col** is the column number of the call in the module.
 - **same_module** is set to true if the called function is in the same module that the calling function.
 - **testcases** is the list of test cases that covered the control coupling, each of them have the following information:
 - **name** is the name of the test case.
 - **requirements** is the list of requirements that is covered by this test case.

Control couplings are listed as an array, as follows:

```

"controlcouplings": [
  {
    "isShortest": true,
    "isLongest": true,
    "calls": [
      {
        "calling_uuid": "3fb6b20659c9b70fc6d01ba797abae1f",
        "called_uuid": "0dd641fbc509e237cb0600f451d27d59",
        "line": 100,
        "col": 19,
        "same_module": false
      }
    ],
    "testcases": [
      {
        "name": "fct_8/1",
        "requirements": [
          {
            "name": "REQ_PTU_123"
          }
        ]
      }
    ]
  }
], ...

```

Overview

Data Coupling is defined as “the manner or degree by which one software component influences the execution of another software component” in the [Clarification of Structural Coverage Analyzes of Data Coupling and Control Coupling](#) document edited by the **Certification Authorities Software Team (CAST)**. The purpose is 'to provide a measurement and assurance of the correctness of these modules/components' interactions and dependencies'. Data Coupling is used to verify that all the global variables of the application under test have been consumed in read (also called *use*) and write (also called *def*) during the tests.

HCL OneTest™ Embedded introduces a new coverage level call “data coupling” for C language that consists to verify that all the global variables of the application under test has been consumed in read (also called *use*) and write (also called *def*) during the tests, as following:

- For each global variable, HCL OneTest™ Embedded identifies the *def* and *use*. Then it considers all the possible *def/use* pair as a data coupling.
- To cover a Data Coupling, i.e. a *def/use* pair, this *def* and this *use* must be executed from at least one test.

HCL OneTest™ Embedded provides a new interactive HTML-based report for Data Coupling.

To identify Data Coupling instances, HCL OneTest™ Embedded analyzes all the global variables of the application, where they are read and written. For one global variable, each pair of write and read constitutes an instance of Data Coupling.

For each data coupling, HCL OneTest™ Embedded provides the following information:

- The name of the global variable.
- The def position (file name, line, and column).
- The use position (file name, line, and column).
- The list of test cases that covered the Data Coupling.
- The list of requirements that are relative to these test cases.

How Data Coupling works

HCL OneTest™ Embedded identifies the position if the *def/use* using coverage information. When you select the Data Coupling option, some coverage options are set automatically: blocks, calls and conditions.

Coverage files (**.fdc** and **.tio**) are the input of the report generator that produces a report in HTML format (and optionally the raw data can be generated in a Json file). A template is provided for this generator. You can provide your own template to modify the report.

If the Data Coupling feature is used with unit testing feature, the report generator could take as input the **.tdc** files. This allows to have also in the report the test cases that covered each Control Coupling and the associated requirements declared in the **.ptu** file. If not, the test cases are identified by its execution date, and there is no requirement.

Set Data Coupling options

You can set the options for Data Coupling to run the build for your project in HCL OneTest™ Embedded Studio.

Execute a build with Data Coupling

- In HCL OneTest™ Embedded Studio, open the Settings of the project and click the **Configuration Properties > Build > Build options** menu.
- In the right panel, click on the **Build options** and edit the options by clicking on the ... button.
- In the dialog window that shows up on the right, you can select the different tools that can be used for the build. Select **Data Coupling analysis** to enable the Data Coupling feature.

Data Coupling options

Options for Data Coupling can be updated in the following menu of the settings: **Configuration Properties > Runtime analysis > Data Coupling**

From this setting page, you can change the following choice:

- **Report Template:** You can change the template of the report generator. By default, this template is **ccreport.template**.

Data Coupling report

From HCL OneTest™ Embedded V8.2.0, you can get a HTML interactive Data Coupling report as a result to your project build.

The default Data Coupling report is in HTML format. It is generated from a template named **dcreport.template** provided as a text file that you can modify to customize the report. It uses four online JavaScript libraries:

- Bootstrap,
- JQuery,
- Font Awesome,
- VisJS.

These libraries are not provided. You need an Internet connection when you open the report. Otherwise, download the libraries (.css and .js files), copy them in the same folder as your report's, and modify the template file as follows:

Replace:

```
<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css"
  integrity="sha384-MCw98/SFnGE8fJT3GXwEOngsV7Zt27NXFoaoApmYm81iuXoPkFOJwJ8ERdknLPMO"
  crossorigin="anonymous">
<link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.5.0/css/all.css"
  integrity="sha384-B4dIYHKNBt8Bc12p+WXckhzcICo0wtJAoU8YZTY5qE0Id1GSseTk6S+L3B1XeVIU"
  crossorigin="anonymous">
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/vis/4.21.0/vis.min.css">
...
<script src="https://code.jquery.com/jquery-3.3.1.slim.min.js"
  integrity="sha384-q8i/X+965Dz00rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8abtTE1Pi6jizo"
  crossorigin="anonymous"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.3/umd/popper.min.js"
  integrity="sha384-ZMP7rVo3mIykV+2+9J3UJ46jBk0WLaUAdn689aCwoqbBJiSnjAK/l8WvCWPIpM49"
  crossorigin="anonymous"></script>
<script src="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/js/bootstrap.min.js"
  integrity="sha384-ChfqxuxZUCnJSK3+MXmPNiY6ZbWh2IMqE241rYiqJxyMiZ6OW/JmZQ5stweULTy"
  crossorigin="anonymous"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/vis/4.21.0/vis.js"></script>
```

with

```
<link rel="stylesheet" href="./bootstrap.min.css">
<link rel="stylesheet" href="./all.css">
<link rel="stylesheet" href="./vis.min.css">
...
<script src="./jquery-3.3.1.slim.min.js"></script>
<script src="./popper.min.js"></script>
<script src="./bootstrap.min.js"></script>
<script src="./vis.js"></script>
```

The Report is made of three parts.

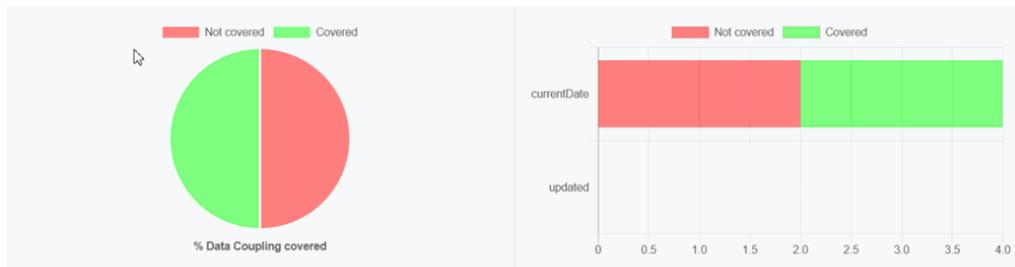
Summary

In the summary section, a table displays the following information:

- The number of global variables in your application.
- The number of Data Couplings in your application.
- The number and the list of global variables without Data Coupling. If you get this information, it means that HCL OneTest™ Embedded has identified global variables that are read but never written, or written but never read. This could be due to the fact that only a part of the application is analyzed.

Two charts display the following information:

- The percentage of Data Coupling in a pie graph.
- A two-colored horizontal graph that provides a number of covered and uncovered Data Couplings for each global variable.



Details

A table lists all the Data Couplings and displays the following information for each of them:

- **Variable:** The name of the global variable.
- **Def:** The Def position of the column: file name [line] and (column).
- **Use:** The Use position of the column: file name [line] and (column).
- **Test Cases:** The list of cases that covered the Data Coupling.
- **Requirements:** The list of requirements relative to these test cases.
- **Covered:** This option is checked if the Data Coupling has been covered.

They are grouped by global variables.

Details

Variable	Def	Use	Test Cases	Requirements	Covered
Global Variable ' currentDate '					
currentDate	main [MAIN.C] (33:2)	main [MAIN.C] (118:7)	• <i>no name #1</i> [Thu Nov 14 14:06:14 2019]		✓
currentDate	main [MAIN.C] (33:2)	main [MAIN.C] (123:2)	• <i>no name #1</i> [Thu Nov 14 14:06:14 2019]		✓
currentDate	DiffDays [DIFFDATES.C] (74:28)	main [MAIN.C] (118:7)			
currentDate	DiffDays [DIFFDATES.C] (74:28)	main [MAIN.C] (123:2)			
Global Variable ' updated '					
This variable ' updated ' is written but never read within the selected compilation units					

Call graph

The call graph displays all the global variables with their interactions with one or more functions of the application that read or/and write them.

- Incoming arrows are 'Def' (write).
- Outcoming arrows are 'Use' (read).

The arrows between them represent a 'Def' or a 'Use' (depending of the sense of the arrow). It is green if the corresponding 'Def' or 'Use' has been covered. These arrows are not representing Data Coupling. A Data Coupling instance is a couple of incoming and outcoming arrows that reach the same global variables.

Filters

Buttons can be used to filter different sections of the report.

- **Show/Hide Graph:** It is used to show or hide the call graph at the end of the report.
- **Show/Hide Requirements:** It is used to show or hide the **Requirements** column in the **Details** section of the report.

Customize Data Coupling Report

The Data Coupling report is based on a template called **ccreport.template** that you can find in the following folder:

Raw data

This template is made of 2 parts:

- The HTML part that is the common part of all reports,
- A JavaScript part that sets the tables and call graph depending of 2 variables initialized dynamically when the report is creating:

```
var data = {{json}};    // the raw data
var d = new Date({{date}}) // the date of the generation
```

Raw data is composed of 4 sections at the top level:

- A summary of the Data Coupling metrics:
 - **nbGlobalVariables** is the number of global variables found in the application.
 - **nbDefUses** is the number of Def/Use pairs found in the application.
 - **nbDefUsesCovered** Def/Use pairs found in the application that have been covered by at least one test.
 - **nbVariablesWithoutDefUse** is the number of global variables that have no Def/Use pairs in the application.
 - **variablesWithoutDefUse** is the list of global variables that have no Def/Use pairs in the application.

```
"nbGlobalVariables": 2,
"nbDefUses": 4,
"nbDefUsesCovered": 2,
"nbVariablesWithoutDefUse": 1,
"variablesWithoutDefUse": [
  "updated"
]
```

- The list of the modules, each of them has the following information:
 - **Name** is the short name of the C file,
 - **Fullname** is the name and path of the C file,
 - **uuid** is a unique identifier of the module,
 - **unknown** is set to true is the module is not part of the information you provided (there is only one unknown module that gathers all the call to functions that are not in the known modules),
 - **functions** is the list of the unique identifiers of functions of the module.

Modules are listed as hashmap with the uuid, as follows:

```
"modules": {
  "f5b5579edeaca82df478a6780c0c4c92": {
    "name": "USAGE.C",
    "fullname": "...",
    "uuid": "f5b5579edeaca82df478a6780c0c4c92",
    "unknown": false,
    "functions": [
      "ba9eb05ad703046fed306b4271b7ead7"
    ]
  },...
}
```

- The list of functions including following information:
 - **name** is the name of the C function,
 - **line** is the first line of the function in the module,
 - **id** is the number used in **.tsf** file to identify this function,
 - **stacksize** is the stack size computed during the execution if this option has been set (otherwise -1),
 - **uuid** is a unique identifier of the function,
 - **module** is a unique identifier of the module in which the function is declared,
 - **calls** is the list of the calls in this function. Each of them have the following information:
 - **calling_uuid** is the unique identifier of the calling function,
 - **called_uuid** is the unique identifier of the called function,
 - **line** is the line number of the call in the module,
 - **col** is the column number of the call in the module,
 - **same_module** is set to true id the called function is in the same module that the calling function.
 - **level** is a number that represent the level of the function in the call graph, starting to 0.
 - **calledby** is the list of unique identifiers of functions that call this one.
- Functions are listed as hashmap with the uuid, as following:

```

"functions": {
  "ba9eb05ad703046fed306b4271b7ead7": {
    "name": "write_usage",
    "line": 9,
    "id": 1,
    "stacksize": -1,
    "uuid": "ba9eb05ad703046fed306b4271b7ead7",
    "module": "f5b5579edeaca82df478a6780c0c4c92",
    "calls": [
      {
        "calling_uuid": "ba9eb05ad703046fed306b4271b7ead7",
        "called_uuid": "7b6cd643b5b44e1e0510f30f62729eba",
        "line": 10,
        "col": 2,
        "same_module": false
      }
    ],
    "level": 1,
    "calledby": [
      "3fb6b20659c9b70fc6d01ba797abae1f"
    ]
  },...
}

```

- The list of the control flows, each of them have the following information:
 - **stacksize** is the size computed for this control flow. This value is -1 if the tool was unable to compute.
 - **calls** is the list of successive calls that composed this Control Flow, each of them have the following information:
 - **calling_uuid** is the unique identifier of the calling function.
 - **called_uuid** is the unique identifier of the called function.
 - **line** is the line number of the call in the module.
 - **col** is the column number of the call in the module.

- **same_module** is set to true if the called function is in the same module that the calling function.
- **alternates** is a list of line and column if the function is called several times in this function
- **isRecursive** is set to true if a recursive call has been found in this control flow.
- **name** is the name of the test case.
- **missingFunctions** is the list of functions (name and unique identifier) in the control flow for which there is no stack size.

Control couplings are listed as an array, as follows:

```
"variables": [
  {
    "name": "currentDate",
    "line": 7,
    "moduleuuid": "e60218b872e86c7d154af4e306e9160a",
    "defs": [
      {
        "variablename": "currentDate",
        "linelocal": -1,
        "line": 33,
        "col": 2,
        "function": "main",
        "moduleuuid": "4306a1f82e1b1400a35d13ac6e2efce7",
        "isdef": true,
        "where": "bloc",
        "variabletype": "global",
        "covered": true
      },...
    ]
  },...
]
"uses": [
  {
    "variablename": "currentDate",
    "linelocal": -1,
    "line": 118,
    "col": 7,
    "function": "main",
    "moduleuuid": "4306a1f82e1b1400a35d13ac6e2efce7",
    "isdef": false,
    "where": "cond",
    "variabletype": "global",
    "covered": true
  },...
],...
"nbDefUses": 4,
"testcases": [
  {
    {
      "name": "fct_8/1",
      "requirements": [
        {
          "name": "REQ_PTU_123"
        }
      ]
    }
  },...
],...
]
```

Application Profiling

Application Profiling is gathering the main features that provide profiling information at the application level: the Worst Stack Size feature and the Worst performance (coming soon) feature.

Worst Stack Size

HCL OneTest™ Embedded introduces the Worst Stack Size feature to compute an estimation of the maximum stack size of the application under test.

Overview

To implement this feature, HCL OneTest™ Embedded uses two mixed technologies:

- Static analysis that computes the call graph of the application (Example: all the calls between functions are analyzed and computed as a graph),
- Dynamic analysis that provides the stack size of each functions when executing them.

This information is used to provide an estimation of the worst stack size. This estimation is accurate under the following conditions:

- All the functions of the application should have been executed at least once in order to have the stack size for each of them.
- Your application should not have recursive calls, because the number of loops in the recursive calls being unpredictable, it is impossible to compute the stack size.
- If your application used libraries (Example: call functions for which we have not the source code), you should provide an additional file that gives an estimation of the stack size for each of them. These estimations do not need to be precise, but should be an upper bound of the exact stack size.
- If your compiler optimizes the Stack Size, you might have different Stack Sizes for the same function. In this case, the Worst Stack Size is computed with the maximum value found in the different runs.
- If your application is multi-threaded, you can provide the list of entry points so that HCL OneTest™ Embedded can calculate the worst total stack size and compare it to the maximum memory stack available on your target to produce a pass/failed verdict.

For the Worst Stack, HCL OneTest™ Embedded provides a brand-new interactive HTML-based report. This report identifies if one or more of these conditions are not met.

How Worst Stack Size Works

When an application node is executed, the source code is instrumented by the Instrumentor (attolcc4 for C language) that produces a static file with the **.tsf** extension that contains information on the functions (this file is common with Control Coupling feature). The resulting source code is then compiled, linked and executed and the Control Coupling feature outputs a dynamic file with the extension **.tzf**.

These 2 types of files are used in input of the report generator that produces a report in HTML format (and optionally the raw data can be generated in a Json file). A template is provided for this generator. You can provide your own template to modify the report. An addition file could be provided to this report generator in order to specify the stack size of the external functions.

**Note:**

To visualize your report in Eclipse, if you are using the default browser option, be sure that JavaScript is enabled. Otherwise, you can choose another browser that is compatible with your version of JavaScript by changing it in Window > Preferences > General > Web Browser.

Set Worst Stack Size Options

Enable Worst Stack Size

- In HCL OneTest™ Embedded Studio, open the settings of the project and click **Configuration Properties > Build > Build options**.
- Then, in the right panel, click on the value field of the **Build options** line and click the ... button to open the Build options editor.
- Then, a dialog window shows you on the right the different tools that you can select during the build. Select **Application profiling** to enable the Worst Stack Size feature.

Multi-thread option

The Multi-thread option for the Worst Stack Size feature can be configured in the following menu of the settings:

- Click **Configuration Properties > Runtime analysis > Multi-Threads**.
- In the right pane, click the ... in the value field of the **Entry points** option to open the **Entry points** editor.
- In the Entry points editor, enter the list of entry points for each thread and click **OK**.

Stack Size options

Options for the Worst Stack Size feature can be updated in the following menu of the settings: **Configuration Properties > Runtime analysis > Application Profiling > Stack size**.

In the setting page, you can change the following options:

- **Trace file name (.tzf)**: set the name of the trace file dedicated to worst stack size. By default this name is the base name of the test with the extension **.tzf**.
- **Report Template**: change the template of the report generator. By default this template is **wssreport.template**.
- **External functions stack size**: this is a file that contains the stack size of the external functions (generally functions that are in libraries and used by your application). The format of this file should be in Json, with the extension **.tzfe**, as follows:

```
[
  {"name":"printf", "stacksize":4},
  {"name":"sin", "stacksize":4},
  {"name":"cos", "stacksize":4},
  {"name":"tan", "stacksize":4}
]
```

- **Maximum Size:** Enter the maximum stack size in bytes that the application should not exceed.
- **Security:** Enter a percentage of available Stack Size for security.

If you provide the maximum Stack Size allowed and a percentage of available Stack Size for security, the report displays the total Stack Size and verify if this size does not go over the available Stack Size.

Worst Stack Size Report

The default Worst Stack Size report is in HTML format. It is generated from a template named **wssreport.template** provided as a text file that you can modify to customize the report. It uses four online JavaScript libraries:

- Bootstrap,
- JQuery,
- Font Awesome,
- VisJS.

These libraries are not provided. You need an Internet connection when you open the report. Otherwise, you need to download the libraries (.css and .js files), copy them in the same folder as your report's, and modify the template file as follows:

Replace:

```
<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css"
  integrity="sha384-MCw98/SFnGE8fJT3GXwEOngsV7Zt27NXFoaoApmYm81iuXoPkFOJwJ8ERdknLPMO"
  crossorigin="anonymous">
<link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.5.0/css/all.css"
  integrity="sha384-B4dIYHKNBt8Bc12p+WXckhzcICo0wtJAoU8YZTY5qE0Id1GSseTk6S+L3BlXeVIU"
  crossorigin="anonymous">
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/vis/4.21.0/vis.min.css">
...
<script src="https://code.jquery.com/jquery-3.3.1.slim.min.js"
  integrity="sha384-q8i/X+965Dz00rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8abtTE1Pi6jizo"
  crossorigin="anonymous"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.3/umd/popper.min.js"
  integrity="sha384-ZMP7rVo3mIykV+2+9J3UJ46jBk0WLaUAdn689aCwoqbBJiSnjAK/l8WvCWPiPm49"
  crossorigin="anonymous"></script>
<script src="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/js/bootstrap.min.js"
  integrity="sha384-ChfqqxuZUCnJSK3+MXmPNIyE6ZbWh2IMqE241rYiqJxyMiZ6OW/JmZQ5stwEULTy"
  crossorigin="anonymous"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/vis/4.21.0/vis.js"></script>
```

with

```
<link rel="stylesheet" href="./bootstrap.min.css">
<link rel="stylesheet" href="./all.css">
<link rel="stylesheet" href="./vis.min.css">
...
<script src="./jquery-3.3.1.slim.min.js"></script>
<script src="./popper.min.js"></script>
<script src="./bootstrap.min.js"></script>
<script src="./vis.js"></script>
```

The Worst Stack Size report is made of three parts.

Summary

Worst Stack Size per Entry Point table

Summary

	main	DiffDays
Worst Stack Size per Entry Point	1616 bytes	
# Control Flows	165	2
# Control Flows without Stack Size	111	2
# Recursive Computed Control Flows	0	0
# Functions	37	3
# Functions without Stack Size	14	3

The Summary section displays a table with the Worst Stack Size calculated by the tools, given the information you provided in the build settings. This number is provided in bytes.

The Worst Stack Size is given per entry point and per thread if you have entered the list of entry point threads of your application in the Build Settings. You can set the list of entry point threads of your application in the Build Settings.

The table displays the following information:

- The number of control flows found in your application. A control flow is a set of successive calls starting from an entry point (each function that is never called by another one is considered as an entry point) to a function with no call or to an external function.
- The number of control flows for which we have no estimation of the stack size. This happens when one of the functions in this control flow has not been executed or if it is an external function for which no estimation of the stack size is provided.

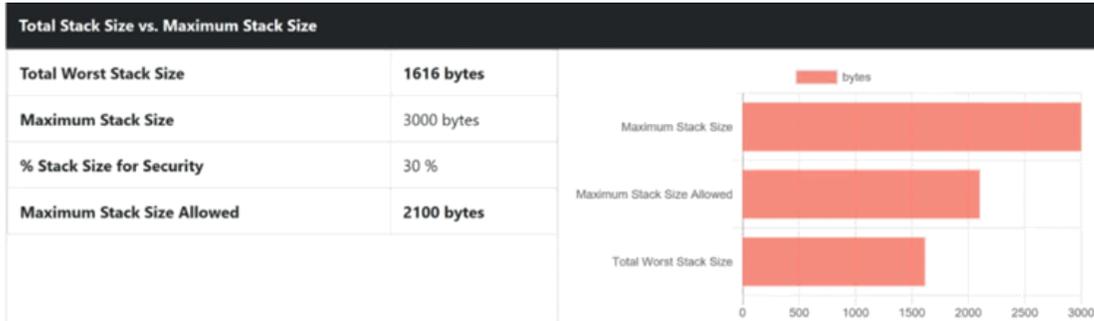
If this number is greater than 0, it is highlighted in red because there is no way to be sure that the worst stack size is really the worst regarding the missing information.

- The number of recursive control flows found in the application. If this number is greater than 0, it is highlighted in red because there is no way to be sure that the worst stack size is really the worst.
- The number of functions in your application.
- The number of functions without stack size estimation. These are the functions that have not been executed or the external functions for which we have not provided an estimation of the stack size. If this number is greater than 0, it is highlighted in red because we can't be sure that the worst stack size is really the worst.

The information is given for each entry thread.

If you don't provide the list of entry points in the build settings, the information is displayed only for the control flow and gives the Worst Stack Size.

Total Stack Size vs. Maximum Stack Size graph



If you provide in the Settings the list of entry points, optionally you can provide the maximum Stack Size allowed and a percentage of available Stack Size for security. In such case, the report displays the total Stack Size and verifies if this size does not go over the available Stack Size.

The **Maximum Stack Size** and **Percentage of available Stack Size for security** options can be set in the Build Settings.

In the report, you can compare the Stack Size or the sum of Stack Size with the maximum of Stack Size allowed and the percentage of available Stack Size for security if both options are provided in the settings.

In the toolbar that is under the graph, you can select the information to display or hide (all entry points, or for only one thread) and the number of control flows in the table. You can also show or hide the graph in the report from a button.

Details

The **Details** table lists by default the 10 first control flows with the biggest Stack Size and displays for each of them the following information:

- The control flow, for example, the successive functions starting from an entry point (any function that is never called by another one is considered as an entry point) to a function with no call, or to an external function. **Each function is identified by its name, its module (example: C file) between brackets, and by the line and column where this call to the next function calls appear in the code in parenthesis.**
- The estimation of the Stack Size. The information is blank if the tool has not been able to calculate the Stack Size for this control flow. In this case, the functions in the control flow that prevent us from computing the Stack Size are highlighted in red.

A drop down menu at the top of the table allows you to choose 10, 20, 30, 50, 100 or all the control flows to display.

Functions

The Functions table lists all the functions of your application, including external functions. The following information is provided for each function:

- The module name (i.e. the C file) where the function is saved.,
- The function name. This name is in red if there is no stack information for this function,
- The number of functions called in the current one.
- The Stack Size of the function in bytes.

Call Graph

The Call Graph part displays all the functions as an interactive call graph from left to right or from the top to the bottom, depending on the selector button position on the top of the call graph.

You can select a control flow in the table to highlight it in the call graph.

Customize the Worst Stack Size Report

The Worst Stack Size report is based on a template called **wssreport.template** that you can find in the folder **<install>/lib/reports**.

This template is made of 2 parts:

- The HTML part that is the common to all reports,
- A JavaScript part that sets the tables and call graph depending on 2 variables dynamically initialized when the report is created:

```
o var data = {{json}}; // the raw data

o var d = new Date({{date}}); // the date of the generation
```

Raw data

Raw data is made of four sections at the top level:

- A summary of the Worst Stack Size metrics:
 - **worstStackSize** is the worst stack size computed by the tools, depending on the information you provided. This number is provided in bytes.
 - **nbFlows** is the number of control flows found in your application. A control flow is a set of successive calls starting from an entry point (each function that is never called by another one is considered as an entry point) to a function without calls or to an external function.
 - **nbFlowsWithoutStack** is the number of control flows for which there is no estimation of the stack size. This happens when one of the functions in this control flow has not been executed, or if it is an external function for which we have not provided an estimation of the stack size.
 - **nbRecursiveFlows** is the number of recursive control flows found in the application.

- **nbFunctions** is the number of functions in your application.
- **nbFunctionsNoValue** is the number of functions without stack size estimation. These are the functions that have not been executed, or the external functions for which there is no estimation of the stack size provided.

```
"worstStackSize": 2139,
"nbFlows": 167,
"nbFlowsWithoutStack": 70,
"nbRecursiveFlows": 0,
"nbFunctions": 40,
"nbFunctionsNoValue": 10
```

The list of the modules, each of them has the following information:

- **name** is the short name of the C file,
- **fullname** is the name and path of the C file,
- **uuid** is a unique identifier of the module,
- **unknown** is set to true if the module is not part of the information you provided (there is only one unknown module that gathers all the function calls that are not in the known modules),
- **functions** is the list of the unique identifiers of functions of the module.

Modules are listed as Hashmap with the uuid, as following:

```
"modules": {
  "f5b5579edeaca82df478a6780c0c4c92": {
    "name": "USAGE.C",
    "fullname": "...",
    "uuid": "f5b5579edeaca82df478a6780c0c4c92",
    "unknown": false,
    "functions": [
      "ba9eb05ad703046fed306b4271b7ead7"
    ]
  }, ...
}
```

The list of functions, each of them have the following information:

- **name** is the name of the C function.
- **line** is the first line of the function in the module.
- **id** is the number used in **.tsf** file to identify this function.
- **stacksize** is the stack size computed during the execution if this option has been set (otherwise -1).
- **uuid** is a unique identifier of the function.
- **module** is a unique identifier of the module in which the function is declared.
- **calls** is the list of the calls in this function. Each of them have the following information:

- **calling_uuid** is the unique identifier of the calling function.
- **called_uuid** is the unique identifier of the called function.
- **line** is the line number of the call in the module.
- **col** is the column number of the call in the module.
- **same_module** is set to true if the called function is in the same module that the calling function.
- **level** is a number that represents the level of the function in the call graph, starting from 0.
- **calledby** is the list of unique identifiers of functions that call the function.

Functions are listed as hashmap with the uuid, as following:

```
"functions": {
  "ba9eb05ad703046fed306b4271b7ead7": {
    "name": "write_usage",
    "line": 9,
    "id": 1,
    "stacksize": -1,
    "uuid": "ba9eb05ad703046fed306b4271b7ead7",
    "module": "f5b5579edeaca82df478a6780c0c4c92",
    "calls": [
      {
        "calling_uuid": "ba9eb05ad703046fed306b4271b7ead7",
        "called_uuid": "7b6cd643b5b44e1e0510f30f62729eba",
        "line": 10,
        "col": 2,
        "same_module": false
      }
    ],
    "level": 1,
    "calledby": [
      "3fb6b20659c9b70fc6d01ba797abae1f"
    ]
  },...
```

The list of the Control Flows, each of them have the following information:

- **stacksize** is the size of the stack computed for the control flow. This value is -1 if the tool was unable to compute it.
- **calls** is the list of successive calls that composed this control flow, each of them is including the following information:
 - **calling_uuid** is the unique identifier of the calling function.
 - **called_uuid** is the unique identifier of the called function.
 - **line** is the line number of the call in the module.
 - **col** is the column number of the call in the module.
 - **same_module** is set to true id. The called function is in the same module that the calling function.
 - **alternates** is a list of line & column in case of the calling function is called several times in this function.

- **isRecursive** is set to true if a recursive call has been found in this control flow.
- **missingFunctions** is the list of functions (name and unique identifier) in the control flow for which we have not the stack size.

Control flows are listed as an array, as follows:

```
"controlflows": [
  {
    "isRecursive": false,
    "stacksize": 2139,
    "calls": [
      {
        "calling_uuid": "3fb6b20659c9b70fc6d01ba797abae1f",
        "called_uuid": "0dd641fbc509e237cb0600f451d27d59",
        "line": 97,
        "col": 19,
        "same_module": false,
        "alternates": [
          {
            "line": 100,
            "col": 19
          }
        ]
      },...
    ],
    "missingfunctions": [],
  },...
],...
```

Testing software components

The test features provided with HCL OneTest™ Embedded allow you to submit your application to a robust test campaign. Each feature uses a different approach to the software testing problem, from the use of test drivers stimulating the code under test, to source code instrumentation testing internal behavior from inside the running application.

- Component Testing for C and Ada performs black box or functional testing of software components independently of other units in the same system.
- Component Testing for C++ uses object-oriented techniques to address embedded software testing.
- System Testing for C is dedicated to testing message-based applications.

These test features each use a dedicated scripting language for writing specialized test cases. HCL OneTest™ Embedded test features can also be used together with any of the runtime analysis tools.

To learn about

See

Black-box or functional testing of C software components independently of other units in the same system.	Component Testing for C and Ada on page 447
Using object-oriented techniques to test your C++ code	Component Testing for C++ on page 511
Testing message-based applications written in C	System Testing for C on page 586

To use a component test feature:

Here is a rundown of the main steps to using the HCL OneTest™ Embedded test features:

1. Set up a new project in HCL OneTest™ Embedded. This can be done automatically with the [New Project Wizard on page 664](#).
2. Follow the [Activity Wizard on page 663](#) to add your application source files to the workspace.
3. Select the source files under test with the Test Generation Wizard to create a test node. The Wizard guides you through process of selecting the right test feature for your needs.
4. Develop the test cases by completing the automatically generated test scripts with the corresponding script language and native code.
5. Use the [Project Explorer on page 1021](#) to set up the test campaign and add any additional runtime analysis or test nodes.
6. [Run the test campaign on page 698](#) to builds and execute a test driver with the application under test.
7. [View and analyze the generated test reports on page 683](#).

Related Topics

[About Component Testing for C and Ada on page 447](#) | [About Component Testing for C++ on page 511](#) | [About System Testing for C on page 586](#) | [Using Runtime Analysis Features on page 306](#)

Component Testing for C overview

Component Testing for C

The Component Testing for C feature of HCL® OneTest™ Embedded provides a unique, fully automated, and proven solution for applications written in C, dramatically increasing test productivity.

Component Testing for C supports ANSI C89 and C99.

How Component Testing for C Works

When a test node is executed, the Test Script Compiler (**attolpreproC**) compiles both the test scripts and the source under test. This preprocessing creates a **.tdc** file. The resulting source code generates a test driver.

If any Runtime Analysis tools are associated with the test node, then the source code is also instrumented with the Instrumentor (**attolcc1**) tool.

The test driver, TDP, stubs and dependency files all make up the test harness.

The test harness interacts with the source code under test and produces test results. Test execution creates a **.rio** file.

The **.tdc** and **.rio** files are processed together the Component Testing Report Generator (**attolpostpro**). The output is the **.xrd** report file, which can be viewed and controlled in the HCL® OneTest™ Embedded GUI.

Of course, these steps are mostly transparent to the user when the test node is executed in the HCL® OneTest™ Embedded GUI.

To learn about	See
Writing test scripts for your software under test	Writing a Test Script on page 450
The types of source files under test	Integrated, Simulated and Additional Files on page 448
Configuration Settings for Component Testing test nodes	Component Testing for C Settings on page 999
Viewing Component Testing for C test results	Viewing Reports on page 507
Upgrading from a pre-2002 version of HCL® OneTest™ Embedded	Importing V2001 Component Testing Files on page 505

Related Topics

[Using Test Features on page 446](#) | [Activity Wizards on page 663](#) | [Manually Creating a Test or Application Node on page 680](#) | [About System Testing for C on page 586](#)

Integrated, simulated and additional files

Component Testing for C

When creating a Component Testing test node for C and Ada, the Component Testing wizard offers the following options for specifying dependencies of the source code under test:

- Integrated files
- Simulated files
- Additional files

Integrated Files

This option provides a list of source files whose components are *integrated* into the test program after linking.

The Component Testing wizard analyzes integrated files to extract any global variables that are visible from outside. For each global variable the Parser declares an external variable and creates a default test which is added to an environment named after the file in the **.ptu** test script.

By default, any symbols and types that could be exported from the source file under test are declared again in the test script.

Simulated Files

This option gives the Component Testing wizard a list of source files to simulate—or stub—upon execution of the test.

A stub is a dummy software component designed to replace a component that the code under test relies on, but cannot use for practicality or availability reasons. A stub can simulate the response of the stubbed component.

The Component Testing parser analyzes the simulated files to extract the global variables and functions that are visible from outside. For each file, a **DEFINE STUB** block, which contains the simulation of the file's external global variables and functions, is generated in the **.ptu** test script.

By default, no simulation instructions are generated.

Additional Files

Additional files are merely dependency files that are added to the Component Testing test node, but ignored by the source code parser. Additional files are compiled with the rest of the test node but are not instrumented.

For example, Microsoft Visual C resource files can be compiled inside a test node by specifying them as additional files.

You can toggle a source file from *under test* to *additional* by using the Properties Window dialog box.

Related Topics

[Component Testing Wizard on page 666](#)

Testing shared libraries

Component Testing for C

In order to test a shared library, you must create a test node containing the **.ptu** component test script that uses the library, and a reference link to the library.

After the execution of the test node, the runtime analysis and component test results are located in the application node.

To test a shared library:

1. Add the library to your project:
 - a. Right-click a group or project node and select **Add Child** and **Library** from the popup menu.
 - b. Enter the name of the Library node
 - c. Right-click the Library node and select **Add Child** and **Files** from the popup menu.
 - d. Select the source files of the shared library.
2. Run the Component Testing wizard as usual on the source file of your library. This creates a test node containing the test scripts and the source file.
3. Delete the source file from the test node.
4. Create a reference to the shared library in the test node:
 - a. Right-click the application or test node that will use the shared library and select **Add Child** and **Reference** from the popup menu.
 - b. Select the library node and click **OK**.
5. Build and execute the test node.

Example

An example demonstrating how to test shared libraries is provided in the **Shared Library** example project. See [Example projects on page 677](#) for more information.

Related Topics

[Using shared libraries on page 686](#) | [Profiling shared libraries on page 307](#)

Writing a Test Script

Component Testing for C

When you first create Component Testing for C test node with the Component Testing Wizard, HCL OneTest™ Embedded produces a **.ptu** test script template based on the source under test.

To write the test script, you can use the Text Editor provided with HCL OneTest™ Embedded.

Component Testing for C uses the C Test Script Language. Full reference information for this language is provided in the Reference section.

To learn about

See

Basic .ptu test script instructions	Test Script Structure on page 451
Initializing and testing variable values	Testing Variables on page 454
Simulating stub functions	Stub Simulation on page 556
Catching exceptions	Unexpected Exceptions on page 580
Other specific C testing notions	Advanced C Testing on page 500

Related Topics

[Structure Statements on page 533](#) | [About the Text Editor on page 693](#)

Test Script Structure

Component Testing for C

The C Test Script Language allows you to structure tests to:

- Describe several test cases in a single test script,
- Select a subset of test cases according to different Target Deployment Port criteria.

Test script filenames must contain only plain alphanumeric characters.

A typical Component Testing **.ptu** test script looks like this:

```
HEADER add, 1, 1
```

```
<variable declarations for the test script>
```

```
BEGIN
```

```
SERVICE add
```

```
<local variable declarations for the service>
```

```
TEST 1
```

```
FAMILY nominal
```

```
ELEMENT
```

```
VAR variable1, INIT=0, EV=0
```

```
VAR variable2, INIT=0, EV=0  
  
#<call to the procedure under test>  
  
END ELEMENT  
  
END TEST  
  
END SERVICE
```

All instructions in a test script have the following characteristics:

- All statements begin with a keyword.
- Statements are not case sensitive (except when C expressions are used).
- Statements start at the beginning of a line and end at the end of a line. You can, however, write an instruction over several lines using the ampersand (&) continuation character at the beginning of additional lines. In this case, the ampersand must be the very first character on that line; no spaces or tabs should precede it.
- Statements must be shorter than 2048 characters, although this limit may be lower on some platforms.

Structure statements

The following statements allow you to describe the structure of a test.

- **HEADER:** For documentation purposes, specifies the name and version number of the module being tested, as well as the version number of the tested source file. This information is displayed in the test report.
- **BEGIN:** Marks the beginning of the generation of the actual test program.
- **SERVICE:** Contains the test cases related to a given service. A service usually refers to a procedure or function. Each service has a unique name (in this case add). A **SERVICE** block terminates with the instruction **END SERVICE**.
- **TEST:** Each test case has a number or identifier that is unique within the block **SERVICE**. The test case is terminated by the instruction **END TEST**.
- **FAMILY:** Qualifies the test case to which it is attached. The qualification is free (in this case **nominal**). A list of qualifications can be specified (for example: **family, nominal, structure**) in the Tester Configuration dialog box.
- **ELEMENT:** Describes a test phase in the current test case. The phase is terminated by the instruction **END ELEMENT**. The different phases of the same test case cannot be dissociated after the tests are run, unlike the test cases introduced by the instruction **NEXT_TEST**. However, the test phases introduced by the instruction **ELEMENT** are included in the loops created by the instruction **LOOP**.

The three-level structure of the test scripts has been deliberately kept simple. This structure allows:

- A clear and structured presentation of the test script and report
- Tests to be run selectively on the basis of the service name, the test number, or the test family.

In the test script, the testers can add an optional **REQUIREMENT** statement in order to link the tests to one or several requirements of the application under test.

The **REQUIREMENT** instruction appears within **TEST** blocks, where it defines the requirements for this test or within **SERVICE** blocks where it defines the requirements for the tests including in this service or before the first **SERVICE** block where it defines the requirements for all the tests in the file.

Euclidian divisions in C

All Euclidian divisions performed by the Test Script Compiler round to the inferior integer.

Therefore, writing **-a/b** returns a different result than **-(a/b)**, as in the following examples:

- $-(9/2)$ returns -4
- $-9/2$ returns -5

Related Topics

[Component Testing Tester Configuration on page 701](#)

Using native C statements

Component Testing for C

In some cases, it can be necessary to include portions of native C code inside a **.ptu** test script. You can use the **#**, **@**, and **!** prefixes to do this.

Analyzed native code -

When lines are prefixed with the **#** character, the Test Script Compiler **analyzes** the line and then **copies** the line into the generated code. You can use the **#** prefix to declare test script variables and to include the files that declare the functions under test.

Variable declarations must be placed outside of C test script blocks preferably at the beginning of scenarios and procedures.

Ignored native code - @

When lines are prefixed with the **@** character, the Test Script Compiler only **copies** the line into the test harness and does **not analyze** the line. You can use the **@** prefix to copy instructions into the test harness, when the test script compiler would not understand these instructions. Assembly instructions are examples of these instructions.

Parsed native code - !

When lines are prefixed with the ! character, the Test Script Compiler **analyzes** the lines, but does **not copy the lines** into the test harness. You can use the ! prefix to declare variables and types that are built into the compiler.

Automatically updating a .ptu test script

Component Testing for C

Changes that are made during the development process can sometimes impact the test script, for example when new functions are added after the test script was generated.

You can update a **.ptu** test script to automatically add new elements to SERVICES and INCLUDE blocks to reflect changes that were made to the source code. An update does not remove or modify any existing statements.

For the update to work, you must not edit any generated comment lines that start with **%c** or **%d** in the test script. The update command only works with .ptu test scripts that were generated by Test RealTime 7.0 or later, which contain these **%c** and **%d** comment lines.

To update a .ptu test script

1. In the **Project Explorer**, right-click the **.ptu** test script that you want to update.
2. From the pop-up menu, select **Update**.
3. Edit the **.ptu** test script.

Related Topics

[Writing a Test Script on page 450](#) | [Stub Simulation on page 475](#)

Testing variables

Component Testing for C

One of the main features of Component Testing for C is its ability to compare initial values, expected values and actual values of variables during test execution. In the C Test Script Language, this is done with the **VAR** statement.

The **VAR** statement specifies both the test start-up procedure and the post-execution test for simple variables. This instruction uses three parameters:

- **Name of the variable under test:** this can be a simple variable, an array element, or a field of a record. It is also possible to test an entire array, part of an array or all the fields of a record.
- **Initial value of the variable:** identified by the keyword **INIT**.
- **Expected value of the variable after the procedure has been executed:** identified by the keyword **EV**.

Declare variables under test with the **VAR** statement, followed by the declaration keywords:

- **INIT** = for an assignment
- **INIT ==** for no initialization
- **EV** = for a simple test.

It does not matter where the **VAR** instructions are located with respect to the test procedure call since the C code generator separates **VAR** instructions into two parts :

- The variable test is initialized with the **ELEMENT** instruction
- The actual test against the expected value is done with the **END ELEMENT** instruction

Many other forms are available that enable you to create more complex test scenarios.

Using C Expressions

Component Testing for C allows you to define initial and expected values with standard C expressions.

All literal values, variable types, functions and most operators available in the C language are accepted by Component Testing for C.

Example

The following example demonstrates typical use of the **VAR** statement

```
HEADER add, 1, 1
```

```
#with add;
```

```
BEGIN
```

```
SERVICE add
```

```
# a, b, c : integer;
```

```
TEST 1
```

```
FAMILY nominal
```

```
ELEMENT
```

```
VAR a, init = 1, ev = init
```

```
VAR b, init = 3, ev = init
```

```
VAR c, init = 0, ev = 4
```

```
#c := add(a,b);
```

```
END ELEMENT
```

END TEST

END SERVICE

Related Topics

[Testing intervals on page 456](#) | [Testing tolerances on page 456](#) | [Initializing without testing on page 458](#) | [Testing expressions on page 458](#) | [Declaring parameters on page 459](#) | [Testing arrays on page 460](#) | [Testing structured variables on page 470](#)

Testing intervals

Component Testing for C

You can test an expected value within a given interval by replacing **EV** with the keywords **MIN** and **MAX**.

You can also use this form on alphanumeric variables, where character strings are considered in alphabetical order ("**A**"<"**B**"<"**C**").

Example

The following example demonstrates how to test a value within an interval:

```
TEST 4
```

```
FAMILY nominal
```

```
ELEMENT
```

```
VAR a, INIT in {1,2,3}, EV = INIT
```

```
VAR b, INIT = 3, EV = INIT
```

```
VAR c, INIT = 0, MIN = 4, MAX = 6
```

```
#c = add(a,b);
```

```
END ELEMENT
```

```
END TEST
```

Related Topics

[Testing variables on page 454](#) | [Testing tolerances on page 456](#) | [Initializing without testing on page 458](#) | [Testing expressions on page 458](#) | [Declaring parameters on page 459](#) | [Testing arrays on page 460](#) | [Testing structured variables on page 470](#)

Testing tolerances

Component Testing for C

You can associate a tolerance with an expected value for numerical variables. To do this, use the keyword **DELTA** with the expected value **EV**.

This tolerance can either be an absolute value (the default option) or relative (in the form of a percentage *<value>%*).

You can rewrite the test from the previous example as follows:

```
TEST 5
```

```
FAMILY nominal
```

```
ELEMENT
```

```
VAR a, INIT in {1,2,3}, EV = INIT
```

```
VAR b, INIT = 3, EV = INIT
```

```
VAR c, INIT = 0, EV = 5, DELTA = 1
```

```
#c = add(a,b);
```

```
END ELEMENT
```

```
END TEST
```

or

```
TEST 6
```

```
FAMILY nominal
```

```
ELEMENT
```

```
VAR a, INIT in {1,2,3}, EV = INIT
```

```
VAR b, INIT = 3, EV = INIT
```

```
VAR c, INIT = 0, EV = 5, DELTA = 20%
```

```
#c = add(a,b);
```

```
END ELEMENT
```

```
END TEST
```

Related Topics

[Testing variables on page 454](#) | [Testing intervals on page 456](#) | [Initializing without testing on page 458](#) | [Testing expressions on page 458](#) | [Declaring parameters on page 459](#) | [Testing arrays on page 460](#) | [Testing structured variables on page 470](#)

Initializing without testing

Component Testing for C

It is sometimes difficult to predict the expected result for a variable; such as if a variable holds the current date or time. In this case, you can avoid specifying an expected output.

Example

The following script show an example of an omitted test:

```
TEST 7
```

```
FAMILY nominal
```

```
ELEMENT
```

```
VAR a, init in {1,2,3}, ev = init
```

```
VAR b, init = 3, ev = init
```

```
VAR c, init = 0, ev ==
```

```
#c = add(a,b);
```

```
END ELEMENT
```

```
END TEST
```

Related Topics

[Testing variables on page 454](#) | [Testing intervals on page 456](#) | [Testing tolerances on page 456](#) | [Testing expressions on page 458](#) | [Declaring parameters on page 459](#) | [Testing arrays on page 460](#) | [Testing structured variables on page 470](#)

Testing expressions

Component Testing for C

To test the return value of an expression, rather than declaring a local variable to memorize the value under test, you can directly test the return value with the **VAR** instruction.

In some cases, you must leave out the initialization part of the instruction.

Example

The following example places the call of the **add** function in a **VAR** statement:

```
TEST 12
```

```

FAMILY nominal
ELEMENT
VAR a, init in {1,2,3}, ev = init
VAR b, init(a) with {3,2,1}, ev = init
VAR add(a,b), ev = 4
END ELEMENT
END TEST

```

In this example, you no longer need the variable `c`.

All syntax examples of expected values are still applicable, even in this particular case.

Related Topics

[Testing variables on page 454](#) | [Testing intervals on page 456](#) | [Testing tolerances on page 456](#) | [Initializing without testing on page 458](#) | [Declaring parameters on page 459](#) | [Testing arrays on page 460](#) | [Testing structured variables on page 470](#)

Declaring parameters

Component Testing for C

ELEMENT blocks contain specific instructions that describe the test start-up procedures and the post-execution tests.

The hash character (`#`) at the beginning of a line indicates a native language statement written in C.

This declaration is introduced after the **SERVICE** instruction because it is local to the **SERVICE** block; it is invalid outside this block.

It is only necessary to declare parameters of the procedure under test. Global variables are already present in the module under test or in any integrated modules, and do not need to be declared locally.

Related Topics

[Testing variables on page 454](#) | [Testing intervals on page 456](#) | [Testing tolerances on page 456](#) | [Initializing without testing on page 458](#) | [Testing expressions on page 458](#) | [Testing arrays on page 460](#) | [Testing structured variables on page 470](#)

Initial and Expected Value settings

Component Testing for C

The Initial and Expected Value settings are part of the [Component Testing Settings for C on page 999](#) dialog box and describes how values assigned to each variable are displayed in the Component Testing report. Component Testing allows three possible evaluation strategy settings.

Variable Only

This evaluation strategy setting generates both the initial and expected values of each variable evaluated by the program during execution.

This is possible only for variables whose expression of initial or expected value is not reducible by the Test Script Compiler. For arrays and structures in which one of the members is an array, this evaluation is not given for the initial values. For the expected values, however, it is given only for *Failed* items.

Value Only

With this setting, the test report displays for each variable both the initial value and the expected value defined in the test script.

Combined Evaluation

The combined evaluation setting combines both settings. The test report thus displays the initial value, the expected value defined in the test script, and the value found during execution if that value differs from the expected value.

Related Topics

[Component Testing Settings for C on page 999](#) | [Understanding Component Testing Reports on page 508](#) | [Array and Structure Display on page 510](#)

Arrays

Testing Arrays

Component Testing for C

With Component Testing for C, you can test arrays in quite the same way as you test variables. In the C Test Script Language, this is done with the **ARRAY** statement.

The **ARRAY** statement specifies both the test start-up procedure and the post-execution test for simple variables. This instruction uses three parameters:

- **Name of the variable under test:** species the name of the array in any of the following ways:
 - To test one array element, conform to the C syntax: **histo[0]**.
 - To test the entire array without specifying its bounds, the size of the array is deduced by analyzing its declaration. This can only be done for well-defined arrays.
 - To test a part of the array, specify the lower and upper bounds within which the test will be run, separated with two periods (..), as in: histo[1..SIZE_HISTO]
- **Initial value of the array:** identified by the keyword **INIT**.
- **Expected value of the array after the procedure has been executed:** identified by the keyword **EV**.

Declare variables under test with the **ARRAY** statement, followed by the declaration keywords:

- **INIT =** for an assignment
- **INIT ==** for no initialization
- **EV =** for a simple test.

It does not matter where the **ARRAY** instructions are located with respect to the test procedure call since the C code generator separates **ARRAY** instructions into two parts :

- The array test is initialized with the **ELEMENT** instruction
- The actual test against the expected value is done with the **END ELEMENT** instruction

To initialize and test an array, specify the same value for all the array elements.

You can use the same expressions for initial and expected values as those used for simple variables (literal values, constants, variables, functions, and C operators).

Use the **ARRAY** instruction to run simple tests on all or only some of the elements in an array.

Testing Arrays with C Expressions

To initialize and test an array, specify the same value for all the array elements. The following two examples illustrate this simple form.

```
ARRAY image, INIT = 0, EV = INIT
```

```
ARRAY histo[1..SIZE_HISTO-1], INIT = 0, EV = 0
```

You can use the same expressions for initial and expected values as those used for simple variables (literal values, constants, variables, functions, and C operators).

Example

The following example highlights the ARRAY instruction syntax for C:

```
HEADER histo, 1, 1

##include "histo.h"

BEGIN

SERVICE COMPUTE_HISTO

#int x1, x2, y1, y2;

#int status;

#T_HISTO histo;

TEST 1

FAMILY nominal

ELEMENT

VAR x1, init = 0, ev = init

VAR x2, init = SIZE_IMAGE-1, ev = init

VAR y1, init = 0, ev = init

VAR y2, init = SIZE_IMAGE-1, ev = init

ARRAY image, init = 0, ev = init

VAR histo[0], init = 0, ev = SIZE_IMAGE*SIZE_IMAGE

ARRAY histo[1..SIZE_HISTO-1], init = 0, ev = 0

VAR status, init = 0, ev = 0

#status = compute_histo(x1, y1, x2, y2, histo);

END ELEMENT

END TEST

END SERVICE
```

Related Topics

[Testing arrays with pseudo-variables on page 463](#) | [Testing large arrays on page 464](#) | [Testing arrays with lists on page 465](#) | [Testing character arrays on page 466](#) | [Testing arrays with other arrays on page 467](#) | [Testing an array of union elements on page 468](#)

Testing arrays with pseudo-variables

Component Testing for C

Another form of initialization consists of using one or more pseudo-variables, as the following example illustrates:

```
TEST 3

FAMILY nominal

ELEMENT

VAR x1, init = 0, ev = init

VAR x2, init = SIZE_IMAGE-1, ev = init

VAR y1, init = 0, ev = init

VAR y2, init = SIZE_IMAGE-1, ev = init

ARRAY image, init=(int)(100*(1+sin((float)(I1+I2))))), ev = init

ARRAY histo[0..200], init = 0, ev ==

ARRAY histo[201..SIZE_HISTO-1], init = 0, ev = 0

VAR status, init ==, ev = 0

#status = compute_histo(x1, y1, x2, y2, histo);

END ELEMENT

END TEST
```

I1 and I2 are two pseudo-variables which take as their value the current values of the array indices (for **image**, from **0** to **199** for **I1** and **I2**). You can use these pseudo-variables like a standard variable in any C expression.

This lets you create more complicated test scripts in the case of very large arrays, where the use of enumerated expressions is limited.

For multidimensional arrays, you can combine these different types of initialization and test expressions, as the following example shows:

```
ARRAY image, init = {0 => I2, 1 => { 0 => 100, others => 0 },
& others => (I1 + I2) % 255 }
```

Related Topics

[Testing arrays on page 460](#) | [Testing large arrays on page 464](#) | [Testing arrays with lists on page 465](#) | [Testing character arrays on page 466](#) | [Testing arrays with other arrays on page 467](#) | [Testing an array of union elements on page 468](#)

Testing large arrays

Component Testing for C

The maximum number of array elements that can be processed is 100. If you need to test an array that contains more than 100 elements, then you must split the initialization of the array over two or more initializations, as shown in the following example.

Example

The following initialization produces a **Too many INIT or VA values** error:

```
#int a[200];

ARRAY a, init=
{1,2,3,4,5,6,7,8,9,10,1,2,3,4,5,6,7,8,9,20,1,2,3,4,5,6,7,8,9,30,1,2,3,4,
5,6,7,8,9,40,1,2,3,4,5,6,7,8,9,50,1,2,3,4,5,6,7,8,9,60,1,2,3,4,5,6,7,8,9,
70,1,2,3,4,5,6,7,8,9,80,1,2,3,4,5,6,7,8,9,90,1,2,3,4,5,6,7,8,9,100,1,2,3,
4,5,6,7,8,9,110,1,2,3,4,5,6,7,8,9,120,1,2,3,4,5,6,7,8,9,130,1,2,3,4,5,6,
7,8,9,140,1,2,3,4,5,6,7,8,9,150,1,2,3,4,5,6,7,8,9,160,1,2,3,4,5,6,7,8,9,
170,1,2,3,4,5,6,7,8,9,180,1,2,3,4,5,6,7,8,9,190,1,2,3,4,5,6,7,8,9,200}

, ev=init
```

Instead, use the following expression:

```
#int a[200];

ARRAY z [0..99],

init={1,2,3,4,5,6,7,8,9,10,1,2,3,4,5,6,7,8,9,20,1,2,3,4,5,6,7,8,9,30,1,2,
3,4,5,6,7,8,9,40,1,2,3,4,5,6,7,8,9,50,1,2,3,4,5,6,7,8,9,60,1,2,3,4,5,6,
7,8,9,70,1,2,3,4,5,6,7,8,9,80,1,2,3,4,5,6,7,8,9,90,1,2,3,4,5,6,7,8,9,100}

, ev=init
```

```
ARRAY z [100..199],
```

```
init={1,2,3,4,5,6,7,8,9,110,1,2,3,4,5,6,7,8,9,120,1,2,3,4,5,6,7,8,9,130,
1,2,3,4,5,6,7,8,9,140,1,2,3,4,5,6,7,8,9,150,1,2,3,4,5,6,7,8,9,160,1,2,3,
4,5,6,7,8,9,170,1,2,3,4,5,6,7,8,9,180,1,2,3,4,5,6,7,8,9,190,1,2,3,4,5,6,
7,8,9,200}
, ev=init
```

Related Topics

[Testing arrays on page 460](#) | [Testing arrays with pseudo-variables on page 463](#) | [Testing arrays with lists on page 465](#) | [Testing character arrays on page 466](#) | [Testing arrays with other arrays on page 467](#) | [Testing an array of union elements on page 468](#)

Testing arrays with lists

Component Testing for C

While an expression initializes all the **ARRAY** elements in the same way, you can also initialize each element using an enumerated list of expressions between brackets (**{}**). In this case, you must specify a value for each array element.

Furthermore, you can precede every element in this list of initial or expected values with the array index of the element concerned followed by the characters "=>". The following example illustrates this form:

```
ARRAY histo[0..3], init = {0 => 0, 1 => 10, 2 => 100, 3 => 10} ...
```

This form of writing the **ARRAY** instruction has the following advantages:

- It improves the readability of the list.
- It allows you to mix values without worrying about the order.

You can also use this form together with the simple form if you follow this rule: once one element has been defined with its array index, you must do the same with all the following elements.

If several elements in an array are to take the same value, specify the range of elements taking this value as follows:

```
ARRAY histo[0..3], init = { 0 .. 2 => 10, 3 => 10 } ...
```

You can also specify a value for all the as yet undefined elements by using the keyword **others**, as the following example illustrates:

```
TEST 2
```

```
FAMILY nominal
```

```
ELEMENT
```

```
VAR x1, init = 0, ev = init
VAR x2, init = SIZE_IMAGE-1, ev = init
VAR y1, init = 0, ev = init
VAR y2, init = SIZE_IMAGE-1, ev = init
ARRAY image, init = {others=>{others=>100}}, ev = init
ARRAY histo, init = 0,
& ev = {100=>SIZE_IMAGE*SIZE_IMAGE, others=>0}
VAR status, init ==, ev = 0
#status = compute_histo(x1, y1, x2, y2, histo);
END ELEMENT
END TEST
```

Note The form **{others => <expression> }** is equivalent to initializing and testing all array elements with the same expression.

You can also initialize and test multidimensional arrays with a list of expressions, as follows. In this case, the previously mentioned rules apply to each dimension.

```
ARRAY image, init = {0, 1=>4, others=>{1, 2, others=>100}} ...
```

Note Some C compilers allow you to omit levels of brackets when initializing a multidimensional array. The Unit Testing Scripting Language does not accept this non-standard extension to the language.

Related Topics

[Testing arrays on page 460](#) | [Testing arrays with pseudo-variables on page 463](#) | [Testing large arrays on page 464](#) | [Testing character arrays on page 466](#) | [Testing arrays with other arrays on page 467](#) | [Testing an array of union elements on page 468](#) | [VAR, ARRAY and STR on page 741](#)

Testing character arrays

Component Testing for C

Character arrays are a special case. Variables of this type are processed as character strings delimited by quotes.

You therefore need to initialize and test character arrays using character strings, as the following list example illustrates.

If you want to test character arrays like other arrays, you must use a format modification declaration (**FORMAT** instruction) to change them to arrays of integers.

Example

The following list example illustrates this type of modification:

```
TEST 2
```

```
FAMILY nominal
```

```
FORMAT str[] = int
```

```
ELEMENT
```

```
VAR l, pointer, init = NIL, ev = NONIL
```

```
VAR s, init = "myfoo", ev = init
```

```
VAR str[0..5], init == , ev = {'m','y','f','o','o',0}
```

```
#l = strcpy(str,s);
```

```
END ELEMENT
```

```
END TEST 2
```

Related Topics

[Testing arrays on page 460](#) | [Testing arrays with pseudo-variables on page 463](#) | [Testing large arrays on page 464](#) | [Testing arrays with lists on page 465](#) | [Testing arrays with other arrays on page 467](#) | [Testing an array of union elements on page 468](#)

Testing arrays with other arrays

Component Testing for C

The following example illustrates a form of initialization that consists of initializing or comparing an array with another array that has the same declaration:

```
TEST 4
```

```
FAMILY nominal
```

```
ELEMENT
```

```
VAR x1, init = 0, ev = init
```

```
VAR x2, init = SIZE_IMAGE-1, ev = init
```

```
VAR y1, init = 0, ev = init
VAR y2, init = SIZE_IMAGE-1, ev = init
ARRAY image, init = extern_image, ev = init
ARRAY histo, init = 0, ev ==
VAR status, init ==, ev = 0
#read_image(extern_image,"image.bmp");
#status = compute_histo(x1, y1, x2, y2, histo);
END ELEMENT
END TEST
```

Read_image and **extern_image** are two arrays that have been declared in the same way. Every element from the **extern_image** array is assigned to the corresponding **read_image** array element.

You can use this form of initialization and testing with one or more array dimensions.

Related Topics

[Testing arrays on page 460](#) | [Testing arrays with pseudo-variables on page 463](#) | [Testing large arrays on page 464](#) | [Testing arrays with lists on page 465](#) | [Testing character arrays on page 466](#) | [Testing an array of union elements on page 468](#)

Testing arrays of union elements

Component Testing for C

When testing an array of unions, detail your tests for each member of the array, using **VAR** lines in the **ELEMENT** block.

Example

Considering the following variables:

```
#typedef struct {
# int test1;
# int test2;
# int test3;
# int test4;
```

```

# int test5;

# int test6;

# } Test;

typedef struct {

# int champ1;

# int champ2;

# int champ3;

# } Champ;

typedef struct {

# int toto1;

# int toto2;

# } Toto;

typedef union {

# Test A;

# Champ B;

# Toto C;

# } T_union;

extern T_union Tableau[4];

```

The test must be written element per element:

TEST 1

FAMILY nominal

ELEMENT

```

VAR Tableau[0], init = {A => { test1 => 0, test2 => 0, test3 => 0, test4 => 0,
& test5 => 0, test6 => 0} }, ev = init

```

```

VAR Tableau[1], init = {B => { champ1 => 0, champ2 => 0, champ3 => 0} }, ev = init

```

```

VAR Tableau[2], init = {B => { champ1 => 0, champ2 => 0, champ3 => 0} }, ev = init

```

```
VAR Tableau[3], init = {B => { champ1 => 0, champ2 => 0, champ3 => 0}} , ev = init
```

```
#ret_fct;
```

```
END ELEMENT
```

```
END TEST -- TEST 1
```

Related Topics

[Testing arrays on page 460](#) | [Testing arrays with pseudo-variables on page 463](#) | [Testing large arrays on page 464](#) | [Testing arrays with lists on page 465](#) | [Testing character arrays on page 466](#) | [Testing arrays with other arrays on page 467](#)

Structured Variables

Testing structured variables

Component Testing for C

To test all the fields of a structured variable, use a single instruction (**STR**) to define their initializations and expected values:

```
TEST 2
```

```
FAMILY nominal
```

```
ELEMENT
```

```
VAR l, init = NIL, ev = NONIL
```

```
STR *l, init == , ev = {"myfoo",NIL,NIL}
```

```
VAR s, init = "myfoo", ev = init
```

```
#l = push(l,s);
```

```
END ELEMENT
```

```
END TEST
```

You can only initialize and test structured variables with the following forms:

- INIT =
- INIT ==
- EV =
- EV ==

If a field of a structured variable needs to be initialized or tested in a different way, you can omit its initial and expected values from the global test of the structured variable, and run a separate test on this field.

The following example illustrates this:

```
TEST 4
```

```
FAMILY nominal
```

```
ELEMENT
```

```
VAR l, init = NIL, ev = NONIL
```

```
VAR *l, init == , ev = {NIL,NIL}
```

```
VAR s, init in {"foo","bar"}, ev = init
```

```
VAR l->str, init ==, ev(s) in {"foo","bar"}
```

```
#l = push(l,s);
```

```
END ELEMENT
```

```
END TEST
```

Using field names, write this as follows:

```
VAR *l, init ==, ev = {next=>NIL,prev=>NIL}
```

Related Topics

[Testing variables on page 454](#) | [Testing structured variables with C expressions on page 471](#) | [Testing structured variables with other structured variables on page 472](#) | [C Unions on page 474](#) | [Omitting a Field's Initial and Test Values on page 473](#)

Testing structured variables with C expressions

Component Testing for C

To initialize and test a structured variable or record, initialize or test all the fields using a list of native language expressions (one per field). The following example (taken from list.ptu) illustrates this form:

```
STR *l, init == , ev = {"myfoo",NIL,NIL}
```

Each element in the list must correspond to the structured variable field as it was declared.

Every expression in the list must obey the rules described so far, according to the type of field being initialized and tested:

- An expression for simple fields or arrays of simple variables initialized using an expression
- A list of expressions for array fields initialized using an enumerated list
- A list of expressions for structured fields

Using Field Names in Native Expressions

You can specify field names in native expressions by following the field name of the structure with the characters "=>", as follows:

```
TEST 3
```

```
FAMILY nominal
```

```
ELEMENT
```

```
VAR l, init = NIL, ev = NONIL
```

```
VAR *l, init == , ev = {str=>"myfoo",next=>NIL,prev=>NIL}
```

```
VAR s, init = "myfoo", ev = init
```

```
#l = push(l,s);
```

```
END ELEMENT
```

```
END TEST
```

If you use this form, you do not have to respect the order of expressions in the list.

You can also use the position of the fields in the structure or record instead of the field names, on the basis that the field numbers begin at 1:

```
VAR *l, init ==, ev = {3 => NIL, 2 => NIL, 1 => "myfoo"}
```

As with arrays, you can also use a range for field positions, as follows:

```
VAR *l, init ==, ev = {1 => "myfoo", 2..3 => NIL}
```

Related Topics

[Testing variables on page 454](#) | [Testing structured variables on page 470](#) | [Testing structured variables with other structured variables on page 472](#) | [C Unions on page 474](#) | [Omitting a Field's Initial and Test Values on page 473](#)

Testing structured variables with other structured variables

Component Testing for C

You can initialize and test a structured variable or record using another structured variable or record of the same type. The following example illustrates this form:

```
STR *l, init == , ev = l1
```

Each field of the structured variable will be initialized or tested using the associated fields of the variable used for initialization or testing.

Related Topics

[Testing Variables on page 454](#) | [Testing a Structured Variable on page 470](#) | [Testing a Structured Variable with C Expressions on page 471](#) | [Testing a Structured Variable with Another Structured Variable on page 472](#) | [C Unions on page 474](#) | [Omitting a Field's Initial and Test Values on page 473](#)

Omitting a Field's Initial and Test Values

Component Testing for C

You can only initialize and test structured variables with the following forms:

- INIT =
- INIT ==
- EV =
- EV ==

If a field of a structured variable needs to be initialized or tested in a different way, you can omit its initial and expected values from the global test of the structured variable, and run a separate test on this field.

The following example illustrates this:

```
TEST 4
```

```
FAMILY nominal
```

```
ELEMENT
```

```
VAR l, init = NIL, ev = NONIL
```

```
VAR *l, init == , ev = {NIL,NIL}
```

```
VAR s, init in {"foo","bar"}, ev = init
```

```
VAR l->str, init ==, ev(s) in {"foo","bar"}
```

```
#l = push(l,s);
```

END ELEMENT

END TEST

Using field names, write this as follows:

```
VAR *l, init ==, ev = {next=>NIL,prev=>NIL}
```

Related Topics

[Testing Variables on page 454](#) | [Testing a Structured Variable on page 470](#) | [Testing a Structured Variable with C Expressions on page 471](#) | [Testing a Structured Variable with Another Structured Variable on page 472](#) | [C Unions on page 474](#)

C Unions

Component Testing for C

If the structured variable involves a C union (defined using the **union** instruction) rather than a structure (defined using the **struct** instruction), you need to specify which field in the union is tested. The initial and test value only relates to one of the fields in the union, whereas, for a structure, it relates to all the fields.

The **list.c** example demonstrates this if you modify the structure of the list, such that the value stored at each node is an integer, a floating-point number, or a character string:

list1.h:

```
enum node_type { INTEGER, REAL, STRING };

typedef struct t_list {

enum node_type type;

union {

long integer_value;

double real_value;

char * string_value;

} value;

struct t_list * next;

struct t_list * prev;

} T_LIST, * PT_LIST;
```

In this case, the test becomes:

```

HEADER list1, 1, 1

##include "list1.h"

BEGIN

SERVICE push1

#PT_LIST I;

#enum node_type t;

#char s[10];

TEST 1

FAMILY nominal

ELEMENT

VAR I, init = NIL, ev = NONIL

VAR t, init = my_string, ev = init

VAR *I, init == ,

& ev = {STRING,{string_value=>"myfoo"}, NIL,NIL}

VAR s, init = "myfoo", ev = init

#I = push1(I, t, s);

END ELEMENT

END TEST

END SERVICE

```

The use of **string_value =>** indicates that the chosen field in the union is **string_value**.

If no field is specified, the first field in the union is taken by default.

Related Topics

[Testing variables on page 454](#) | [Testing structured variables on page 470](#) | [Testing structured variables with C expressions on page 471](#) | [Testing structured variables with other structured variables on page 472](#)

Stub simulation

Component Testing for C

Stub simulation is based on the idea that certain functions are to be simulated and are therefore replaced with other functions which are generated in the test driver. These generated functions, or stubs, have the same interface as the simulated functions, but the body of the functions is replaced.

These stubs have the following roles:

- To store input values to simulated functions
- To assign output values from simulated functions

To generate these stubs, the Test Script Compiler must have the following information:

- The prototypes of the functions that are to be simulated from the stub point of view.
- A method of passing each parameter (input, output, or input/output).

When using the Component Testing Wizard, you specify the functions that you want to stub. This automatically adds the corresponding code to the `.ptu` test script. On execution of the test, Component Testing for C generates the stub in the test driver, which includes:

- a variable array for the input values of the stub
- a variable array for the output values of the stub
- a body declaration for the stub function

Function Prototypes

When generating a stub for a function, HCL® OneTest™ Embedded considers both the original and the simulation version of the first prototype of the function that is encountered, which can be:

- The declaration of the function in an included header file.
- The declaration **DEFINE STUB** statement in the `.ptu` test script, which declares how the stub is used by the application under test and how the check code is generated.

If the first declaration is not found HCL® OneTest™ Embedded considers that original is identical to the simulated function.

Both can differ when the original prototype does not declare explicitly how the application uses it. For example, a void * parameter can be used as `char*` or `int*`.

It is possible to stub a function that is located in the source file under test. In this case, the source file must be included in the `.ptu` test script. If an existing body of stubbed function is encountered in the source code under test, HCL® OneTest™ Embedded renames the existing body to `_atu_stub_ <function-name>` and the stubbed version of the function is used in the test driver.

An example is provided in the **StubInUseFunc** test node of the **Stub C** example project.

Note: To comply with the DO178B standard, the source code under test must be compiled separately from the **.ptu**. If you choose to include the source file in the **.ptu** script, then you will need to justify this with the DO178B authority.

Passing Parameters

Passing parameters by pointer can lead to problems of ambiguity regarding the data actually passed to the function. For example, a parameter that is described in a prototype by `int *x` can be passed in the following way:

`int *x` as input ==> `f(x)`

`int x` as output or input/output ==> `f(&x)`

`int x[10]` as input ==> `f(x)`

`int x[10]` as output or input/output ==> `f(x)`

Therefore, to describe the stubs, you should specify the following:

- The data type in the calling function
- The method of passing the data

Example

An example project called **Stub C** is available from the Examples section of the Start page. This example demonstrates the use of stubs in Component Testing for C. See [Example projects on page 677](#) for more information.

Related Topics

[Stub Definition in C on page 477](#) | [Stub Usage in C on page 480](#) | [Sizing Stubs on page 484](#) | [Replacing Stubs on page 482](#) | [Advanced Stubs on page 485](#) | [Example projects on page 677](#)

Stub Definition

Component Testing for C

The following simulation describes a set of function prototypes to be simulated in an instruction block called **DEFINE STUB ... END DEFINE**:

HEADER file, 1, 1

BEGIN

DEFINE STUB file

```
#int open_file(char _in f[100]);
```

```
#int create_file(char _in f[100]);  
  
#int read_file(int _in fd, char _out l[100]);  
  
#int write_file(int fd, char _in l[100]);  
  
#int close_file(int fd);  
  
END DEFINE
```

The prototype of each simulated function is described in ANSI form. The following information is given for each parameter:

- The type of the calling function (**char f[100]** for example, meaning that the calling function supplies a character string as a parameter to the open_file function)
- The method of passing the parameter, which can take the following values:
 - **_in** for an input parameter
 - **_out** for an output parameter
 - **_inout** for an input/output parameter

These values describe how the parameter is used by the called function, and, therefore, the nature of the test to be run in the stub.

- The **_in** parameters only will be tested.
- The **_out** parameters will not be tested but will be given values by a new expression in the stub.
- The **_inout** parameters will be tested and then given values by a new expression.

Any returned parameters are always taken to be **_out** parameters.

You must always define stubs after the BEGIN instruction and outside any **SERVICE** block.

Modifying Stub Variable Values

You can define stubs so that the variable pointed to is updated with different values in each test case. For example, to stub the following function:

```
extern void function_b(unsigned char * param_1);
```

Declare the stub as follows:

```
DEFINE STUB code_c
```

```
#void function_b(unsigned char _out param_1);
```

```
END DEFINE
```

Note Any **_out** parameter is automatically a pointer, therefore the asterisk is not necessary.

To return '255' in the first test case and 'a' in the second test case, you would write the following in your test script:

```
SERVICE function_a
SERVICE_TYPE extern
-- By function returned type declaration
#int ret_function_a;
TEST 1
FAMILY nominal
ELEMENT
VAR ret_function_a, init = 0, ev = 1
STUB function_b (255)
#ret_function_a = function_a();
END ELEMENT
END TEST -- TEST 1
TEST 2
FAMILY nominal
ELEMENT
VAR ret_function_a, init = 1, ev = 0
STUB function_b ('a')
#ret_function_a = function_a();
END ELEMENT
END TEST -- TEST 2
END SERVICE -- function_a
```

Simulating Global Variables

The simulated file can also contain global variables that are used by the functions under test. In this case, as with simulated functions, you can simulate the global variables by declaring them in the **DEFINE STUB** block, as shown in the following example:

```
DEFINE STUB file

#int fic_errno; /* simulated global variable */

#char fic_err_msg[100]; /* simulated global variable */

#int open_file(char _in f[100]);

#int create_file(char _in f[100]);

#int read_file(int _in fd, char _out l[100]);

#int write_file(int fd, char _in l[100]);

#int close_file(int fd);

END DEFINE
```

The global variables are created as if they existed in the simulated file. The global variables must be initialized within the **.ptu** test script.

Using stubs

Component Testing for C

Use the **STUB** statement to declare that you want to use a stub rather than the original function. You can use the **STUB** instruction within environments or test scenarios.

This **STUB** instruction tests input parameters and assigns a value to output parameters each time the simulated function is called.

The following information is required for every stub called in a scenario:

- Test values for the input parameters
- Return values for the output parameters
- Test and return values for the input/output parameters
- Where appropriate, the return value of the called stub

Example

The following example illustrates use of a stub which simulates file access.

```

SERVICE copy_file

#char file1[100], file2[100];

#int s;

TEST 1

FAMILY nominal

ELEMENT

VAR file1, init = "file1", ev = init

VAR file2, init = "file2", ev = init

VAR s, init == , ev = 1

STUB open_file ("file1")3

STUB create_file ("file2")4

STUB read_file (3,"line 1")1, (3,"line 2")1, (3,"")0

STUB write_file (4,"line 1")1, (4,"line 2")1

STUB close_file (3)1, (4)1

#s = copy_file(file1, file2);

END ELEMENT

END TEST

END SERVICE

```

The following example specifies that you expect three calls of *foo*.

```

STUB STUB1.foo(1)1, (2)2, (3)3

...

#foo(1);

#foo(2);

#foo(4);

```

The first call has a parameter of 1 and returns 1. The second has a parameter of 2 and returns 2 and the third has a parameter of 3 and returns 3. Anything that does not match is reported in the test report as a failure.

Replacing Stubs

Component Testing for C

Stubs can be used to replace a component that is still in development. Later in the development process, you might want to replace a stubbed component with the actual source code.

To replace a stub with actual source code:

1. Right-click the test node and select Add Child and Files
2. Add the source code files that will replace the Stubbed functions.
3. If you do not want a new file to be instrumented, right-click the file select Properties. Set the Instrumentation property to No.
4. Open the .ptu test script, and remove the **STUB** sections from your script file.

Multiple stub calls

Component Testing for C

For a large number of calls to a stub, use the following syntax for a more compact description:

```
<call i> .. <call j> =>
```

You can describe each call to a stub by adding the specific cases before the preceding instruction, for example:

```
<call i> =>
```

or

```
<call i> .. <call j> =>
```

The call count starts at 1 as the following example shows:

```
TEST 2
```

```
FAMILY nominal
```

```
COMMENT Reading of 100 identical lines
```

```
ELEMENT
```

```

VAR file1, init = "file1", ev = init
VAR file2, init = "file2", ev = init
VAR s, init == , ev = 1
STUB open_file 1=>("file1")3
STUB create_file 1=>("file2")4
STUB read_file 1..100(3,"line")1, 101=>(3,"")0
STUB write_file 1..100=>(4,"line")1
STUB close_file 1=>(3)1,2=>(4)1
#s = copy_file(file1,file2);
END ELEMENT
END TEST

```

Multiple stub calls

If a stub is called several times during a test, either of the following are possible:

- Describe the different calls in the same **STUB** instruction, as described previously.
- Use several **STUB** instructions to describe the different calls. (This allows a better understanding of the test script when the **STUB** calls are not consecutive.)

The following example rewrites the test to use this syntax for the call to the **STUB close_file**:

```

STUB close_file (3)1
STUB close_file (4)1

```

No stub calls

To check that a **STUB** is never called, even if an **ENVIRONMENT** containing the **STUB** is used, use the following syntax:

```

STUB write_file 0=>(4,"line")

```

No testing of the maximum number of stub calls

If you do not want to test the maximum number of calls to a stub, you can use the keyword **others** in place of the call number to describe the behavior of the stub for the calls to the stub that are not yet described.

The minimum number of calls to a stub is checked against the maximum call number that is specified without the **others** keyword.

For example, the following instruction lets you specify the first call and all the following calls without knowing the exact number. In this example, the test checks that the stub has been called at least once:

```
STUB write_file 1=>(4,"line")1,others=>(4,"")1
```

Stub memory usage

Component Testing for C

For each STUB, the test allocates memory to:

- Store the expected value of the input parameters during the test
- Store the obtained value of the input parameters during the test when error is detected
- Store the values assigned to output parameters before the test

A stub can be called several times during the execution of a test.

The test allocates memory for expected and returned values in accordance with the maximum number of STUB calls used in the tests.

In the following example, the script allocates storage space for expected and returned values for 4 ranges for **read_file** and 3 ranges for **write_file**:

```
TEST 1
```

```
STUB read_file 1..10(3,"line")1,11..20(1,"line")2, 21..100(1,"line")3, 101=>(3,"")0
```

```
STUB write_file 1..5=>(4,"line")1,others=>(4,"")1
```

```
...
```

```
END TEST
```

```
.....
```

```
TEST 2
```

```
STUB read_file 1..100(3,"line")1, 101=>(3,"")0
```

```
STUB write_file 1..2=>(4,"line")1,3=>(4,"line")4,others=>(4,"")1
```

```
...
```

```
END TEST
```

By default, when you define a **STUB**, the test allocates space for obtained values for the 10 first call in error.

In the following example, the script allocates storage space for the first 17 call errors to the stub:

```
DEFINE STUB file 17

#int open_file(char _in f[100]);

#int create_file(char _in f[100]);

#int read_file(int _in fd, char _out l[100]);

#int write_file(int fd, char _in l[100]);

#int close_file(int fd);

END DEFINE
```

In this case, only the first 17 errors are shown in the report. Any more errors are not recorded.

You can also reduce the stub allocation value to a lower value when running tests on a target platform that is short on memory resources.

Advanced stubs

Component Testing for C

This section covers some of the more complex notions when dealing with stub simulations in Component Testing for Ada.

To learn about	See
Writing complex stubs in C	Native Code in Stubs on page 486
Specifying items that are not to be tested	Excluding a Parameter from a Stub on page 564
Stubbing functions that take arrays in _inout mode	Simulating Functions with _inout Mode Arrays on page 567
Stubbing functions that use type modifiers	Simulating Functions with Type Modifiers on page 489
Stubbing functions for which the number of parameters may vary	Simulating Functions with Varying Parameters on page 490
Stubbing functions that use const parameters	Simulating Functions with const Parameters on page 489

Stubbing functions that use void* parameters

[Simulating Functions with void* Parameters on page 491](#)

Stubbing functions that use char* parameters

[Simulating Functions with char* Parameters on page 492](#)

Creating complex stubs

Component Testing for C

If necessary, you can make stub operation more complex by inserting native code into the body of the simulated function. You can do this easily by adding the lines of native code after the prototype, as shown in the following example:

```
DEFINE STUB file
```

```
#int fic_errno;
```

```
#char fic_err_msg[100];
```

```
#int open_file(char _in f[100])
```

```
# { errno = fic_errno; }
```

```
#int create_file(char _in f[100])
```

```
# { errno = fic_errno; }
```

```
#int read_file(int _in fd, char _out l[100])
```

```
# { errno = fic_errno; }
```

```
#int write_file(int fd, char _in l[100])
```

```
# { errno = fic_errno; }
```

```
#int close_file(int fd)
```

```
# { errno = fic_errno; }
```

```
END DEFINE
```

Excluding a Parameter from a Stub

Component Testing for C

Stub Definition

You can specify in the stub definition that a particular parameter is not to be tested or given a value. You do this using a modifier of type **_no** instead of **_in**, **_out** or **_inout**, as shown in the following example:

```
DEFINE STUB file

#int open_file(char _in f[100]);

#int create_file(char _in f[100]);

#int read_file(int _no fd, char _out l[100]);

#int write_file(int _no fd, char _in l[100]);

#int close_file(int fd);

END DEFINE
```

In this example, the fd parameters to read_file and write_file are never tested.

Note You need to be careful when using **_no** on an output parameter, as no value will be assigned to it. It will then be difficult to predict the behavior of the function under test on returning from the stub.

Stub Usage

Parameters that have not been tested (preceded by **_no**) are completely ignored in the stub description. Therefore, changing **_in** to **_no** in the DEFINE STUB means that you must remove the corresponding input value in each STUB check.

The easiest way to disable the check value is to add the **_nocheck** keyword before the **_in**.

The two values of the input/output parameters are located between brackets as shown in the following example:

```
DEFINE STUB file

#int open_file(char _in f[100]);

#int create_file(char _in f[100]);

#int read_file(int _no fd, char _inout l[100]);

#int write_file(int _no fd, char _in l[100]);

#int close_file(int _no fd);

END DEFINE
```

...

```
STUB open_file ("file1")3
STUB create_file ("file2")4
STUB read_file (("", "line 1"))1, (("line 1", "line 2"))1,
&(("line2", ""))0
STUB write_file ("line 1")1, ("line 2")1
STUB close_file ()1, ()1
```

If a stub is called and if it has not been declared in a scenario, an error is raised in the report because the number of the calls of each stub is always checked.

Functions Using _inout Mode Arrays

Component Testing for C

To stub a function taking an array in **_inout** mode, you must provide storage space for the actual parameters of the function.

The function prototype in the **.ptu** test script remains as usual:

```
#extern void function(unsigned char *table);
```

The **DEFINE STUB** statement however is slightly modified:

```
DEFINE STUB Funct
#void function(unsigned char _inout table[10]);
END DEFINE
```

The declaration of the pointer as an array with explicit size is necessary to memorize the actual parameters when calling the stubbed function. For each call you must specify the exact number of required array elements.

```
ELEMENT
STUB Funct.function 1 => (({'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 0x0},
&{'i', 'h', 'g', 'f', 'e', 'd', 'c', 'b', 'a', 0x0}))
#call_the_code_under_test();
END ELEMENT
```

This naming convention compares the actual values and not the pointers.

The following line shows how to pass **_inout** parameters:

```
{{<in_parameter>},{<out_parameter>}}
```

Functions Containing Type Modifiers

Component Testing for C

Type modifiers can appear in the signature of the function but should not be used when manipulating any passed variables. When using type modifiers, add **@** prefix to the type modifier keyword.

HCL® OneTest™ Embedded recognizes **@**-prefixed type modifiers in the function prototype, but ignores them when dealing internally with the parameters passed to and from the function.

This behavior is the default behavior for the "const" keyword, the '@' is not necessary for const.

Example

Consider a type modifier **__foo**

```
DEFINE STUB tst_cst
```

```
#int ModifParam(@__foo float _in param);
```

```
END DEFINE
```

Note In this example, **__foo** is not a standard ANSI-C feature. To force HCL® OneTest™ Embedded to recognize this keyword as a type modifier, you must add the following line to the **.ptu** test script:

```
##pragma attol type_modifier = __foo
```

Functions Using *const* Parameters

Component Testing for C

Functions using *const* parameters sometimes produce compilation errors when stubbed with HCL® OneTest™ Embedded.

This is because the preprocessor generates variables that are used for testing calls to the **STUBs**. These variables have the same type as the parameter to the function being stubbed: *const int*. These *const* variables cannot be modified, causing the compilation errors.

To work around this problem, you can indicate that type modifiers for a STUB parameter should be used in the function definition, but not in the declaration of the variables used to control the STUBs.

To do this, add an **@** character as a prefix to the type modifier. If your function takes a *const* pointer, then you don't need the **@** prefix:

This technique can be used with any type modifier.

Example

Consider the following function:

```
extern int ConstParam(const int param);
```

To stub the function, you would normally write the following lines in the **.ptu** test script. These will produce compilation error messages:

```
DEFINE STUB Example
```

```
#int ConstParam(const int _in param);
```

```
END DEFINE
```

Instead, use the following syntax to define the stub:

```
DEFINE STUB Example
```

```
#int ConstParam(@const int _in param);
```

```
END DEFINE
```

If your function takes a *const* pointer:

```
DEFINE STUB Example
```

```
#int ConstParam(const int _in *param);
```

```
END DEFINE
```

Simulating functions with varying parameters

Component Testing for C

In some cases, functions may be designed to accept a variable number of parameters on each call.

You can still stub these functions with the Component Testing feature by using the '...' syntax indicating that there may be additional parameters of unknown type and name.

In this case, Component Testing can only test the validity of the first parameter.

Example

The standard *printf* function is a good example of a function that can take a variable number of parameters:

```
int printf (const char* param, ...);
```

Here is an example including a STUB of the *printf* function:

```

HEADER add, 1, 1

#extern int add(int a, int b);

##include <stdio.h>

BEGIN

DEFINE STUB MultiParam

#int printf (const char param[200], ...);

END DEFINE

SERVICE add

#int a, b, c;

TEST 1

FAMILY nominal

ELEMENT

VAR a, init = 1, ev = init

VAR b, init = 3, ev = init

VAR c, init = 0, ev = 4

STUB printf("hello %s\n")12

#c = add(a,b);

END ELEMENT

END TEST

END SERVICE

```

Simulating Functions with *void** Parameters

Component Testing for C

When stubbing a function that takes *void** type parameters, such as `fct_sim(double c, void * d)`, the Source Code Parser generates incomplete code that might not compile.

Using `void* _out` means that the stub has to dereference a pointer to void, which is not possible.

When you are stubbing functions that take `void*` parameters, you must check and edit the `.ptu` test script in order to specify the real type that the stub has to dereference.

Example

Consider the following test script generated by the C Source Code Parser:

```
DEFINE STUB fct_sim_c
#int fct_sim(double _in c, void _inout d);
END DEFINE
```

You should modify the `.ptu` script like this:

```
DEFINE STUB fct_sim_c
#int fct_sim(double _in c, unsigned char _inout d);
END DEFINE
```

Or, if testing the parameters is not required:

```
DEFINE STUB fct_sim_c
#int fct_sim(double _no c, unsigned char _no d);
END DEFINE
```

Simulating Functions with `char*` parameters

You can use Component Testing for C to stub functions that take a parameter of the `char*` type.

This feature applies to Component Testing for C.

The `char*` type causes problems with the Component Testing feature because of the ambiguity built into the C programming language. The `char*` type can represent:

- Pointers
- Pointers to a single `char`
- Arrays of characters of indeterminate size
- Arrays of characters of which the last character is the character `\0`, a C string.

By default, the product treats all variables of this type as C strings. To specify a different behavior, you must use one of the following methods.

Pointers

Use the **FORMAT** command to specify that the test required is that of a pointer. For example:

```

HEADER charp, ,
#extern int CharPointer(char* pChar);

BEGIN

DEFINE STUB CH

#int CharPointer(void* pChar);

END DEFINE

SERVICE CharPointer1

#char *Chars;

#int ret;

TEST 1

ELEMENT

FORMAT Chars = void*

VAR Chars, init = NIL, ev = init

VAR ret, init = 0, ev = 0

STUB CharPointer(NIL)0

#ret = CharPointer(Chars);

END ELEMENT

END TEST -- TEST 1

END SERVICE -- CharPointer1

```

Pointers to a Single *char*

Define the type as **_inout**, as in the following example.

```

HEADER charp, ,
#extern int CharPointer(char* pChar);

BEGIN

```

```

DEFINE STUB CH
#int CharPointer(char Char);
END DEFINE
SERVICE CharPointer1
#char AChar;
#int ret;
TEST 1
ELEMENT
VAR AChar, init = 'A', ev = init
VAR ret, init = 0, ev = 'A'
STUB CharPointer('A')A'
#ret = CharPointer(&AChar);
END ELEMENT
END TEST -- TEST 1
END SERVICE -- CharPointer1

```

Arrays of Characters of Indeterminate Size

Use the **FORMAT** command to specify that the array is in fact an array of unsigned chars not chars, as the product considers that char arrays are C strings. For example:

```

HEADER charp, ,
#extern int CharPointer(char* pChar);
BEGIN
DEFINE STUB CH
#int CharPointer(unsigned char Chars[4]);
END DEFINE
SERVICE CharPointer1
#char Chars[4];
#int ret;

```

```

TEST 1
ELEMENT
FORMAT Chars = unsigned char[4]
ARRAY Chars, init = {'a','b','c','d'}, ev = init
VAR ret, init = 0, ev = 'a'
STUB CharPointer({'a','b','c','d'})0
#ret = CharPointer(Chars);
END ELEMENT
END TEST -- TEST 1
END SERVICE -- CharPointer1

```

C strings

Use an array of characters in which the last character is the character '\0', a C string.

```

HEADER charp, ,
#extern int CharPointer(char* pChar);
BEGIN
DEFINE STUB CH
#int CharPointer(char* pChar);
END DEFINE
SERVICE CharPointer1
#char Chars[10];
#int ret;
TEST 1
ELEMENT
VAR Chars, init = "Hello", ev = init
VAR ret, init = 0, ev = 'H'
STUB CharPointer("Hello")'H'

```

```
#ret = CharPointer(Chars);  
  
END ELEMENT  
  
END TEST -- TEST 1  
  
END SERVICE -- CharPointer1
```

Environments

Testing environments

Component Testing for C

When drawing up a test script for a service, you usually need to write several test cases. It is likely that, except for a few variables, these scenarios will be very similar. You could avoid writing a whole series of similar scenarios by factorizing items that are common to all scenarios.

Furthermore, when a test harness is generated, there are often side-effects from one test to another, particularly as a result of unchecked modification of global variables.

To avoid these two problems and leverage your test script writing, the Test Script Language lets you define test environments introduced by the keyword **ENVIRONMENT**.

These test environments are effectively a set of default tests performed on one or more variables.

Declaring environments

Component Testing for C

A test environment consists of a list of variables for which you specify:

- Default initialization conditions for before the test
- Default expected results for after the test

Use the **VAR**, **ARRAY**, and **STR** instructions described previously to specify the status of the variables before and after the test.

You can only use an environment once you have defined it.

Delimit an environment using the instructions **ENVIRONMENT** <environment_name> and **END ENVIRONMENT**. You must place it after the **BEGIN** instruction. When you have declared it, an environment is visible to the block in which it was declared and to all blocks included therein.

Example

The following example illustrates the use of environments:

```

HEADER histo, 1, 1

#include <math.h>

#include "histo.h"

BEGIN

ENVIRONMENT image

ARRAY image, init = 0, ev = init

END ENVIRONMENT

USE image

SERVICE COMPUTE_HISTO

#int x1, x2, y1, y2;

#int status;

#T_HISTO histo;

#T_IMAGE image1;

ENVIRONMENT compute_histo

VAR x1, init = 0, ev = init

VAR x2, init = SIZE_IMAGE?1, ev = init

VAR y1, init = 0, ev = init

VAR y2, init = SIZE_IMAGE?1, ev = init

ARRAY histo, init = 0, ev = 0

VAR status, init == , ev = 0

END ENVIRONMENT

USE compute_histo

```

Specifying parameters for environments

Component Testing for C

You can specify parameters for environments.

Declare the parameters in the **ENVIRONMENT** instruction as you would for a service:

```
ENVIRONMENT compute_histo1(a,b,c,d)

VAR x1, init = a, ev = init

VAR x2, init = b, ev = init

VAR y1, init = c, ev = init

VAR y2, init = d, ev = init

ARRAY histo[0..SIZE_HISTO?1], init = 0, ev = 0

VAR status, init ==, ev = 0

END ENVIRONMENT
```

The parameters are identifiers, which you can use in variable status instructions, as follows:

- In initial or expected value expressions
- In expressions delimiting bounds of arrays in extended mode

The parameters are initialized when they are used:

```
USE compute_histo1(0,0,SIZE_IMAGE?1,SIZE_IMAGE?1)
```

The number of values must be strictly equal to the number of parameters defined for the environment. The values can be expressions of any type.

Environment override

Component Testing for C

To provide more flexibility in using environments, you can override the initialization and test specifications in an **ENVIRONMENT** block for one or more variables, one or more array elements, or one or more fields of a structured variable by using either of the following:

- A new environment
- The instructions **VAR**, **ARRAY**, or **STR** in the **ELEMENT** block

The **ENVIRONMENT** concept greatly improves test robustness. You can use this approach to group default initialization and test specifications with all the variables that are global to a module under test, allowing you to check that unexpected global variables in tests on a service are indeed not modified.

The following steps are used to handle environments:

- **VAR**, **ARRAY** and **STR** instructions are stored between **ENVIRONMENT** and **END ENVIRONMENT** instructions.
- When the Test Script Compiler comes across the instruction **USE**, it determines the scope of the environment that has been stored.
- At every **END ELEMENT** instruction, the Test Script Compiler browses through all visible environments beginning with the most recently declared one. The Test Script Compiler then checks every environment variable to see if it has been fully or partially tested. If it has only been partially tested, the Test Script Compiler generates the necessary tests to complete the testing of the variable.

This process means that:

- Tests linked to environments are always carried out last.
- The higher the environment's precedence, the earlier the tests it contains will be carried out.

Example

The following example illustrates an override of an array element in two tests:

TEST 1

FAMILY nominal

ELEMENT

VAR histo[0], init = 0, ev = SIZE_IMAGE*SIZE_IMAGE

#status = compute_histo(x1,y1,x2,y2,histo);

END ELEMENT

END TEST

TEST 2

FAMILY nominal

ELEMENT

ARRAY image, init = {others => {others => 100}}, ev = init

ARRAY histo[100], init = 0, ev = SIZE_IMAGE*SIZE_IMAGE

#status = compute_histo(x1,y1,x2,y2,histo);

END ELEMENT

END TEST

In the first test, only *histo[0]* has an override. Therefore, all the default tests were generated except for the test on the *histo* variable, which had its 0 element removed, and a test was generated on *histo[1..255]*.

In the second test, the override is more noticeable; the *histo[100]* element has been removed to generate two tests: one on *histo[0..99]*, and the other on *histo[101..255]*.

Using environments

Component Testing for C

The **USE** keyword declares the use of an environment (in other words, the beginning of that environment's visibility).

The impact or visibility of an environment is determined by the position at which you declare the environment's use with the **USE** statement.

The initial values and tests associated with the environment are applied as follows, depending on the position of the declaration:

- To all the tests in a program
- To all the tests in a service
- To all the **ELEMENT** blocks of a particular test
- Within one **ELEMENT** block of a given test.

Advanced C testing

Advanced C Testing

Component Testing for C

This section covers some of the more complex notions behind Component Testing for C.

To learn about	See
Macro definition conditions	Test Script Compiler Macro Definitions on page 501
Testing the <i>main()</i> function of C programs	Testing Main Functions on page 502
Initializing and testing pointer variables	Initializing Pointer Variables while Preserving the Pointed Value on page 505
Testing pointers against structure elements which are also pointers	Testing Pointers against Pointer Structure Elements on page 502
Working around the ambiguity of the C language between arrays and pointers	Testing a String Pointer as a Pointer on page 504

Writing cleaner **.ptu** test scripts

[C Syntax Extensions on page 482](#)

Using **SERVICES** and **FAMILY** statements.

[Component Testing Tester Configuration on page 701](#)

Test Script Compiler Macro Definitions

Component Testing for C

You can specify a list of conditions to be applied when starting the Test Script Compiler. You can then generate the test harness conditionally. In the test script, you can include blocks delimited with the keywords **IF**, **ELSE**, and **END IF**.

If one of the conditions specified in the **IF** instruction is present, the code between the keywords **IF** and **ELSE** (if **ELSE** is present), and **IF** and **END IF** (if **ELSE** is not present) is analyzed and generated. The **ELSE / END IF** block is eliminated.

If none of the conditions specified in the **IF** instruction is satisfied, the code between the keywords **ELSE** and **END IF** is analyzed and generated.

By default, no generation condition is specified, and the code between the keywords **ELSE** and **END IF** is analyzed and generated.

Testing Long Types

Component Testing for C

HCL® OneTest™ Embedded does not support 64-bit *long* types as standard. The **long long** and **_int64** types do not exist in the C Testing Language. However, a workaround does permit the use of long types within a **.ptu** test script.

1. Locate the **ana/atus_c.def** file in the TDP directory and verify that the following customization point exists.

```
#define _int64 long
```

If the line does not exist, you must add this customization point to the **ana/atus_c.def** file.

2. Locate the following line:

```
#pragma attol sizeof(long)=32
```

and replace the line with the two following lines:

```
#pragma attol sizeof(long)=64
```

```
#pragma attol sizeof(int)=64
```

If the line does not exist, you must add both lines to the **ana/atus_c.def** file.

3. Within the **.ptu** test script, append an **L** to the notation of initial and expected *long* values, and use **h64** to format the results. For example:

```
VAR MyVarLong, long#h64, init = 0xAAAAAAAAAAAAAAAAAL, ev = 0xFFFFFFFFFFFFFFFFL
```

Testing Main Functions

Component Testing for C

You can use the Component Testing feature to test C language *main* functions. To do so, you must rename those functions.

Example

```
#ifdef ATTOL

int test_main (int argc, char** argv)

#else

int main (int argc, char** argv)

#endif

{

...

}
```

If you are running an runtime analysis feature on the Component Testing test node, you can also use the **-rename** command line option to rename the *main* function name.

See the command line interface pages of Studio reference section in the **Reference category of the help..**

Testing Pointers against Pointer Structure Elements

Component Testing for C

To test pointers against structure elements which are also pointers, specify for each pointer the variable it is pointing to.

For example, consider the following code:

```
typedef struct st_Test

{
```

```

int a;

int b;

struct st_Test *Ptr1;

}st_Toto;

int FunctionTest (st_Toto *p_toto)

{

int res=0;

if (p_toto != 0)

{

if(p_toto->Ptr1 == 0)

{

res = 1;

}

}

else

{

res = 2;

}

return(res);

}

```

To test the pointer p_toto, write the following test script:

```
SERVICE TestFunction
```

```
SERVICE_TYPE extern
```

```
-- Tested service parameter declarations
```

```
#st_toto *p_toto;
```

```
-- By function returned type declaration
```

```

#int ret_TestFunction;

ENVIRONMENT ENV_TestFunction

VAR ret_TestFunction, init = 0, ev = init

END ENVIRONMENT -- ENV_TestFunction

USE ENV_TestFunction

TEST 1

FAMILY nominal

ELEMENT

STR *p_toto, init = { a => 0, b => 0, Ptr1 => NIL }, ev= init

STR *p_toto->Ptr1, init = {a=>2,b=>32, Ptr1=>NIL}, ev= init

VAR ret_TestFunction, init = 0, ev = init

#ret_TestFunction = TestFunction(p_toto);

END ELEMENT

END TEST -- TEST 1

FIN SERVICE -- TestFunction

```

Testing a String Pointer as a Pointer

Component Testing for C

Use the **string_ptr** keyword on a **VAR** line to work around the ambiguity of the C language between arrays and pointers.

For example the following **VAR** line (supposing the declaration *char* string;*) will generate C code that will copy the string into the memory location pointed by string.

```
VAR string, init = "foo", ev = init
```

-- This is the "traditional" way

Of course, if no memory was allocated to the variable, this is not possible.

The following alternative approach causes the string to point to the memory location containing **"foo"**. The string is then compared to **"foo"** using a string comparison function:

```
VAR string, string_ptr, init = "foo", ev = init
```

-- Note the additional field in the line

This syntax allows you to initialize the variable to "**NIL**", and to compare its contents to a given string after the test.

Initializing Pointer Variables while Preserving the Pointed Value

Component Testing for C

To initialize a variable as a pointer while keeping the ability to test the value of the pointed element, use the **FORMAT string_ptr** statement in your **.ptu** test script.

This allows you to initialize your variable as a pointer and still perform string comparisons using **str_comp**.

Example:

```
TEST 1
```

```
FAMILY nominal
```

```
ELEMENT
```

```
FORMAT pointer_name = string_ptr
```

-- Then your variable pointer_name will be first initialized as a pointer

```
....
```

```
VAR pointer_name, INIT="l11c01pA00", ev=init
```

-- It is initialized as pointing at the string "l11c01pA00",

--and then string comparisons are done with the expected values using str_comp.

Importing legacy component testing files

Component Testing for C

The file format of ATTOL UniTest and HCL OneTest™ Embedded v2001A Component Testing for C and Ada is not compatible with the current file format used by HCL OneTest™ Embedded.

This means that any **.prj**, **.cmp**, and **.ses** files created with pre-v2002 versions of the product must be imported and converted in order to be used in a current HCL OneTest™ Embedded project.

The **Import** feature creates a new workspace with the updated Component Testing script files.

Note This problem only affects the Component Testing for C and Ada feature. You can use previous Component Testing for C++ and System Testing tests in your current projects without importing them.

1. From the **File** menu, select **Import**.
2. In the window **Import V2001A Component Testing Files Into a New Workspace**, select the **Add...** button and then select those V2001A Component Testing files that you wish to import. To import a complete UniTest or Test RealTime v2001A project, you must select all the **.prj**, **.cmp**, and **.ses** files from that project.
3. Click the **OK** button
4. In the window **Name Workspace**, type in a name for the new workspace and click **OK**.

Limitations

This feature imports the session, project and campaign data from the old version of Component Testing, including references to and from test scripts as well as tested and integrated source files.

After the importation, you must manually check and update the following items:

- **Target Deployment Port:** Use the TDP Editor to reconfigure any custom ATTOL Target Package settings. The Target Deployment Guide contains advanced information about upgrading from an old Target Package.
- **Configuration Settings:** The Import feature retrieves **-D** condition information and *include* directories. Check the **General**, **Build** and **Component Testing for C** tabs of the **Configuration Settings** dialog box to identify any other settings that need updating.
- **Service and Family parameters:** These are not imported and require manual updating with the Tester Configuration function.

Related Topics

[About Component Testing for C and Ada on page 447](#) | [Manually Creating a Test or Application Node on page 680](#) | [Tester Configuration on page 701](#) | [Migrating from previous versions of HCL OneTest Embedded](#) | [Upgrading from v2001 target deployment ports on page 28](#)

Link tests to Requirements

HCL OneTest™ Embedded allows you to link a test or a set of tests to a requirement coming from another tool to create a traceability matrix between requirements and test results.

- To link a test or set of tests to a requirement, enter the following command line:

```
REQUIREMENT <name> { , [<attrName> =|:] <attrValue> }
```

Where:

- - **<name>** is the name of the requirement. Optionally, this name could be followed by attributes.
 - **<attrName>** is the name of the attribute. This name is optional. It is automatically added if it is missing.
 - **<attrValue>** is the value of the attribute.

Example:

```
REQUIREMENT REQ_TEST2ELEM_025, type=robustness, level:1, John
```

The tests linked by a requirement depend on the position of the keyword REQUIREMENT in the script:

```
HEADER add, 1, 1
<variable declarations for the test script>
BEGIN
  REQUIREMENT... -- Requirement defined for all tests in the script
SERVICE add
  <local variable declarations for the service>
  REQUIREMENT... -- Requirement defined for all tests in the service
  TEST 1
    REQUIREMENT...      -- Requirement defined for the test only
  FAMILY nominal
  ELEMENT
    VAR variable1, INIT=0, EV=0
    VAR variable2, INIT=0, EV=0
    #<call to the procedure under test>
  END ELEMENT
  END TEST
END SERVICE
```

Attribute values can be overloaded by environment variables during pre-processing phase. For example, if \$TARGETNAME is set, the value of the attribute \$TARGETNAME in the script will be overloaded by this environment. This allows you to dynamically configure some attributes in your build chain depending on the execution context.

After the tests execution, a requirement status is computed for each requirement, based on the result of the tests that are linked to this requirement.

A tool rod2req generates an XML file with all the requirement status and a coverage status.

Viewing Reports

Component Testing for C

After test execution, depending on the options selected, a series of Component Testing for C test reports are produced.

To learn about	See
Accessing the test reports	Opening a Report on page 683
Navigating through test reports	Using the Report Viewer on page 705
Interpreting test results	Understanding Component Testing Test Reports on page 582
Interpreting sequence diagrams of a test report	Understanding Component Testing UML Sequence Diagrams on page 509
Performing a <i>diff</i> between two test reports	Comparing C Test Reports on page 510

How the test report handles arrays and structured variables [Array and Structure Display on page 510](#)

Understanding Component Testing Reports

Component Testing for C

Test reports for Component Testing are displayed in the HCL® OneTest™ Embedded Report Viewer.

The test report is a hierarchical summary report of the execution of a test node. Parts of the report that have *Passed* are displayed in green. *Failed* tests are shown in red.

Report Explorer

The [Report Explorer on page 1024](#) displays each element of a test report with a *Passed* , *Failed*  symbol.

- Elements marked as *Failed*  are either a failed test, or an element that contains at least one failed test.
- Elements marked as *Passed*  are either passed tests or elements that contain only passed tests.

Test results are displayed for each instance, following the structure of the **.ptu** test script.

Report Header

Each test report contains a report header with:

- The version of HCL® OneTest™ Embedded used to generate the test as well as the date of the test report generation
- The path and name of the project files used to generate the test
- The total number of test cases *Passed* and *Failed*. These statistics are calculated on the actual number of test elements listed in the sections below

Test Results

The graphical symbols in front of the node indicate if the test, item, or variable is *Passed*  or *Failed* .

- A test is *Failed* if it contains at least one failed variable. Otherwise, the test is considered *Passed*.

You can obtain the following data items if you click with the pointer on the Information node:

- Number of executed tests
- Number of correct tests
- Number of failed tests

A variable is incorrect if the expected value and the value obtained are not identical, or if the value obtained is not within the expected range.

If a variable belongs to an environment, an environment header is previously edited.

In the report variables are edited according to the value of the Display Variables setting of the Component Testing test node.

The following table summarizes the editing rules:

Results	Display Variable	Display Variable	Display Variable
	All Variables	Incorrect Variables	Failed Tests Only
✓ Passed test	Variable edited automatically	Variable not edited	Variable not edited
✗ Failed test	Variable edited automatically	Variable edited automatically	Variable edited if incorrect

The [Initial and Expected Values on page 459](#) option changes the way initial and expected values are displayed in the report.

Related Topics

[Opening a report on page 683](#) | [Using the Report Viewer on page 705](#) | [Array and structure display on page 510](#) | [Initial and expected values on page 459](#) | [Exporting reports on page 705](#)

Understanding Component Testing UML Sequence Diagrams

Component Testing for C

During the execution of the test, Component Testing generates trace data this is used by the UML/SD Viewer. The Component Testing sequence diagram uses standard UML notation to represent both Component Testing results.

When using Component Testing for C with Runtime Tracing or other HCL OneTest™ Embedded features that generate UML sequence diagrams, all results are merged in the same sequence diagram.

You can click any element of the UML sequence diagram to open the test report at the corresponding line. Click again in the test report, and you will locate the line in the **.pts** test script.

Related Topics

[About the UML/SD Viewer on page 402](#) | [UML/SD Viewer Toolbar on page 1028](#) | [Understanding Component Testing Reports for C and Ada on page 508](#) | [Understanding Component Testing Reports for C++ on page 522](#)

Comparing C Test Reports

Component Testing for C

The Component Testing comparison capability allows you to compare the results of the last two consecutive tests.

To activate the comparison mode, select **Compare two test runs** in the [Component Testing Settings for C on page 999](#) dialog box.

In comparison mode an additional check is performed to identify possible regressions when compared with the previous test run.

The Component Testing Report displays an extra column named "Obtained Value Comparison" containing the actual difference between the current report and the previous report.

Related Topics

[Component Testing Settings for C on page 999](#) | [Understanding Component Testing Reports on page 508](#)

Array and Structure Display

Component Testing for C

The Array and Structure Display option indicates the way in which Component Testing processes variable array and structure statements. This option is part of the [Component Testing Settings for C on page 999](#) dialog box.

Standard Array and Structure Display

This option processes arrays and structures according to the statement with which they are declared. This is the default operating mode of Component Testing. The default report format is the **Standard** editing.

Extended Array and Structure Display

Arrays of variables may be processed after the keywords **VAR** or **ARRAY**, and structured variables after the keywords **VAR**, **ARRAY**, or **STRUCTURE**:

- After a **VAR** statement, each element in the array is initialized and tested one by one. Likewise, each member of a structure that is an array is initialized and tested element by element.
- After an **ARRAY** statement, the entire array is initialized and checked. Likewise, each member of a structure is initialized and checked element by element.
- After a **STRUCTURE** statement, the whole of the structure is initialized and checked.

When **Extended editing** is selected, Component Testing interprets **ARRAY** and **STRUCTURE** statements as **VAR** statements.

The output records in the unit test report are then detailed for each element in the array or structure.

Note This setting slightly slows down the test execution because checks are performed on each element in the array.

Packed Array and Structure Display

This command has the opposite effect of the Extended editing option. When **Packed editing** is selected, Component Testing interprets **VAR** statements as **ARRAY** or **STRUCTURE** statements.

Array and structure contents are fully tested, only the output records are more concise.

Note This setting slightly improves speed of execution because checks are performed on each array as a whole.

Related Topics

[Component Testing for C and Ada Settings on page 999](#)

Component Testing for C++

Component Testing for C++ overview

Component Testing for C++

Component Testing for C++ is a fully integrated feature of HCL OneTest™ Embedded that uses object-oriented techniques to address automated testing of C++ embedded and native software.

Object-oriented testing does not mean that the Component Testing for C++ feature is designed solely for testing object-oriented languages. Whether the target application is object-oriented or not, Component Testing for C++ adapts to the environment.

In fact, Component Testing for C++ can be used for:

- Software feature tests,
- Component integration tests,
- Software validation,
- Non-regression tests.

Component Testing for C++ supports ISO/IEC 14882:1998.

Overview

Basically, Component Testing for C++ interacts with your source code through a scripting language called *C++ Test Script Language*. You use the HCL OneTest™ Embedded GUI or command line tools to set up your test campaign, write your test scripts, run your tests and view the test results. Object Testing's mode of operation is twofold:

- *C++ Test Driver* scripts describe a test harness that stimulates and checks basic I/O of the code under test.
- *C++ Contract Check* scripts, which instrument the code under test, verifying behavioral assertions during execution of the code.

Note: Contract Check is part of the Component Testing for C++ feature. However, contract check scripts can also be used in application nodes, as a Runtime Analysis feature.

When the test is executed, Component Testing for C++ compiles both the test scripts and the source under test, then instruments the source code and generates a test driver. Both the instrumented application and the test driver provide output data which is displayed within HCL OneTest™ Embedded.

How Component Testing for C++ Works

When a test node is executed, the Test Compiler (**atoprepro**) compiles both the test scripts and the source under test. This preprocessing creates an **.ots** file. The resulting source code generates a test driver.

If any Runtime Analysis tools are associated with the test node, then the source code is also instrumented with the Instrumentor (**attolcpp**) tool.

The test driver, TDP, stubs and dependency files all make up the test harness.

The test harness interacts with the source code under test and produces test results. Test execution creates a **.tdf** file.

The **.ots** and **.tdf** files are processed together the Component Testing Report Generator (**atopospro**). The output is the **.xrd** report file, which can be viewed and controlled in the HCL OneTest™ Embedded GUI.

Of course, these steps are mostly transparent to the user when the test node is executed in the HCL OneTest™ Embedded GUI.

Related Topics

[Using Test Features on page 446](#) | [Manually Creating a Test or Application Node on page 680](#)

C++ testing overview

C++ test nodes

Component Testing for C++

The project structure of HCL OneTest™ Embedded GUI uses *test nodes* to represent your Component Testing test harness.

Test nodes created for Component Testing for C++ use the following structure

- **C++ Test Node:** represents the Component Testing for C++ test harness
- `<script>.otc`: is the Contract-Check test script
- `<script>.otd`: is the test driver script
- `<source>.cpp`: is the source file under test
- `<source>.cpp`: is an additional source file

Related Topics

[Component Testing for C++ on page 511](#) | [Setting up a Project on page 674](#) | [Additional Source Files on page 517](#)

C++ contract check Script

Component Testing for C++

The C++ Contract Check script allows you to test invariants and state charts as well as wraps for each method of the class.

The Contract Check script is contained in an `.otc` file, whose name matches the name of the file containing the class definition.

C++ Contract Check scripts are written in C++ Contract Check Language, which is part of the C++ Test Script language designed for Component Testing for C++.

A typical Contract Check `.otc` test script is structured as follows:

```
CLASS <class to wrap>
{
  WRAP <method>
  REQUIRE <expression>
  ENSURE <expression>
  WRAP <method>
  REQUIRE <expression>
  ENSURE <expression>
}
```

See the Reference section for the semantics of the C++ Contract Check Language.

Note When an **.otc** contract check script is used in a test node, the related source files are always instrumented even if they are displayed as not instrumented in Project Explorer.

Contract Check in a Component Test

You can use the Component Testing wizard to set up a test node and create the C++ contract-check script templates or you can manually create a Component Testing for C++ test node to reuse existing test scripts.

The **.otc** contract-check script must be executed before an **.otd** Test Driver script, therefore the order in which both script types appear in the Test node is critical. This is important if you are manually creating a test node.

Contract Check Runtime Analysis

C++ Contract Check scripts can also be used in a simple application node.

In this case, you can either copy the **.otc** contract from an existing C++ component test node, or you can create an **.otc** contract check script manually.

The **.otc** contract-check script must be placed before any other item in the application node.

Related Topics

[C++ Test Driver Script on page 514](#) | [Component Testing Wizard on page 666](#) | [Manually Creating a Node on page 680](#) | [Using native C++ statements on page 517](#)

C++ Test Driver Script

Component Testing for C++

The C++ Test Driver Script stimulates the source code under test to test assertions on a cluster of classes.

The test driver script itself is contained in an **.otd** file and may call two optional files:

- A declaration file (**.dcl**) that contains C++ code that ensures the types, class, variables and functions needed by your test script will be available in your code.
- A stub file (**.stb**) whose purpose is to define variables, functions and methods which are to be stubbed.

Using a separate declaration and stub files is optional. It is possible to include all or certain declarations and stubs directly within the test driver script file.

C++ Contract Check scripts are written in C++ Contract Check Language, which is part of the C++ Test Script Language designed for Component Testing for C++.

A typical Component Testing **.otd** test script looks like this:

```
INCLUDE "Test.dcl";
```

```

TEST CLASS TestClass1 {
    PROLOGUE {
        <Declarations of variables>

        <Actions to be performed before executing this test class.>
    }
    TEST CASE Test1 {
        #method_under_test();

        CHECK (expression_must_be_true == true);
    }
    EPILOGUE {
        <Actions to be performed when leaving the test class>
    }
    RUN {
        Test1;
    }
}

RUN {
    TestClass1 (File<char*>);
}

```

See the Reference section for the semantics of the C++ test driver language.

You can use the Component Testing wizard to set up a test node and create the C++ Test Driver script templates or you can manually create a Component Testing for C++ test node to reuse existing test scripts.

An **.otc** contract-check script must be executed before an **.otd** Test Driver script, therefore the order in which both script types appear in the Test node is critical. This is important if you are manually creating a test node.

See the Reference section for the semantics of the C++ Contract Check Language.

Related Topics

[C++ Contract-Check Script on page 513](#) | [Component Testing Wizard on page 666](#) | [Using native C++ statements on page 517](#)

Files and classes under test

Component Testing for C++

Source Files

The **Source under test** are source files containing the code you want to test. These files must contain either the definition of the classes targeted by the test, or method implementations of those classes.

Note Source files can be either body files (.C, .cc, .cpp...) or header files (.h), but it is usually recommended to select the body file. Specifying both header and body files as **Source under test** is unnecessary.

When using a C++ Test Driver Script, the wizard generates:

- A template test driver script (.otd) to test each class defined in the **Candidate classes** box.
- Declaration (.dcl) and stub (.stb) files to make the environment of the source under test available to the test script.

When using a C++ Contract Check script, the wizard generates:

- A template contract script (.otc) containing template code allowing you to add invariants and state charts as well as empty wraps for each method of the class.

Note If a source under test is a header file (a file containing only declarations, typically a .h file), the source file under test is automatically included in the C++ Test Driver script.

Candidate Classes

For source files containing several classes, you may only want to submit a restricted number of classes to testing.

If no classes are selected, the wizard automatically selects all classes that are defined or implemented in the source(s) under test as follow:

- The class is defined within the source file (*i.e.* the sequence class <name>{ };).
- At least one of the methods of the class is defined within the source file (*i.e.* a method's body).

Note Classes can only be selected if you have refreshed the File View before running the Test Generation Wizard.

Related Topics

[Additional Files or Directories on page 517](#) | [Component Testing Wizard on page 666](#)

Using native C++ statements

Component Testing for C++

In some cases, it can be necessary to include portions of C++ native code inside an **.otc** or **.otd** test script for one the following reasons:

- To declare native variables that participate in the flow of a scenario. Such statements must be analyzed by the Component Testing Parser.
- To insert native code into a scenario. In this case, the code is ignored by the parser, but carried on into the generated code.

Analyzed native code

Lines prefixed with a # character are analyzed by Component Testing Parser.

Prefix statements with a # character to include native C++ variable declarations as well as any code that can be analyzed by the parser.

```
#int i;
```

```
#char *foo;
```

Variable declarations must be placed outside of Component Testing Language blocks or preferably at the beginning of scenarios and procedures.

Ignored native code

Lines prefixed with a @ character are ignored by the parser, but copied into the generated code.

To use native C++ code in the test script, start instructions with a @ character:

```
@for(i=0; i++; i<100) func(i);
```

```
@foo(a,&b,c);
```

You can add native code either inside or outside of C++ Test Script Language blocks.

Related topics

[C++ contract check script on page 513](#) | [C++ test driver script on page 514](#)

Additional and included files

Component Testing for C++

When creating a Component Testing test node for C++, the Component Testing wizard offers the following options for specifying dependencies of the source code under test:

- Additional files
- Included files

Additional Files

Additional source files are source files that are required by the test script, but not actually tested. For example, with Component Testing for C++, Visual C++ resource files can be compiled inside a test node by specifying them as additional files.

Additional header files (.h) are not handled in the same way as additional body files (.cc, .C, or .cpp):

- **Body files:** With a body file, the Test Generation Wizard considers that the compiled file will be linked with your test program. This means that all *defined* variables and routines are considered as defined, and therefore not stubbed.
- **Header files:** With a header file (a file containing only *declarations*), the Test Generation Wizard considers that all the entities *declared* in the source file itself (not in included files) are defined. Typically, you would use additional header files if you only have a .h file under test and a matching object file (.o or .obj), but not the actual source file (.cc, .C, or .cpp).

You can toggle a source file from *under test* to *additional* by changing the **Instrumentation** property in the Properties Window dialog box.

Additional directories are directories that are declared to only contain additional source files.

Functions which are not located in an additional file or in a tested file are simulated by Component Testing for C++.

Included Files

Included files are normal source files under test. However, instead of being compiled separately during the test, they are included and compiled with the C++ Test Driver script.

Header files are automatically considered as included files, even if they are not specified as such.

Source files under test should be specified as included when:

- The file contains the class definition of a class you want to test
- A function or a variable definition depends upon a type which is defined in the file under test itself
- You need access in your test script to a static variable or function, defined in the file under test

In most cases, you do not have to specify files to be included. The Component Testing wizard automatically generates a warning message in the [Output Window on page 1021](#), when it detects files that should be specified as included files. If this occurs, rerun the Component Testing wizard, and select the files to be included in the **Include source files** section of the [Advanced Options on page 673](#) dialog box.

To specify included files while creating a test node:

1. Select a valid C++ configuration and run the Component Testing wizard.
2. On the **Test Script Generation Settings** page (Step 3/5), expand **Components Under Test** and **<Test Name>** . where *<Test Name>* is the name of the Test Node.
3. Scroll down the list to **Included Files**, select the value field and click the '...' button to enter a list of files.
4. Enter any other advanced settings and continue with the Component Testing wizard.

To specify additional files while creating a test node:

1. Select a valid C++ configuration and run the Component Testing wizard.
2. On the **Test Script Generation Settings** page (Step 3/5), select **General** and switch the **Test Mode** setting to **Expert Mode**.
3. Expand **Components Under Test** and select Test Boundaries.
4. Under **Additional Files or Directories**, select the value field and click the '...' button to enter a list of files or directories
5. Enter any other advanced settings and continue with the Component Testing wizard.

Related Topics

[Files and Classes Under Test on page 516](#) | [Component Testing Wizard on page 666](#) | [Advanced Options on page 673](#)

Declaration files

Component Testing for C++

A declaration file (**.dcl**) ensures that the types, class, variables and functions needed by your test script will be available in your code.

Using a separate **.dcl** file is optional, since it is merely included within the C++ Test Driver script. It is possible to declare types, classes, variables and functions directly within an C++ Test Driver script file.

Typically, **.dcl** files are created by the Component Testing Wizard and do not need to be edited by the user. If you do need to define your own declarations for a test, it is recommended that you do this within the Test Driver script. Declaration files appear in the Component Testing for C++ test node.

Declaration files must be written in C++ Test Script Language and contain native code declarations. See the **Reference section** for details about the language.

Related Topics

[C++ Test Driver Script on page 514](#) | [Simulated, Additional and Included Files on page 517](#)

Error Handling

An error may be generated by either native code or any of the following instructions in a test script:

- CHECK
- CHECK PROPERTY
- CHECK EXCEPTION
- [CHECK STUB on page 761](#)
- CHECK METHOD
- REQUIRE
- ENSURE
- Native statement

Refer to each of these keywords to see when the instructions generate an error.

Error handling behavior is specified with the keyword [ON ERROR on page 767](#). According to the choice specified by ON ERROR, the script may continue normal execution, skip the current block, or exit.

Test Results

When no errors occur during execution of a C++ Test Script Language script, the script receives *Passed* status. Otherwise, it is considered *Failed*.

When the test is completed, the errors appear in the Report Viewer or in the UML/SD Viewer as red notes.

Template Classes

Component Testing for C++ supports assertions only for fully generic and fully specialized template classes. Partial specializations are not supported.

A contract referring to a generic template class is applied to every instance of this template class, unless a specific contract has been defined for an instance of this template class.

There may be a state machine description associated with the template class, and another with a template specialization. In such a case, the latter applies to the specific template instance, and the first applies to any other instance.

Same mechanism for invariant definition (There may be invariants associated with the template class, and other invariants with a template specialization. In such a case, the latter ones apply to the specific template instance, while the first one apply to any other instance.)

A wrap defined within a generic template class contract does not apply to specialization of the associated method. If you want to test a method specialization, you must define a WRAP into the contract associated to the class instance the method specialization belongs to.

It is not possible to define WRAPs for template methods within a non-template class.

Specialization

Specialized templates are templates for which some of the parameters are real. Full-specialization of a template is an instance of the template (all parameters are real).

Example

```
template <class T,int N> class C; // generic template, not a specialization
```

```
template <class T> class C<T,2>; // partial specialization (not supported by Component Testing for C++)
```

```
template <> class C<char *,2>; // full-specialization
```

Note When using full-specializations, latest ISO/IEC C++ standards suggest using the template prefix **template<>**.

Testing shared libraries

Component Testing for C++

In order to test a shared library, you must create a test node containing the **.otd** component test script that uses the library, and a reference link to the library.

After the execution of the test node, the runtime analysis and component test results are located in the application node.

To test a shared library:

1. Add the library to your project:
 - a. Right-click a group or project node and select **Add Child** and **Library** from the popup menu.
 - b. Enter the name of the Library node
 - c. Right-click the Library node and select **Add Child** and **Files** from the popup menu.
 - d. Select the source files of the shared library.
2. Run the Component Testing wizard as usual on the source file of your library. This creates a test node containing the **.otc** and **.otd** test scripts and the source file.
3. Delete the source file from the test node.
4. Create a reference to the shared library in the test node:

- a. Right-click the application or test node that will use the shared library and select **Add Child** and **Reference** from the popup menu.
 - b. Select the library node and click **OK**.
5. Build and execute the test node.

Example

An example demonstrating how to test shared libraries is provided in the **Shared Library** example project. See [Example projects on page 677](#) for more information.

Related Topics

[Using shared libraries on page 686](#) | [Profiling shared libraries on page 307](#)

C++ test reports

Understanding Component Testing for C++ reports

Component Testing for C++

Test reports for Component Testing for C++ are displayed in HCL OneTest™ Embedded Report Viewer.

The test report is a hierarchical summary report of the execution of a test node. Parts of the report that have **Passed** are displayed in green. **Failed** tests are shown in red.

Report Explorer

The Report Explorer displays each element of a Test Verdict report with a Passed , Failed  or Undefined  symbol:

- Elements marked as Failed  are either a failed test, or an element that contains at least one failed test.
- An Undefined  marker means either that the test was not executed, or that the element contains a test that was not executed AND all executed tests were passed.
- Elements marked as Passed  are either passed tests or elements that contain only passed tests.

Test results are displayed in two parts:

- Test Classes, Test Suites and Test Cases of all the executed C++ Test scripts.
- Class results for the entire Test. Each class contains assertions (WRAP statement), invariants, states and transitions.

Report Header

Each Test Verdict report contains a report header with:

- The path and name of the **.xrd** report file.
- A general verdict for the test campaign: Passed or Failed.
- The number of test cases Passed and Failed. These statistics are calculated on the actual number of test elements (Test Case, Procedure, Stub and Classes) listed sections below.

Note The total number counts the actual test elements, not the number of times each element was executed. For instance, if a test case is run 5 times, of which 2 runs have failed, it will be counted as one *Failed* test case.

Test Script

Each script is displayed with a metrics table containing the number of Test Suite, Test Class, Test Case, Epilogue, Procedure, Prologue and Stub blocks encountered. In this section, statistics reflect the number of times an element occurs in a C++ Test script.

Test Results

For each Test Case, Procedure and Stub, this section presents a summary table of the test status. The table contains the number of times each verification was executed, failed and passed.

For instance, if a Test Case containing three **CHECK** statements is run twice, the reported number of executions will be six, the number of failed verifications will be two, and the number of passed verifications will be four.

The general status is calculated as follows:

Condition	Result	Status
A verification fails		Failed
A verification does not occur		Undefined
All verifications pass on each execution		Passed

Tested Classes

Class results are grouped at the end of the report and sorted in alphabetical order.

For each class the report shows the general status of assertions (**WRAP** statement), invariants, states and transitions.

The general status is computed as follows:

Condition	Result	Status
An assertion or invariant fails		Failed
An assertion or invariant does not occur		Undefined
All assertions or all invariants pass on each execution		Passed
A state is not reached		Not reached
A state has no exit transition		Not fired

When a class does not behave as expected, a table of violations is displayed. A violation is observed at the exit of a state and can be one of the following:

- Multiple: means that several states were reachable at the same time,
- Illegal: means that no state was reachable.

The displayed table gives the number of times a violation has occurred for each state. The status of this table is always Failed.

Related Topics

[Understanding Component Testing for C++ UML Sequence Diagrams on page 524](#) | [Using the Report Viewer on page 705](#) | [Opening a report on page 683](#) | [Exporting reports on page 705](#)

Understanding Component Testing for C++ UML Sequence Diagrams

Component Testing for C++

During the execution of the test, Component Testing for C++ generates trace data this is used by the UML/SD Viewer. The Component Testing for C++ sequence diagram uses standard UML notation to represent both Contract-Check and Test Driver results.

- Class Contract-check sequence diagrams,
- Test Driver Sequence Diagrams.

Both types of results can appear simultaneously in the same sequence diagram. When using Runtime Tracing with Component Testing for C++, all results are generated in the same sequence diagram.

Related Topics

[Understanding Component Testing for C++ Reports on page 522](#) | [Opening a Report on page 683](#) | [About the UML/SD Viewer on page 402](#)

Illegal and multiple transitions

Component Testing for C++

When dealing with state or transition diagrams, Component Testing for C++ adds a custom *observation state*, which is both the initial state and error state. All user-defined states can make a transition towards the *initial/error* state, and this state can transition towards all user-defined states.

At the beginning of test execution, the object is in the initial/error state.

During the test, the object is continuously tested to comply to the user-defined **STATES** and **TRANSITIONS**. There are three possible cases.

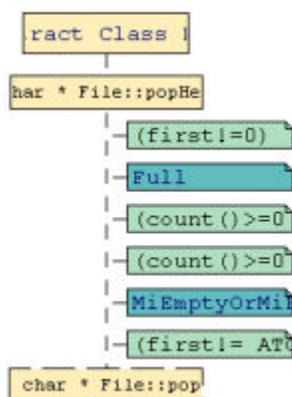
- The transition can be fired to a single state: the current state is set.
- The transition cannot be fired to any of the defined states: in this case, the state switches to the observation state and Component Testing for C++ generates an **ILLEGAL TRANSITION** note.
- The transitions can be fired to two or more states. In this case, the transition diagram is no longer unambiguous. The state is set to the observation state and Component Testing for C++ generates a **MULTIPLE TRANSITION**.

When the state diagram is in the *initial/error* state, the transition is still continuously checked, however all user defined states can be potentially fired.

Contract-Check sequence diagrams

Component Testing for C++

The following example shows how a typical class contract is represented by Component Testing for C++. C++ classes are represented as vertical lines, like object instances. The events related to the class - method entry and exit, assertion and state chart checks - are attached to the class lifeline.



Methods

For each class, methods are shown with method entry and exit actions:

- Method entry actions have a solid border,
- Method exit actions have a dotted border.

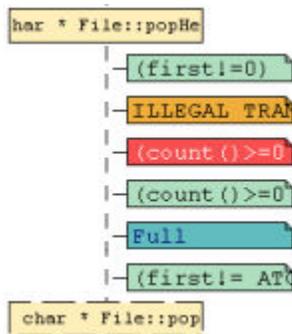
Contract-Checks

Pre and post-conditions, invariants and state verifications are displayed as Notes, attached to the class instance, and contained within the method.

You can click a note to highlight the corresponding OTC Contract-Check script line in the Text Editor window.

Illegal and Multiple Transitions

State or transition diagram errors are identified as ILLEGAL TRANSITION or MULTIPLE TRANSITION Notes as shown in the following figure:



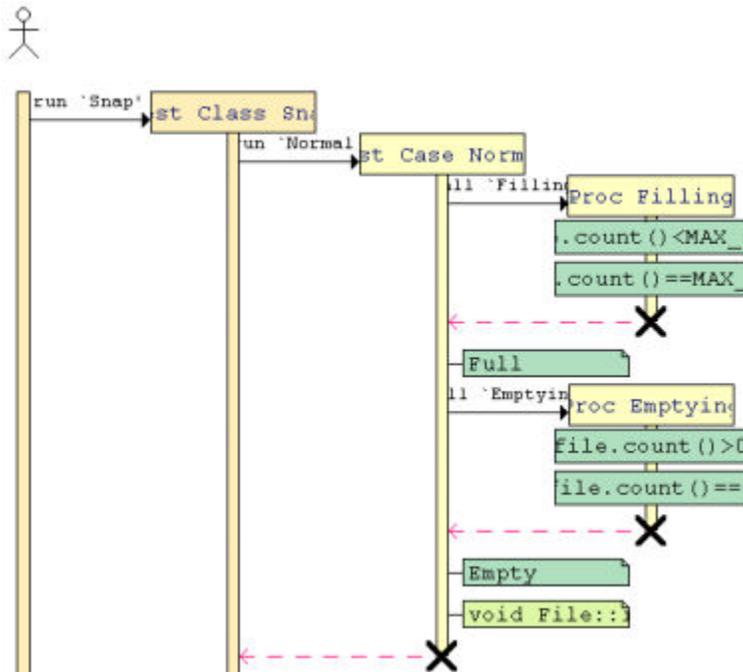
Related Topics

[Test Driver Sequence Diagrams on page 526](#) | [About the UML/SD Viewer on page 402](#) | [UML Sequence Diagrams on page 397](#)

Test Driver Sequence Diagrams

Component Testing for C++

The following example illustrates typical results generated by a Test Driver script:



Instances

When using a Test Driver script, each of the following C++ Test Script Language keywords are represented as a distinct object instance:

- TEST CLASS
- TEST SUITE
- TEST CASE
- STUB
- PROC

You can click an instance to highlight the corresponding statement in the Text Editor window.

Checks

Test Driver checks are displayed as Passed ("✓") or Failed ("✗") glyphs attached to the instances.

You can click any of these glyphs to highlight the corresponding statement in the Text Editor window.

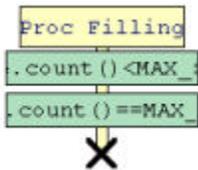
- CHECK
- CHECK PROPERTY
- CHECK STUB

- CHECK METHOD
- CHECK EXCEPTION

To distinguish checks that occur immediately from checks that apply to a stub, method or exception, the three latter use different shades of red and green.

You can click an instance to highlight the corresponding statement in the Text Editor window.

Pre and Post-conditions



The following pre and post-condition statements are green (Passed) or red (Failed) actions contained in **STUB** or **PROC** instances.

- REQUIRE
- ENSURE

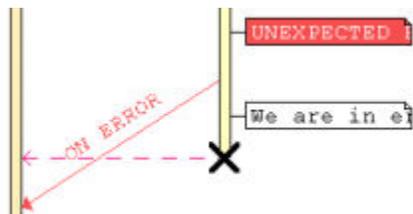
Exceptions

Component Testing for C++ generates **UNEXPECTED EXCEPTION** Notes whenever an unexpected exception is encountered. These notes will be followed by the **ON ERROR** condition.

Error Handling

Whenever a check and a pre- or post-condition generates an error, or an **UNEXPECTED EXCEPTION** occurs, the **ON ERROR** condition is displayed as shown in the following diagrams.

An **ON ERROR BYPASS** condition:



An **ON ERROR CONTINUE** condition:

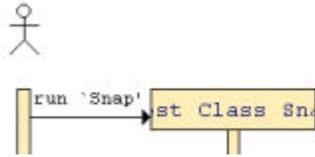


Comments and Prints

COMMENT and **PRINT** statements generate a white note, attached to the corresponding instance.

Messages

Messages can represent either a **RUN** or a **CALL** statement, or a native code stub call, as shown below:



Related Topics

[Contract-Check Sequence Diagrams on page 525](#) | [About the UML/SD Viewer on page 402](#) |

Component Testing for Ada

The Component Testing feature of HCL OneTest™ Embedded provides a unique, fully automated, and proven solution for the Ada language, dramatically increasing test productivity.

To learn about	See
General information on the Component Testing feature	Component Testing for Ada Overview on page 529
Writing test scripts for your software under test	Writing a Test Script on page 532
The types of source files under test	Integrated, Simulated and Additional Files (Ada) on page 530
Viewing Component Testing test results	Viewing Reports on page 582
Upgrading from a pre-2002 version of HCL OneTest™ Embedded	Importing V2001 Component Testing Files on page 505

Related Topics

[Using Test Features on page 446](#) | [Activity Wizards on page 663](#) | [Manually Creating a Test or Application Node on page 680](#) | [About System Testing for C on page 586](#)

Component Testing for Ada Overview

Component Testing for Ada

Component Testing for Ada interacts with your source code through the Ada Test Script Language.

Testing with Component Testing for Ada is as simple as following these steps:

- Set up your test project in the GUI
- Write a **.ptu** test script
- Run your tests
- View the results.

Component Testing for Ada supports Ada 83 and Ada 95.

How Component Testing for Ada Works

When a test node is executed, the Test Script Compiler (**attolpreproADA**) compiles both the test scripts and the source under test. This preprocessing creates a **.tdc** file. The resulting source code generates a test driver.

If any Runtime Analysis tools are associated with the test node, then the source code is also instrumented with the Instrumentor (**attolada**) tool.

The test driver, TDP, stubs and dependency files all make up the test harness.

The test harness interacts with the source code under test and produces test results. Test execution creates a **.rio** file.

The **.tdc** and **.rio** files are processed together the Component Testing Report Generator (**attolpostpro**). The output is the **.xrd** report file, which can be viewed and controlled in the HCL OneTest™ Embedded GUI.

Of course, these steps are mostly transparent to the user when the test node is executed in the HCL OneTest™ Embedded GUI.

To learn about

Source file types for code under test

Configuration Settings for Ada Component Testing test nodes

See

[Integrated, Simulated and Additional Files on page 530](#)

[Initial and Expected Value Settings on page 531](#)

Integrated, simulated and additional Files

Component Testing for Ada

When creating a Component Testing test node for Ada, the Component Testing wizard offers the following options for specifying dependencies of the source code under test:

- Integrated files
- Simulated files
- Additional files

Integrated Files

This option provides a list of source files whose components are *integrated* into the test program after linking.

The Component Testing wizard analyzes integrated files to extract any global variables that are visible from outside. For each global variable the Parser creates a default test which is added to an environment named after the file in the `.ptu` test script.

Simulated Files

This option gives the Component Testing wizard a list of source files to simulate—or stub—upon execution of the test.

A stub is a dummy software component designed to replace a component that the code under test relies on, but cannot use for practicality or availability reasons. A stub can simulate the response of the stubbed component.

The Component Testing parser analyzes the simulated files to extract the global variables and functions that are visible from outside. For each file, a **DEFINE STUB** block is generated in the `.ptu` test script.

By default, no simulation instructions are generated.

Additional Files

Additional files are merely dependency files that are added to the Component Testing test node, but ignored by the source code parser. Additional files are compiled with the rest of the test node but are not instrumented.

You can toggle instrumentation of a source file by using the Properties Window dialog box.

Related Topics

[Component Testing Wizard on page 666](#)

Initial and expected value settings

Component Testing for Ada

The Initial and Expected Value settings are part of the [Component Testing Settings for Ada on page 999](#) dialog box and describe how values assigned to each variable are displayed in the Component Testing report. Component Testing allows three possible evaluation strategy settings.

Variable Only

This evaluation strategy setting generates both the initial and expected values of each variable evaluated by the program during execution.

This is possible only for variables whose expression of initial or expected value is not reducible by the Test Script Compiler. For arrays and structures in which one of the members is an array, this evaluation is not given for the initial values. For the expected values, however, it is given only for *Failed* items.

Value Only

With this setting, the test report displays for each variable both the initial value and the expected value defined in the test script.

Combined evaluation

The combined evaluation setting combines both settings. The test report thus displays the initial value, the expected value defined in the test script, and the value found during execution if that value differs from the expected value.

Related Topics

[Component Testing Settings on page 999](#) | [Understanding Component Testing Reports on page 582](#)

Writing a Test Script

Component Testing for Ada

When you first create Component Testing for Ada test node with the Component Testing Wizard, HCL OneTest™ Embedded produces a **.ptu** test script template based on the source under test.

To write the test script, you can use the Text Editor provided with HCL OneTest™ Embedded.

Component Testing for Ada uses the Ada Test Script Language. Full reference information for this language is provided in the **Reference.** section.

section

To learn about	See
Basic .ptu test script instructions	Structure Statements on page 533
Specifying the main Ada entry unit	Test Program Entry Point on page 573
Initializing and testing variable values	Initial and Expected Values on page 535

Simulating stub functions	Stub Simulation on page 556
Catching exceptions	Unexpected Exceptions on page 580
Other specific Ada testing notions	Advanced Ada Testing on page 570

Related Topics

[Structure Statements on page 533](#) | [About the Text Editor on page 693](#)

Test Script Structure

Component Testing for Ada

The Ada Test Script Language allows you to structure tests to:

- Describe several test cases in a single test script,
- Select a subset of test cases according to different Target Deployment Port criteria.

All instructions in a test script have the following characteristics:

- All statements begin with a keyword.
- Statements are not case sensitive.
- Statements start at the beginning of a line and end at the end of a line. You can, however, write an instruction over several lines using the ampersand (&) continuation character at the beginning of additional lines.
- Statements must be shorter than 2048 characters, although this limit may be lower on some platforms.

The basic structure of a Component Testing **.ptu** test script for Ada looks like this:

```
HEADER add, 1, 1
```

```
<variable declarations for the test script>
```

```
BEGIN
```

```
SERVICE add
```

```
<local variable declarations for the service>
```

```
TEST 1
```

```
FAMILY nominal
```

ELEMENT

VAR variable1, INIT=0, EV=0

VAR variable2, INIT=0, EV=0

#<call to the procedure under test>

END ELEMENT

END TEST

END SERVICE

Structure statements

The following statements allow you to describe the structure of a test.

- **HEADER:** For documentation purposes, specifies the name and version number of the module being tested, as well as the version number of the tested source file. This information is displayed in the test report.
- **BEGIN:** Marks the beginning of the generation of the actual test program.
- **SERVICE:** Contains the test cases related to a given service. A service usually refers to a procedure or function. Each service has a unique name (in this case add). A **SERVICE** block terminates with the instruction **END SERVICE**.
- **TEST:** Each test case has a number or identifier that is unique within the block **SERVICE**. The test case is terminated by the instruction **END TEST**.
- **FAMILY:** Qualifies the test case to which it is attached. The qualification is free (in this case **nominal**). A list of qualifications can be specified (for example: **family, nominal, structure**) in the Tester Configuration dialog box.
- **ELEMENT:** Describes a test phase in the current test case. The phase is terminated by the instruction **END ELEMENT**. The different phases of the same test case cannot be dissociated after the tests are run, unlike the test cases introduced by the instruction **NEXT_TEST**. However, the test phases introduced by the instruction **ELEMENT** are included in the loops created by the instruction **LOOP**.

The three-level structure of the test scripts has been deliberately kept simple. This structure allows:

- A clear and structured presentation of the test script and report
- Tests to be run selectively on the basis of the service name, the test number, or the test family.

In the test script, the testers can add an optional **REQUIREMENT** statement in order to link the tests to one or several requirements of the application under test.

The **REQUIREMENT** instruction appears within **TEST** blocks, where it defines the requirements for this test or within **SERVICE** blocks where it defines the requirements for the tests including in this service or before the first **SERVICE** block where it defines the requirements for the all the tests in the file.

Related Topics

[Ada Test Script Basics on page 532](#) | [Test Iterations on page 581](#)

Using native Ada statements

Component Testing for Ada

In some cases, it can be necessary to include portions of Ada native code inside a **.ptu** test script. You can use the **#**, **@**, and **!** prefixes to do this.

Analyzed native code -

When lines are prefixed with the **#** character, the Test Script Compiler **analyzes** the line and then **copies** the line into the generated code. You can use the **#** prefix to declare test script variables and to include the files that declare the functions under test.

Variable declarations must be placed outside of Ada test script blocks preferably at the beginning of scenarios and procedures.

Ignored native code - @

When lines are prefixed with the **@** character, the Test Script Compiler only **copies** the line into the test harness and does **not analyze** the line. You can use the **@** prefix to copy instructions into the test harness, when the test script compiler would not understand these instructions. Assembly instructions are examples of these instructions.

Parsed native code - !

When lines are prefixed with the **!** character, the Test Script Compiler **analyzes** the lines, but does **not copy the lines** into the test harness. You can use the **!** prefix to declare variables and types that are built into the compiler.

Related Topics

[Test script structure on page 533](#)

Testing Variables

Component Testing for Ada

One of the main features of Component Testing for Ada is its ability to compare initial values, expected values and actual values of variables during test execution. In the Ada Test Script Language, this is done with the **VAR** statement.

The **VAR** statement specifies both the test start-up procedure and the post-execution test for simple variables. This instruction uses three parameters:

- **Name of the variable under test:** this can be a simple variable, an array element, or a field of a record. It is also possible to test an entire array, part of an array or all the fields of a record.
- **Initial value of the variable:** identified by the keyword **INIT**.
- **Expected value of the variable after the procedure has been executed:** identified by the keyword **EV**.

Declare variables under test with the **VAR** statement, followed by the declaration keywords:

- **INIT =** for an assignment
- **INIT ==** for no initialization
- **EV =** for a simple test.

Component Testing for Ada allows you to define initial and expected values with standard Ada expressions.

All literal values, variable types, functions and most operators available in the Ada language are accepted by Component Testing for Ada.

It does not matter where the **VAR** instructions are located with respect to the test procedure call since the Ada code generator separates **VAR** instructions into two parts :

- The variable test is initialized with the **ELEMENT** instruction
- The actual test against the expected value is done with the **END ELEMENT** instruction

Many other forms are available that enable you to create more complex test scenarios.

Example

The following example demonstrates typical use of the **VAR** statement

```
HEADER add, 1, 1
```

```
#with add;
```

```
BEGIN
```

```
SERVICE add
```

```
# a, b, c : integer;
```

```
TEST 1
```

```
FAMILY nominal
```

```
ELEMENT
```

```
VAR a, init = 1, ev = init
```

```
VAR b, init = 3, ev = init
```

```
VAR c, init = 0, ev = 4
```

```
#c := add(a,b);
```

```
END ELEMENT
```

```
END TEST
```

```
END SERVICE
```

Related Topics

[Testing intervals on page 537](#) | [Testing tolerances on page 538](#) | [Reporting a variable without testing on page 539](#) | [Testing expressions on page 539](#)

Testing Intervals

Component Testing for Ada

You can test an expected value within a given interval by replacing **EV** with the keywords **MIN** and **MAX**.

You can also use this form on alphanumeric variables, where character strings are considered in alphabetical order ("**A**"<="**B**"<="**C**").

Example

The following example demonstrates how to test a value within an interval:

```
TEST 4
```

```
FAMILY nominal
```

```
ELEMENT
```

```
VAR a, init in (1,2,3), ev = init
```

```
VAR b, init = 3, ev = init
```

```
VAR c, init = 0, min = 4, max = 6
```

```
#c = add(a,b);
```

```
END ELEMENT
```

```
END TEST
```

Related Topics

[Testing variables on page 535](#) | [Testing intervals on page 537](#) | [Testing tolerances on page 538](#) | [Reporting a variable without testing on page 539](#) | [Testing expressions on page 539](#)

Testing Tolerances

Component Testing for Ada

You can associate a tolerance with an expected value for numerical variables. To do this, use the keyword **DELTA** with the expected value **EV**.

This tolerance can either be an absolute value (the default option) or relative (in the form of a percentage *<value>%*).

Example

```
TEST 5
```

```
FAMILY nominal
```

```
ELEMENT
```

```
VAR a, INIT in (1,2,3), EV = INIT
```

```
VAR b, INIT = 3, EV = INIT
```

```
VAR c, INIT = 0, EV = 5, DELTA = 1
```

```
#c = add(a,b);
```

```
END ELEMENT
```

```
END TEST
```

or

```
TEST 6
```

```
FAMILY nominal
```

```
ELEMENT
```

```
VAR a, INIT in (1,2,3), EV = INIT
```

```
VAR b, INIT = 3, EV = INIT
```

```
VAR c, INIT = 0, EV = 5, DELTA = 20%
```

```
#c = add(a,b);
```

```
END ELEMENT
```

```
END TEST
```

Related Topics

[Testing variables on page 535](#) | [Testing intervals on page 537](#) | [Reporting a variable without testing on page 539](#)
| [Testing expressions on page 539](#)

Testing expressions

Component Testing for Ada

To test the return value of an expression, rather than declaring a local variable to memorize the value under test, you can directly test the return value with the VAR instruction.

In some cases, you must leave out the initialization part of the instruction.

Example

The following example places the call of the **add** function in a **VAR** statement:

```
TEST 12
FAMILY nominal
ELEMENT
VAR a, init = 1, ev = init
VAR b, init = 3, ev = init
VAR add(a,b), ev = 4
END ELEMENT
FIN TEST
```

In this example, you no longer need the variable **c**. The resulting test report an *Unknown*  status indicating that it has not been tested.

All syntax examples of expected values are still applicable, even in this particular case.

Related Topics

[Testing variables on page 535](#) | [Testing intervals on page 537](#) | [Testing tolerances on page 538](#) | [Initializing without testing on page 539](#)

Initializing without testing

Component Testing for Ada

It is sometimes difficult to predict the expected result for a variable; such as if a variable holds the current date or time. In this case, you might want to avoid specifying an expected output but still have the value of the variable initialized in the test script. To do this, use the **EV ==** syntax.

Example

In the following script **a**, **b**, and **c** are initialized, but only **a** and **b** are tested.

```
TEST 7
```

```
FAMILY nominal
```

```
ELEMENT
```

```
VAR a, init in (1,2,3), ev = init
```

```
VAR b, init = 3, ev = init
```

```
VAR c, init = 0, ev ==
```

```
#c = add(a,b);
```

```
END ELEMENT
```

```
END TEST
```

Related Topics

[Testing variables on page 535](#) | [Testing intervals on page 537](#) | [Testing expressions on page 539](#) |

Declaring global variables for testing

Component Testing for Ada

The Target Deployment Ports for Ada do not provide any variables that can be used freely by the tester.

To avoid having to modify the code under test, it is easier to add an extra C package, which is actually just the *spec* part of the package, to provide a set of globally accessible variables. You can do this directly in the **.ptu** test script.

Declaring Global Variables

Any code inserted between the **HEADER** and **BEGIN** keywords is copied into the generated code as is. For example:

Header Code_Under_Test, 1.0, 1.0

```
#With Code_Under_Test; -- only if Code_Under_Test is used within My_Globals
-- this context clause goes into the package My_Globals
#package My_Globals is
# Global_Var_Integer : Integer := 0;
#end My_Globals;
```

```
#with Code_Under_Test;
#with My_Globals;
```

-- these two context clauses go into the generated test harness

Begin

-- etc..

Note Any Ada instruction between **HEADER** and the **BEGIN** instruction must be encapsulated into a procedure or a package. Context clauses are possible.

Accessing Global Variables

The extra global variable package is visible from within all units of the test driver.

Variables can be accessed like this:

```
#My_Globals.Global_Var_Integer := 1;
```

Variables can be accessed from a **DEFINE STUB** block for example:

```
Define Stub Another_Package
```

```
#with My_Globals;
```

```
#procedure some_proc (param : in out some_type) is
```

```
#begin
```

```
# My_Globals.Global_Var_Integer := 2;
```

```
#end some_proc;
```

-- however, no "return" statement is possible within this block

```
End Define
```

Variables can be accessed in the **ELEMENT** blocks, just like any other variable:

```
VAR My_Globals.Global_Var_Integer, init = 0, EV = 1
```

HCL OneTest™ Embedded processes the **.ptu** test script in such a way that global variable package automatically becomes a separate compilable unit.

Related Topics

[Testing variables on page 535](#) | [Testing intervals on page 537](#) | [Testing expressions on page 539](#) | [Handling global variables with stubs on page 568](#)

Testing arrays

Component Testing for Ada

With Component Testing for Ada, you can test arrays in quite the same way as you test variables. In the Ada Test Script Language, this is done with the **ARRAY** statement.

The **ARRAY** statement specifies both the test start-up procedure and the post-execution test for simple variables. This instruction uses three parameters:

- **Name of the variable under test:** species the name of the array in any of the following ways:
 - To test one array element, conform to the Ada syntax: **histo(0)**.
 - To test the entire array without specifying its bounds, the size of the array is deduced by analyzing its declaration. This can only be done for well-defined arrays.
 - To test a part of the array, specify the lower and upper bounds within which the test will be run, separated with two periods (..), as in: **histo(1..SIZE_HISTO)**
- **Initial value of the array:** identified by the keyword **INIT**.
- **Expected value of the array after the procedure has been executed:** identified by the keyword **EV**.

Declare variables under test with the **ARRAY** statement, followed by the declaration keywords:

- **INIT =** for an assignment
- **INIT ==** for no initialization
- **EV =** for a simple test.

It does not matter where the **ARRAY** instructions are located with respect to the test procedure call since the Ada code generator separates **ARRAY** instructions into two parts :

- The array test is initialized with the **ELEMENT** instruction
- The actual test against the expected value is done with the **END ELEMENT** instruction

Testing an Array with Ada Expressions

To initialize and test an array, specify the same value for all the array elements. The following two examples illustrate this simple form.

```
ARRAY image, init = 0, ev = init
```

```
ARRAY histo[1..SIZE_HISTO-1], init = 0, ev = 0
```

You can use the same expressions for initial and expected values as those used for simple variables (literal values, constants, variables, functions, and Ada operators).

Example

```

HEADER histo, 1, 1

#with histo; use histo;

BEGIN

SERVICE COMPUTE_HISTO

# x1, x2, y1, y2 : integer;

# histo : T_HISTO;

TEST 1

FAMILY nominal

ELEMENT

VAR x1, init = 0, ev = init

VAR x2, init = SIZE_IMAGE#1, ev = init

VAR y1, init = 0, ev = init

VAR y2, init = SIZE_IMAGE#1, ev = init

ARRAY image(1..200,1..200), init = 0, ev = init

VAR histo(1), init = 0, ev = SIZE_IMAGE*SIZE_IMAGE

ARRAY histo(1..SIZE_HISTO), init = 0, ev = 0

#compute_histo(x1, y1, x2, y2, histo);

END ELEMENT

END TEST

END SERVICE

```

Related Topics

[Testing variables on page 535](#) | [Testing an array with pseudo-variables on page 544](#) | [Testing character arrays on page 545](#) | [Testing large arrays on page 545](#) | [Testing arrays with lists on page 547](#) | [Testing arrays with other arrays on page 548](#)

Testing arrays with pseudo-variables

Component Testing for Ada

Another form of initialization consists of using one or more pseudo-variables, as the following example illustrates:

```
TEST 3

FAMILY nominal

ELEMENT

VAR x1, init = 0, ev = init

VAR x2, init = SIZE_IMAGE-1, ev = init

VAR y1, init = 0, ev = init

VAR y2, init = SIZE_IMAGE-1, ev = init

ARRAY image, init=(int)(100*(1+sin((float)(I1+I2))))), ev = init

ARRAY histo[0..200], init = 0, ev ==

ARRAY histo[201..SIZE_HISTO-1], init = 0, ev = 0

VAR status, init ==, ev = 0

#status = compute_histo(x1, y1, x2, y2, histo);

END ELEMENT

END TEST
```

I1 and **I2** are two pseudo-variables which take as their value the current values of the array indices (for image, from **0** to **199** for **I1** and **I2**). You can use these pseudo-variables like a standard variable in any Ada expression.

This allows you to create more complex test scripts when using large arrays when the use of enumerated expressions is limited.

For multidimensional arrays, you can combine these different types of initialization and test expressions, as demonstrated in the following example:

```
ARRAY image, init = (0 => I2, 1 => ( 0 => 100, others => 0 ),
& others => (I1 + I2) % 255 )
```

Related Topics

[Testing variables on page 535](#) | [Testing arrays on page 542](#) | [Testing character arrays on page 545](#) | [Testing large arrays on page 545](#) | [Testing arrays with lists on page 547](#) | [Testing arrays with other arrays on page 548](#)

Testing Character Arrays

Component Testing for Ada

Character arrays are a special case. Variables of this type are processed as character strings delimited by quotes.

You therefore need to initialize and test character arrays using character strings, as the following list example illustrates.

If you want to test character arrays like other arrays, you must use a format modification declaration (FORMAT instruction) to change them to arrays of integers.

Example

The following list example illustrates this type of modification:

```
TEST 2
```

```
FAMILY nominal
```

```
ELEMENT
```

```
VAR l, init = NIL, ev = NONIL
```

```
VAR s, init = "foo", ev = init
```

```
VAR l.str(1..5), init = "foo", ev = ('f','o','o')
```

```
#l := stack(s, l);
```

```
END ELEMENT
```

```
END TEST
```

Related Topics

[Testing variables on page 535](#) | [Testing arrays on page 542](#) | [Testing an array with pseudo-variables on page 544](#) | [Testing large arrays on page 545](#) | [Testing arrays with lists on page 547](#) | [Testing arrays with other arrays on page 548](#)

Testing large arrays

Component Testing for Ada

The maximum number of array elements that can be processed is 100. If you need to test an array that contains more than 100 elements, then you must split the initialization of the array over two or more initializations, as shown in the following example.

Example

The following initialization produces a **Too many INIT or VA values** error:

```
ARRAY a, init=
(1,2,3,4,5,6,7,8,9,10,1,2,3,4,5,6,7,8,9,20,1,2,3,4,5,6,7,8,9,30,1,2,3,4,
5,6,7,8,9,40,1,2,3,4,5,6,7,8,9,50,1,2,3,4,5,6,7,8,9,60,1,2,3,4,5,6,7,8,9,
70,1,2,3,4,5,6,7,8,9,80,1,2,3,4,5,6,7,8,9,90,1,2,3,4,5,6,7,8,9,100,1,2,3,
4,5,6,7,8,9,110,1,2,3,4,5,6,7,8,9,120,1,2,3,4,5,6,7,8,9,130,1,2,3,4,5,6,
7,8,9,140,1,2,3,4,5,6,7,8,9,150,1,2,3,4,5,6,7,8,9,160,1,2,3,4,5,6,7,8,9,
170,1,2,3,4,5,6,7,8,9,180,1,2,3,4,5,6,7,8,9,190,1,2,3,4,5,6,7,8,9,200)
, ev=init
```

Instead, use the following expression:

```
ARRAY z [0..99],
init=(1,2,3,4,5,6,7,8,9,10,1,2,3,4,5,6,7,8,9,20,1,2,3,4,5,6,7,8,9,30,1,2,
3,4,5,6,7,8,9,40,1,2,3,4,5,6,7,8,9,50,1,2,3,4,5,6,7,8,9,60,1,2,3,4,5,6,
7,8,9,70,1,2,3,4,5,6,7,8,9,80,1,2,3,4,5,6,7,8,9,90,1,2,3,4,5,6,7,8,9,100)
, ev=init
```

```
ARRAY z [100..199],
init={1,2,3,4,5,6,7,8,9,110,1,2,3,4,5,6,7,8,9,120,1,2,3,4,5,6,7,8,9,130,
1,2,3,4,5,6,7,8,9,140,1,2,3,4,5,6,7,8,9,150,1,2,3,4,5,6,7,8,9,160,1,2,3,
4,5,6,7,8,9,170,1,2,3,4,5,6,7,8,9,180,1,2,3,4,5,6,7,8,9,190,1,2,3,4,5,6,
7,8,9,200}
, ev=init
```

Related Topics

[Testing variables on page 535](#) | [Testing arrays on page 542](#) | [Testing an array with pseudo-variables on page 544](#) | [Testing character arrays on page 545](#) | [Testing arrays with lists on page 547](#) | [Testing arrays with other arrays on page 548](#)

Testing arrays with lists

Component Testing for Ada

While an expression initializes all the array elements in the same way, you can also initialize each element by using an enumerated list of expressions between brackets "()" . In this case, you must specify a value for each array element.

Furthermore, you can precede every element in this list of initial or expected values with the array index of the element concerned followed by the characters "=>" . The following example illustrates this form:

```
ARRAY histo[0..3], init = (0 => 0, 1 => 10, 2 => 100, 3 => 10) ...
```

This form of writing the **ARRAY** statement has several advantages:

- Improved readability of the list
- Ability to mix values without worrying about the order

You can also use this form together with the simple form if you follow this rule: once one element has been defined with its array index, you must do the same with all the following elements.

If several elements in an array are to take the same value, specify the range of elements taking this value as follows:

```
ARRAY histo[0..3], init = ( 0 .. 2 => 10, 3 => 10 ) ...
```

You can also initialize and test multidimensional arrays with a list of expressions, as follows. In this case, the previously mentioned rules apply to each dimension.

```
ARRAY image, init = (0, 1=>4, others=>(1, 2, others=>100)) ...
```

Example

You can specify a value for all the as yet undefined elements by using the keyword `others`, as the following example illustrates:

```
TEST 2
```

```
FAMILY nominal
```

```
ELEMENT
```

```
VAR x1, init = 0, ev = init
```

```
VAR x2, init = SIZE_IMAGE-1, ev = init
```

```
VAR y1, init = 0, ev = init
```

```
VAR y2, init = SIZE_IMAGE-1, ev = init
```

```

ARRAY image, init = (others=>(others=>100)), ev = init
ARRAY histo, init = 0,
& ev = (100=>SIZE_IMAGE*SIZE_IMAGE, others=>0)
VAR status, init ==, ev = 0
#status = compute_histo(x1, y1, x2, y2, histo);
END ELEMENT
END TEST

```

Related Topics

[Testing variables on page 535](#) |
 [Testing arrays on page 542](#) |
 [Testing an array with pseudo-variables on page 544](#) |
 [Testing character arrays on page 545](#) |
 [Testing large arrays on page 545](#) |
 [Testing arrays with other arrays on page 548](#)

Testing arrays with other arrays

Component Testing for Ada

Component Testing for Ada is flexible enough to allow complex array comparisons. You can initialize or compare an array with another array that shares the same declaration.

You can use this form of initialization and testing with one or more array dimensions.

Example

The following example tests the two arrays **read_image** and **extern_image**, which have been declared in the same way. Every element from the **extern_image** array is assigned to the corresponding **read_image** array element.

```

TEST 4
FAMILY nominal
#read_image(extern_image,"image.bmp");
ELEMENT
VAR x1, init = 0, ev = init
VAR x2, init = SIZE_IMAGE-1, ev = init
VAR y1, init = 0, ev = init
VAR y2, init = SIZE_IMAGE-1, ev = init

```

```

ARRAY image, init = extern_image, ev = init

ARRAY histo, init = 0, ev ==

VAR status, init ==, ev = 0

#status = compute_histo(x1, y1, x2, y2, histo);

END ELEMENT

END TEST

```

Related Topics

[Testing variables on page 535](#) | [Testing arrays on page 542](#) | [Testing an array with pseudo-variables on page 544](#) | [Testing character arrays on page 545](#) | [Testing large arrays on page 545](#) | [Testing arrays with lists on page 547](#)

Testing Records

Component Testing for Ada

To test all the fields of a structured variable or record, use a single **STR** instruction to define the initializations and expected values of the structure.

The **STR** statement specifies both the test start-up procedure and the post-execution test for simple variables. This instruction uses three parameters:

- **Name of the variable under test:** this can be a simple variable, an array element, or a field of a record. It is also possible to test an entire array, part of an array or all the fields of a record.
- **Initial value of the variable:** identified by the keyword **INIT**.
- **Expected value of the variable after the procedure has been executed:** identified by the keyword **EV**.

Declare variables under test with the **STR** statement, followed by the declaration keywords:

- **INIT =** for an assignment
- **INIT ==** for no initialization
- **EV =** for a simple test.

It does not matter where the **STR** instructions are located with respect to the test procedure call since the Ada code generator separates **STR** instructions into two parts :

- The variable test is initialized with the **ELEMENT** instruction
- The actual test against the expected value is done with the **END ELEMENT** instruction

Many other forms are available that enable you to create more complex test scenarios.

Example

The following example demonstrates typical use of the **STR** statement:

```
--procedure push(l: in out list; s:string);
```

```
TEST 2
```

```
FAMILY nominal
```

```
ELEMENT
```

```
VAR l, init = NIL, ev = NONIL
```

```
STR l.all, init == , ev = ("myfoo",NIL,NIL)
```

```
VAR s, init = "myfoo", ev = init
```

```
#push(l,s);
```

```
END ELEMENT
```

```
END TEST
```

Related Topics

[Testing Records on page 549](#) | [Testing a Record with Ada Expressions on page 550](#) | [Testing a Record with Another Record on page 551](#) | [Testing Records with Discriminants on page 552](#) | [Testing Tagged Records on page 553](#) | [No Test on page 555](#)

Testing a Record with Ada Expressions

Component Testing for Ada

To initialize and test a structured variable or record, you must initialize or test all the fields using a list of native language expressions (one per field). The following example illustrates this form:

```
STR l.all, init == , ev = ("myfoo",NIL,NIL)
```

Each element in the list must correspond to the structured variable field as it was declared.

Every expression in the list must obey the rules described so far, according to the type of field being initialized and tested:

- An expression for simple fields or arrays of simple variables initialized using an expression
- In Ada, an aggregate for fields of type record or array

Using Field Names in Native Expressions

As with arrays, you can specify field names in native expressions by following the field name of the structure with the characters =>, as follows:

```
TEST 3

FAMILY nominal

ELEMENT

VAR l, init = NIL, ev = NONIL

VAR l.all, init == , ev = (str=>"myfoo",next=>NIL,prev=>NIL)

VAR s, init = "myfoo", ev = init

#l = push(l,s);

END ELEMENT

END TEST
```

When using this form, you do not have to respect the order of expressions in the list.

Related Topics

[Testing Records on page 549](#) | [Testing a Record with Another Record on page 551](#) | [Testing Records with Discriminants on page 552](#) | [Testing Tagged Records on page 553](#) | [No Test on page 555](#)

Testing a Record with Another Record

Component Testing for Ada

As with arrays, you can initialize and test a record using another record of the same type. The following example illustrates this form:

```
STR l.all, init == , ev = l1.all
```

Each field of the structured variable will be initialized or tested using the associated fields of the variable used for initialization or testing.

Related Topics

[Testing Records on page 549](#) | [Testing a Record with Ada Expressions on page 550](#) | [Testing Records with Discriminants on page 552](#) | [Testing Tagged Records on page 553](#) | [No Test on page 555](#)

Testing Records with Discriminants

Component Testing for Ada

You can use record types with discriminants, with the following Ada restrictions:

- The initialization part must be complete.
- The evaluation can omit every field except discriminant fields.

Initialization and expected value expressions are Ada aggregates beginning with the value of the discriminant.

Example

Ada example:

```
type rec (discr:boolean:=TRUE)
case discr is
when TRUE =>
ch2:float;
when FALSE =>
ch3:integer;
end case;
end record;
```

Test Script Sample:

```
#r1: rec(TRUE);
#r2: rec;
TEST 1
FAMILY nominal
ELEMENT
var r1, init = (TRUE, 0.0), ev ==
var r2, init = (FALSE, 1), ev = (TRUE, 1.0)
#func (r);
END ELEMENT
```

END TEST

Related Topics

[Testing Records on page 549](#) | [Testing a Record with Ada Expressions on page 550](#) | [Testing a Record with Another Record on page 551](#) | [Testing Tagged Records on page 553](#) | [No Test on page 555](#)

Testing Tagged Records

Component Testing for Ada

Component Testing for Ada supports tagged record types. As with other classic records, you can omit a field in the initialization or evaluation part. You can also define tagged types with a discriminant part. In such cases, the only limitation is that of the discriminant.

Example

The following example illustrates tagged records. First, the source code:

```
Package Items Is
```

```
Type Item Is Tagged Record
```

```
X_Coord : Float;
```

```
Y_Coord : Float;
```

```
End Record;
```

```
Procedure foo_test;
```

```
End Items;
```

```
With Items; Use Items;
```

```
Package Forms Is
```

```
Type Point Is New Item With Null Record;
```

```
Type Circle Is New Item With Record
```

```
Radius : Float;
```

```
End Record;
```

```
Type Triangle Is New Item With Record
```

```
A,B,C : Float;
```

```
End Record;
```

Type Cylinder Is New Circle With Record

Height : Float;

End Record;

End Forms;

Following is the associated test script:

HEADER Items, ,

#With Items; Use Items;

#With Forms; Use Forms;

BEGIN Items

#I : Item := (1.0,0.5);

#C : Circle := (0.0,1.0,13.5);

#T : Triangle;

#P : Point;

#Cyl : Cylinder;

SERVICE Compute_Items

SERVICE_TYPE extern

TEST 1

FAMILY Nominal

ELEMENT

Var T, Init = (0.0,1.5,4.5,5.0,6.5), Ev = (I with A=>4.0, B=>5.0, C=>6.0)

Var P, Init = I, Ev = (Y_coord => 1.0, X_coord => 0.0)

Var I, Init = (0.0,1.0), Ev = Item(C)

Var P, Init = (I with NULL record), Ev = (Y_coord => 1.0, X_coord => 0.0)

End Element

END TEST -- Test 1

TEST 2

FAMILY Nominal

ELEMENT

Var I, Init = (2.0,3.0), Ev ==

Var T, Init = (2.0,3.0,4.0,5.0,6.0), Ev = (I with A=>4.0, B=>5.0, C=>6.0)

Var Cyl, Init = (2.0, 3.0, 4.0, 5.0), Ev ==

Var I, Init ==, Ev = Item(Cyl)

END ELEMENT

END TEST -- Test 2

END SERVICE -- Compute_Items

Related Topics

[Testing Records on page 549](#) | [Testing a Record with Ada Expressions on page 550](#) | [Testing a Record with Another Record on page 551](#) | [Testing Records with Discriminants on page 552](#) | [No Test on page 555](#)

No Test

Component Testing for Ada

You can only initialize and test records with the following forms:

- INIT =
- INIT ==
- EV =
- EV ==

If a field of a structured variable needs to be initialized or tested in a different way, you can omit its initial and expected values from the global test of the structured variable, and run a separate test on this field.

The following example illustrates this:

TEST 4

FAMILY nominal

ELEMENT

VAR I, init = NIL, ev = NONIL

```
VAR l.all, init == , ev = (next=>NIL,prev=>NIL)
```

```
VAR s, init in ("foo","bar"), ev = init
```

```
VAR l.str, init ==, ev(s) in ("foo","bar")
```

```
#push(l,s);
```

```
END ELEMENT
```

```
END TEST
```

Related Topics

[Testing Records on page 549](#) | [Testing a Record with Ada Expressions on page 550](#) | [Testing a Record with Another Record on page 551](#) | [Testing Records with Discriminants on page 552](#) | [Testing Tagged Records on page 553](#)

Stub Simulation

Component Testing for Ada

Stub simulation is based on the idea that subroutines to be simulated are replaced with other subroutines generated in the test driver. These simulated subroutines are often referred to as *stubs*.

Stubs use the same interface as the simulated subroutines, only the body of the subroutine is replaced.

Stubs have the following roles:

- Check **in** and **in out** parameters against the simulated subroutine. If there is a mismatch, the values are stored.
- Assign **out** and **in out** parameters from the simulated procedure
- Return a value for a simulated function

To generate stubs, the Test Script Compiler needs to know the specification of the compilation units that are to be simulated.

Passing parameters by pointer can lead to problems of ambiguity regarding the data actually passed to the function. For example, a parameter that is described in a prototype by `int *x` can be passed in the following way:

```
int *x as input ==> f(x)
```

```
int x as output or input/output ==> f(&x)
```

```
int x[10] as input ==> f(x)
```

```
int x[10] as output or input/output ==> f(x)
```

Therefore, to define a stub, you must specify the following information:

- The data type in the calling function
- The method of passing the data

Example

An example project called Stub Ada is available from the Examples section of the Start page. This example demonstrates the use of stubs in Component Testing for Ada. See Example projects for more information.

Related Topics

[Defining stubs on page 557](#) | [Using Stubs on page 559](#) | [Ada Syntax Extensions on page 561](#) | [Advanced Stubs \(Ada\) on page 563](#) | [Example projects on page 677](#)

Defining stubs

Component Testing for Ada

The following example highlights the simulation of all functions and procedures declared in the specification of file_io. A new body is generated for file_io in file <testname>_fct_simule.ada.

```
HEADER file, 1, 1
```

```
BEGIN
```

```
DEFINE STUB file_io
```

```
END DEFINE
```

You must always define stubs after the **BEGIN** instruction and outside any **SERVICE** block.

Simulation of Generic Units

You can stub a generic unit like an ordinary unit with the following restrictions:

Parameters of a procedure or function, and function return types of a type declared in a generic unit or parameter of this unit must use the **_NO** mode.

For example, if you want to stub the following generic package:

```
GENERIC
```

```
TYPE TYPE_PARAM is .....
```

```
Package GEN is
```

```

TYPE TYPE_INT0 is ...;

procedure PROC(x:TYPE_PARAM,y:in out TYPE_INT0,Z:out integer);

function FUNC return TYPE_INT0;

end GEN;

```

Use the following stub definition:

```

DEFINE STUB GEN

# procedure PROC(x: _NO TYPE_PARAM,y: _NO TYPE_INT0,Z:out integer);

# function FUNC return _NO TYPE_INT0;

END DEFINE

```

You can add a body to procedures and functions to process any parameters that required the **_NO** mode.

Note With some compilers, when stubbing a unit by using a **WITH** operator on the generic package, cross dependencies may occur.

Separate Body Stub

In some cases, you might need to define the body stub separately, with a proprietary behavior. Declare the stub separately as shown in the following example, and then you can define a body for it:

```

DEFINE STUB <STUB NAME>

# procedure My_Procedure(...) is separate ;

END DEFINE

```

The Ada Test Script Compiler will not generate a body for the service *My_Procedure*, but will expect you to do so.

Initializing variables with a stub

When a stub that returns a value is called before the main program initialization (for example if the stubbed function is used to initialize a variable), the initialization should be declared in the stub as in the following example:

```

DEFINE STUB PACKAGE_1

# Function Proc_1 ... return <type> is

# BEGIN

# if Attol_f_idx < 0

```

```
# THEN
# RETURN <value of type>;
# end if;
# END;
END DEFINE
```

Related Topics

[Stub Simulation on page 556](#) | [Using Stubs on page 559](#) | [Sizing Stubs on page 562](#) | [Ada Syntax Extensions on page 561](#) | [Advanced Stubs \(Ada\) on page 563](#)

Using Stubs

Component Testing for Ada

Range of Values of STUB Parameters

When using stubs, you may need to define an authorized range for each **STUB** parameter. Furthermore, you can summarize several calls in one line associated with this parameter.

Write such **STUB** lines as follows:

```
STUB F 1..10 => (1<->5)30
```

This expression means that the **STUB F** will be called 10 times with its parameter having a value between 1 and 5, and its return value is always 30.

You can combine this with several lines; the result looks like the following example:

```
STUB F 1..10 => (1<->5)30,
```

```
& 11..19 => (1<->5)0,
```

```
& 20..30 => (<->) 1,
```

```
& others =>(<->)-1
```

To check that a **STUB** is never called, even if an **ENVIRONMENT** containing the **STUB** is used, use the the following syntax:

```
STUB F 0=>(<->)
```

Raise-exception Stubs

You can force to raise a user-defined (or pre-defined) exception when a **STUB** is called with particular values.

The appropriate syntax is as follows:

```
STUB P(1E+307<->1E+308) RAISE STORAGE_ERROR
```

If the **STUB F** happens to be called with its parameter between **1E+307** and **1E+308**, the exception **STORAGE_ERROR** will be raised during execution of the application; the test will be **FALSE** otherwise.

Suppose that the current stubbed unit contains at least one overloaded sub-program. When calling this particular **STUB**, you will need to qualify the procedure or function. You can do this easily by writing the **STUB** as follows:

```
STUB A.F (1<->2:REAL)RAISE STANDARD.CONSTRAINT_ERROR
```

The **STUB A.F** is called once and will raise a **CONSTRAINT_ERROR** if its parameter, of type **REAL**, has a value between **1** and **2**.

Compilation Sequence

The Ada Test Script Compiler generates three files:

- `<testname>_fct_simule.ada` for the body of simulated functions and procedures
- `<testname>_var_simule.ada` for the declaration of simulation variables
- `<testname>_var_simule_B.ada` for the body of test procedures

You must compile your packages in the following order:

1. Simulated unit (specification)
2. `<testname>_var_simule.ada`
3. `<testname>_var_simule_B.ada`
4. Test program
5. `<testname>_fct_simule.ada`

Replacing Stubs

Stubs can be used to replace a component that is still in development. Later in the development process, you might want to replaced a stubbed component with the actual source code.

To replace a stub with actual source code:

1. Right-click the test node and select Add Child and Files
2. Add the source code files that will replace the Stubbed functions.
3. If you do not want a new file to be instrumented, right-click the file select Properties. Set the Instrumentation property to No.
4. Open the **.ptu** test script, and remove the **STUB** sections from your script file.

Related Topics

[Stub Simulation on page 556](#) | [Defining Stubs on page 557](#) | [Sizing Stubs on page 562](#) | [Ada Syntax Extensions on page 561](#) | [Advanced Stubs \(Ada\) on page 563](#)

Multiple stub calls

Component Testing for Ada

For a large number of calls to a stub, use the following syntax for a more compact description:

```
<call i> .. <call j> =>
```

You can describe each call to a stub by adding the specific cases before the preceding instruction, for example:

```
<call i> =>
```

or

```
<call i> .. <call j> =>
```

The call count starts at 1 as the following example shows:

```
TEST 2
```

```
FAMILY nominal
```

```
COMMENT Reading of 100 identical lines
```

```
ELEMENT
```

```
VAR file1, init = "file1", ev = init
```

```
VAR file2, init = "file2", ev = init
```

```
VAR s, init == , ev = 1
```

```
STUB open_file 1=>("file1")3
```

```
STUB create_file 1=>("file2")4
```

```
STUB read_file 1..100(3,"line")1, 101=>(3,"")0
```

```
STUB write_file 1..100=>(4,"line")1
```

```
STUB close_file 1=>(3)1,2=>(4)1
```

```
#s = copy_file(file1,file2);
```

```
END ELEMENT
```

END TEST

Several Calls to a Stub

If a stub is called several times during a test, either of the following are possible:

- Describe the different calls in the same **STUB** instruction, as described previously.
- Use several **STUB** instructions to describe the different calls. (This allows a better understanding of the test script when the **STUB** calls are not consecutive.)

The following example rewrites the test to use this syntax for the call to the **STUB close_file**:

```
STUB close_file (3)1
```

```
STUB close_file (4)1
```

No Testing of the Number of Calls of a Stub

If you don't want to test the number of calls to a stub, you can use the keyword **others** in place of the call number to describe the behavior of the stub for the calls to the stub not yet described.

For example, the following instruction lets you specify the first call and all the following calls without knowing the exact number:

```
STUB write_file 1=>(4,"line")1,others=>(4,"")1
```

Related Topics

[Stub Simulation on page 556](#) | [Defining Stubs on page 557](#) | [Using Stubs on page 559](#) | [Sizing Stubs on page 562](#) | [Advanced Stubs on page 563](#)

Stub memory allocation

Component Testing for Ada

For each **STUB**, the Component Testing feature allocates memory to:

- Store the value of the input parameters during the test
- Store the values assigned to output parameters before the test

A stub can be called several times during the execution of a test. By default, when you define a **STUB**, the Component Testing feature allocates space for 10 calls. This means that only the 10 first errors found in stub calls are displayed in the report and that any more errors are ignored. If you call the **STUB** more than 10 times, then you must specify the number of expected calls in the **STUB** declaration statement.

In the following example, the script allocates storage space for the first 17 calls to the stub:

```

DEFINE STUB file 17

#procedure proc_inout (param1 : in out integer) is

#begin

# param1:=param1+1

#end proc_inout

END DEFINE

```

You can also reduce the stub allocation value to a lower value when running tests on a target platform that is short on memory resources.

Related Topics

[Stub Simulation on page 556](#) | [Defining Stubs on page 557](#) | [Using Stubs on page 559](#) | [Ada Syntax Extensions on page 561](#) | [Advanced Stubs \(Ada\) on page 563](#)

Advanced Stubs

Component Testing for Ada

This section covers some of the more complex notions when dealing with stub simulations in Component Testing for Ada.

To learn about	See
Writing complex stubs in Ada	Native Code in Stubs on page 563
Defining stubs of generic Ada units	Simulating Generic Units on page 566
Stubbing functions that take arrays in _inout mode	Simulating Functions with _inout Mode Arrays on page 567
Stubbing functions for which the number of parameters may vary	Simulating Functions with Varying Parameters on page 568
Defining a stub in a separate body	Separate Body Stub on page 570

Creating complex stubs

Component Testing for Ada

If necessary, you can make stub operation more complex by inserting native Ada code into the body of the simulated function. You can do this easily by adding the lines of native code after the prototype.

Example

The following stub definition makes extensive use of native Ada code.

```
DEFINE STUB file

#function open_file(f:string) return file_t is

#begin

# raise file_error;

#end;

END DEFINE
```

Related Topics

[Advanced Stubs on page 563](#) | [Excluding a Parameter from a Stub on page 564](#) | [Simulating Generic Units on page 566](#) | [Simulating Functions with _inout Mode Arrays on page 567](#) | [Simulating Functions that Use a Variable Number of Parameters on page 568](#) | [Separate Body Stub on page 570](#)

Excluding a parameter from a stub

Component Testing for Ada

You can specify in the stub definition that a particular parameter is not to be tested or given a value. This is done using a modifier of type **no** instead of **in**, **out** or **in out**.

Note You must be careful when using **_no** on an output parameter, as no value will be assigned. It will then be difficult to predict the behavior of the function under test on returning from the stub.

Example

In this example, the **f** parameters to **read_file** and **write_file** are never tested.

```
DEFINE STUB file

#procedure read_file(f: _no file_t; l:out string; res:out BOOLEAN);

#procedure write_file(f: _no file_t, l : string);

END DEFINE
```

Related Topics

[Advanced Stubs on page 563](#) | [Native Code in Stubs on page 563](#) | [Simulating Generic Units on page 566](#) | [Simulating Functions with _inout Mode Arrays on page 567](#) | [Simulating Functions that Use a Variable Number of Parameters on page 568](#) | [Separate Body Stub on page 570](#)

Stubbing separate compilation units

It is possible to create stubs for separate compilation units, such as procedures or packages, even for protected packages.

For the stubbing of a protected object to work, you must either:

- Stub the package containing the protected object, or
- A body exists for the package in which the protected body is declared as separate.

To stub a protected object you must use the following syntax:

```
DEFINE STUB SEPARATE(<package>) <compilation unit>
```

```
...
```

```
END DEFINE
```

If the compilation unit does contain an **entry** statement, the **entry** itself cannot be stubbed. In this case you must define the **entry** body within the DEFINE STUB block as in the following example:

```
DEFINE STUB SEPARATE(<package>) <compilation unit>
```

```
# entry body E1 ... is ...
```

```
END DEFINE
```

Example

The following example is a **.ptu** test script implementing a stub of a separate compilation unit. It is available in the **StubAda** example project provided with the product.

```
HEADER PARENT, ,
```

```
#With PARENT;
```

```
BEGIN
```

```
DEFINE STUB package
```

```
END DEFINE
```

```
DEFINE STUB SEPARATE(package) MY_VALUE
```

```
END DEFINE
```

```
SERVICE SOMETHING
```

```
SERVICE_TYPE extern
```

```
-- Declaration of service's parameters
```

```
#X : INTEGER;
```

```
#Ret : INTEGER;
```

```
TEST 1
```

```
FAMILY nominal
```

```
ELEMENT
```

```
-- stub of the protected object "get"
```

```
STUB My_Value.Get()2
```

```
Var X, Init = 0, ev = Init
```

```
Var Ret, Init = 0, ev = 2
```

```
#Ret := PARENT.SOMETHING(X);
```

```
END ELEMENT
```

```
END TEST -- TEST 1
```

```
END SERVICE -- SOMETHING
```

Related Topics

[Advanced stubs on page 563](#) | [Creating Complex Stubs on page 563](#)

Stubbing generic units

Component Testing for Ada

You can stub generic units just as ordinary units by using the following syntax:

```
DEFINE STUB STUB_NAME < dimension>
```

```
# optional declarations
```

```
END DEFINE
```

The Unit Testing Ada Test Script Compiler generates a stub body for this unit to perform the desired simulations.

Related Topics

[Advanced Stubs on page 563](#) | [Native Code in Stubs on page 563](#) | [Excluding a Parameter from a Stub on page 564](#) | [Simulating Functions with _inout Mode Arrays on page 567](#) | [Simulating Functions that Use a Variable Number of Parameters on page 568](#) | [Separate Body Stub on page 570](#)

Simulating functions with _inout mode arrays

Component Testing for Ada

To stub a function that takes an array in **_inout** mode, you must provide storage space for the actual parameters of the function.

The function prototype in the **.ptu** test script remains as usual:

```
#extern void function(unsigned char *table);
```

The **DEFINE STUB** statement however is slightly modified:

```
DEFINE STUB Funct
```

```
#void function(unsigned char _inout table[10]);
```

```
END DEFINE
```

The declaration of the pointer as an array with explicit size is necessary to memorize the actual parameters when calling the stubbed function. For each call you must specify the exact number of required array elements.

```
ELEMENT
```

```
STUB Funct.function 1 => (({'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 0x0},
```

```
& {'i', 'h', 'g', 'f', 'e', 'd', 'c', 'b', 'a', 0x0}))
```

```
#call_the_code_under_test();
```

```
END ELEMENT
```

This naming convention compares the actual values and not the pointers.

The following line shows how to pass **_inout** parameters:

```
{{<in_parameter>},{<out_parameter>}}
```

Related Topics

[Advanced Stubs on page 563](#) | [Native Code in Stubs on page 563](#) | [Excluding a Parameter from a Stub on page 564](#) | [Simulating Generic Units on page 566](#) | [Simulating Functions that Use a Variable Number of Parameters on page 568](#) | [Separate Body Stub on page 570](#)

Handling global variables with stubs

Component Testing for Ada

You can enable stubs to manipulate global variables in your test. For example, you can use this method to test a system input function that sets a value to a global variable.

In the stub declaration, declare an alias for the global variable in a procedure or function, as in the following examples:

```
DEFINE STUB tostub

# with toto;

# procedure pr1(x:integer)[alias1:in out va];

# function return_false(param: in boolean) [alias1:in out va,alias2:in out toto.a] return BOOLEAN;

END DEFINE
```

To use the global variable, specify the behavior of the global variable as in the following examples:

```
STUB tostub.pr1(5)[(3,6)]

...

STUB return_false 1=>(FALSE)[alias1=>(8,0),alias2=>((1,3),(4,5))] TRUE

STUB return_false 2=>(TRUE)[(0,10),((1,3),(7,9))] TRUE
```

Related Topics

[Testing variables on page 535](#) | [Defining stubs on page 557](#) | [Declaring global variables for testing on page 540](#)

Stubbing functions with varying parameters

Component Testing for Ada

In some cases, functions may be designed to accept a variable number of parameters on each call.

You can still stub these functions with the Component Testing feature by using the '...' syntax indicating that there may be additional parameters of unknown type and name.

In this case, Component Testing can only test the validity of the first parameter.

Example

The standard *printf* function is a good example of a function that can take a variable number of parameters:

```
int printf (const char* param, ...);
```

Here is an example including a STUB of the *printf* function:

```
HEADER add, 1, 1

#extern int add(int a, int b);

##include <stdio.h>

BEGIN

DEFINE STUB MultiParam

#int printf (const char param[200], ...);

END DEFINE

SERVICE add

#int a, b, c;

TEST 1

FAMILY nominal

ELEMENT

VAR a, init = 1, ev = init

VAR b, init = 3, ev = init

VAR c, init = 0, ev = 4

STUB printf("hello %s\n")12

#c = add(a,b);

END ELEMENT

END TEST

END SERVICE
```

Related Topics

[Advanced Stubs on page 563](#) | [Native Code in Stubs on page 563](#) | [Excluding a Parameter from a Stub on page 564](#) | [Simulating Generic Units on page 566](#) | [Simulating Functions with _inout Mode Arrays on page 567](#) | [Separate Body Stub on page 570](#)

Stubbing a body separately

Component Testing for Ada

Under certain circumstances, it may be useful to define the body stub separately, with a proprietary behavior.

To do this, declare the stub separately and then define a body for it.

Example

In the following example, Component Testing for Ada will not generate a body for the service `My_Procedure`, but will expect you to do so:

```
DEFINE STUB <STUB NAME>
```

```
# procedure My_Procedure(...) is separate ;
```

```
END DEFINE
```

Related Topics

[Advanced Stubs on page 563](#) | [Native Code in Stubs on page 563](#) | [Excluding a Parameter from a Stub on page 564](#) | [Simulating Generic Units on page 566](#) | [Simulating Functions with _inout Mode Arrays on page 567](#) | [Simulating Functions that Use a Variable Number of Parameters on page 568](#)

Advanced Ada testing

Component Testing for Ada

This section covers some of the more complex notions behind Component Testing for Ada.

To learn about	See
Internal procedures, internal variables and private variables	Testing Internal Procedures and Internal and Private Variables on page 571
Generic units	Testing Generic Compilation Units on page 571
Initializing and testing pointer variables	Testing Pointer Variables while Preserving the Pointed Value on page 575
Testing in a asynchronous environment	Testing Tasks on page 576
Making private type structures, internal procedures and variables visible to the test program	Separate Compilation on page 578
Generating the test harness as a separate package	Generating a Separate Test Harness on page 578

Macros definition conditions	Test Script Compiler Macro Definitions on page 580
Invalid values produced by the Ada Component Testing Wizard.	Unknown Values on page 580
Link tests to Requirement.	Requirement. on page 581

Testing Internal Procedures and Internal and Private Variables

Component Testing for Ada

Black box testing is not sufficient as soon as you want to test the following:

- Internal procedures of packages
- Internal variables of packages
- Private type variables

For packages, you can test internal procedures via external procedures. However, it is sometimes easier to test them directly.

You cannot modify or test internal variables with a black box approach. Internal variables are generally tested via external procedures, but it is sometimes easier to modify and test them directly also.

Private type variables are also a problem because their structure is not visible from outside the package.

Testing Generic Compilation Units

Component Testing for Ada

Types and objects in a generic unit depend on generic formal parameters that are not known by the Test Script Compiler. Therefore, Component Testing for Ada cannot directly test a generic package.

To test a generic package, you must first instantiate the package and then call the instance. Such instances must appear in compilation units or at the beginning of the test script (in any case before the BEGIN statement), as follows:

```
WITH <generic>;
```

```
PACKAGE <instance> IS NEW <generic> (...);
```

Depending on the nature of the source code under test, there are two ways to test an instantiation of a generic package:

- If the code cannot contain a specific procedure for testing purposes and the test does not need access to internal variables, then the test body can be generated as an external package. The test body can view the instance under test through the use of a **WITH** instruction.

In the **.ptu** test script, after the generic instantiation, add the **WITH <instance>** ; statement before the **BEGIN** keyword. For example:

```
WITH <Generic_Package>;
```

```
PACKAGE <Instance> IS NEW <Generic_Package> (...);
```

```
WITH <Instance>;
```

```
BEGIN
```

where *<Generic_Package>* is the name of the generic unit under test, and *<Instance>* is the name of the instantiated unit from the generic.

- If you need to test private types within the generic package and the test needs access to all internal variables, then the test body must be part of the generic package as a specific test procedure.

In the **.ptu** test script, specify the generic package, the instance package and the test procedure on the **BEGIN** line. For example:

```
WITH <Generic_Package>;
```

```
PACKAGE <Instance> IS NEW <Generic_Package> (...);
```

```
BEGIN GENERIC(<Generic_Package>, <Instance>), <Procedure_Name>
```

where *<Generic_Package>* is the name of the generic unit under test, and *<Instance>* is the name of the instantiated unit from the generic. The *<Procedure_Name>* parameter is not mandatory. Component Testing uses **Attol_Test** by default.

This instruction generates the test body into *<Procedure_Name>* as a separate unit of the Generic package as well as the **WITH** to this instance, as requested by the test body.

If specified, *<Procedure_Name>* must be part of the generic package as separate procedure.

Example

Consider the following Ada compilation unit:

```
Generic
```

```
Type t is private ;
```

```
Procedure swap(x,y :in out t) ;
```

```
Procedure swap(x,y :in out t) is
```

```
Z :t ;
```

```
Begin
```

```

Z := x ;

X := y;

Y := z;

End swap ;

With swap ;

Procedure swap_integer is new swap(integer) ;

```

You can test the ***swap_integer*** procedure just like any other procedure:

```

HEADER swap_integer,,
#with swap_integer;

BEGIN

SERVICE swap_integer

#x,y:integer;

TEST 1

FAMILY nominal

ELEMENT

VAR x , init = 1 , ev = 4

Var Y , init=4 ,ev = 1

#swap_integer(x,y) ;

END ELEMENT

END TEST

END SERVICE

```

Related Topics

[Simulating Generic Units on page 566](#) | [Advanced Ada Testing on page 570](#)

Test Program Entry Point

Component Testing for Ada

Since **ATTOL_TEST** is a sub-unit and not a main unit, Component Testing for Ada generates a main procedure at the end of the test program with the name provided on the command line.

Two methods are available to start the execution of the test program:

- Call during the elaboration of the unit under test.
- Call by the main procedure.

Call During the Elaboration of the Unit

In this case, you must add an additional line in the body of the unit tested:

```
PACKAGE <name>
...
END;
PACKAGE BODY <name>
...
PROCEDURE ATTOL_TEST is SEPARATE;
BEGIN
...
ATTOL_TEST;
END;
```

The package specification is not modified, but the test procedure is called at every elaboration of the package. Therefore, you need to remove or replace this call with an empty procedure after the test phase.

Call by the Main Procedure

In this case, you must add an additional line in the specification of the unit tested:

```
PACKAGE < name>
...
PROCEDURE ATTOL_TEST;
...
END;
```

```
PACKAGE BODY <name> is
```

```
...
```

```
PROCEDURE ATTOL_TEST is SEPARATE;
```

```
END;
```

Component Testing will then automatically generate a call to the **ATTOL_TEST** procedure in the main procedure of the test program. The test will be executed during the execution of the main program.

Limitations

Consider the following limitations:

- The unit under test must be of type *package*.
- The root body of **ATTOL_TEST** (procedure **ATTOL_TEST** is separate) cannot appear inside a generic package.

Testing Pointer Variables while Preserving the Pointed Value

Component Testing for Ada

To initialize a variable as a pointer while keeping the ability to test the value of the pointed element, use the **FORMAT string_ptr** statement in your **.ptu** test script.

This allows you to initialize your variable as a pointer and still perform string comparisons using **str_comp**.

Example:

```
TEST 1
```

```
FAMILY nominal
```

```
ELEMENT
```

```
FORMAT pointer_name = string_ptr
```

```
-- Then your variable pointer_name will be first initialized as a pointer
```

```
....
```

```
VAR pointer_name, INIT="l11c01pA00", ev=init
```

```
-- It is initialized as pointing to the string "l11c01pA00",
```

```
--and then string comparisons are done with the expected values using str_comp.
```

Testing Ada Tasks

Component Testing for Ada

As a general matter, HCL OneTest™ Embedded Component Testing for Ada was designed for synchronous programming. However, it is possible to achieve component testing even in an asynchronous environment.

The important detail is that any task which might be producing Runtime Analysis information (especially by calling stubbed procedures or functions) must be terminated when control reaches the **END ELEMENT** instruction in the **.ptu** test script.

If the code under test does not provide select statements or entry points in order to request the termination of the task, an abort call to the task might be necessary. For tasks who terminate after a certain time (not entering a infinite loop), the tester might check the task's state and sleep until termination of the task. In the **.ptu** test script, this might read as follows:

```
#while not TaskX'Terminated loop
# delay 1;
#end loop;
```

This instruction block is placed just before the **END ELEMENT** statement of the Test Script.

Example

The source files and complete **.ptu** script for following example are provided in the **examples/Ada_Task** directory.

In this example, the task calls a stubbed procedure. Therefore the task must be terminated from within the Test Script. Two different techniques of starting and stopping the task are shown here in **Test 1** and **Test 2**.

```
HEADER Prg_Under_Tst, 0.3, 0.0

#with Pck_Stub;

BEGIN Prg_Under_Tst

DEFINE STUB Pck_Stub

#with Text_IO;

#procedure Proc_Stubbed is

#begin

# Text_IO.Put_Line("Stub called.");

#end;

END DEFINE
```

```

SERVICE S1

SERVICE_TYPE extern

#Param_1 : duration;

#task1 : Prioritaire;

TEST 1

FAMILY nominal

ELEMENT

VAR Param_1, init = duration(0), ev = init

STUB Pck_Stub.Proc_Stubbed 1..1 => ()

#Task1.Unit_Testing_Exit_Loop;

#delay duration(5);

#Task1.Unit_Testing_Wait_Termination;

END ELEMENT

END TEST -- TEST 1

TEST 2

FAMILY nominal

ELEMENT

VAR Param_1, init = duration(2), ev = init

STUB Pck_Stub.Proc_Stubbed 1..1 => ()

#declare

# Task2 : T_Prio := new Prioritaire;

#begin

# Task2.Do_Something_Useful(Param_1);

# Task2.Unit_Testing_Exit_Loop;

# Task2.Unit_Testing_Wait_Termination;

#end;

```

END ELEMENT

END TEST -- TEST 2

END SERVICE --S1

In the **BEGIN** line of the script, it is not necessary to add the name of the separate procedure **Atto_Test**, as this is the default name;

The user code within the **STUB** contains a context clause and some custom native Ada instructions.

In both **Test 1** and **Test 2** it is necessary not only to stop the main loop of the task before reaching the **END ELEMENT** instruction, but also the task itself in order to have the tester return.

Task1 and **Task2** could run in parallel, however, the test Report would be unable to distinguish between the **STUB** calls coming in from either task, and would show the calls in a cumulative manner.

The entry points **Unit_Testing_Exit_Loop** and **Unit_Testing_Wait_Termination** can be considered as implementations for testing purposes only. They might not be used in the deployment phase.

The second test is *False* in the Report, the loop runs twice. This allows to check that the dump goes through smoothly.

Separate Compilation

Component Testing for Ada

You can make internal procedures and variables and the structure of private types visible from the test program, by including them in the body of the unit under test with a separate Ada instruction.

You must add the following line at the end of the body of the unit tested:

```
PACKAGE BODY <name>
```

```
...
```

```
PROCEDURE Test is separate;
```

```
END;
```

Defining the procedure **Test** this way allows you to access every element of the specification and also those defined in the body.

Generating a Separate Test Harness

Component Testing for Ada

Because of restrictions of the Ada language, Component Testing cannot generate a test harness which is a separate of more than one package.

You can however generate the main test harness as a separate of one of the packages and declare additional procedures as separates of other packages. This is done in the header of the **.ptu** test script, as in the following example:

```
Header Code_Under_Test, 1.0, 1.0

#separate (Second_Package);

#procedure Something is

#begin

# -- here internal variables of Second_Package are

# -- visible; private types can be accessed etc.

# null;

#end Something;

#with Second_Package;

-- this is to gain visibility on the package

-- from within the test harness

Begin First_Package, Test_Entry_Point

-- this causes HCL OneTest™ Embedded to generate a procedure

-- "Test_Entry_Point" as a separate of "First_Package" as

-- "main" procedure of the Test Harness

-- etc.
```

If the test script requires access to items from *Second_Package*, it can call the corresponding procedure from within an ELEMENT block of this **.ptu** test script.

```
Element

-- some VAR instructions here

#Second_Package.Something;

#-- here is the call to the tested procedure
```

End Element

Related Topics

[Declaring Global Variables for Testing on page 540](#)

Test Script Compiler Macro Definitions

Component Testing for Ada

You can specify a list of conditions to be applied when starting the Test Script Compiler. You can then generate the test harness conditionally. In the test script, you can include blocks delimited with the keywords **IF**, **ELSE**, and **END IF**.

If one of the conditions specified in the **IF** instruction is present, the code between the keywords **IF** and **ELSE** (if **ELSE** is present), and **IF** and **END IF** (if **ELSE** is not present) is analyzed and generated. The **ELSE** / **END IF** block is eliminated.

If none of the conditions specified in the **IF** instruction is satisfied, the code between the keywords **ELSE** and **END IF** is analyzed and generated.

By default, no generation condition is specified, and the code between the keywords **ELSE** and **END IF** is analyzed and generated.

Unexpected Exceptions

Component Testing for Ada

The generated test driver detects all raised exceptions. If a raised exception is not specified in the test script, it is displayed in the report.

When the exception is a standard Ada exception (**CONSTRAINT_ERROR**, **NUMERIC_ERROR**, **PROGRAM_ERROR**, **STORAGE_ERROR**, **TASKING_ERROR**), the exception name is displayed in the test report.

Unknown Values

Component Testing for Ada

In some cases, Component Testing for Ada is unable to produce a default value in the **.ptu** test script. When this occurs, Component Testing produces an invalid value with the prefix **_Unknown**.

Such cases include:

- Private values: **_Unknown_private_value**
- Function pointers: **_Unknown_access_to_function**
- Tagged limited private: **_Unknown_access_to_tagged_limited_private**

Before compiling you must manually replace these **_Unknown** values with valid values.

Test Iterations

Component Testing for Ada

You can execute the test case several times by adding the number of iterations at the end of instruction **TEST**, for example:

```
TEST <name> LOOP <number>
```

You can add other test cases to the current test case by using the instruction **NEXT_TEST**:

```
TEST <name>
```

```
...
```

```
NEXT_TEST
```

```
...
```

```
END TEST
```

This instruction allows a new test case to be added that will be linked to the preceding test case. Each loop introduced by the instruction **LOOP** relates to the test case to which it is attached.

Test cases introduced by the instruction **NEXT_TEST** can be dissociated after the tests are run. With the **ELEMENT** structure, the different phases of the same test case can be dissociated.

Test phases introduced by the instruction **ELEMENT** can be included in the loops created by the **LOOP** instruction.

Requirement

- To link a test or set of tests to a requirement, enter the following command line:

```
REQUIREMENT <name> { , [<attrName> =|:] <attrValue> }
```

Where:

- ▪ <name> is the name of the requirement. Optionally, this name could be followed by attributes.
- <attrName> is the name of the attribute. This name is optional. It is automatically added if it is missing.
- <attrValue> is the value of the attribute.

Example:

```
REQUIREMENT REQ_TEST2ELEM_025, type=robustness, level:1, John
```

The tests linked by a requirement depend on the position of the keyword REQUIREMENT in the script:

```

HEADER add, 1, 1
<variable declarations for the test script>
BEGIN
  REQUIREMENT... -- Requirement defined for all tests in the script
SERVICE add
  <local variable declarations for the service>
  REQUIREMENT... -- Requirement defined for all tests in the service
TEST 1
  REQUIREMENT...      -- Requirement defined for the test only
FAMILY nominal
ELEMENT
  VAR variable1, INIT=0, EV=0
  VAR variable2, INIT=0, EV=0
  #<call to the procedure under test>
END ELEMENT
END TEST
END SERVICE

```

Attribute values can be overloaded by environment variables during pre-processing phase. For example, if \$TARGETNAME is set, the value of the attribute \$TARGETNAME in the script will be overloaded by this environment. This allows you to dynamically configure some attributes in your build chain depending on the execution context.

After the tests execution, a requirement status is computed for each requirement, based on the result of the tests that are linked to this requirement.

A tool rod2req generates an XML file with all the requirement status and a coverage status.

Viewing Reports

Component Testing for Ada

After test execution, depending on the options selected, a series of Component Testing for Ada test reports are produced.

To learn about	See
Accessing the test reports	Opening a Report on page 683
Navigating through test reports	Using the Report Viewer on page 705
Performing a <i>diff</i> between two test reports	Comparing Ada Test Reports on page 584
Interpreting test results	Understanding Component Testing Test Reports on page 582

Understanding Component Testing reports

Component Testing for Ada

Test reports for Component Testing are displayed in the Report Viewer.

The test report is a hierarchical summary report of the execution of a test node. Parts of the report that have *Passed* are displayed in green. *Failed* tests are shown in red.

Report Explorer

The [Report Explorer on page 1024](#) displays each element of a test report with a *Passed* ✓, *Failed* ✗ symbol.

- Elements marked as *Failed* ✗ are either a failed test, or an element that contains at least one failed test.
- Elements marked as *Passed* ✓ are either passed tests or elements that contain only passed tests.

Test results are displayed for each instance, following the structure of the **.ptu** test script.

Report Header

Each test report contains a report header with:

- The version of HCL OneTest™ Embedded used to generate the test as well as the date of the test report generation
- The path and name of the project files used to generate the test
- The total number of test cases *Passed* and *Failed*. These statistics are calculated on the actual number of test elements listed in the sections below

Test Results

The graphical symbols in front of the node indicate if the test, item, or variable is *Passed* ✓ or *Failed* ✗:

- A test is *Failed* if it contains at least one failed variable. Otherwise, the test is considered *Passed*.

You can obtain the following data items if you click with the pointer on the Information node:

- Number of executed tests
- Number of correct tests
- Number of failed tests

A variable is incorrect if the expected value and the value obtained are not identical, or if the value obtained is not within the expected range.

If a variable belongs to an environment, an environment header is previously edited.

In the report variables are edited according to the value of the Display Variables setting of the Component Testing test node.

The following table summarizes the editing rules:

Results	Display Variable	Display Variable	Display Variable
	All Variables	Incorrect Variables	Failed Tests Only
✔ Passed test	Variable edited automatically	Variable not edited	Variable not edited
✘ Failed test	Variable edited automatically	Variable edited automatically	Variable edited if incorrect

The [Initial and Expected Values on page 531](#) option changes the way initial and expected values are displayed in the report.

Tests that contain only a **STUB** statement and no **VAR**, **ARRAY**, or **STR** statement are reported as empty tests. The **STUB** instruction is not considered as part of the the **TEST** as **STUBs** are always tested whether there is a **STUB** statement present or not.

Related Topics

[Opening a report on page 683](#)

[Using the Report Viewer on page 705](#)

[Initial and expected values on page 531](#)

[Exporting reports on page 705](#)

Comparing Ada Test Reports

Component Testing for Ada

The Component Testing comparison capability allows you to compare the results of the last two consecutive tests.

To activate the comparison mode, select **Compare two test runs** in the [Component Testing for C and Ada Settings on page 999](#) dialog box.

In comparison mode an additional check is performed to identify possible regressions when compared with the previous test run.

The Component Testing Report displays an extra column named "Obtained Value Comparison" containing the actual difference between the current report and the previous report.

Related Topics

[Component Testing for Ada Settings on page 999](#) | [Understanding Component Testing reports on page 582](#)

Array and structure display

Component Testing for Ada

The Array and Structure Display option indicates the way in which Component Testing processes variable array and structure statements. This option is part of the [Component Testing Settings for C on page 999](#) dialog box.

Standard array and structure display

This option processes arrays and structures according to the statement with which they are declared. This is the default operating mode of Component Testing. The default report format is the **Standard** editing.

Extended array and structure display

Arrays of variables may be processed after the keywords **VAR** or **ARRAY**, and structured variables after the keywords **VAR**, **ARRAY**, or **STRUCTURE**:

- After a **VAR** statement, each element in the array is initialized and tested one by one. Likewise, each member of a structure that is an array is initialized and tested element by element.
- After an **ARRAY** statement, the entire array is initialized and checked. Likewise, each member of a structure is initialized and checked element by element.
- After a **STRUCTURE** statement, the whole of the structure is initialized and checked.

When **Extended editing** is selected, Component Testing interprets **ARRAY** and **STRUCTURE** statements as **VAR** statements.

The output records in the unit test report are then detailed for each element in the array or structure.

Note This setting slightly slows down the test execution because checks are performed on each element in the array.

Packed array and structure display

This command has the opposite effect of the Extended editing option. When **Packed editing** is selected, Component Testing interprets **VAR** statements as **ARRAY** or **STRUCTURE** statements.

Array and structure contents are fully tested, only the output records are more concise.

Note This setting slightly improves the speed of execution because checks are performed on each array as a whole.

Related Topics

[Component Testing for C and Ada Settings on page 999](#)

System Testing for C

About System Testing for C

System Testing for C is the first commercial automated feature dedicated to testing message-based applications. Until now most of the projects developing real-time, embedded or distributed systems spent a fair amount of resources building dedicated test beds. Project managers can now save time and money by avoiding this costly, non-core-business activity.

System Testing for C helps you solve complex testing issues related to system interaction, concurrency, and time and fault tolerance by addressing the functional, robustness, load, performance and regression testing phases from small, single threads or tasks up to very large, distributed systems.

With the System Testing tool, test engineers can easily design, code and execute virtual testers that represent unavailable portions of the system under test - SUT - and its environment.

System Testing for C is recommended for testing:

- Telecommunication and networking equipment using standard protocols
- Aerospace equipment using standard or proprietary operating systems and a communication bus
- Automotive Electronic Control Units (ECUs) or appliance systems
- Distributed applications based on message-oriented middleware

System Testing for C supports C89 and C99.

Related Topics

[Using Test Features on page 446](#) | [About Component Testing for C and Ada on page 447](#) | [About Component Testing for C++ on page 511](#)

Agents and Virtual Testers

About Virtual Testers

System Testing for C

Virtual Testers are multiple contextual incarnations of a single **.pts** System Testing test script.

One Virtual Tester can be deployed simultaneously on one or several targets, with different test configurations. A same virtual tester can also have multiple clones on the same target host machine.

System Testing generates Virtual Testers from a test script according to the declared [instances on page 622](#). The System Testing Supervisor, which runs on the HCL OneTest™ Embedded host computer, is in charge of deploying and controlling remote Virtual Testers.

Note A System Testing Agent must be installed and running on each target host before deploying Virtual Testers to those targets.

Following the execution architecture and constraints needed to comply, the Test Script Compiler provides several ways to generate the Virtual Testers.

Related Topics

[Configuring Virtual Testers on page 591](#) | [Deploying Virtual Testers on page 592](#) | [Debugging Virtual Testers on page 592](#) | [System Testing Supervisor on page 654](#) | [System Testing in a Multi-Threaded or RTOS Environment on page 596](#)

System Testing Agents

Installing System Testing Agents

System Testing for C

When using Virtual Testers on remote target hosts, a daemon must be running on the target to act as an interface between the virtual tester and the System Testing Supervisor. This daemon is known as the System Testing Agent.

Note Always make sure that the version of the System Testing Agent matches the version of HCL OneTest™ Embedded. If you have upgraded from a previous version of HCL OneTest™ Embedded, you must also update all System Testing Agents on remote machines.

The installation directory of System Testing includes the following necessary agent files:

- **atsagtd.bin**: the agent executable binary for UNIX
- **atsagtd.exe**: the agent executable binary for Windows
- **atsagtd**: the agent launcher for UNIX when using *inetd*
- **atsagtd.sh**: a UNIX shell script that starts **atsagtd.bin**

On Windows platforms, the **ATS_DIR** environment variable must be set to indicate the directory where the **atsagtd.exe** and **atsagtd.ini** files are located. If the file cannot be found, only the current user on the current computer will be authorized.

Installing the Agent

There are two methods for installing the System Testing Agent:

- **Manual launch**
- **Inetd daemon installation**

To install a System Testing Agent for manual execution:

This procedure does not require system administrator access, but launching of the agent is not fully automated.

1. Copy **atsagtd.bin** or **atsagtd.exe** to a directory on the target machine.
2. On the target machine, set the **ATS_DIR** environment variable to the directory containing the agent binaries.
3. Add that same agent directory to your **PATH** environment variable.

Note You can add these commands to the user configuration file: **login**, **.cshrc** or **.profile**.

1. On UNIX systems, create an agent access file **.atsagtd** file in your home directory. On Windows create an **atsagtd.ini** file in the agent installation directory. See [System Testing Agent Access Files on page 590](#).
2. Move the agent access file to your chosen base directory, such as the directory where the Virtual Testers will be launched.
3. Launch the agent as a background task, with the port number as a parameter. By default, this number is 10000.

```
atsagtd.bin <port number>&
atsagtd <port number>
```

To install a System Testing Agent with *inetd*:

This procedure is for UNIX only. Launching agents on target machines is automatic with *inetd*.

With this method, the *inetd* daemon runs the **atsagtd.sh** shell script that initializes environment variables on the target machine and launches the System Testing Agent.

1. Copy **atsagtd.sh** and **atsagtd.bin** to a directory on the target machine.
2. On the target machine, set the **ATS_DIR** environment variable to the directory containing the agent binaries.
3. Add that same agent directory to your **PATH** environment variable.

Note You can add these commands to the user configuration file: **login**, **.cshrc** or **.profile**.

1. Log on as root on the target machine.
2. Add the following line to the **/etc/services** file:

```
atsagtd <port number>/tcp
```

The agent waits for a connection to *<port number>*. By default, System Testing uses port 10000.

Note If NIS is installed on the target machine, you may have to update the NIS server. You can check this by typing **yycat services** on the target host.

1. Add the following line to the **/etc/inetd.conf** file:

```
atsagtd stream tcp nowait <username> <atsagtd path> <atsagtd path>
```

where *<username>* is the name of the user that will run the agent on the target machine and *<atsagtd path>* is the full path name of the System Testing Agent executable file **atsagtd**.

To reconfigure the inetd daemon, use one of the following methods:

- Type the command **/etc/inetd -c** on the target host.
- Send the **SIGHUP** signal to the running *inetd* process.
- Reboot the target machine.

In some cases, you might need to update the file **atsagtd.sh** shell script to add some environment variables to the target machine.

Return to your user account and create an agent access file **.atsagtd** file in your home directory. See [System Testing Agent Access Files on page 590](#).

Forcing IPv4 only

You can force the agent to bind a socket listener using only IPv4 (excluding IPv6) by using the **-IPv4** option:

```
atsagtd <port number> -IPv4
```

Troubleshooting the agent

To check the installation, type the following command on the host running HCL OneTest™ Embedded:

```
telnet <target machine> <port number>
```

where *<port number>* is the port number you specified during the installation procedure. By default, System Testing uses port 10000. The System Testing Agent should answer with the following message:

```
210 hello, please to meet you.
```

After the connection succeeds, press **Enter** to close the connection or type the following command to check that *<username>* is set up as a user:

```
Jef <username>
```

If the connection fails, try the following steps to troubleshoot the problem:

- Check the target hostname and port.
- Check the Agent Access File.
- Check the target hostname and port in the *atsagtd.sh* shell script.
- Check the **/etc/services** and **/etc/inetd.conf** files on the target machine.
- If you are using NIS services on your network, check the NIS configuration.

To see the current working directory, type the following command:

PWD

To close the connection, type:

QUIT

Related Topics

[System Testing Agent Access Files on page 590](#) | [About Virtual Testers on page 586](#)

System Testing Agent Access Files

System Testing for C

The **.atsagtd** (UNIX) or **atsagtd.ini** (Windows) agent access file is an editable configuration file that secures access to System Testing Agents and contains a list of machines and users authorized to execute agents on that machine, with the following syntax:

```
<computer name> <username> [#<comment>]
```

On Windows platforms, the System Testing Agent uses the **ATS_DIR** environment variable to locate the **atsagt.ini** file.

A plus sign **+** can be used as a wildcard to provide access to all users or all workstations.

The minus sign **-** suppresses access to a particular user.

You can add comments to the agent access file by starting a line with the **#** character. Blank lines are not allowed.

Example

```
# This is a sample .atsagtd or atsagtd.ini file.
```

```
# The following line allows access from user jdoe on a machine named workstation
```

```
workstation jdoe
```

```
# The following line allows access from all users of workstation
```

```
workstation +
```

```
# The following allows access from jdoe on any host
```

```
+ jdoe
```

```
# The following allows access to all users except anonymous from the machine workstation
```

```
workstation +
```

```
workstation -anonymous
```

Related Topics

[Installing System Testing Agents on page 587](#) | [System Testing Supervisor - atsspv on page 1106](#)

Configuring Virtual Testers

System Testing for C

The Virtual Tester Configuration dialog box allows you to create and configure a set of Virtual Testers that can be deployed for System Testing.

To open the Virtual Test Configuration dialog box:

1. In the **Project Explorer**, right-click a **.pts** test script.
2. From the pop-up menu, select **Virtual Tester Configuration**.

Note The Virtual Tester Configuration box is also included as part of the [System Testing Wizard on page 669](#) when you are setting up a new activity.

Virtual Tester List

Use the Virtual Tester List to create a **New** Virtual Tester, **Remove** or **Copy** an existing one.

Select a Virtual Tester in the Virtual Tester List to apply any changes in the property tabs on the right.

General Tab

This tab specifies an instance and target deployment to be assigned to the selected Virtual Tester.

- **VT Name:** This is the name of the Virtual Tester currently selected in the **Virtual Tester List**. The name of the virtual tester must be a standard C identifier.
- **Implemented INSTANCE:** Use this box to assign an instance, defined in the **.pts** test script, to the selected virtual tester. This information is used for Virtual Tester deployment. Select **Default** to specify the instance during deployment.
- **Target:** This specifies the Target Deployment Port compilation parameters for the selected Virtual Tester.
- **Configure Settings:** This button opens the [Configuration Settings on page 658](#) dialog for the selected Virtual Tester node.

Scenario Tab

Use this tab to select one or several scenarios as defined in the **.pts** test script. During execution, the Virtual Tester plays the selected scenarios.

Family Tab

Use this tab to select one or several families as defined in the **.pts** test script. During execution, the Virtual Tester plays the selected families.

Related Topics

[About Virtual Testers on page 586](#) | [About Configuration Settings on page 658](#) | [Deploying Virtual Testers on page 592](#)

Debugging Virtual Testers

System Testing for C

In some cases, you may want to observe how your system under test reacts when an error occurs and the consequences of this error on the whole process, without stopping the Virtual Tester.

By default, when an error occurs in a block, the execution of the block is interrupted. To prevent interruption, use the virtual tester debug mode.

You can statically activate the debug mode by compiling the generated Virtual Tester with the **ATL_SYSTEMTEST_DEBUG** variable, as in the following example:

```
cc -c -I$ATLTGT/lib/ -DATL_SYSTEMTEST_DEBUG <source.c>
```

where **\$ATLTGT** is the current TDP directory.

Related Topics

[About Virtual Testers on page 586](#) | [On-the-Fly Tracing on page 656](#)

Deploying Virtual Testers

System Testing for C

The Virtual Tester Deployment Table allows to deploy previously created Virtual Testers.

To open the Virtual Tester Deployment Table

1. Make sure that **Execution** is selected in your Build options.
2. In the **Project Explorer**, right-click a System Testing node.
3. From the pop-up menu, select **Deployment Configuration**.
4. Select **Advanced Options** and click **Rendezvous**.

Note The Virtual Tester Deployment Table is also included in the [System Testing Wizard on page 669](#) when you are setting up a new activity.

Virtual Tester Deployment Table

Use the **Add**, **Remove** or **Copy** buttons to modify the list. Each line represents one or several executions of a Virtual Tester assigned to an instance, target host, and other parameters.

- **Number of Occurrences:** Specifies the number of simultaneous executions of the current line.
- **Virtual Tester Name:** Specifies one of the previously created Virtual Testers.
- **Instance:** Specifies the instances assigned to this Virtual Tester. If an instance was specifically assigned in the [Virtual Tester Configuration on page 591](#) box, this cannot be changed. Select **<all>** only if no INSTANCE is defined in the test script.
- **Network Node:** This defines the target host on which the current line is to be deployed. You can enter a machine name or an IP address. Leave this field blank if you want to use the IP address specified in the Host Configuration section of the [General Settings on page 987](#).

Note If the IP address line in the Host Configuration settings is blank, then the Virtual Tester Deployment Table retrieves the IP address of the local machine when generating the deployment script.

Advanced Options

Click the **Advanced Options** button to add the following columns to the Virtual Tester Deployment Table, and to add the **Rendezvous...** button.

- **Agent TCP/IP Port:** This specifies the port used by the [System Testing Agents on page 587](#) to communicate with HCL OneTest™ Embedded. By default, System Testing uses port 10000.
- **Delay:** This allows you to set a delay between the execution of each line of the table.
- **First Occurrence ID:** This specifies the unique occurrence ID identifier for the first Virtual Tester executed on this line. The occurrence ID is automatically incremented for each number of instances of the current line. See [Communication Between Virtual Testers on page 621](#) for more information.
- **RIO filename:** This specifies the name of the .rio file containing the Virtual Tester output, for use in [multi-threaded or RTOS environments on page 596](#).

Click the **Rendezvous Configuration** button to [set up any rendezvous members on page 595](#).

File System Limitations

Deployment of the Virtual Testers results in the creation of an **.spv** deployment script. This script contains file system commands, such as **CHDIR**. If you are deploying the test to a target platform that does not support a file system, you must edit the **.spv** script manually.

For the **.spv** supervisor script to be generated, the **Execution** option must be selected in the Build options.

Related Topics

[About Virtual Testers on page 586](#) | [Configuring Virtual Testers on page 591](#) | [Setting Up Rendezvous Members on page 595](#) | [Editing the Deployment Script on page 594](#) | [System Testing supervisor script \(.spv\) on page 893](#) | [System Testing Supervisor - atsspv on page 1106](#)

Editing the Deployment Script

System Testing for C

The System Testing Supervisor actually runs a script, which is automatically generated by [configuring Virtual Testers on page 591](#) and [deploying Virtual Testers on page 592](#).

In some cases, you will need to manually edit the script. To do this, you first have to generate an **.spv** deployment script in your workspace.

To generate a deployment script

1. Make sure that **Execution** is selected in your Build options.
2. In the **Project Explorer**, right-click a System Testing node.
3. From the pop-up menu, select **Generate Deployment Script**.
4. Enter a name for the generated script.

If you decide to manually maintain a deployment script, you must ensure that any pathnames and other parameters remain up to date with the rest of the System Testing node.

For information on the **.spv** script command language, please refer to the Reference section.

Related Topics

[About Virtual Testers on page 586](#) | [System Testing Supervisor on page 654](#) | [Installing System Testing Agents on page 587](#)

Optimizing Execution Traces

System Testing for C

Each Virtual Tester generates a trace file during its execution. This trace file is used to generate the [System Testing Report on page 639](#).

You may want to adapt the volume of traces generated at execution time. For example, each Virtual Tester saves its execution traces in an internal buffer that you can configure.

To optimize execution trace output, use the Execution Traces area in the [Test Script Compiler Settings on page 1003](#) dialog box.

- By default, System Testing generates a normal trace file.
- Select **Time stamp only** to generate traces for each scenario begin and end, all events, and for error cases. This option also generates traces for each **WAITTIL** and **PRINT** instruction. Use this option for load and performance testing, if you expect a large quantity of execution traces and you want to store all timing data.
- Select **Block start/end only** to generate traces for each scenario beginning and end, all events, and for all error cases.
- Select **Error only** to generate traces only if an error is detected during execution of the application. This report will be incomplete, but the report will show failed instructions as well as a number of instructions that preceded the error. This number depends on the virtual tester trace buffer size. Use this option for endurance testing, if you expect a large quantity execution traces.

In addition to the above, you can select the **Circular trace** option for strong real-time constraints when you need full control over the flush of traces to disk. If you want to still store a large amount of trace data, specify a large buffer.

Related Topics

[Test Script Compiler Settings on page 1003](#) | [Circular Trace Buffer on page 655](#)

Setting Up Rendezvous Members

System Testing for C

When you have used Rendezvous points in your **.pts** test script, it is necessary to indicate the number of members that the supervisor must expect at each rendezvous.

The **Rendezvous Members** dialog box is an advanced option of the [Virtual Tester Configuration on page 591](#).

To specify the number of members for each rendezvous:

1. In the **Project Explorer**, right-click a System Testing node.
2. From the pop-up menu, select **Deployment Configuration**.
3. Select **Advanced Options** and click **Rendezvous**.
4. For each rendezvous encountered in the **.pts** test script, select a number of rendezvous members.
5. Select **AutoGenerate** to automatically compute the number of members in each Rendezvous. In some cases, such as when rendezvous are placed in an exception, this option cannot provide correct information to the supervisor.
6. Click **OK**.

Related Topics

[Deploying Virtual Testers on page 592](#) | [System Testing Supervisor on page 654](#)

System Testing in a Multi-Threaded or RTOS Environment

System Testing for C

When Virtual Testers must be executed as a threaded part of a UNIX or Windows process, or on RealTime Operating Systems (RTOS) you must take several constraints into account:

- The Virtual Tester should be generated as a function and not a main program.
- You must consider the configuration of the Virtual Testers' execution.

There are memory management constraints:

- There is no dynamic memory allocation.
- Stacks are small.
- Virtual Testers share global data.
- Configuration of Virtual Tester execution.

Virtual Tester as a Thread or Task

When using a flat-memory RTOS model, the Virtual Testers can run as a process thread or as a task in order to avoid conflicts with the application under test's global variables.

Moreover, the Target Deployment Port is fully reentrant. Therefore, you can run multiple instances of a Virtual Tester in the same process. The system runs each process as a different process thread.

In this case, the Test Script Compiler generates the virtual tester source code without a *main()* function, but with a user function.

To configure System Testing to run in multi-threaded mode, select the **Not shared** option in [Test Script Compiler Settings on page 1003](#).

Multiple Instances of a Same Virtual Tester

Multiple instances of a same Virtual Tester can run simultaneously on a same target. In this case, you need to protect the Virtual Tester threads in the same process against access to global variables.

The **Not Shared** setting in [Test Script Compiler Settings on page 1003](#) allows you to specify global variables in the test script that should remain unshared by separate Virtual Tester threads. When selected, multiple instances of a Virtual Tester can all run in the same process.

You can share some global static variables in order to reuse data among different Virtual Testers by using the **SHARE** command in the **.pts** test script. See the Reference section for information about the System Testing Language.

Related Topics

[About Virtual Testers on page 586](#) | [System Testing in a Multi-Threaded or RTOS Environment on page 596](#) | [Test Script Compiler Settings on page 1003](#)

Launching virtual tester threads

System Testing for C

In a multi-threaded environment, there are two methods of starting the virtual tester threads:

- From a specially designed thread launcher program that you must write to include in your project.
- From a TDP thread launcher if available.

TDP thread launcher from TDP

Some TDPs can launch the virtual tester threads without needing a special program. If your TDP supports this method, the only requirement is to specify this in the Configuration settings of the System Testing test node.

To use the TDP thread launcher:

1. In the **Project Explorer**, click the **Settings**  button.
2. Select a System Testing test node in the **Project Explorer** pane.
3. In the Configuration Settings list, expand **System Testing** and select **Test Script Compiler**.
4. Set the **Use thread launcher from TDP** setting to **Yes**.
5. When you have finished, click **OK** to validate the changes.

TDP thread launcher program

If the TDP does not contain a TDP thread launcher, the only way to start Virtual Tester threads is to write a program, specifying:

- The name of the execution trace file
- The name of the instance to be started

To do this, use the **ATL_T_ARG** structure, defined in the **ats.h** header file of the Target Deployment Port.

Example

The following example is a sample program for launching virtual tester threads.

```
#include <stdio.h>

#include <sched.h>
```

```
#include <pthread.h>

#include <errno.h>

#include "TP.h"

extern ATL_T_THREAD_RETURN *start(ATL_PT_ARG);

int main(int argc, char *argv[])

{

pthread_t thrTester_1,thr_Tester_2;

pthread_attr_t pthread_attr_default;

ATL_T_ARG arg_Tester_1, arg_Tester_2;

int status;

arg_Tester_1.atl_riofilename = "Tester_1.rio";

arg_Tester_1.atl_filters = "";

arg_Tester_1.atl_instance = "Tester_1";

arg_Tester_1.atl_occid = 0;

arg_Tester_2.atl_riofilename = "Tester_2.rio";

arg_Tester_2.atl_filters = "";

arg_Tester_2.atl_instance = "Tester_2";

arg_Tester_2.atl_occid = 0;

pthread_attr_init(&pthread_attr_default);

/* Start Thread Tester 1 */

pthread_create(&thrTester_1,&pthread_attr_default,start,&arg_Tester_1);

/* Start Thread Tester 2 */

pthread_create(&thrTester_2,&pthread_attr_default,start,&arg_Tester_2);

/* Both Testers are running */

/* Wait for the end of Thread Tester 1 */

pthread_join(thrTester_1, (void *)&status);
```

```

/* Wait for the end of Thread Tester 2 */

pthread_join(thrTester_2, (void *)&status);

return(0);

}

```

An example demonstrating how to use System Testing for C on multi-threaded applications is provided in the **Broadcast Server** example project. See [Example projects on page 677](#) for more information.

Related Topics

[About Virtual Testers on page 586](#) | [System Testing in a Multi-Threaded or RTOS Environment on page 596](#)

System Testing for C Test Scripts

Flow control

Flow Control Instructions

System Testing for C

Several execution flow instructions let you develop algorithms with multiple branches.

System Testing **.pts** test script flow control instructions include:

- [Function calls on page 599](#)
- [Conditions on page 601](#)
- [Iterations on page 602](#)
- [Multiple Conditions on page 603](#)

Related Topics

[Instances on page 622](#) | [Event Management on page 606](#) | [Time Management on page 634](#) | [Using Native Language on page 638](#)

Function calls

System Testing for C

The CALL instruction lets you call functions or methods in a test script and to check return values of functions or methods.

For the following example, you must pre-declare the **param1**, **param2**, **param4**, and **return_param** variables in the test script, using native language.

CALL function ()

-- indicates that the return parameter is neither checked nor stored in a variable.

CALL function () @ "abc"

-- indicates that the return parameter to the function must be compared with the string "abc", but its value is not stored in a variable.

CALL function () @@return_param

-- indicates that the return parameter is not checked, but is stored in the variable return_param.

CALL function () @ 25 @return_param

-- indicates that the return parameter is checked against 25 and is stored in the variable return_param.

Related Topics

[Using Native Language on page 638](#) | [CALL on page 839](#)

Include Statements

System Testing for C

To avoid writing large test scripts, you can split test scripts into several files and link them using the **INCLUDE** statement.

This instruction consists of the keyword **INCLUDE** followed by the name of the file to include, in quotation marks (" ").

INCLUDE instructions can appear in high- and intermediate-level scenarios, but not in the lowest-level scenarios.

You can specify both absolute or relative filenames. There are no default filename extensions for included files. You must specify them explicitly.

Example

```
HEADER "Socket validation", "1.0", "beta"
```

```
INCLUDE "../initialization"
```

```
SCENARIO first
```

```
END SCENARIO
```

```
SCENARIO second
```

```
INCLUDE "scenario_3.pts"
```

SCENARIO level2

FAMILY nominal, structural

...

END SCENARIO

END SCENARIO

Conditions

System Testing for C

The **IF** statement comprises the keywords **IF**, **THEN**, **ELSE**, and **END**. It lets you define branches and follows these rules:

- The test following the keyword **IF** must be a Boolean expression in C or C++.
- IF instructions can be located in scenarios, procedures, or environment blocks.
- The **ELSE** branch is optional.

The sequence **IF** (test) **THEN** must appear on a single line. The keywords **ELSE** and **END IF** must each appear separately on their own lines.

Example

```
HEADER "Instruction IF", "1.0", "1.0"
```

```
#int IdConnection;
```

```
SCENARIO Main
```

```
COMMENT connection
```

```
CALL socket(AF_UNIX, SOCK_STREAM, 0)@@IdConnection
```

```
IF (IdConnection == -1) THEN
```

```
EXIT
```

```
END IF
```

```
END SCENARIO
```

Related Topics

[Iterations on page 602](#) | [Multiple Conditions on page 603](#) | [IF on page 855](#)

Iterations

System Testing for C

The **WHILE** instruction comprises the keywords **WHILE** and **END**. It lets you define loops and follows these rules:

- The test following the keyword **WHILE** must be a C Boolean expression.
- The **WHILE** instructions can be located in scenarios, procedures, or environment blocks.

The sequence **WHILE** (test) and the keyword **END WHILE** must each appear separately on their own lines.

Example

```
HEADER "Instruction WHILE", "", ""
```

```
#int count = 0;
```

```
#appl_id_t id;
```

```
#message_t message;
```

```
SCENARIO One
```

```
FAMILY nominal
```

```
CALL mbx_init(&id) @ err_ok
```

```
VAR id.applname, INIT="JUPITER"
```

```
CALL mbx_register(&id) @ err_ok
```

```
VAR message, INIT={
```

```
& type=>DATA,
```

```
& applname=>"SATURN",
```

```
& userdata=>"hello world!"}
```

```
WHILE (count<10)
```

```
CALL mbx_send_message(&id,&message) @ err_ok
```

```
VAR count, INIT=count+1
```

```
END WHILE
```

```
CALL mbx_unregister(&id) @ err_ok
```

```
CALL mbx_end(&id) @ err_ok
```

END SCENARIO

Related Topics

[Conditions on page 601](#) | [Multiple Conditions on page 603](#) | [WHILE on page 890](#)

Multiple Conditions

System Testing for C

The multiple-condition statement **CASE** comprises the keywords **CASE**, **WHEN**, **END**, **OTHERS** and the arrow symbol **=>**.

CASE instructions follow these rules:

- The test following the keyword **CASE** must be a C or C++ Boolean expression. The keyword **WHEN** must be followed by an integer constant.
- The keyword **OTHERS** indicates the default branch for the **CASE** instruction. This branch is optional.
- **CASE** instructions can be located in scenarios, procedures, or environment blocks.

Example

```
HEADER "Instruction CASE", "", ""
```

```
...
```

```
MESSAGE message_t: response
```

```
SCENARIO One
```

```
...
```

```
CALL mbx_send_message(&id,&message) @ err_ok
```

```
DEF_MESSAGE response, EV={}
```

```
WAITTIL(MATCHING(response),WTIME == 10)
```

```
-- Checking the just received event type
```

```
CASE (response.type)
```

```
WHEN ACK =>
```

```
CALL mbx_send_message(&id,&message) @ err_ok
```

```
WHEN DATA =>
```

```
CALL mbx_send_message(&id,&ack) @ err_ok
```

```
WHEN NEG_ACK =>  
  
CALL mbx_send_message(&id,&error) @ err_ok  
  
WHEN OTHERS => ERROR  
  
END CASE  
  
END SCENARIO
```

Related Topics

[Conditions on page 601](#) | [Iterations on page 602](#) | [System Testing language reference on page 893](#) | [CASE on page 842](#)

Procedures

System Testing for C

You can also use procedures to build more compact test scripts. The following are characteristics of procedures:

- They must be defined before they are used in scenarios.
- They do not return any parameters.

A procedure begins with the keyword **PROC** and ends in the sequence **END PROC**. For example:

```
HEADER "Socket Validation", "1.0", "beta"
```

```
PROC function ()
```

```
...
```

```
END PROC
```

```
SCENARIO first
```

```
...
```

```
CALL function ()
```

```
...
```

```
END SCENARIO
```

```
SCENARIO second
```

```
SCENARIO level2
```

```
FAMILY nominal, structural
```

...

END SCENARIO

END SCENARIO

A procedure can call sub-procedures as long as these sub-procedures are located above the current procedure.

Procedure blocks can take parameters. When defining a procedure, you must also specify the input/output parameters.

Each parameter is described as a type followed by the name of the variable.

The declaration syntax requires, for each argument, a type identifier and a variable identifier. If you want to use complex data types, you must use either a macro or a C or C++ type declaration.

Example

In the following example, the argument to procedure **function1** is a character string of 35 bytes. The arguments to procedure **function2** are an integer and a pointer to a character.

```
HEADER "Socket Validation", "1.0", "beta"
```

```
#typedef char string[35];
```

```
##define ptr_car char *
```

```
PROC function1 (string a)
```

...

```
END PROC
```

```
PROC function2 (int a, ptr_car b)
```

...

```
END PROC
```

```
SCENARIO first
```

...

```
CALL function1 ( "foo" )
```

...

```
END SCENARIO
```

Adaptation layer

Adaptation Layer

System Testing for C

The adaptation Layer helps you describe communication between the Virtual Tester and the system under test.

Many different means of communication allow your systems to talk with each other. At the software application level, a communication type is identified by a set of services provided by specific functions.

For example, a UNIX system provides several means of communication between processes, such as *named pipes*, *message queues*, *BSD sockets*, or *streams*. You address each communication type with a specific function.

Furthermore, each communication type has its own data type to identify the application you are sending messages to. This type is often an *integer* (message queues, BSD sockets, ...), but sometimes a *structure* type.

Data exchanged this way must be interpreted by all communicating applications. For this reason, each type of exchanged data must be well identified and well known. By providing the type of exchanged data to the Virtual Tester, it will be able to automatically print and check the incoming messages.

- [Basic Declarations on page 606](#)
- [Sending Messages on page 608](#)
- [Receiving Messages on page 610](#)
- [Messages and Data Management on page 615](#)
- [Communication Between Virtual Testers on page 621](#)

Basic Declarations

System Testing for C

COMMTYPE Instruction

For each communication type, there is a specific data type that identifies the application you are sending messages to. In a test script, the **COMMTYPE** instruction is used to identify clearly this data type, and then, the communication type.

The **COMMTYPE** instruction cannot handle basic types. Therefore, you must previously define the type with a **typedef** statement.

For example, on UNIX systems, the data type for the BSD sockets is an integer. The **COMMTYPE** instruction is therefore used as follows:

```
#typedef int bsd_socket_id_t;
```

```
COMMTYPE ux_bsd_socket IS bsd_socket_id_t
```

In the **stack** example provided with the product, the following line defines a new communication type called **appl_comm**:

```
COMMTYPE appl_comm IS appl_id_t
```

MESSAGE Instruction

The **MESSAGE** instruction identifies the type of the data exchanged between applications. It also defines a set of reference messages.

The type of the messages exchanged between applications using our **stack** example is **message_t**.

The following instruction also declares three reference messages:

```
MESSAGE message_t: ack, neg_ack, data
```

CHANNEL Instruction

The **CHANNEL** instruction is used to declare a communication channel on a specific communication type. Thanks to channels of communication, the user can easily manage a large number of opened connections.

```
CHANNEL appl_comm: appl_channel_1, appl_channel_2
```

ADD_ID Instruction

A communication channel is a logical medium of communication that multiplexes several opened connections of the same type between the Virtual Tester and applications under test. When opening a new connection, it has to be linked to a communication channel, so that the Virtual Tester knows about this new connection.

```
CALL mbx_init( &id ) @ err_ok @ errcode
```

```
ADD_ID (appl_channel, id)
```

In this example, the function call to **mbx_init** opens a connection between the Virtual Tester and the system under test. This connection is identified by the value of **id** after the call. The **ADD_ID** instruction add this new connection to the channel **appl_channel**.

Related Topics

[COMMTYPE on page 845](#) | [MESSAGE on page 864](#) | [CHANNEL on page 843](#) | [ADD_ID on page 838](#) | [Adaptation layer on page 606](#)

Sending Messages

System Testing for C

PROCSEND Instruction

Event management provides a mechanism to send messages. This mechanism needs the definition of a message sending procedure or **PROCSEND** for each couple communication type, message type.

The **PROCSEND** instruction is then called automatically by the **SEND** instruction to send a message to the system under test (SUT).

In the following example, **msg** is a **message_t** typed input formal parameter specifying the message to send. The input formal parameter **id** is used to know where to send the message on the communication type **appl_comm**.

```
PROCSEND message_t: msg ON appl_comm: id

CALL mbx_send_message ( &id, &msg ) @ err_ok

END PROCSEND
```

The sending is done by the API function call to **mbx_send_message**. The return code is treated to decide whether the message was correctly sent. Another value than **err_ok** means that an error occurred during the sending.

The script must have one **PROCSEND** for each message type and channel type pair used by any of the **SEND** instructions.

The name of each **PROCSEND** in the generated C code is made up with the signature of the *message* type and *channel* type for each **PROCSEND** found in the test script, as follows.

VAR Instruction

The instruction **VAR** allows you to initialize messages declared using **MESSAGE** instructions. This message may also be initialized by any other C or C++ function or method:

```
VAR ack, INIT= { type => ACK }

VAR data, INIT= {

& type => DATA,

& applname => "SATURN",

& userdata => "hello world !" }
```

To learn all the nuts and bolts of the **DEF_MESSAGE** instruction, see the Messages and Data Management chapter.

SEND Instruction

This instruction allows you to invoke a message sending on one communication channel .

It has two arguments:

- the message to send,
- the communication channel where the message should be sent.

The send instruction is as follows:

```
SEND ( message , appl_ch )
```

In the previous figure, the **SEND** instruction allows the test program to send a message on a known connection (see the **ADD_ID** instruction). If an error occurs during the sending of the message, the **SEND** exits with an error. The scenario execution is then interrupted.

To send the message on the appropriate channel, the generated code calls the **PROCSEND** named with the signature of the *message* type to be sent (first parameter) and the *channel* type to be used (second parameter).

The message type is provided by the **MESSAGE** instruction. The channel type is provided by the **CHANNEL** instruction.

Therefore, in the generated code, the **SEND** instruction calls the following function:

```
PROCSEND_message_t_appl_comm(message, appl_ch)
```

which corresponds to the following line in the test script:

```
PROCSEND message_t ... ON appl_comm
```

Example

The following test script describes a simple use of our stack. First of all, some resources are allocated and a connection is established with the communication stack (**mbx_init**). This connection is made known by the Virtual Tester with the **ADD_ID** instruction. Then, the Virtual Tester registers (**mbx_register**) onto the stack by giving its application name (**JUPITER**). The Virtual Tester sends a message to an application under test (**SATURN**). Finally, the Virtual Testers unregisters itself (**mbx_unregister**) and frees the allocated resources (**mbx_end**).

```
HEADER "SystemTest 1st example: sending a message","1.0","
```

```
COMMTYPE appl_comm IS appl_id_t
```

```
MESSAGE message_t: message, ack, data, neg_ack
```

```
CHANNEL appl_comm: appl_ch
```

```
#appl_id_t id;

#int errcode;

PROCSEND message_t: msg ON appl_comm: id

CALL mbx_send_message ( &id, &msg) @ err_ok

END PROCSEND

SCENARIO first_scenario

FAMILY nominal

COMMENT Initialize, register, send data

COMMENT wait acknowledgement, unregister and release

CALL mbx_init(&id) @ err_ok @ errcode

ADD_ID(appl_ch,id)

VAR id.applname, INIT="JUPITER"

CALL mbx_register(&id) @ err_ok @ errcode

VAR message, INIT={

& type=>DATA,

& applname=>"SATURN",

& userdata=>"hello Saturn!"}

SEND ( message, appl_ch )

CALL mbx_unregister(&id) @ err_ok @ errcode

CLEAR_ID(appl_ch)

CALL mbx_end(&id) @ err_ok @ errcode

END SCENARIO
```

Receiving Messages

System Testing for C

CALLBACK Instruction

The event management provides an asynchronous mechanism to receive messages. This mechanism needs the definition of a callback for each couple communication type, message type.

A procedure should do a non-blocking read for a specific message type on a specific communication type.

The **MESSAGE_DATE** instruction lets you mark the right moment of the reception of messages. The **NO_MESSAGE** instruction exits from the callback and indicates that no message has been read.

The callback to receive messages from our system under test could be:

```
CALLBACK message_t: msg ON appl_comm: id

CALL mbx_get_message ( &id, &msg, 0 ) @@ errcode

MESSAGE_DATE

IF ( errcode == err_empty ) THEN

NO_MESSAGE

END IF

IF ( errcode != err_ok ) THEN

ERROR

END IF

END CALLBACK
```

In this example, **msg** is an output formal parameter of the callback. Its type is **message_t**.

When multiple connections are used, the input formal parameter **id** is used to know where to read a message on the communication type **appl_comm**.

The reading is done by the function call to **mbx_get_message**. The return code is stored into the variable **errcode**. The value **err_empty** for the return code means that no message has been read. Another value than **err_ok** or **err_empty** means that an error occurred during the reading. The **NO_MESSAGE** and **ERROR** instructions make the callback to return.

The script must have one **CALLBACK** for each *message* type - *channel* type pair used by any **WAITTIL** instructions.

The name of each **CALLBACK** in the generated C code is made up with the signature of the *message* type and *channel* type for each **CALLBACK** found in the test script.

DEF_MESSAGE Instruction

The **DEF_MESSAGE** instruction defines the values of a reference message declared with the **MESSAGE** instruction. A reference message is a set of field values as expected by the virtual tester from the system under test. Any undefined fields are not compared to the receive message.

```
DEF_MESSAGE ack, EV= { type => ACK }
```

```
DEF_MESSAGE data, EV= {
```

```
& type => DATA,
```

```
& applname => "SATURN",
```

```
& userdata => "hello world !" }
```

To learn more about the **DEF_MESSAGE** instruction, see the Messages and Data Management chapter.

WAITTIL Instruction

The **WAITTIL** instruction allows the test script to wait for events or conditions. **WAITTIL** is made of two Boolean expressions: an expected condition, and a failure condition. The script execution pauses until one of the two expressions becomes true.

In the following example, the **WAITTIL** instruction receives all the messages sent to the Virtual Tester on a known connection. As soon as a received message matches the reference message **ack**, the **WAITTIL** exits normally. Otherwise, if any message matching the reference message **ack** is received during 3000ms (300 x 10ms, the default time unit), the **WAITTIL** exits with an error. The scenario execution is interrupted.

```
WAITTIL ( MATCHING(ack, appl_ch), WTIME == 300)
```

The time unit is configurable in the Target Deployment Port depending on the target platform.

To receive a message on the appropriate channel, the generated code calls a **CALLBACK** named with the signature of the expected message type (first parameter) and the channel type (second parameter).

The message type is provided by the **MESSAGE** instruction. The channel type is provided by the **CHANNEL** instruction.

Therefore, in the generated code, the **SEND** instruction calls the following function:

```
CALLBACK_message_t_appl_comm(message, appl_ch)
```

which corresponds to the following line in the test script:

```
CALLBACK message_t ... ON appl_comm
```

If the channel parameter is omitted in the **WAITTIL** instruction, the generated code calls all **CALLBACK** instructions that read the corresponding message type on all known channel types.

In the example given above, the status of the reference event variable **ack** is tested using the function **MATCHING()** which identifies if the last incoming event corresponds to the content of the variable **ack**. **WTIME** is a reserved keyword valuated with the time expired since the beginning of the **WAITTIL** instruction.

The **WAITTIL** Boolean conditions are described using C or C++ conditions including operators to manipulate events:

- **MATCHING**: does the last event match the specified reference event?
- **MATCHED**: did the Virtual Tester receive an event matching the specified event?
- **NOMATCHING**: is the last event different from the specified reference event?
- **NOMATCHED**: did the Virtual Tester receive an event different from the specified event?

The different combinations of these operators allow an easy an extensive definition of event sequences:

– I expect evt1 on channel1 before my_timeout is reached

```
WAITTIL (MATCHING(evt1, channel1), WTIME>my_timeout)
```

– I expect evt1 then evt2 on one channel before my_timeout is reached

```
WAITTIL (MATCHED(evt1)&& MATCHING(evt2), WTIME>my_timeout)
```

– I expect to receive nothing during my_time

```
WAITTIL (WTIME>my_time, MATCHING(empty_evt))
```

– I expect evtA or evtB before my_timeout is reached

```
WAITTIL (MATCHING(evtA)||MATCHING(evtB), WTIME>my_timeout)
```

*

After the **WAITTIL** instruction, the value of these operators is available until the next call to **WAITTIL**.

Example

The following test script describes a simple use of our stack. First of all, some resources are allocated and a connection is established with the communication stack (**mbx_init**). This connection is made known by the Virtual Tester with the **ADD_ID** instruction. Then, the Virtual Tester registers (**mbx_register**) onto the stack giving its application name (**JUPITER**).

The Virtual Tester sends a message to an application under test (**SATURN**), and waits for the acknowledgment sent back by the stack with the **WAITTIL** instructions. Finally, the Virtual Tester unregisters (**mbx_unregister**) and frees the allocated resources (**mbx_end**).

```
HEADER "SystemTest 1st example: sending & receiving a message","1.0","
```

```
COMMTYPE appl_comm IS appl_id_t
```

MESSAGE message_t: message, ack, data, neg_ack

CHANNEL appl_comm: appl_ch

#appl_id_t id;

#int errcode;

PROCSEND message_t: msg ON appl_comm: id

CALL mbx_send_message (&id, &msg) @ err_ok

END PROCSEND

CALLBACK message_t: msg ON appl_comm: id

CALL mbx_get_message (&id, &msg, 0) @@ errcode

MESSAGE_DATE

IF (errcode == err_empty) THEN

NO_MESSAGE

END IF

IF (errcode != err_ok) THEN

ERROR

END IF

END CALLBACK

SCENARIO first_scenario

FAMILY nominal

COMMENT Initialize, register, send data

COMMENT wait acknowledgement, unregister and release

CALL mbx_init(&id) @ err_ok @ errcode

ADD_ID(appl_ch,id)

VAR id.applname, INIT="JUPITER"

CALL mbx_register(&id) @ err_ok @ errcode

VAR message, INIT={

```

& type=>DATA,
& applname=>"SATURN",
& userdata=>"hello Saturn!"}
SEND ( message, appl_ch )
COMMENT Negative acknowledgment expected
COMMENT (Saturn is not running !)
DEF_MESSAGE ack, EV={type=>ACK}
WAITTIL (MATCHING(ack), WTIME==10)
CALL mbx_unregister(&id) @ err_ok @ errcode
CLEAR_ID(appl_ch)
CALL mbx_end(&id) @ err_ok @ errcode
END SCENARIO

```

Related Topics

[Event Management on page 606](#) | [Basic Declarations on page 606](#) | [Sending Messages on page 608](#) | [Messages and Data Management on page 615](#)

Messages and Data Management

System Testing for C

The instruction VAR allows you to initialize and check the contents of simple or complex variables.

The process of initializing or checking variables is performed independently by the following two sub-instructions:

```
VAR <variable> , INIT = <init_expr>
```

or

```
VAR <variable> , EV = <expec_expr>
```

This instruction allows you to initialize and check the contents of structured variables, such as messages.

The field <variable> represents a variable or part of a structured variable.

<init_expr> and <expec_expr> let you describe the contents of structured variables using a simple syntax.

To describe a sequence of fields at the same level in a structured variable, you enclose the sequence in braces '{} ' or brackets '[]' and separate the fields with a comma ','.

You can reference members of a structured variable in the following ways:

- Reference by name
- Reference by position

You cannot however mix both methods.

The System Testing report does not show **VAR** instructions relating to initializations. Only **VAR** instructions relating to content checks on variables or messages are recorded in the test report.

The **DEF_MESSAGE** instruction allows you to define reference messages using the **DEF_MESSAGE** instruction, using exactly the same syntax. The following examples are presented using the **VAR** instruction, but are also applicable to **DEF_MESSAGE**.

The report does not show **DEF_MESSAGE** instruction as they appear in the test script, but only when they are used within a **WAITTIL** instruction.

Reference by Name

You can describe the contents of a structure by naming each field in the structure. This is very useful if you do not know the order of the fields in the declaration of the structure.

When referencing by name, a parameter is described by the name of the field in the structure followed by the arrow symbol (**=>**) and the initialization or checking expression.

```
#typedef struct
# {
# int Integer;
# char String [ 15 ];
# float Real;
# } block;
# block variable;

VAR variable, INIT={Real=>2.0, Integer=>26, String=>"foo"}
```

You can omit the specification of structure elements by name if you know the order of the fields within the structure. For the block type defined above, you can write the following **VAR** statement:

```
VAR variable, INIT={ 26, "foo", 2.0 }
```

Reference by Position

You can describe the contents of an array by giving the position of elements within the array.

When referencing by position, define a parameter by giving the position of the field in the array followed by the arrow symbol (\Rightarrow) and the initialization or checking expression.

Note that numbering begins at zero.

```
#int array[5];
```

```
VAR array, EV=[4=>5, 1=>12, 2=>-18, 5=>15-26, 3=>0, 0=>123]
```

You can use ranges of positions when referencing by position. These ranges are specified by two bounds separated by the symbol double full-stop (\dots).

```
#typedef int matrix[3][150];
```

```
VAR matrix, EV= [
```

```
& 2=>[0..99=>1, 100..149=>2],
```

```
& 0=>[99..0=>2, 100..149=>1],
```

```
& 1=>[0..80=>-1, 81..149=>0]]
```

Note that the bounds of an interval can be reversed.

When referencing by position, you must reference an entire sequence at a given level.

Partial Initialization and Checks

With a **VAR** instruction, you can partially initialize and check a structured variable.

```
#float array[10];
```

```
VAR array, INIT=[5..7=>2.1]
```

The array elements **5**, **6** and **7** are initialized to **2.1**. Other elements are not initialized.

Multi-dimension Initialization and Checks

With a **VAR** instruction, you can initialize and check multi-dimension variables with judicious use of bracket '[' and brace '}' separators.

The separators delimit the description of a structured variable to a given dimension. The absence of separators at a given level indicates that the initialization or checking value is valid for all the sub-dimensions of the variable.

In the following example:

- **Ex. 1:** The set of 300 integer values of the matrix variable are initialized to zero.
- **Ex. 2:** The 100 integer values contained in **matrix[0]** are initialized to **1**, the 100 values of **matrix[1]** are initialized to **2**, and the 100 values of **matrix[2]** are initialized to **3**.
- **Ex. 3:** Only the **matrix[0][0]** is initialized to zero.
- **Ex. 4:** Only the first 100 values of **matrix[0]** are initialized to zero.

```
#int matrix[3][100];
```

```
--Ex. 1- Global initialization
```

```
VAR matrix, INIT=0
```

```
--Ex. 2- Global initialization of lines
```

```
VAR matrix, INIT=[1,2,3]
```

```
--Ex. 3- Initialization of only one element
```

```
VAR matrix, INIT=[[0]]
```

```
--Ex. 4- Initialization of only one line
```

```
VAR matrix, INIT=[0]
```

The following example provides a set of **VAR** instructions that are semantically identical:

```
#int matrix[3][3];
```

```
VAR matrix, EV=0
```

```
VAR matrix, EV=[0,0,0]
```

```
VAR matrix, EV=[[0,0,0],[0,0,0],[0,0,0]]
```

In the three **VAR** instructions above, all the matrix elements are checked against zero.

Array Indices

With a **VAR** instruction, you can initialize and check array elements according to their index at a given level.

The index is specified by a capital I followed by the level number. Levels begin at **1**. You can use **I1**, **I2**, **I3**, etc. as implicit variables.

```
#int matrix[3][100];
```

```
VAR matrix, EV=I1*I2
```

Each element of the above matrix is checked against the product of variables **I1** and **I2**, which indicate, respectively, a range from 0 to 2 and a range from 0 to 99. The above matrix is checked against the 3 by 100 multiplication table.

Reference by Default

You can reference the remaining set of fields in an array, structure, or object in a **VAR** instruction. To do this, use the keyword **OTHERS**, followed by the arrow symbol =>, and an expression in C or C++.

Note: To use **OTHERS**, the remaining fields must be the same type and must be compatible with the expression following **OTHERS**.

```
#typedef struct {
# char String[25];
# int Value;
# int Value2;
# int Array[30];
#} block;
# block variable;
VAR variable, INIT=[
& String=>"chaine",
& Array=>[0..10=>0, OTHERS=>1] ,
& OTHERS=>2]
```

In the previous example, **OTHERS** has two functions:

- When initializing the array, the values indexed from 11 to 29 begin at 1.
- When initializing the structure, the value and value2 fields begin at 2.

Checking Pointers

With a **VAR** instruction, you may use **NIL** and **NONIL**, to check for null and non-null pointers.

```
#typedef struct {
# int a;
# float b;
#} block, *PT_block;
```

```
#PT_block addr[10];

VAR addr, EV=[0..5=>NIL, OTHERS=>NONIL]
```

In the above example, the pointers indexed from 0 to 5 of the `addr` array are compared with the null address. The test of the pointers indexed from 6 to 9 is correct if these pointers are different from the null address.

Checking Ranges

You may use ranges of acceptable values instead of immediate values. To do this, use the following syntax:

```
VAR <variable>, EV=[Min..Max]

DEF_MESSAGE <variable>, EV=[Min..Max]
```

The following example demonstrates this syntax:

```
#typedef struct {

# int a;

# float b;

#} block, *PT_block;

#PT_block addr[10];

VAR addr, EV=[0..5=>{a=>[0..100]}, OTHERS=>NONIL]
```

In the previous example, the elements indexed from 0 to 5 of the `addr` array are checked with the following constraint: `a` should be greater than 0 and lower than 100.

The test of the pointers indexed from 6 to 9 is correct if these pointers are different from null address

Character Strings

When you use the **VAR** instruction for character strings, you may alter it. In C, a character string can also be an array. This flexibility is retained in the VAR instruction.

In the following example, the first variable **String** initializes as in C (null-terminated). The second **String** initializes as an array of characters (not null-terminated).

```
#char String[15];

VAR String, INIT="abcdef"

VAR String, INIT=['a', 'b', 'c', 'd', 'e', 'f']
```

Note You must define the **VAR** instruction either as a character string or an array of characters.

Communication Between Virtual Testers

System Testing for C

Virtual Testers can communicate between themselves with simple messages by using the **INTERSEND** and **INTERRECV** statements. Virtual Tester messages can be either an *integer* or a *text string*.

For information about the **INTERSEND** and **INTERRECV** statements, please refer to the System Testing Script Language section in the **Studio Reference pages of the help**.

For these statements to be active, you must enable On-the-fly Runtime Tracing in the Configuration Settings.

To enable Virtual Tester communication:

1. In the Project Explorer, select the System Testing test node, and click **Settings**.
2. In the **Configuration Settings** dialog box, select **System Testing** and **Target Deployment Port for System Testing**.
3. Set **Enable On-the-fly Runtime Tracing** to **Yes** and click **OK**.

Identifier

For message delivery purposes, each Virtual Testers carries a unique *identifier*. The virtual tester identifier is constructed with the following rules:

- If the Virtual Tester is run as an instance named *<instance>*:

<instance>_<occid>

- If the Virtual Tester is running in multi-threaded mode, with its entry point in *<function>*:

<function_name>_<occid>

- In any other case, the identifier uses the **.rio** file name:

<filename>.rio_<occid>

By default the occurrence identification number *<occid>* for each Virtual Tester is 0, but you can set different *<occid>* values in the [Virtual Tester Deployment on page 592](#) dialog box.

There must never be two Virtual Testers at the same time with the same identifier. If an **INTERSEND** message cannot be delivered because of an ambiguous identifier, the System Testing supervisor returns an error message.

Related Topics

[About Virtual Testers on page 586](#) | [Configuring Virtual Testers on page 591](#) | [System Testing in a Multi-Threaded or RTOS Environment on page 596](#) | [Deploying Virtual Testers on page 592](#)

Instances

Instances

System Testing for C

In a distributed environment, you can merge the description of several entities, Virtual Testers, in a unique test script. This is possible through the concept of interaction instances, as defined in UML.

Hence, you create Virtual Testers, all based on a same test script, with distinct behaviors such as a client and a server or both.

The use of instances in a test script must be split into two parts, as follows:

- The [declaration of the instances on page 622](#) used in test script
- The [description of the instances on page 623](#) by specific blocks containing declarations or instructions.

Related Topics

[Instance Declaration on page 622](#) | [Instance Synchronization on page 623](#) | [About Virtual Testers on page 586](#)

Instance Declaration

System Testing for C

The **DECLARE_INSTANCE** instruction lets you declare the set of the instances included in the test script.

Note Each instance behavior will be translated into different Virtual Testers executed within a process or a thread.

The **DECLARE_INSTANCE** instruction must be located before the top-level scenario.

The instance declaration can be done by one or several **DECLARE_INSTANCE** instructions. They must appear in the test script in such a way that no **INSTANCE** block containing global declarations uses an instance that has not been previously declared.

Example

```
HEADER "Multi-server / Multi-client example","1.0","
```

```
DECLARE_INSTANCE server1, server2
```

```
...
```

```
DECLARE_INSTANCE client1, client2, client3
```

...

SCENARIO Principal

...

Related Topics

[DECLARE_INSTANCE on page 846](#) | [Instance synchronization on page 623](#)

Instance Synchronization

System Testing for C

The **RENDEZVOUS** statement, provides a way to synchronize Virtual Testers to each instance.

When a scenario is executed, the **RENDEZVOUS** instruction stops the execution until all Virtual Testers sharing this synchronization point (the identifier) have reached this statement.

When all Virtual Testers have met the rendezvous, the scenario resumes.

SCENARIO first_scenario

FAMILY nominal

– Synchronization point shared by both Instances

RENDEZVOUS sync01

INSTANCE JUPITER:

RENDEZVOUS sync02

...

END INSTANCE

INSTANCE SATURN:

RENDEZVOUS sync02

...

END INSTANCE

END SCENARIO

Synchronization can be shared with other parts of the test bench such as in-house Virtual Testers, specific feature , and so on. This can be done easily by linking these pieces with the current Target Deployment Port.

Then, to define a synchronization point, you must make a call to the following function:

```
atl_rdv("sync01");
```

This synchronization point matches the following instruction used in a test script:

```
RENDEZVOUS sync01
```

Example

The following test script is based on the example developed in the [Event Management on page 606](#) section. The script provides an example of the usefulness of instances for describing several applications in a same test script.

```
HEADER "SystemTest Instance-including Scenario Example", "1.0", ""
```

```
DECLARE_INSTANCE JUPITER, SATURN
```

```
COMMTYPE appl_comm IS appl_id_t
```

```
MESSAGE message_t: message, data, my_ack, neg_ack
```

```
CHANNEL appl_comm: appl_ch
```

```
#appl_id_t id;
```

```
#int errcode;
```

```
PROCSEND message_t: msg ON appl_comm: id
```

```
CALL mbx_send_message( &id, &msg ) @ err_ok
```

```
END PROCSEND
```

```
CALLBACK message_t: msg ON appl_comm: id
```

```
CALL mbx_get_message ( &id, &msg, 0 ) @@ errcode
```

```
MESSAGE_DATE
```

```
IF ( errcode == err_empty ) THEN
```

```
NO_MESSAGE
```

```
END IF
```

```
IF ( errcode != err_ok ) THEN
```

```
ERROR
```

```
END IF
```

```
END CALLBACK
```

```

SCENARIO first_scenario

FAMILY nominal

COMMENT Initialize, register, send data

COMMENT wait acknowledgement, unregister and release

CALL mbx_init(&id) @ err_ok @ errcode

ADD_ID(appl_ch,id)

INSTANCE JUPITER:

VAR id.applname, INIT="JUPITER"

END INSTANCE

INSTANCE SATURN:

VAR id.applname, INIT="SATURN"

END INSTANCE

CALL mbx_register(&id) @ err_ok @ errcode

COMMENT Synchronization of both instances

RENDEZVOUS start_RDV

INSTANCE JUPITER:

VAR message, INIT={type=>DATA,num=>id.s_id,
& applname=>"SATURN",
& userdata=>"Hello Saturn!"}

SEND( message , appl_ch )

DEF_MESSAGE my_ack, EV={type=>ACK}

WAITTIL (MATCHING(my_ack), WTIME==300)

DEF_MESSAGE data, EV={type=>DATA}

WAITTIL (MATCHING(data), WTIME==1000)

END INSTANCE

INSTANCE SATURN:

```

```

DEF_MESSAGE data, EV={type=>DATA}

WAITTIL (MATCHING(data), WTIME==1000)

VAR message, INIT={type=>DATA,num=>id.s_id,
& applname=>"JUPITER",
& userdata=>"Fine, Jupiter!"}

SEND( message , appl_ch )

DEF_MESSAGE my_ack, EV={type=>ACK}

WAITTIL (MATCHING(my_ack), WTIME==300)

END INSTANCE

CALL mbx_unregister(&id) @ err_ok @ errcode

CLEAR_ID(appl_ch)

CALL mbx_end(&id) @ err_ok @ errcode

COMMENT Termination Synchronization

RENDEZVOUS term_RDV

END SCENARIO

```

The scenario describes the behavior of two applications (**JUPITER** and **SATURN**) exchanging messages by using a communications stack.

Some needed resources are allocated and a connection is established with the communication stack (**mbx_init**). This connection is made known by the Virtual Tester with the **ADD_ID** instruction. Note that this is a common part to both instances.

Then, the two applications register (**mbx_register**) onto the stack by giving their application name (**JUPITER** or **SATURN**). These operations are specific to each instance, which is why these operations are done in two separate instance blocks.

The application **JUPITER** sends the message "Hello Saturn!" to the **SATURN** application (through the communication stack) which is supposed to have set itself in a message waiting state (**WAITTIL (MATCHING(data), ...)**).

Once the message has been sent, **JUPITER** waits for an acknowledgment from the communication stack (**WAITTIL(my_ack),...**). Then, it waits for the response of **SATURN** (**WAITTIL (MATCHING(data),...)**) which answers by the message "Fine, Jupiter!" (**SEND(message , appl_ch)**). These operations are specific to each instance.

Finally, the applications unregister themselves and free the allocated resources in the last part, which is common to both instances.

Related Topics

[Instance declarations on page 622](#) | [RENDEZVOUS on page 875](#) | [MATCHED\(\) on page 862](#) | [MATCHING\(\) on page 863](#) | [NOTMATCHED\(\) on page 867](#) | [NOTMATCHING\(\) on page 868](#) | [WAITTIL on page 889](#)

Environments

Environments

System Testing for C

When creating a test script, you typically write several test scenarios. These scenarios are likely to require the same resources to be deployed and then freed. You can avoid writing a series of scenarios containing similar code by factorizing elements of the scenario.

To resolve these problems and leverage your test script writing, you can define environments introduced by the keywords **INITIALIZATION**, **TERMINATION**, and **EXCEPTION**.

This section describes

- [Error Handling on page 627](#)
- [Exception Environment \(Error Recovery Block\) on page 629](#)
- [Initialization Environment on page 630](#)
- [Termination Environment on page 632](#)

Error Handling

System Testing for C

The **ERROR** Statement

The **ERROR** instruction lets you interrupt execution of a scenario where an error occurs and continue on to the next scenario at the same level.

ERROR instructions follow these rules:

- **ERROR** instructions can be located in scenarios, in procedures, or in environment blocks.
- If an **ERROR** instruction is encountered in an **INITIALIZATION** block, the Virtual Tester exits with an error from the set of scenarios at the same level.

Note In debug mode, the behavior of **ERROR** instructions is different (see Debugging Virtual Testers).

The following is an example of an **ERROR** instruction:

```
HEADER "Instruction ERROR", "1.0", "1.0"

#int IdConnection;

SCENARIO Main

COMMENT connection

CALL socket(AF_UNIX, SOCK_STREAM, 0)@@IdConnection

IF (IdConnection == -1) THEN

ERROR

END IF

END SCENARIO
```

The EXIT Statement

The **EXIT** instruction lets you interrupt execution of a Virtual Tester. Subsequent scenarios are not executed.

EXIT instructions follow these rules:

- **EXIT** instructions can be located in scenarios, procedures, or environment blocks.
- If an **EXIT** instruction is encountered, the **EXCEPTION** blocks are not executed.

The following is an example of an **EXIT** instruction:

```
HEADER "Instruction EXIT", "1.0", "1.0"

#int IdConnection;

SCENARIO Main

COMMENT connection

CALL socket(AF_UNIX, SOCK_STREAM, 0)@@IdConnection

IF (IdConnection == -1) THEN

EXIT

END IF

END SCENARIO
```

Related Topics

[Exception Environment \(Error Recovery Block\) on page 629](#) | [Environments on page 627](#) | [Initialization Environment on page 630](#) | [Termination Environment on page 632](#)

Exception Environment (Error Recovery Block)

System Testing for C

A test script is composed of a hierarchy of scenarios. An exception environment can be defined at a given scenario level.

When an error occurs in a scenario all exception blocks at the same level or above are executed sequentially.

The syntax for exception environments can take two different forms, as follows:

- **A block:** This begins with the keyword **EXCEPTION** and ends with the sequence **END EXCEPTION**. A termination block can contain any instruction.
- **A procedure call:** This begins with the keyword **EXCEPTION** followed by the name of the procedure and, where appropriate, its arguments.

Example

In the following example, the highest level of the test script is made up of two scenarios called first and second. The exception environment that precedes them is executed once if scenario premier finished with an error, and once if scenario second finishes with an error.

```
HEADER "Validation", "01a", "01a"
```

```
PROC Unload_mem()
```

```
...
```

```
END PROC
```

```
EXCEPTION Unload_mem()
```

```
SCENARIO first
```

```
...
```

```
END SCENARIO
```

```
SCENARIO second
```

```
EXCEPTION
```

```
...
```

END EXCEPTION

SCENARIO level2_1

FAMILY nominal, structural

...

END SCENARIO

SCENARIO level2_2

FAMILY nominal, structural

...

END SCENARIO

END SCENARIO

Scenario second is made up of two sub-scenarios, level2_1 and level2_2. The second exception environment is executed after incorrect execution of scenarios level2_1 and level2_2. The highest-level exception environment is not re-executed if scenarios level2_1 and level2_2 finish with an error.

Only one exception environment can appear at a given scenario level.

An exception environment can appear among scenarios at the same level. It does not have to be placed before a set of scenarios at the same level.

In a test report, the execution of an exception environment is shown even if you decided not to trace the execution.

Related Topics

[Error Handling on page 627](#) | [Environments on page 627](#)

Initialization Environment

System Testing for C

A test script is composed of scenarios in a tree structure. An initialization environment can be defined at a given scenario level.

This initialization environment is executed before each scenario at the same level.

The syntax for initialization environments can take two different forms, as follows:

- **A block:** This begins with the keyword **INITIALIZATION** and ends with the sequence **END INITIALIZATION**. An initialization block can contain any instruction.
- **A procedure call:** This begins with the keyword **INITIALIZATION** followed by the name of the procedure and, where appropriate, its arguments.

Example

In the following example, the highest level of the test script is made up of two scenarios called first and second. The initialization environment that precedes them is executed twice: once before scenario first is executed and once before scenario second is executed.

```
HEADER "Validation", "01a", "01a"
```

```
PROC Load_mem()
```

```
...
```

```
END PROC
```

```
INITIALIZATION Load_mem()
```

```
SCENARIO first
```

```
...
```

```
END SCENARIO
```

```
SCENARIO second
```

```
INITIALIZATION
```

```
END INITIALIZATION
```

```
SCENARIO level2_1
```

```
FAMILY nominal, structural
```

```
...
```

```
END SCENARIO
```

```
SCENARIO level2_2
```

```
FAMILY nominal, structural
```

```
...
```

```
END SCENARIO
```

```
END SCENARIO
```

Scenario *second* is made up of two sub-scenarios, *level2_1* and *level2_2*. The second initialization environment is executed before scenarios *level2_1* and *level2_2* are executed. The highest-level initialization environment is not re-executed between scenarios *level2_1* and *level2_2*.

Only one initialization environment can appear at a given scenario level.

An initialization environment can appear among scenarios at the same level. The initialization environment does not have to be placed before a set of scenarios at the same level.

In a test report, the execution of an initialization environment is shown beginning with the word **INITIALIZATION** and ending with the words **END INITIALIZATION**.

Related Topics

[Termination Environment on page 632](#) | [Environments on page 627](#)

Termination Environment

System Testing for C

A test script is composed of scenarios in a tree structure. A termination environment can be defined at a given scenario level.

This termination environment is executed at the end of every scenario at the same level, provided that each scenario finished without any errors.

The syntax for termination environments can take two different forms, as follows:

- **A block:** This begins with the keyword **TERMINATION** and ends with the sequence **END TERMINATION**. A termination block can contain any instruction.
- **A procedure call:** This begins with the keyword **TERMINATION** followed by the name of the procedure and, where appropriate, its arguments.

Example

In the previous example, the highest level of the test script is made up of two scenarios called *first* and *second*. The termination environment that precedes them is executed twice:

- once after scenario *first* is executed correctly
- once after scenario *second* is executed correctly

```
HEADER "Validation", "01a", "01a"
```

```
PROC Unload_mem()
```

```
...
```

```

END PROC

TERMINATION Unload_mem()

SCENARIO first

...

END SCENARIO

SCENARIO second

TERMINATION

...

END TERMINATION

SCENARIO level2_1

FAMILY nominal, structural

...

END SCENARIO

SCENARIO level2_2

FAMILY nominal, structural

...

END SCENARIO

END SCENARIO

```

Scenario *second* is made up of two sub-scenarios, *level2_1* and *level2_2*. The second termination environment is executed after the correct execution of scenarios *level2_1* and *level2_2*. The highest-level termination environment is not re-executed between scenarios *level2_1* and *level2_2*.

Only one termination environment can appear at a given scenario level.

A termination environment can appear among scenarios at the same level. The termination environment does not have to be placed before a set of scenarios at the same level.

In a test report, the execution of a termination environment is shown beginning with the word **TERMINATION** and ending with the words **END TERMINATION**.

Related Topics

[Initialization Environment on page 630](#) | [Environments on page 627](#) | [Basic Structure](#)

Time management

Time management

System Testing for C

In some cases, you will need information about execution time within a test script.

The following instructions provide a way to dump timing data, define a timer, clear a timer, get the value of a timer, and temporarily suspend test script execution:

- [TIME instruction on page 634](#)
- [TIMER instruction on page 635](#)
- [RESET instruction on page 636](#)
- [PRINT instruction on page 637](#)
- [PAUSE instruction on page 637](#)

TIME instruction

System Testing for C

The **TIME** instruction returns the current value of a timer. You must use a C expression or scripting instruction (**IF**, **PRINT**, and so on).

Before using **TIME**, you must declare the timer with the [TIMER on page 635](#) instruction.

Example

```
HEADER "Socket validation", "1.0", "beta"
```

```
TIMER globalTime
```

```
PROC first
```

```
TIMER firstProc
```

```
...
```

```
PRINT globalTimeValue, TIME (globalTime)
```

```
END PROC
```

```
SCENARIO second
```

```
SCENARIO level2
```

```
TIMER level2Scn
...
PRINT level2ScnValue, TIME (level2Scn)
END SCENARIO
END SCENARIO
```

Related Topics

[TIMER Instruction on page 635](#) | [RESET Instruction on page 636](#) | [PRINT Instruction on page 637](#) | [PAUSE Instruction on page 637](#)

TIMER instruction

System Testing for C

The **TIMER** instruction declares a timer in the test script.

You may declare a timer in any test script block: global, initialization, termination, exception, procedure, or scenario.

The timer lasts as long as the block in which the timer is defined. This means that a timer defined in the global block can be used until the end of the test script.

You may define multiple timers in the same test script. The timer starts immediately after its declaration.

The unit of the timer unit is defined during execution of the application, with the **WAITTIL** and **WTIME** instructions.

Example

```
HEADER "Socket validation", "1.0", "beta"
```

```
TIMER globalTime
```

```
PROC first
```

```
TIMER firstProc
```

```
...
```

```
END PROC
```

```
SCENARIO second
```

```
SCENARIO level2
```

```
TIMER level2Scn
```

```
...
```

END SCENARIO

END SCENARIO

Related Topics

[TIME Instruction on page 634](#) | [RESET Instruction on page 636](#) | [PRINT Instruction on page 637](#) | [PAUSE Instruction on page 637](#)

RESET instruction

System Testing for C

The **RESET** instruction lets you reset a timer to zero.

The timer restarts immediately when the **RESET** statement is encountered.

A timer must be declared before using **RESET**.

Example

```
HEADER "Socket validation", "1.0", "beta"
```

```
TIMER globalTime
```

```
PROC first
```

```
TIMER firstProc
```

```
RESET globalTime
```

```
...
```

```
END PROC
```

```
SCENARIO second
```

```
SCENARIO level2
```

```
TIMER level2Scn
```

```
...
```

```
RESET level2Scn
```

```
END SCENARIO
```

```
END SCENARIO
```

Related Topics

[TIME Instruction on page 634](#) | [TIMER Instruction on page 635](#) | [PRINT Instruction on page 637](#) | [PAUSE Instruction on page 637](#)

PRINT instruction

System Testing for C

You can print the result of an expression in a performance report by using the **PRINT** statement. The **PRINT** instruction prints an identifier before the expression.

Example

```
HEADER "Socket validation", "1.0", "beta"
```

```
#long globalTime = 45;
```

```
SCENARIO first
```

```
PRINT timeValue, globalTime
```

```
END SCENARIO
```

```
SCENARIO second
```

```
SCENARIO level2
```

```
PRINT time2Value, globalTime*10+5
```

```
...
```

```
END SCENARIO
```

```
END SCENARIO
```

Related Topics

[TIME Instruction on page 634](#) | [TIMER Instruction on page 635](#) | [RESET Instruction on page 636](#) | [PAUSE Instruction on page 637](#)

PAUSE instruction

System Testing for C

The **PAUSE** instruction lets you temporarily stop test script execution for a given period.

The unit of the **PAUSE** instruction is defined during execution of the application, with the **WAITTIL** and **WTIME** instructions.

Example

```
HEADER "Socket validation", "1.0", "beta"
```

```
#long time = 20;

PROC first

PAUSE 10

...

END PROC

SCENARIO second

SCENARIO level2

PAUSE time*10

...

END SCENARIO

END SCENARIO

Related Topics
```

[TIME Instruction on page 634](#) | [TIMER Instruction on page 635](#) | [RESET Instruction on page 636](#) | [PRINT Instruction on page 637](#)

Using native C statements

System Testing for C

In some cases, it can be necessary to include portions of C native code inside a **.pts** test script for one the following reasons:

- To declare native variables that participate in the flow of a scenario. Such statements must be analyzed by the System Testing Parser.
- To insert native code into a scenario. In this case, the code is ignored by the parser, but carried on into the generated code.

Analyzed native code

Lines prefixed with a **#** character are analyzed by Component Testing Parser.

Only prefix statements with a **#** character to include native C variable declarations that must be analyzed by the parser.

```
#int i;

#char *foo;
```

Variable declarations must be placed outside of System Testing Language blocks or preferably at the beginning of scenarios and procedures.

Ignored native code

Lines prefixed with a @ character are ignored by the parser, but copied into the generated code.

To use native C code in the test script, start instructions with a @ character:

```
@for(i=0; i++; i<100) func(i);
```

```
@foo(a,&b,c);
```

You can add native code either inside or outside of System Testing Language blocks.

Related Topics

[CALL Instruction on page 599](#) | [Basic Structure](#) | [System Testing language reference on page 893](#)

Understanding System Testing for C Reports

Understanding System Testing for C Reports

System Testing for C

Test reports for System Testing are displayed in HCL OneTest™ Embedded Report Viewer.

The test report is a hierarchical summary report of the execution of a test node. Parts of the report that have *Passed* are displayed in green. *Failed* tests are shown in red.

Report Explorer

The [Report Explorer on page 1024](#) displays each element of a test report with a *Passed* , *Failed*  symbol.

- Elements marked as *Failed*  are either a failed test, or an element that contains at least one failed test.
- Elements marked as *Passed*  are either passed tests or elements that contain only passed tests.

Test results are displayed for each instance, following the structure of the **.pts** test script.

Report Header

Each test report contains a report header with:

- The version of HCL OneTest™ Embedded used to generate the test as well as the date of the test report generation
- The path and name of the project files used to generate the test

- The total number of test cases *Passed* and *Failed*. These statistics are calculated on the actual number of test elements listed in the sections below
- Virtual Tester information.

Main Report Sections

For each Virtual Tester execution, the report lists the details of test script execution, with time stamps and test result tables.

- **Messages:** The report displays fields and values for each field
- **Tests Results:** For each message, the report compares initial values, expected values and obtained values

Related Topics

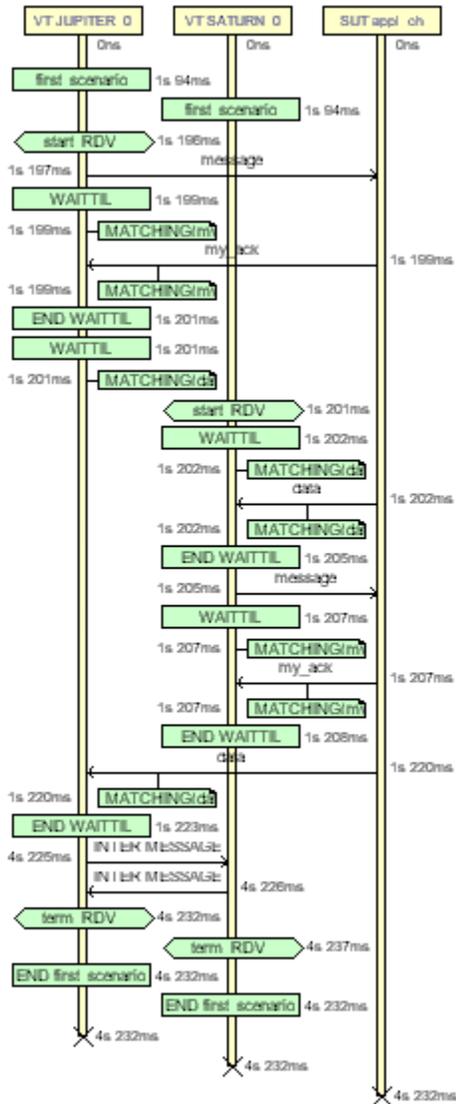
[Understanding System Testing UML Sequence Diagrams on page 640](#) | [Using the Report Viewer on page 705](#) | [Opening a Report on page 683](#) | [Exporting reports on page 705](#)

Understanding System Testing UML Sequence Diagrams

System Testing for C

During the execution of the test, System Testing generates trace data this is used by the UML/SD Viewer. The System Testing sequence diagram uses standard UML notation to represent both System Testing results.

This is an example of a typical System Testing UML sequence diagram.



You can modify the appearance of UML sequence diagrams by changing the [UML/SD Viewer Preferences](#) on page 1016.

When using System Testing with Runtime Tracing or other HCL OneTest™ Embedded features that generate UML sequence diagrams, all results are merged in the same sequence diagram.

You can click any element of the UML sequence diagram to open the [System Testing reports](#) on page 639 at the corresponding line. Click again in the test report, and you will locate the line in the `.pts` test script.

Virtual Testers and System Under Test

The system under test (SUT) and the Virtual Testers (VT) are represented as vertical instances. Messages sent and received by the Virtual Tester are represented along the Virtual Tester lifeline.

Messages

Messages are sent and received between Virtual Tester and system instances.

Rendezvous

RENDEZVOUS statements are displayed as [Synchronizations on page 653](#) in the Virtual Tester lifeline.

Test Script Events and Errors

Test script events and errors are represented as UML [actions on page 642](#). Only significant instructions, such as **INITIALIZATION**, **WAITTIL** blocks and test errors are represented.

By default, errors appear in red. Other events are green.

WAITTIL blocks are displayed with their start and end events. Matching conditions are represented as [notes on page 650](#). Use the mouse cursor tool-tip to get more information about the matching conditions.

On-the-Fly Tracing

If you are using the On-the-Fly option, only the following information can be displayed in real-time during the execution of the application:

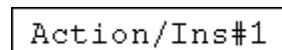
- Virtual Tester and system under test
- Messages
- Rendezvous
- Test script blocks

Related Topics

[About the UML/SD Viewer on page 402](#) | [UML/SD Viewer Toolbar on page 1028](#) | [Understanding System Testing Reports on page 639](#) | [On-the-Fly Tracing on page 656](#)

Actions

An action is represented as shown below:



```
Action/Ins#1
```

The action box displays the name of the action.

The action is linked to its source file. In the UML/SD Viewer, click an action to open the Text Editor at the corresponding line in the source code.

Related Topics

[UML Sequence Diagrams on page 397](#) | Model Elements and Relationships in Sequence Diagrams

Actors

An actor is a model element that describes a role that a user plays when interacting with the system being modeled. Actors, by definition, are external to the system. Although an actor typically represents a human user, it can also represent an organization, system, or machine that interacts with the system. An actor can correspond to multiple real users, and a single user may play the role of multiple actors.

Shape

An actor usually appears as a "stick man" shape.



In models depicting software applications, actors represent the users of the system. Examples include end users, external computer systems, and system administrators.

Naming Conventions

Each actor has a unique name that describes the role the user plays when interacting with the system.

Related Topics

[Objects on page 650](#) | [UML Sequence Diagrams on page 397](#)

Activations

An activation (also known as a focus of control) is a notation that can appear on a lifeline to indicate the time during which an instance (an actor instance, object, or classifier role) is active. An active instance is performing an action, such as executing an operation or a subordinate operation. The top of the activation represents the time at which the activation begins, and the bottom represents the time at which the activation ends.

For example, in a sequence diagram for a "Place Online Order" interaction, there are lifelines for a ":Cart" object and ":Order" object. An "updateTotal" message points from the ":Order" object to the ":Cart" object. Each lifeline has an activation to indicate how long it is active because of the "updateTotal" message.

Shape

An activation appears as a thin rectangle on a lifeline. You can stack activations to indicate nested stack frames in a calling sequence.

Using Activations

Activations can appear on your sequence diagrams to represent the following:

- On lifelines depicting instances (actors, classifier roles, or objects), an activation typically appears as the result of a message to indicate the time during which an instance is active.
- On lifelines involved in complex interactions, nested activations (also known as stacked activations or nested focuses of control) are displayed to indicate nested stack frames in a calling sequence, such as those that happen during recursive calls.
- On lifelines depicting concurrent operations, the entire lifeline may appear as an activation (thin rectangles) instead of dashed lines.

Naming Conventions

An activation is usually identified by the incoming message that initiates it. However, you may add text labels that identify activations either next to the activation or in the left margin of the diagram.

Related Topics

[Classifier Roles on page 644](#) | [Lifelines on page 647](#) | [Messages on page 649](#) | [Objects on page 650](#) | [UML Sequence Diagrams on page 397](#) | [Stimuli on page 652](#)

Classifier Roles

A classifier role is a model element that describes a specific role played by a classifier participating in a collaboration without specifying an exact instance of a classifier. A classifier role is neither a class nor an object. Instead, it is a model element that specifies the kind of object that must ultimately fulfill the role in the collaboration. The classifier role limits the kinds of classifier that can be used in the role by referencing a base classifier. This reference identifies the operations and attributes that an instance of a classifier will need in order to fulfill its responsibilities in the collaboration.

Classifier roles are commonly used in collaborations that represent patterns. For example, a subject-observer pattern may be used in a system. One classifier role would represent the subject, and one would represent the observer. Each role would reference a base class that identifies the attributes and operations that are needed to participate in the subject-observer collaboration. When you use the pattern in the system, any class that has the specified operations and behaviors can fill the role.

Shape

A classifier role appears as a rectangle. Its name is prefixed with a slash and is not underlined. In sequence diagrams, a lifeline (a dashed, vertical line) is attached to the bottom of a classifier role to represent its life over a period of time. For details about lifelines, see [Lifelines on page 647](#).

Classifier Role	Classifier Role with Life-line
------------------------	---------------------------------------

Using Classifier Roles

Classifier roles can appear on a model to represent the following:

- In models depicting role-based interactions, a classifier role represents an instance in an interaction. Using classifier roles instead of objects can provide two advantages: First, a class can serve as the base classifier for multiple classifier roles. Second, instances of a class can realize multiple classifier roles in one or more collaborations.
- In models depicting patterns, a classifier role specifies the kind of object that must ultimately fulfill a role in the pattern. The classifier role shows how the object will participate in the pattern, and its reference to a base class defines the attributes and operations that are required for participation in the pattern. When the pattern is used in the model, classes are bound to the collaboration to identify the type of objects that realize the classifier roles.

The classifier roles in a model are usually contained in a collaboration and usually appear in sequence diagrams.

Naming Conventions

The name of a classifier role consists of a role name and base class name. You can omit one of the names. The following table identifies the variations of the naming convention.

Convention	Example	Description
/rolename:base-class	/courseOffering:course	The courseOffering role is based on the course class.
/rolename	/courseOffering	Role name. The base class is hidden or is not defined.
:baseclass	:course	Unnamed role based on the course class.

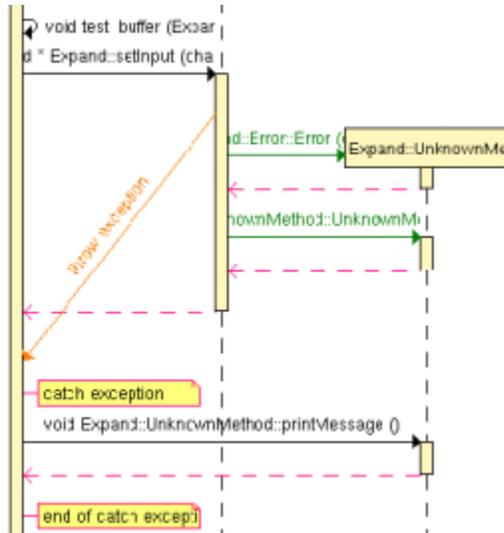
Related Topics

[Objects on page 650](#)

Exceptions

When tracing C++ exceptions, Runtime Tracing locates the throw point of the exception (the throw keyword in C++) as well as its catch point.

Exceptions are displayed as a slanted red line, as shown in the example below, generated by Runtime Tracing.



To jump to the corresponding portion of source code:

1. Click an instance to open the Text Editor at the line in the source code where the exception is thrown.
2. Click the **catch exception** or **end of catch exception** notes to open the Text Editor at the line where the exception is caught.

To filter an instance out of the UML sequence diagram:

1. Right-click an exception and select **Filter instance** in the pop-up menu.

Related Topics

[UML Sequence Diagrams on page 397](#) | Model Elements and Relationships in Sequence Diagrams

Destruction Markers

A destruction marker (also known as a termination symbol) is a notation that can appear on a lifeline to indicate that an instance (object or classifier role) has been destroyed. Usually, the destruction of an object results in the memory occupied by the data members of the object being freed.

For example, when a customer exits the Web site for an e-commerce application, the ":Cart" object that held information about the customer's activities is destroyed, and the memory that it used is freed. The destruction of the ":Cart" object can be shown in a sequence diagram by adding a destruction marker on the ":Cart" object's lifeline.

Shape

A destruction marker appears as an X at the end of a lifeline.

Naming Conventions

Destruction markers do not have names.

Related Topics

[Classifier Roles on page 644](#) | [Lifelines on page 647](#) | [Messages on page 649](#) | [Objects on page 650](#)

Lifelines

A lifeline is a notation that represents the existence of an object or classifier role over a period of time. Lifelines appear only in sequence diagrams, where they show how each instance (object or classifier role) participates in the interaction.

For example, a "Place Online Order" interaction in an e-commerce application includes a number of lifelines in a sequence diagram, including lifelines for a ":Cart" object, ":OnlineOrder" object, and ":CheckoutCart" object. As the interaction is developed, stimuli are added between the lifelines.

Shape

A lifeline appears as a vertical dashed line in a sequence diagram.

Lifeline for an Object	Lifeline for a Classifier Role
-------------------------------	---------------------------------------

Using Lifelines

When a classifier role or object appears in a sequence diagram, it will automatically have a lifeline. Lifelines indicate the following:

- **Creation** – If an instance is created during the interaction, its lifeline starts at the level of the message or stimulus that creates it; otherwise, its lifeline starts at the top of the diagram to indicate that it existed prior to the interaction.
- **Communication** – Messages or stimuli between instances are illustrated with arrows. A message or stimulus is drawn with its end on the lifeline of the instance that sends it and its arrowhead on the lifeline of the instance that receives it.
- **Activity** – The time during which an instance is active (either executing an operation directly or through a subordinate operation) can be shown with activations.
- **Destruction** – If an instance is destroyed during the interaction, its lifeline ends at the level of the message or stimulus that destroys it, and a destruction marker appears; otherwise, its lifeline extends beyond the final message or stimulus to indicate that it exists during the entire interaction.

Naming Conventions

A lifeline has the name of an object or classifier role. For details, see [Objects on page 650](#) or Classifier Roles.

Related Topics

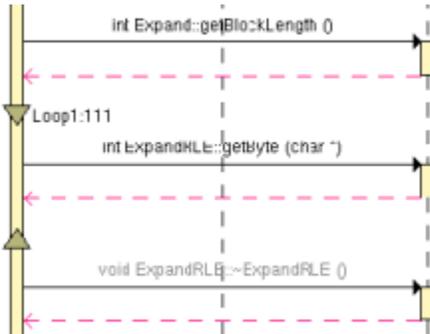
[Classifier Roles on page 644](#) | Model Elements and Relationships in Sequence Diagrams | [Messages on page 649](#) | [Objects on page 650](#) | [Stimuli on page 652](#)

Loops

Loop detection simplifies UML sequence diagrams by summarizing repeating traces into a loop symbol.

Note Loops are extensions to UML Sequence Diagrams and are not supported by the UML standard.

A loop is represented as shown below:



A tag displays the name of the loop and the number of executions.

The loop is linked to its source file. In the UML/SD Viewer, click a loop to open the Text Editor at the corresponding line in the source code.

To configure Runtime Tracing to detect loops:

1. From the Project Explorer, select the highest level node to which you want to apply the option, such as the Workspace.
2. Right-click the node, and select **Settings...** from the pop-up menu.
3. In the **Configuration Settings** dialog, select the **Runtime Tracing** node, and **Trace Control**.
4. From the options box, set the **Automatic Loop Detection** to **Yes**.
5. Click **OK**.

Related Topics

[UML Sequence Diagrams on page 397](#) | Model Elements and Relationships in Sequence Diagrams

Messages

A message is a model element that specifies a communication between classifier roles and usually indicates that an activity will follow. The types of communications that messages model include calls to operations, signals to classifier roles, the creation of classifier roles, and the destruction of classifier roles. The receipt of a message is an instance of an event.

For example, in the observer pattern, the instance that is the subject sends an "Update" message to instances that are observing it. You can illustrate this behavior by adding "Subject" and "Observer" classifier roles and then adding an "Update" message between them.

Shape

A message appears as a line with an arrow. The direction of the arrow indicates the direction in which the message is sent. In a sequence diagram, messages usually connect two classifier role lifelines.

Message shapes can be adorned with names and sequence numbers.

Types of Messages

Different types of messages can be used to model different flows of control.

Type	Shape	Description
Procedure Call or Nested Flow of Control		Models either a call to an operation or a call to a nested flow of control. When calling a nested flow of control, the system waits for the nested flow of control to complete before continuing with the outer flow.
Asynchronous Flow of Control		Models an asynchronous message between two objects. The source object sends the message and immediately continues with the next step.
Return From a Procedure Call		Models a return from a call to a procedure. This type of message can be omitted from diagrams because it is assumed that every call has a return.

Using Messages

Messages can appear in a sequence diagram to represent the communications exchanged between classifier roles during dynamic interactions.

Note Both messages and stimuli are supported. Stimuli are added to collaboration instances, and messages are added to collaborations. For details about stimuli, see [Stimuli on page 652](#).

The messages in a model are usually contained in collaborations and usually appear in sequence diagrams.

Naming Conventions

Messages can be identified by a name or operation signature.

Type	Example	Description
Name	// Get the Password	A name identifies only the name of the message. Simple names are often used in diagrams developed during analysis because the messages are identified by their responsibilities and not operations. One convention uses double slashes (//) to indicate that the stimulus name is not associated with an operation.
Signature	getPassword(String)	When an operation is assigned to a message, you can display the operation signature to identify the name of the operation and its parameters. Signatures are often used in diagrams developed during design because they provide the detail that developers need when they code the design.

Related Topics

[Classifier Roles on page 644](#) | [Lifelines on page 647](#) | [UML Sequence Diagrams on page 397](#)

Notes

Notes appear as shown below and are centered on, and attached to, the element to which they apply:



sequence diagram notes can be associated to messages and instances.

The note is linked to its source file. In the UML/SD Viewer, click a note to open the Text Editor at the corresponding line in the source code.

Notes can be automatically added to the diagram by Component Testing and Runtime Analysis tools. For example, memory profiling adds notes indicating where memory errors and warnings were detected. Component Testing for C and Ada adds notes when a test fails.

You can manually add notes to your C and C++ source code by clicking **Add Note**  in the text editor. This inserts the `_ATT_USER_NOTE` instrumentation pragma into your source code.

Related Topics

| [Instrumentation Pragmas on page 1046](#)

Objects

An object is a model element that represents an instance of a class. While a class represents an abstraction of a concept or thing, an object represents an actual entity. An object has a well-defined boundary and is meaningful in the application. Objects have three characteristics: state, behavior, and identity. State is a condition in which the object may exist, and it usually changes over time. The state is implemented with a set of attributes. Behavior determines how an object responds to requests from other objects. Behavior is implemented by a set of operations. Identity

makes every object unique. The unique identity lets you differentiate between multiple instances of a class if each has the same state.

The behaviors of objects can be modeled in sequence and activity diagrams. In sequence diagrams, you can display how instances of different classes interact with each other to accomplish a task. In activity diagrams, you can show how one or more instances of an object changes states during an activity. For example, an e-commerce application may include a "Cart" class. An instance of this class that is created for a customer visit, such as "cart100:Cart." In a sequence diagram, you can illustrate the stimuli, such as "addItem()," that the "cart100:Cart" object exchanges with other objects. In an activity diagram, you can illustrate the states of the "cart100:Cart" object, such as empty or full, during an activity such as a user browsing the online catalog.

Shape

In sequence and activity diagrams, an object appears as a rectangle with its name underlined. In sequence diagrams, a lifeline (a dashed, vertical line) is attached to the bottom of an object to represent the existence of the object over a period of time. For details about lifelines, see [Lifelines on page 647](#).

Object Object with Life-line

Types of Objects

The following table identifies three types of objects.

Types of Objects	Description
Active	Owns a thread of control and may initiate control activity. Processes and tasks are kinds of active objects.
Passive	Holds data, but does not initiate control.
Multiobject	Is a collections of object or multiple instances of the same class. It is commonly used to show that a set of objects interacts with a single stimulus.

Using Objects

Objects can appear in a sequence diagram to represent concrete and prototypical instances. A concrete instance represents an actual person or thing in the real world. For example, a concrete instances of a "Customer" class would represent an actual customer. A prototypical instance represents an example person or thing. For example, a prototypical instance of a "Customer" class would contain the data that a typical customer would provide.

Naming Conventions

Each object must have a unique name. A full object name includes an object name, role name, and class name. You may use any combination of these three parts of the object name. The following table identifies the variations of object names.

Syntax	Example	Description
<u>ob-</u> <u>ject/role:clas</u>	<u>cart100/</u> <u>storage:cart</u>	Named instance (cart100) of the cart class that is playing the storage role during an interaction.
<u>objec-</u> <u>t:class</u>	<u>cart100:cart</u>	Named instance (cart100) of the cart class.
<u>/role:class</u>	<u>/stor-</u> <u>age:cart</u>	Anonymous instance of the cart class playing the storage role in an interaction.
<u>ob-</u> <u>ject/role</u>	<u>cart/stor-</u> <u>age</u>	An object named cart playing the storage role. This object is either an object that is hiding the name of the class or an instance that is not associated with a class.
<u>object</u>	<u>cart100</u>	An object named cart100. This object is either an instance that is hiding the name of the class or an instance that is not associated with a class.
<u>/role</u>	<u>/storage</u>	An anonymous instance playing the storage role. This object is either an instance that is hiding the name of the object and class or an instance that is not associated with an object or class.
<u>:class</u>	<u>:cart</u>	Anonymous instance of the customer class.

Related Topics

[UML Sequence Diagrams on page 397](#) | [Stimuli on page 652](#)

Stimuli

A stimulus is a model element that represents a communication between objects in a sequence diagram and usually indicates that an activity will follow. The types of communications that stimuli model include calls to operations, signals to objects, the creation of objects, and the destruction of objects. The receipt of a stimulus is an instance of an event.

Shape

A stimulus appears as a line with an arrow. The direction of the arrow indicates the direction in which the stimulus is sent. In a sequence diagram, a stimulus usually connects two object lifelines.

Stimulus shapes can be adorned with names and sequence numbers.

Types of Stimuli

Different types of stimuli can be used to model different flows of control.

Type	Shape	Description
Procedure Call or Nested Flow of Control		Models either a call to an operation or a call to a nested flow of control. When calling a nested flow of control, the system waits for the nested flow of control to complete before continuing with the outer flow.
Asynchronous Flow of Control		Models an asynchronous stimulus between two objects. The source object sends the stimulus and immediately continues with the next step.
Return from a Procedure Call		Models a return from a call to a procedure. This type of stimulus can be omitted from diagrams because it is assumed that every call has a return.

Naming Conventions

Stimuli can have either names or signatures.

Type	Example	Description
Name	// Get the Password	A name identifies only the name of the stimulus. Simple names are often used in diagrams developed during analysis because the stimuli are identified by their responsibilities and not by their operations. One convention uses double slashes (//) to indicate that the stimulus name is not associated with an operation.
Signature	getPassword(String)	When an operation is assigned to a stimulus, you can display the operation signature to identify the name of the operation and its parameters. Signatures are often used in diagrams developed during design because they provide the detail that developers need when they code the design.

Related Topics

[Lifelines on page 647](#) | [Objects on page 650](#) | [UML Sequence Diagrams on page 397](#)

Synchronizations

Synchronizations are an extension to the UML standard that only apply when using the split trace file feature of Runtime Tracing. They are used to show that all instance lifelines are synchronized at the beginning and end of each split TDF file.

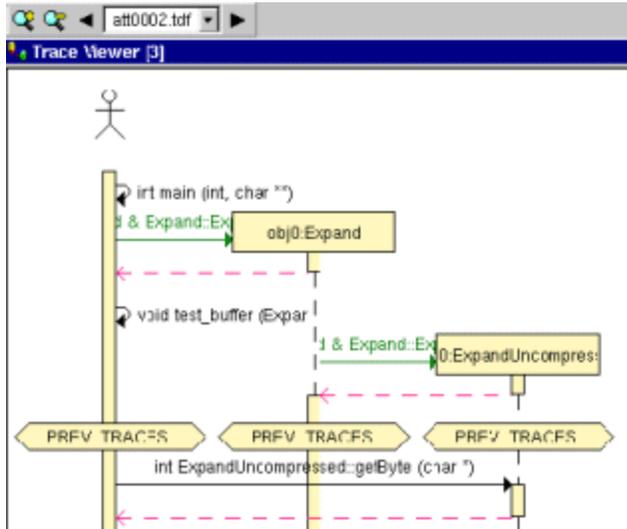
Shape

A synchronization is represented as shown below:

Synchro

The synchronization box displays the name of the synchronization.

The synchronization is linked to its source file. In the UML/SD Viewer, click a synchronization to open the Text Editor at the corresponding line in the source code.



When the Split Trace capability is enabled, the UML/SD Viewer displays the list of TDF files generated in the UML/SD Viewer toolbar.

At the beginning of each diagram, before the Synchronization, the Viewer displays the context of the previous file.

Another synchronization is displayed at the end of each file, to insure that all instance lifelines are together before viewing the next file.

Related Topics

[UML Sequence Diagrams on page 397](#) | Model Elements and Relationships in Sequence Diagrams

Advanced System Testing

System Testing Supervisor

System Testing for C

HCL OneTest™ Embedded System Testing manages the simultaneous execution of Virtual Testers distributed over a network. When using System Testing feature of HCL OneTest™ Embedded, the machine running HCL OneTest™ Embedded runs a Supervisor process, whose job is to:

- Set up target hosts to run the test
- Launch the Virtual Testers, the system under test and any other tools.
- Synchronize Virtual Testers during execution
- Retrieve the execution traces after test execution

The System Testing Supervisor uses a [deployment script on page 594](#), generated by the [Virtual Tester Configuration on page 591](#) and [Virtual Tester Deployment on page 592](#) dialog boxes, to control System Testing Agents installed on each distributed target host. Agents can launch either applications or Virtual Testers.

While the agent-spawned processes are running, their standard and error outputs are redirected to the supervisor.

Note You must [install and configure the agents on page 587](#) on the target machines before execution.

Related Topics

[About Virtual Testers on page 586](#) | [Installing System Testing Agents on page 587](#) | [Editing the Deployment Script on page 594](#)

Circular Trace Buffer

System Testing for C

The *circular trace buffer* memorizes System Testing for C traces and flushes them to the **.rio** output file when the Virtual Tester ends or at a specified point in the **.pts** test script.

To activate the circular trace buffer option or to set the size of the buffer, see [Test Script Compiler Settings on page 1003](#).

How the Circular Buffer Works

During execution of the test node, System Testing accumulates traces in the buffer. When the buffer fills up, new traces replace old ones, as shown in the following diagram, without flushing to file.

Contents of the Buffer

By default, the buffer stores all traces.

Use the **TRACE_OFF** instruction in your **.pts** System Testing for C test script to trace only scenario begins and ends, environment blocks, procedure blocks, PRINT instructions, and failed instructions.

Use the **TRACE_ON** instruction to resume default behavior.

See the Reference section for detailed information on **.pts** test script instructions.

Flushing the Buffer on the Disk

By default, the buffer is flushed to a file when the Virtual Tester ends.

You may flush the buffer at any point in the **.pts** test script by using the **FLUSH_TRACE** instruction.

You cannot call the **FLUSH_TRACE** instruction, either directly or indirectly, from a **CALLBACK** or **PROCSEND** block.

See the Reference section for detailed information on **.pts** test script instructions.

Note The **TRACE_ON**, **TRACE_OFF** and **FLUSH_TRACE** instructions only apply when the Circular Trace Buffer option is selected.

Related Topics

[Test Script Compiler Settings on page 1003](#) | [System Testing for C Settings on page 1003](#)

On-the-Fly Tracing

System Testing for C

The System Testing for C on-the-fly tracing capability allows you to monitor the Virtual Testers during the test execution in a UML sequence diagram. Information provided by dynamic tracking includes:

- Beginning and end of scenarios
- Rendezvous
- Sent and received messages
- Inter-tester messages (only received messages)
- Beginning and end of termination, initialization and exception blocks
- End of Testers

On-the-fly tracing output is displayed in the [UML/SD Viewer on page 402](#) in real-time. You can click any item in the sequence diagram to instantly highlight the corresponding test script line in the Text Editor window.

To activate System Testing dynamic tracking:

1. select **On-the-fly tracing** in the System Testing Advanced Settings for the System Testing test node
2. ensure that the **Allow remote connections** option is selected in the General Preferences.

Related Topics

[Understanding System Testing UML Sequence Diagrams on page 640](#)

Using the graphical user interface

The graphical user interface (GUI) of HCL OneTest™ Embedded provides an integrated environment designed to act as a single, unified work space for all automated testing and runtime analysis activities.

This section describes the features and capabilities included within the GUI.

GUI Philosophy

In addition to acting as an interface with your usual development tools, the GUI provides navigation facilities, allowing natural hypertext linkage between source code, test, analysis reports, UML sequence diagrams. For example:

- You can click any element of a test report to highlight the corresponding test script line in the embedded text editor.
- You can click any element of a runtime analysis report to highlight and edit the corresponding item in your application source code
- You can click a file name in the output window to open the file in the Text Editor

In addition, the GUI provides easy-to-use Activity Wizards to guide you through the creation of your project components.

To learn about	See
Starting a new activity	Activity Wizards on page 663
Using the Project Explorer to create, develop and execute your project nodes	Setting Up a Project on page 674
Understanding Configurations and Configuration Settings	About Configuration Settings on page 658
Identifying and using various the components of the GUI	Discovering the GUI on page 1019
Launching a GUI node from the command line	Running a Node from the Command Line on page 963
Viewing and editing a source file or test script	About the Text Editor on page 693
Controlling source code versions	Working with Configuration Management on page 32
Viewing a report	Using the Report Viewer on page 705
Customizing the GUI	Editing Preferences on page 1009
Adding external tools to the GUI	About the Tools Menu on page 712

Related Topics

[Command Line Interface on page 962](#) |

GUI components and tools

The HCL OneTest™ Embedded GUI provides a comprehensive set of tools and components that make it an efficient and customizable development environment.

- The [text editor on page 693](#) is a full-featured editor for source code
- The [Tools on page 712](#) menu is a convenient way of integrating any command-line tool into the GUI
- The [test process monitor on page 709](#) provides ongoing activity statistics and metrics
- The [report viewer on page 705](#) displays runtime analysis reports
- The [UML/SD viewer on page 402](#) displays UML sequence diagrams provided by Runtime Tracing feature.

Related Topics

[Using the Graphical User Interface on page 657](#) | [Activity Wizards on page 663](#) | [Discovering the GUI on page 1019](#)

Configurations and settings

Configurations and Settings

Two major concepts of HCL OneTest™ Embedded are Configurations and Configuration Settings:

- A Configuration is an instance of a Target Deployment Port (TDP) as used in your project.
- Configuration Settings are the particular properties assigned to each node in your project for a given Configuration.

A Configuration is not the actual Target Deployment Port. Configurations are derived from the Target Deployment Port that you select when the project is created, and contain a series of Settings for each individual node of your project.

This provides extreme flexibility when you are using multiple platforms or development environments. For example:

- You can create a Configuration for each programming language or compiler involved in your project.
- If you are developing for an embedded platform, you can have one Configuration for native development on your Unix or Windows development platform and another Configuration for running and testing the same code on the target platform.

- You can set up several Configurations based on the same TDP, but with different libraries or compilers.
- If you are using multiple programming languages in your project, you can even override the TDP on one or several nodes of a project.

The Configuration Settings allow you to customize test and runtime analysis configuration parameters for each node or group of your project, as well as for each Configuration. You reach the **Configuration Settings** for each node by right-clicking any node in the Project Explorer window and selecting **Settings**.

The left-hand section of the **Configuration Settings** window allows you to select the settings families related to the node, as well as the Configuration itself, to which changes will be made. The right-hand pane lists the individual setting properties.

The right-hand section contains the various settings available for the selected node.

Propagation Behavior of Configuration Settings

The Project Explorer displays a hierarchical view of the nodes that constitute your project.

Settings for each node are inherited by child nodes from parent nodes. For instance, Settings of a project node will be cascaded down to all nodes in that project.

Child settings can be set to *override* parent settings. In this case, the overridden settings will, in turn, be cascaded down to lower nodes in the hierarchy. Overridden settings are displayed in bold.

Settings are changed only for a particular Configuration. If you want your changes to a node to be made throughout all Configurations, be sure to select **All Configurations** in the Configuration box.

To change the settings for a node:

1. In the **Project Explorer**, click the **Settings** button.
2. Use the **Configuration** box to change the Configuration for which the changes will be made.
3. In the left pane, select the settings family that you want to edit.
4. In the right pane, select and change the setting properties that you want to override.
5. When you have finished, click **OK** to validate the changes.



Note: The **Enter** and **Esc** keys do not work in the **Configuration Settings** window. Use the **OK**, **Apply**, and **Cancel** buttons.

By default, the settings of each node are inherited from those of the parent node. When you override the settings of a parent node, changes are propagated to all child nodes within the same Configuration. Overridden fields are displayed in bold.

Configuration Settings Structure

The Configuration Settings provides access to the following settings families:

- General
- Build
- Runtime Analysis
- Component Testing

The actual settings available for each node depend on the type of node and the language of the selected Configuration.

General Settings

To learn about	See
Configuring the compiler and linker options	Build Settings on page 983
General project settings	General Settings on page 987
Adding a user-specified command line to the project	External Command Settings on page 1007
Controlling System Testing Probes	Probe Control Settings on page 1006

Runtime Analysis

The Runtime Analysis setting family covers Configuration Settings for Memory Profiling, Performance Profiling, Code Coverage and Runtime Tracing.

To learn about	See
Setting up instrumentation and file storage locations	General Runtime Analysis Settings on page 989
Configuring Memory Profiling error and warning detection	Memory Profiling Settings on page 995
Specifying a trace file name for Performance Profiling	Performance Profiling settings on page 997
Setting coverage levels and instrumentation options for Code Coverage	Code Coverage Settings on page 992
Configuring sequence diagram output	Runtime Tracing Control Settings on page 997

Automated Testing Settings

This setting family covers Configuration Settings for Component Testing and System Testing features.

To learn about	See
Setting up C and Ada test execution	Component Testing Settings for C and Ada on page 999
Setting up C++ test execution	Component Testing for C++ Settings on page 1001
Setting up system test execution	System Testing Settings on page 1003

Related Topics

[Modifying Configurations on page 662](#) | [Selecting Configurations on page 210](#) | [Project Explorer on page 1021](#)

Switching test configurations

Although a project can use multiple configurations, as well as multiple TDPs, there must always be at least one active configuration. You can switch from one configuration to another at any time, except during build activity.

About this task

The active configuration affects compiler and deployment options for each resource in the project.



Note: You can also run a test harness with two different test configurations by creating a test suite. See [Creating test suites on page 207](#).

To change the active test configuration:

1. In the project explorer, right-click the project and click **Properties**.
2. Expand **C/C++ Build**, select **Settings**, and click **Manage Configurations**.

Result

The **Manage Configurations** window for the project opens.

3. Select the configuration that you want to use to build and run the test and click **Set Active**.
4. Click **OK** to close the **Manage Configurations** window.

Related information

[Creating test configurations on page 209](#)

Modifying Configurations

Configurations are based on the Target Deployment Ports (TDP) that are specified when you create a new project. In fact, a Configuration contains basic Configuration Settings for a given TDP applied to a project, plus any node-specific overridden settings.

Remember that although a project can use multiple Configurations, as well as multiple TDPs, there must always be at least one active Configuration.

When you create a new configuration, the settings are initialized using the default settings of the TDP.

Configuration Settings are a main characteristic of the project and can be individually customized for any single node in the Project Explorer.

To create a new Configuration for a Project:

1. From the **Project** menu, select **Configurations**.
2. In the **Configurations** dialog box, click the **New...** button.
3. Enter a **Name** for the Configuration.
4. Select the **Target Deployment Port** to be used to create the Configuration.
5. Enter the **Hostname, Address** and **Port** of the machine on which the Target Deployment Port is to be compiled.
6. Click **OK**.
7. Click **Close**.

To remove a Configuration from a Project:

If you choose to remove a Configuration, all custom settings for that Configuration will be lost.

1. From the **Project** menu, select **Configurations**.
2. In the **Configurations** dialog box, select the Configuration to be removed.
3. Click the **Remove** button.
4. Click **Yes** to confirm the removal of the Configuration

To copy an existing Configuration:

This can be useful if you want several Configurations, with different custom settings, based on a unique Target Deployment Port.

1. From the **Project** menu, select **Configurations**.
2. In the **Configurations** dialog box, select an existing Configuration.
3. Click the **Copy To...** button
4. Enter a **Name** for the new Configuration.
5. Click **OK**.

Related Topics

[Opening the Target Deployment Port Editor on page 24](#) | [About Configuration Settings on page 658](#) | [Selecting Configurations on page 210](#) | [Target deployment port overview on page 12](#)

Creating tests and applications

Activity wizards overview

The Start Page provides with a full set of activity wizards to help you get started with a new project or activity.

To start a new activity wizard:

1. From the **Start Page**, click **New Activities**
2. Select the activity of your choice.

To learn about

Creating a new project

Creating a new application node configured for a Runtime Analysis feature

Creating a new test node for Component Testing

Creating a new test node for System Testing

See

[New Project Wizard on page 664](#)

[Runtime Analysis Wizard on page 664](#)

[Component Testing Wizard on page 666](#)

[System Testing Wizard on page 669](#)

Related Topics

[Discovering the GUI on page 1019](#) | [Start Page on page 1020](#) | [Manually Creating a Test or Application Node on page 680](#)

Creating a new project

When HCL OneTest™ Embedded starts, the Start Page offers to either open an existing project or create a new project. The New Project wizard creates a brand new project.

To create a new project:

1. From the Start Page, select New Project. If you want to use a project template as the basis for your project, select **New Project from Template**.
2. In the **Project Name**, enter a name for the project.
3. In the **Location** box, change the default directory if necessary and click **Next** to continue.
4. Select one or several Target Deployment Ports for the new project.

The Wizard creates a Configuration based on each selected Target Deployment Port. Later, when working with the project, any changes are made to the [Configuration Settings on page 658](#), not to the Target Deployment Port itself.

1. Click the **Set as Active** button to set the current TDP. The active port is the default Configuration to be used in your project.
2. Click **Finish**.

Once your project has been created, the wizard opens the **Activities** page.

Related Topics

[Target deployment port overview on page 12](#) | [Activity wizards on page 663](#) | [Start page on page 1020](#) | [Using project templates on page 688](#)

Creating a runtime analysis application node

The Runtime Analysis Wizard helps you create a new application node in the Project Explorer. Basically, an application node represents the build of your C, C++, or Ada source code, which is very similar to most other integrated development environments (IDE). You can actually use this graphical user interface as your primary IDE.

Once you have created your application node, you simply add the options required to run any of the runtime analysis features:

- Memory Profiling
- Performance Profile
- Code Coverage
- Runtime Tracing

To create an application node with the Runtime Analysis Wizard:

1. Use the **Start Page** or the **File** menu to open or create a project.
2. Ensure that the correct Configuration is selected in the **Configuration** box.
3. On the **Start Page**, select **Activities** and choose the **Runtime Analysis** activity.
4. The **Application Files** page opens. Use the **Add** and **Remove** buttons to build a list of source files and header files (for C and C++) to add to your project.

The **Configuration Settings** button allows you to override the default Configuration Settings.

Use the **Move Up** and **Move Down** buttons to change the order in which files appear in the application node, and subsequently are compiled.

Use the **Remove** button to remove files from the selection.

Click **Next** to continue.

1. Select the C procedures and functions, or C++ classes or Ada units that you want to analyze.

Use the **Select File** and **Deselect File** buttons to specify the files that contain the components that you want to analyze. The **Select All** and **Deselect All** buttons to select or clear all components.

Click **Next** to continue.

Enter a name for the application node.

By default, the new application node inherits Configuration Settings from the current project. If necessary, click **Settings** to access the Configuration Settings dialog box. This allows you to change any particular settings for the new application node as well as its contents.

Click **Next** to continue.

1. In the **Summary** page, check that all the parameters are correct, and click **Finish**.

The wizard creates an application node that includes all of the associated source files.

You can now select your build options to apply any of the runtime analysis tools to the application under analysis.

Related Topics

[Activity Wizards on page 663](#) | [Component Testing Wizard on page 666](#) | [Using Runtime Analysis Features on page 306](#) | [Selecting Build Options on page 699](#)

Creating a component test

The Component Testing Wizard helps you create a new Component Testing test node in your project for C, C++ and Ada.

For each script type, the wizard analyzes the source code under test to extract unit information and will produce a corresponding test script template using the following test script types:

- C Test Script Language
- C++ Test Driver Script Language
- C++ Contract-Check Language
- Ada Test Script Language
- JUnit Test Harness

You use the generated test script template to elaborate your own test cases.

You can later add to this test node any of the [runtime analysis features on page 306](#) included in HCL OneTest™ Embedded.

There are two methods of creating a test node with the Component Testing wizard:

- **From the Start page:** this method allows you to specify a set of files or components to test.
- **From the Asset Browser:** this method rapidly creates a test from a single file or source code component selected in the Asset Browser.

Once the test node has been generated, you can complete your Component Testing test scripts in the Text Editor. Refer to the **HCL OneTest™ Embedded Reference section of the help** for information about the actual language semantics.

To run the Component Testing Wizard from the Start Page:

1. Use the **Start Page** or the **File** menu to open or create a project. Ensure that the correct Configuration is selected in the **Configuration** box. The selected programming language impacts the type of Component Testing test node to be created.
2. On the **Start Page**, select **Activities** and choose the **Component Testing** activity.
3. On the **Application Files** page, use the **Add** and **Remove** buttons to build a list of source files and header files (for C and C++) to add to your project. The **Configuration Settings** button allows you to override the default configuration settings.
4. Select **Compute Static Metrics** to run the analysis of static testability metrics.
5. Click **Next** to continue.



Note: If the static metrics analysis takes too much time, you can clear the **Compute Static Metrics** option. In this case, the calculation and display of static metrics in any further steps are disabled.



Note: With Component Testing for Ada, it is not possible to submit only an Ada procedure file. Instead, you must include the single procedure in a package.

6. On the **Components Under Test** page, select the units or files for the selected source files. In order to help you choose which components you want to test, this page displays the metrics for each file or unit (packages, classes or functions depending on the language).
7. Select **File Selection** to choose files under test or **Unit Selection** to choose the source code units that require testing. The selection mode toggles the static metrics displayed between file metrics or unit metrics.



Note: If the Unit Selection view seems incomplete, cancel the wizard, from the **Project** menu, select **Refresh File Information** and restart the wizard.

8. Click **Metrics Diagram** to select the units under test from a [graph representation on page 672](#).
9. Click **Next** to continue or **Generate** to skip any further configuration and to use default settings.
10. On the **Test Script Generation Settings** page, specify the test node generation options. The **General** settings specify how the wizard creates the test node.
 - **Test Name:** Enter a name for the test node.
 - **Test Mode:** Disables or enables the test boundaries.
 - **Typical Mode:** No test boundaries are specified. This is the default setting.
 - **Expert Mode:** This mode allowing you to manually drive generation of the test harness. This provides more flexibility in sophisticated software architectures.
 - **Node Creation Mode:** Selects how the test node is created:
 - **Single Mode:** In C and Ada, this mode creates one test node for each source file under test. In C++, it creates one test node for all selected source code components.
 - **Multiple Mode:** This creates a single test node for each selected source code unit.

The **Components Under Test** settings specify advanced settings for each component of the test node. These settings depend on the language and Configuration.

11. Click **Next** to continue.
12. Review the **Summary**. This page provides a summary of the selected options and the files that are to be generated by the wizard.
13. Click **Next** to create the test node based on this information.
14. The **Test Generation Result** page displays progression of the test node creation process. Click **Settings** to set the [Configuration Settings on page 658](#). You can always modify the test node Configuration Settings later if necessary, from the Project Explorer.



Note: If you apply new settings after the test generation, the wizard reruns the test generation. This allows you to fine-tune any settings that may cause the test generation to fail.

15. Once a test node has been successfully generated, click **Finish** to quit the Component Testing Wizard and update the project.

To run the Component Testing Wizard from the Asset Browser:

1. Use the **Start Page** or the **File** menu to open or create a project.
2. Ensure that the correct Configuration is selected in the **Configuration** box. The selected programming language impacts the type of Component Testing test node to be created.
3. In the Project Explorer, select the **Asset Browser** tab.
4. Right click an object, package or source file under test. From the pop-up menu, select **Test**.
5. On the **Test Script Generation Settings**, specify the test node generation options. The **General** settings specify how the wizard creates the test node.
 - **Test Name:** Enter a name for the test node.
 - **Test Mode:** Disables or enables the test boundaries.
 - **Typical Mode:** No test boundaries are specified. This is the default setting.
 - **Expert Mode:** This mode allowing you to manually drive generation of the test harness. This provides more flexibility in sophisticated software architectures.
6. The **Components Under Test** settings specify advanced settings for the component of the test node. These settings depend on the language and Configuration.
7. Click **Next** to continue.
8. Review the **Summary**. This page provides a summary of the selected options and the files that are to be generated by the wizard.
9. Click **Next** to create the test node based on this information.
10. The **Test Generation Result** page displays progression of the test node creation process. Click **Settings** to set the [Configuration Settings on page 658](#). You can always modify the test node Configuration Settings later if necessary, from the Project Explorer.



Note: If you apply new settings after the test generation, the wizard reruns the test generation. This allows you to fine-tune any settings that may cause the test generation to fail.

11. Once a test node has been successfully generated, click **Finish** to quit the Component Testing Wizard and update the project.

Related Topics

[Component Testing for C and Ada on page 447](#) | [Component Testing for C++ on page 511](#) |

Creating a system test

The System Testing Wizard helps you create a new System Testing test node in your project.

Basically, a System Testing node contains a **.pts** test script as well as a set of Virtual Testers for message-based testing.



Note: System Testing for C does not support paths or file names which contain spaces. When naming files or directories, make sure that these do not contain any spaces.

To create a System Testing node:

1. Enter the name of the new System Testing test node.
2. On the **Test Script Selection (1/7)** page, select the source files that are used to build your application among the source files that are currently in your workspace.
 - a. Select whether you want to create a new **.pts** test script file, or if you want to reuse an existing test script. In both cases you will need to enter a name for the **.pts** test script.
 - b. Next, use the **Add** and **Remove** buttons to build a list of interface files. The **Interface Files List** must contain **.h** header files that define the message structures used by your application.
 - c. Click **Next** to continue.
3. On the **Include Directories List (2/7)** page, specify the directories that contain include files that can be required by the interface files and the messaging API.
 - a. Use the **Add** and **Remove** buttons to build a list of include directories. These are the directories that contain files that are included by your application's source code. If necessary, you can use the **Up** and **Down** buttons to indicate the order in which they are searched.
 - b. Click **Next** to continue. If you chose to use an existing **.pts** test script, this brings you straight to step 5.
4. If you chose to create a new **.pts** test script, on the **Generate New Test Script (3/7)** page, specify the message type to be used by the test.
 - a. **Message type:** Select the type definition that will be used for messages.
 - b. If you want to use an existing **.hts** adaptation layer file choose **Select an adaptation layer file** and add your **.hts** files to the list.
 - c. If you want to create a new messaging API for the test, select **Create a messaging API** and enter the following information:
 - **Generate with INSTANCE blocks:** Select this option if you want INSTANCE statements to be created in the **.pts** test for a multi-process or multi-threaded test driver.
 - **Base filename:** Specify the name of the generated API files. The wizard generates **.c**, **.h** and **.hts** files based on this filename.
 - **Directory:** Specify the location where the API files will be generated.
 - d. Click **Next** to continue.
5. On the **Generate New Test Script (4/7)** page, change the configuration settings of the test node or click **Next**.
6. On the **Virtual Tester Driver Creation (5/7)** page, you can create a set of virtual testers.

- a. Use the **New** button to create and name a new virtual tester. You can create and duplicate several virtual testers. You can also skip this page and decide to create your virtual testers later on.
 - b. When a virtual tester is selected, in the **General** tab, specify an instance and target deployment port for the virtual tester.
 - **VT Name:** This is the name of the selected virtual tester. This must be a standard C identifier.
 - **Implemented INSTANCE:** Use this box to assign an **INSTANCE** statement, defined in the **.pts** test script, to the selected virtual tester. This information is used to deploy the virtual tester. Select **Default** to manually specify the instance during deployment.
 - **Target:** Select the Target Deployment Port that will be used for the selected Virtual Tester.
 - c. In the **Scenario** tab, select one or several scenarios as defined in the **.pts** test script. During execution, the Virtual Tester plays the selected scenarios.
 - d. In the **Family** tab, select one or several families as defined in the **.pts** test script. During execution, the Virtual Tester plays the selected families.
 - e. If necessary, click the **Configure Settings** button to change the configuration settings for the selected virtual tester.
 - f. The **API source files** list displays the generated messaging API source files. Use the **Add** or **Remove** buttons to modify this list if your messaging API requires more files.
 - g. Click **Next** to continue.
7. On the **Deploy Configuration (6/7)** page, specify how to deploy the virtual testers onto host and target computers. Use the **Add**, **Remove** buttons to modify the list. Each line represents one or several parallel executions of a virtual tester assigned to an instance, target host, and other parameters.
- **Number of Occurrences:** Specifies the number of simultaneous executions of the current line.
 - **Virtual Tester Name:** Specifies one of the previously created virtual testers.
 - **INSTANCE:** Specifies the instances assigned to this virtual testers. If an instance was specifically assigned in the [Virtual Tester Configuration on page 591](#) box, this cannot be changed. Select **<default>** only if no **INSTANCE** is defined in the test script.
 - **Network Node:** This defines the target host on which the current line is to be deployed. You can enter a machine name or an IP address. Leave this field blank if you want to use the IP address specified in the Host Configuration section of the [General Settings on page 987](#).
-  **Note:** If the IP address line in the Host Configuration settings is blank, then the Virtual Tester Deployment Table retrieves the IP address of the local machine when generating the deployment script.
8. Click the **Advanced Options** button to add the following columns to the Virtual Tester Deployment Table, and to add the **Rendezvous...** button.
- **Agent TCP/IP Port:** This specifies the port used by the [System Testing Agents on page 587](#) to communicate with HCL OneTest™ Embedded.
 - By default, System Testing uses port 10000.
 - **Delay:** This allows you to set a delay between the execution of each line of the table.

- **First Occurrence ID:** This specifies the unique occurrence ID identifier for the first Virtual Tester executed on this line. The occurrence ID is automatically incremented for each number of instances of the current line. See [Communication Between Virtual Testers on page 621](#) for more information.
 - **RIO filename:** This specifies the name of the .rio file containing the Virtual Tester output, for use in [multi-threaded or RTOS environments on page 596](#).
 - If necessary, click the **Rendezvous Configuration** button to set up any rendezvous members.
 - Click **Next** to continue.
9. Review the options in the **Test Generation Summary (7/7)** page and use the **Back** button if necessary to make any changes.
- **Test Script File:** indicates the name of the .pts test script.
 - **Interface Files:** lists the interface files defining the communication routines of your application.
 - **Included Directories:** lists the directories containing files included by your application.
 - **Virtual Testers:** lists the virtual testers that are to be deployed by the test.
10. Click the **Finish** button to launch the generation of the System Testing node with the corresponding virtual testers.

The wizard creates a test node with the associated test scripts. The test node appears in the Project Explorer.

If you chose to create a new .pts test script, you can now complete the generated System Testing test script in the Text Editor and then [configure on page 591](#) and [deploy on page 592](#) your virtual testers.

Refer to the [System Testing language reference on page 893](#) for information about the System Testing script language.

Related Topics

[Activity Wizards on page 663](#) | [Configuring Virtual Testers on page 591](#) | [Deploying Virtual Testers on page 592](#) | [Setting Up Rendezvous Members on page 595](#) | [INSTANCE on page 858](#) | [SCENARIO on page 876](#) | [FAMILY on page 852](#) | [System Testing settings on page 1003](#) | [System Testing supervisor on page 654](#) |

Metrics Diagram Options

The Metrics Diagram displays a simple two-axis plot based on the static metrics calculated by the wizard. The metrics on each axis can be changed in the Metrics Diagram Options dialog box:

- **Axis Selection:** Set the most relevant metrics for your application on the Vertical Axis and Horizontal Axis.
- **Horizontal Axis Scale:** Use this setting to display a gray line at the specified step.
- **Vertical Axis Scale:** Use this setting to display a gray line at the specified step.
- **Display unit name:** select this option to display the names of units next to the diagram checkboxes.

See [About Static Metrics on page 225](#) for information about the available metrics.

To modify the Testability Metrics Graph:

1. Run the Component Testing wizard.
2. From the **Components under Test** page, click **Metrics Diagram**.
3. Click **Options**.
4. Change the display options and click **OK**.

Related Topics

[Metrics Diagram on page 672](#) | [Component Testing Wizard on page 666](#) | [About Static Metrics on page 225](#)

Viewing a static metrics diagram

As part of the Component Testing wizard, HCL OneTest™ Embedded provides static testability metrics to help you pinpoint the critical components of your application. You can use these static metrics to prioritize your test efforts.

The graph displays a simple two-axis plot based on the static metrics calculated by the wizard. The actual metrics on each axis can be changed in the [Metrics Diagram Options on page 671](#) dialog box.

Each unit (function, package or class, depending on the current Configuration language) is represented by a checkbox located at the intersection of the selected testability metrics values.

Move the mouse pointer over a checkbox to display a tooltip with the names of the associated units. To test a unit, select the corresponding checkbox.

HCL OneTest™ Embedded also provides a [Static Metrics Viewer on page 226](#), which is independent from the Component Testing wizard and can be accessed at any time.

To access the wizard Metrics Diagram:

1. From the Start Page, run the Component Testing wizard.
2. From the **Components under Test** page, click **Metrics Diagram**.

To select a unit for test:

1. If necessary, click **Options** to set the two most relevant metrics for your application. This displays each unit at the intersection point of the two values.
2. Move the mouse pointer over a checkbox to display a tooltip with the name of the unit.
3. Select the most relevant units to test. Units under test are displayed in the list box.
4. Click **OK** to validate the selection.

Related Topics

[Component Testing Wizard on page 666](#) | [About Static Metrics on page 225](#) | [Metrics Diagram Options on page 671](#)

Specifying advanced component test options

The **Advanced Options** dialog box allows you to specify a series of advanced test generation parameters in the Component Testing wizard. In most cases, you can leave the default values.

The actual options available in this dialog box depend on the programming language of the current Configuration:

- C or Ada
- C++

Component Testing for C and Ada

The following advanced options are available in the Component Testing wizard with a C or Ada Target Deployment Port:

- **Tested file:** name of the source file under test
- **Test script and path:** location and name of the generated test script template
- **Test static/private data or functions:** specifies whether the file under test is included in a `#include` statement.
- **Additional options:** allows you to add specific command line options for the C or Ada Source Code Parser. See the **Command Line Reference** section in the help for further information.

Component Testing for C++

The following advanced options are available in the Component Testing wizard for C++:

- **Tested file:** name of the source file under test.
- **Test driver script:** specifies whether an `.otd` test driver script is to be generated.
- **Contract-Check script:** specifies whether an `.otc` Contract Check driver script is to be generated.
- **Test script and path:** location and name of the generated `.otd` test driver script template.
- **Directory for Contract-Check script files:** sets the location where the `.otc` Contract Check script files are created.
- **Additional options:** allows you to add specific command line options for the C++ Source Code Parser. See the **Line Command** section in the **Reference section of the help** for further information.
- **Ignore #line directive:** by default, the Test Generation Wizard analyzes `#line` directives, although use of preprocessed files with Component Testing for C++ is not recommended. Select this option when `#line` directives should be ignored.

- **Test union and struct as class:** tells the Test Generation Wizard to consider classes defined with the *struct* or *union* keyword as candidate classes. This option is only available if the auto-select candidate classes was selected on the File and Classes under Test page.
- **Test each template instance:** tells the wizard to generate C++ Test Script Language code for each instance of a template class. If this option is selected, there must be template class instances in the source file under test. By default, the Test Generation Wizard generates a single portion of C++ Test Script Language code for a template class.
- **Overwrite previous test scripts:** tells the wizard to overwrite any previously generated **.otc** or **.otd** test scripts. If this option is not selected, no changes will be made to any existing **.otc** or **.otd** test scripts.
- **Path for included header files:** specifies how include file names must be analyzed.
 - Select **Relative** for relative filenames.
 - Select **Absolute** for absolute filenames.
 - Select **Copy** to use include the path as specified.
- **Included files:** use the **Add** and **Remove** buttons to add and remove files in the list. The include file list used by the Component Testing wizard is kept in the generated test node settings.

Related Topics

[Component Testing Wizard on page 666](#) | [About Configuration Settings on page 658](#)

Working with Projects

The project is your main work area in HCL OneTest™ Embedded , as displayed in the **Project Explorer** window.

A project is a tree representation that contains nodes. Projects can contain one or more sub-projects which are actually links to other projects.

Note Previous versions of the product used Workspaces instead of sub-projects. Workspaces are automatically converted to sub-projects when loaded into the current version of the product.

Within the project tree, each node has its own individual Configuration Settings –inherited from its parent node– and can be individually executed.

To learn about	See
Creating a new project	New Project Wizard on page 664
Understanding how projects work	Understanding Projects on page 675
Creating a group folder inside a project	Creating a Group on page 682

Adding a new application node to a project without using an activity wizard	Manually Creating an Application Node on page 680
Adding files from an existing makefile	Importing a Makefile on page 690
Adding a command line to a project	Creating an External Command Node on page 681
Adding source files to an existing node	Adding Files to the Project on page 688
Excluding a node from execution	Excluding a Node from a Build on page 700
Removing a node from a project	Deleting a Node on page 682
Renaming an existing node in the project	Renaming a Node on page 687
Changing build options and Runtime Analysis tools	Selecting Build Options on page 699
Executing the project or an individual node	Building and Running a Node on page 698
Using the Debug setting of your compiler	Debug Mode on page 702
Removing all previously generated files	Cleaning Up Generated Files on page 702

Related Topics

[Project Explorer on page 1021](#)

[About Configuration Settings on page 658](#)

[Activity Wizards on page 663](#)

Project overview

A project is a tree representation that contains nodes.

Within the project tree, each node has its own individual Configuration Settings —inherited from its parent node— and can be individually executed.

Project Nodes

The project is your main work area in HCL OneTest™ Embedded.

A project is materialized as a directory in your file system, which contains everything you need to test and analyze your code:

- Source code
- Test scripts
- Analysis and test result files

In the HCL OneTest™ Embedded graphical user interface, a project is organized as follows:

- **Project node:**  this node contains any of the following nodes:
 - **Group node:**  Allows you to group together several application or test nodes.
 - **Application node:**  contains a complete application.
 - **Results node:** contains your runtime analysis result files, once the application has been executed. Use this node to control the result files in Rational ClearCase or any other configuration management system.
 - **Source node:** these are the actual source files under test. They can be instrumented  or not instrumented .
 - **Test node:** represents a complete test harness, for Component Testing for C and Ada , C++ , or System Testing . A test node containing:
 - **Results node:** contains your test result files, once the test has been executed. Use this node to control the result files in Rational ClearCase or any other configuration management system.
 - **Test Script node:**  contains the test driver script for the current test.
 - **Source node:** these are the actual source files under test. They can be instrumented  or not instrumented .
 - **External Command node:**  this node allows you to execute a command line anywhere in the project. Use this to launch applications or to communicate with the application under test.

Application and test nodes can be moved around the project to change the order in which they are executed. The order of files inside a Test node cannot be changed; for example the test script must be executed before the source under test.

Projects and sub-projects

Projects can contain one or more sub-projects which are actually links to other project directories. The behaviour of a sub-project is the same as a project. In fact, a sub-project can be opened separately as a stand-alone project.

Note Previous versions of the product used Workspaces instead of sub-projects. Workspaces are automatically converted to sub-projects when loaded into the current version of the product.

Here are several examples of the use of super-projects and sub-projects:

- In a team, users work on their own projects to develop and test portions of a larger development project. For testing the whole project, a single master project can be created to integrate, build, and test multiple sub-projects in one go.
- A single project may contain different sub-projects for different target platforms.

Results Node

By default, each application and test node contains a Results node.

Once the test or runtime analysis results have been generated, this node contains the report files. Right-click the result node or the report files to bring up the **Source Control** pop-up menu.

If you are not controlling result files in a configuration management system, you can hide the Results node by setting the appropriate option in the Project Preferences.

Related Topics

[Project Preferences on page 1015](#) | [Working with Projects on page 674](#) | [Working with Configuration Management on page 32](#) | [Creating a super-project with sub-projects on page 691](#)

Example projects

HCL OneTest™ Embedded is provided with a range of example projects aimed at demonstrating most of the features of the product. You may use them to familiarize yourself with those features. Do not hesitate to review and manipulate the source files and scripts provided in these examples.

Most examples are designed to run directly with a default Configuration.

To access open an example project:

1. From the Start page, click **Examples** on the left side of the page. This opens the Examples page.
2. Click any of the example projects to open them in the product.

Example	Language	Description
BaseStation C	C/C++	Main sample project covering most test and runtime analysis features. This sample is used for the C and C++ tutorial.
Broadcast Server	C	A sample that demonstrates the use of System Testing for C with a single VT as a process, a double VT as a double process and a double VT as one multithreaded process.
Chained List	C++	This sample shows how to test chained lists with Component Test for C++.

ABWL Check Fre- quency	C	This sample demonstrates the Memory Profiling manual check feature for checking ABWL and FMWL errors. See Checking for ABWL and FMWL errors on page 378 for more information.
Data Pool	C	This project shows how to incorporate data tables in .csv format in your tests. See Importing a Data Table (.csv File) on page 692 .
Enum Ada	Ada	A project that demonstrates Component Testing for Ada for testing <i>enum</i> types.
Generic Ada	Ada	A sample demonstrating how to test generic units in Ada. See Testing Generic Compilation Units on page 571 .
Histogram Ada	Ada	This project demonstrates array handling and overriding ENVIRONMENT statements with Component Test for Ada.
Philosopher	C	A sample C application for Runtime Analysis in a multithreaded environment (Windows only)
Dinner Party	C++	A Component Testing for C++ sample application in a multithreaded environment with class inheritance (Windows only).
Shape	C++	A simple class inheritance example for Component Testing for C++.
Shape Ada	Ada	This example demonstrates Component Testing on object-oriented Ada.
Shared Li- brary	C++	This sample demonstrates how to use shared library files in your applications. See Using shared libraries on page 686 for more information.
Stack	C	A simple example project on a stack application for System Testing for C.
Stub Ada	Ada	A simple example project demonstrating the usage of stubs with Component Testing for Ada. See Stub simulation Ada on page 556 .
Stub C	C	A simple example project demonstrating the usage of stubs with Component Testing for C. See Stub Simulation in C on page 475 .
Task Ada	Ada	A simple example project demonstrating how to test Ada tasks with Component Testing for Ada.
Add	C	An extremely short example to help you to develop new TDPs.
Template Cpp	C++	An example of Component Testing for C++ on C++ templates
Testing Ada	Ada	This sample demonstrates how to test various variable types in Ada.
Testing C	C	This sample demonstrates how to test various variable types in C.
Test Suite Ada	Ada	Use this sample to validate any changes made to an Ada Target Deployment Port.
Test Suite C	C	Use this sample to validate any changes made to a C Target Deployment Port.

Troubleshooting a project

When executing a node for the first time in HCL OneTest™ Embedded, it is not uncommon to experience compilation issues. Most common problems are due to some common oversights pertaining to library or include paths or Target Deployment Port settings.

To help debug such problems during execution, you can prompt the GUI to report more detailed information in the Output window by selecting the verbose output option.

To set the verbose output option from the GUI:

1. From the **Edit** menu, select **Preferences**.
2. Select the **Project** preferences.
3. Select **Verbose output** and click **OK**.

To set the verbose output option from the command line:

1. Set the environment variable **\$ATTOLSTUDIO_VERBOSE**.
2. Rerun the command line tools.

Related Topics

[Project Preferences on page 1015](#) | [Configurations and Settings on page 658](#) | [S on page 699](#) electing [Build Options on page 699](#) | [Troubleshooting Command Line Usage on page 976](#)

Refreshing the asset browser

The **Asset Browser** view of the **Project Explorer** window analyzes source files and extracts information about source code components (classes, methods, functions, etc...) as well as any dependency files. This capability, known as *file tagging*, allows you to navigate through your source files more easily and provides direct access to the source code components through the [Text Editor on page 693](#).

When the automatic file tagging option is selected, HCL OneTest™ Embedded refreshes the file information whenever a change is detected. However, you can use the Refresh Information command to update a single file or the entire project.

You can change the way files are tagged by changing the **Source File Information** Configuration Settings for the current project.

Note When many files are involved in the tagging process, the Refresh Information command may take several minutes.

To manually refresh a single file in the Asset Browser:

1. In the **Project Explorer**, select the **Asset Browser** tab.
2. Right-click the file or object that you want to refresh.
3. From the pop-up menu, select **Refresh Information**.

To refresh all project files:

1. From the **Project** menu, select **Refresh File Information**.

To activate or deactivate the automatic refresh:

With the automatic file tagging option, files are automatically refreshed whenever a file is loaded into the workspace or selected in the Project Explorer.

1. From the **Edit** menu, select **Preferences**.
2. Select the **Project** preferences node.
3. Select or clear the **Activate file tagging** option, and then click **OK**.

To edit the Source File Information settings for the project:

1. In the **Project Explorer**, click the **Settings**  button.
2. Select the project node in the **Project Explorer** pane.
3. In the Configuration Settings list, expand **General**.
4. Select **Source File Information**.
5. When you have finished, click **OK** to validate the changes.

Related Topics

[About the Text Editor on page 693](#)

[Project Preferences on page 1015](#)

[General Settings on page 987](#)

Manually creating an application or test node

Application nodes and test nodes are the main building blocks of your workspace. An application node typically contains the source files required to build the application.

Test nodes contain the source under test, test scripts and any dependency files required for the test.

The preferred method to create an application or test node is to use the [Activity Wizard on page 663](#), which guides you through the entire creation process.

However, if you are re-using existing components, you might want to create an empty application node and manually add its components to the workspace.

The GUI allows you to freely create and modify test or application nodes. However, you must follow the logical rules regarding the order of execution of the items contained in the node. When using Component Testing for C++, **.otc** scripts must be placed before **.otd** scripts.

To manually add components to the application node.

1. In the Project Explorer, right-click a Project node or a Group node.
2. From the pop-up menu, select **Add Child** and **Files**.
3. In the File Selector, select the files that you want to add to the application node.
4. Click **Ok**.

Note Before running an application node created with this method, please ensure that all necessary files are present in the application node and that all [Configuration Settings on page 658](#) have been correctly set.

Related Topics

[Deleting a Node on page 682](#)

[Creating an External Command Node on page 681](#)

[Creating a Group on page 682](#)

Creating an external command node

External Command nodes are custom nodes that allow you to add a user-defined command line at any point in the project tree.

This is particularly useful when you need to run a custom command line during test execution.

To add an external command to a workspace:

1. In the **Project Explorer**, right-click the node inside which you want to create the test, application or external command node
2. From the pop-up menu, select **Add Child** and **External Command**.
3. To move the node up or down in the workspace, right-click the external command node and select **Move Up**  or **Move Down** .

To specify a command line for the external node:

Once the External Command node has been created, you can specify the command line that it will be carrying in the Configuration Settings dialog box:

1. In the **Project Explorer**, click the **Settings**  button.
2. Click the **External Command** node.
3. Enter the command in the **Command** box.
4. Click **OK**.

Note External Commands support the GUI Macro Language so that you can send variables from the GUI environment to your command line. See the GUI Macro Language section in the **Reference Manual** for further details.

Related Topics

[About Configuration Settings on page 658](#) | [External Command Settings on page 1007](#) | [GUI macro variables on page 1029](#)

Creating a group

The Group node is designed to contain several application nodes. This allows you to organize workspace by grouping applications together.

This also allows you to build and run a specific group of application nodes without running the entire workspace.

To create a group node:

1. In the **Project Explorer**, right-click the workspace node or right-click any application node.
2. From the pop-up menu, select **Add Child** and **Group**.
3. In the **New Group** box, enter the name of the group.
4. Click **OK**.

Related Topics

[Building and Running a Node on page 698](#) | [Project Explorer on page 1021](#)

Deleting a node

Removing nodes from a project does not actually delete the files, but merely removes them from the Project Explorer's representation.

To delete a node from the Project Explorer:

1. Select one or several nodes that you want to delete.
2. From the **Edit** menu, select **Delete** or press the **Delete** key.

Related Topics

[Report Explorer on page 1024](#)

Opening a report

Because of the links between the various views of the GUI, there are many ways of opening a test or runtime analysis report in HCL OneTest™ Embedded . The most common ones are described here.

Note Some reports require opening several files. For example, when manually opening a UML sequence diagram, you must select the complete set of **.tsf** files as well as the **.tdf** file generated at the same time. A mismatch in **.tsf** and **.tdf** files would result in erroneous tracing of the UML sequence diagram.

To open a report from the Project Explorer:

1. Execute your test with the **Build** command.
2. Right-click the application or test node.
3. From the pop-up menu, select **View Report** and then the appropriate report.

Note Reports cannot be viewed before the application or test has been executed.

To manually open a report made of several files:

1. From the **File** menu, select **Browse Reports**. Use the **Browse Reports** window to create a list of files to be opened in a single report. For example, a **.tdf** dynamic trace file with the corresponding **.tsf** static trace files.
2. Click the **Add**  button. In the **Type** box of the File Selector, select the appropriate file type. For example, select **.tdf**.
3. Locate and select the report files that you want to open. Click **Open**.
4. Click the **Add**  button. In the **Type** box of the File Selector, select the appropriate file type. For example, select **.tsf**.
5. Locate and select the report files that you want to open. Click **Open**.
6. In the **Browse Reports** window, click **Open**.

Report Viewers

The GUI opens the report viewer adapted to the type of report:

- The UML/SD Viewer displays UML sequence diagram reports.
- The Report Viewer displays test reports
- The [Code Coverage Viewer on page 349](#) displays code coverage reports.
- The [Memory Profiling Viewer on page 377](#) and [Performance Profiling Viewer on page 391](#) display Memory Profiling for C++ and Performance Profiling results.

Related Topics

[Understanding Reports on page 706](#) | [Using the Report Viewer on page 705](#) | [Using the Memory Profiling Viewer on page 377](#) | [Using the Performance Profiling Viewer on page 391](#) | [About the UML/SD Viewer on page 402](#) | [About the Code Coverage Viewer on page 349](#)

Creating a source file folder

The Project Explorer Asset Browser provides a convenient way of viewing the source files in your project.

To make this even more convenient, you can create custom folders to accommodate any file types. This makes navigation through your source files even easier.

Note The Asset Browser provides a virtual navigation interface. The actual files do not change location. Use the **Properties Window** to view the actual file locations.

To create a custom folder:

1. In the **Asset Browser**, select the **By File** sort method.
2. Right-click on an existing folder.
3. From the popup menu, select **New Folder...**
4. Enter a name for the new folder and a file filter for the desired file type.

Related Topics

[Discovering the GUI on page 1019](#) | [Project Explorer on page 1021](#) | [Properties Window on page 1023](#)

Using assembler source files

HCL OneTest™ Embedded provides support for using assembler source code in your projects. Due to their nature, you cannot use Component Testing or Runtime Analysis tools directly on assembler files.

Because assembler file extensions are not standard and depend on your development environment, it is necessary to configure HCL OneTest™ Embedded to recognize the file extension used for assembler files. You must specify the assembler file extension:

- In the [Project Preferences on page 1015](#) in order for the GUI to recognize the file type.
- In the [TDP Editor on page 25](#) for the TDP to recognize assembler files.

To specify the file type preferences:

1. From the **Edit** menu, select **Preferences** and select the **Project > Source File Types** page.
2. Click **Add** to create a new line.
3. In the **Extension** column, enter the file extension. For example: ***.asm**.

4. In the **Description** column, enter the description of the file type. For example: **Assembler source files**.
5. Click OK.

To change ASMEXT in the TDP Editor:

1. Open the TDP Editor: from the Tools menu, select Target Deployment Port Editor > Start.
2. In the TDP Editor select **Basic Settings** and the native language of the TDP.
3. Double-click the **ASMEXT** customization point, and add the assembler file extension. For example: **asm**.
4. Save the TDP and quit the TDP Editor.

To add the assembler files to your project.

1. In the Project explorer, right click antest or application node and select **Add Child > Files**.
2. Select the corresponding file type; and locate and select the assembler files that you want to use in your project.
3. Click OK.

Related Topics

[Project Preferences on page 1015](#) | [Using shared libraries on page 686](#) | [Using the TDP Editor on page 25](#)

Unloadable libraries

In some cases, the architecture of an application requires that shared libraries are loaded and unloaded dynamically during the execution in order to optimizing memory usage.

HCL OneTest™ Embedded supports this behavior by allowing you to specify this in the Configuration settings of the project. There are two steps to this:

- Define a shared library as unloadable
- Specify an application as using unloadable libraries

To use unloadable libraries in a project:

1. In the **Project Explorer**, click the **Settings**  button.
2. Select an application or test node in the **Project Explorer** pane.
3. In the **Configuration Settings** list, expand **Build > Build Target Deployment Port**.
4. On **Use unloadable library**, select **Yes**.
5. Select the library node of your unloadable shared library in the **Project Explorer** pane.
6. In the **Configuration Settings** list, expand **Build > Build Target Deployment Port**.
7. On **Build as unloadable library**, select **Yes**.
8. When you have finished, click **OK** to validate the changes.

Related Topics

[Using shared libraries on page 686](#) | [Build Settings on page 983](#)

Using shared libraries

HCL OneTest™ Embedded provides support for using, testing and profiling shared libraries with any C or C++ test or application node.

Shared libraries must be stored inside library nodes within the project in order for them to be accessed by test or application nodes. The library node is a container for the source files of the shared library.

Once the library has been included in the project, you must create link the library to the test or application by creating a reference node in the test or application node.

There are three steps that you must follow in order to use a shared library in your project:

- Create a library node in the project.
- Specify how the library is to be linked (statically or dynamically).
- Create a reference to the library in the test or application node.

To add a shared library to your project:

1. Right-click a group or project node and select **Add Child** and **Library** from the pop-up menu.
2. Enter the name of the Library node
3. Right-click the Library node and select **Add Child** and **Files** from the popup menu.
4. Select the source files of the shared library and click **OK**.

To specify link settings for a library node:

1. Select a library node in the **Project Explorer** pane.
2. In the **Project Explorer**, click the **Settings**  button.
3. Select the Build > Linker page and select the **Generation Format**:
 - Static library (.lib, .a)
 - Dynamic library (.dll, .so)
 - Executable file (.exe)
4. When you have finished, click **OK** to validate the changes.

To link a library node to a test or application node:

1. Right-click the test or application node that will use the shared library and select **Add Child** and **Reference** from the pop-up menu.
2. Select the library that you want to reference and click **OK**.

Example

An example demonstrating how to test and profile shared libraries is provided in the **Shared Library** example project. See [Example projects on page 677](#) for more information.

Related Topics

[Profiling shared libraries on page 307](#) | [Testing shared libraries on page 449](#)

Viewing node properties

You can obtain and change file or node properties by opening the **Properties** window.

To view file properties:

1. Right-click a file in the **Project Explorer**.
2. Select **Properties...** from the pop-up menu.

Related Topics

[Properties Window on page 1023](#)

Renaming a node

Renaming a node in the Project Explorer involves modifying the properties of the node.

To change the name of a node:

1. In the **Project Explorer**, right-click the node that you want to modify.
2. Select **Properties** in the pop-up menu.
3. Change the **Name** of the node.
4. Click **OK**.

Related Topics

[Viewing File Properties on page 687](#) | [Working with Projects on page 674](#)

Adding files to a project

The Project Explorer centralizes all Project files in a unique location. For HCL OneTest™ Embedded to access and analyze source files, they must be accessible from the [Project Explorer on page 1021](#).

Files are automatically added when you use the [Activity Wizard on page 663](#).

To add files to the Project Explorer:

1. In the **Project Explorer**, select the **Object Browser** tab
2. In the **Sort Method** box, select **By Files**.
3. Select **Project > Add to Current Project > New File**.
4. This opens the file selector. In the file **Type** box, select the type of files that are to be added.
5. Locate and select one or several files to be added, and click **Open**.

The selected files will appear under the Source sections of the Project Explorer.

If you have the Automatic source browsing option enabled, your source files will be analyzed, making their components directly accessible in the Project Explorer.

You can also create new files by right-clicking a node and selecting **Add Child > Add New File**.

Related Topics

[Editing Preferences on page 1009](#)

Using project templates

You can save _HCL OneTest™ Embedded projects as **.rtpl** project templates. Project templates allow you to accelerate the creation of a new project by using a template that contains your basic test project environment, including settings, test or application nodes, or common libraries.

To create a project template:

1. In the **Project Browser**, set up a basic project that you will use as a template.
2. Select **File > Save Project As Template**.

To create a new project based on a template:

1. On the **Start Page**, select **Get Started** and **New Project from Template** or click **File > New > New Project from Template**.
2. Locate the **.rtpl** project template and click **Open**.
3. Select **File > Save Project As Template**.

Related Topics

[Working with Projects on page 674](#) | [Creating a new project on page 664](#)

Importing files

Importing files from a Microsoft Visual Studio project

HCL OneTest™ Embedded Studio offers the ability to create a project by importing source files from an existing Microsoft Visual Studio 6.0 or .NET project.

Note The Import feature merely imports a list o

f files as referenced in the Visual Studio project. It does not import everything you need to immediately build a project in HCL OneTest™ Embedded.

The makefile import feature creates a new project, reads the **.dsp** or **.vcproj** project file and adds the source files found in the Visual Studio project to the HCL OneTest™ Embedded project. The project is created with the default Configuration Settings of the current Target Deployment Port (TDP).

Any other information contained in the Visual Studio project, such as compilation options, must be entered manually in the **Configuration Settings** dialog box.

Alternatively, you can import the files as a sub-project of the HCL OneTest™ Embedded current project. In this case, the sub-project inherits the Configuration Settings of the master project.

To import files from a Microsoft Visual Studio project as a new project:

1. Close any open projects.
2. From the **File** menu, select **Import > Import from Visual Studio 6.0 Project** or **Import from Visual Studio .NET Project**.
3. Use the file selector to locate a valid **.dsp** or **.vcproj** project file and click **Open**.
4. Enter a name for the new project and click **OK**.
5. Select the correct Configuration in the Configuration toolbar.
6. In the **Project Explorer**, click **Settings** .
7. Enter any specific compilation options in the **Build** settings and click **OK**.

To import files from a Microsoft Visual Studio project as a sub-project:

1. With a project open, select the project node.
2. Right-click the project node and select **Add Child > Import**.
3. Use the file selector to locate a valid **.dsp** or **.vcproj** project file and click **Open**.
4. In the **Project Explorer**, click **Settings** .
5. Enter any specific compilation options in the **Build** settings and click **OK**.

Related Topics

[Adding Files to the Project on page 688](#) | [Importing Files from a Makefile on page 690](#) | [Manually Creating a Test or Application Node on page 680](#) | [Selecting Configurations on page 210](#) | [Working with Projects on page 674](#)

Importing files from a makefile or a build log

The HCL OneTest™ Embedded GUI offers the ability to create a project by importing source files from an existing makefile.

Note The Import Makefile feature merely imports a list of files as referenced in the makefile or build log. It does not import everything you need to immediately build a project in HCL OneTest™ Embedded.

The makefile import feature creates a new project, reads the makefile or build log and adds the source files to the project. The project is created with the default Configuration Settings of the current Target Deployment Port (TDP).

Any other information contained in the makefile, such as compilation options must be entered manually in the **Configuration Settings** dialog box. The following limitations apply:

- Source files must be referenced in the build line
- The makefile cannot be recursive
- Any external commands such as Unix Shell commands are not imported
- Complex operations with variables cannot be imported

Any environment variables used within the makefile must be valid.

You can also use Import Makefile feature to import any list of files contained in a plain text file.

Alternatively, you can import the project as a sub-project of the HCL OneTest™ Embedded current project. In this case, the sub-project inherits the Configuration Settings of the master project.

To import files from a makefile as a new project:

1. Close any open projects.
2. From the **File** menu, select **Import > Import from Makefile**. Use the file selector to locate a valid makefile and click **Open**.
3. Enter a name for the new project and click **OK**.
4. Select the correct Configuration in the Configuration toolbar.
5. In the **Project Explorer**, click **Settings** .
6. Enter any specific compilation options in the **Build** settings and click **OK**.

To import files from a makefile as a sub-project:

1. With a project open, select the project node.
2. Right-click the project node and select **Add Child > Import**.
3. Use the file selector to locate a valid makefile and click **Open**.
4. In the **Project Explorer**, click **Settings** .
5. Enter any specific compilation options in the **Build** settings and click **OK**.

Related Topics

[Adding Files to the Project on page 688](#) | [Importing Files from a Microsoft Visual Studio Project file on page 689](#) | [Manually Creating a Test or Application Node on page 680](#) | [Selecting Configurations on page 210](#) | [Working with Projects on page 674](#)

Importing sub-projects

Sub-projects are projects that are grouped together within a master project. Projects can contain one or more sub-projects which are actually links to other project directories. The behaviour of a sub-project is the same as a project.

There are two ways of setting up a master project:

- Add the projects manually to a new or existing project. Use this method to import projects one by one from different locations or to add sub-projects to an existing project.
- Imports all the projects contained in a specific directory into a master project. Use this method to automatically import many sub-projects when they are all located in the same directory.

To add an existing sub-project:

1. Create a new project or open an existing project.
2. Select **File > Add Project > Existing Project**. This opens the file selector.
3. Locate and select an **.rtp** project file and click **OK**.

To create a new sub-project:

1. Create a new project or open an existing project.
2. Select **File > Add Project > New Project**. This opens the Add New Project wizard.
3. Enter a name and location for the new project, and click **Finish**. The new sub-project is created with the configuration settings of the super-project.

To create a master project containing all sub-projects from a directory:

1. Close any open projects
2. Select **File > Import > Import multiple HCL OneTest™ Embedded projects**.
3. Enter the name of the new master project and the location of the existing projects and click **OK**. The new project is created in the selected directory and imports all the projects found in all sub-directories of that location. When browsing many directories, the import can take a long time.

Related Topics

[New Project Wizard on page 664](#) | [Understanding projects on page 675](#) | [Working with Projects on page 674](#)

Importing a data table (.csv file)

HCL OneTest™ Embedded Component Testing for C and C++ provide the ability to import **.csv** table files and to turn these into standard **.h** header files. The resulting header file uses the same filename with a **.h** extension. Once included in your **.ptu** or **.otd** test script, this data can be used by the test driver script or the application under test.

Such **.csv** files can be produced by most spreadsheet programs or a text editor.

To import a **.csv** file into a test node:

1. From the **Project Explorer**, right click an existing test node.
2. From the pop-up menu, select **Add File...**
3. Locate and select the **.csv** file and click **OK**.
4. By default, added files are excluded from the build. Click the **Excluded** marker to allow the file to be built. The **.csv** table file must be located before the **.ptu** test script in the test node.
5. Edit the **.ptu** test script to manually add an include statement of the resulting **.h** header file.

Note The **.csv** data table file must be located before the **.ptu** test script in the test node. If not, then you must manually build the **.csv** data table file before building the test node.

CSV File Format

The formatting rules for the **.csv** file are as follow:

- The first line contains the names of the variable arrays separated by the default CSV separator specified in the preferences or the Configuration settings.
- The second line optionally specifies the data type: **string, char, rint, long, float** and **double**, which can be **signed** or **unsigned**. If this information is not specified, then **int** is assumed by default.
- Each following line contains the data for the corresponding array
- When a blank value is encountered, an end of array is assumed. Any further values for that array will be ignored.

When the test node is built, HCL OneTest™ Embedded produces a *<filename>*.h header file, where *<filename>* is based on the name of the input *<filename>*.csv file.

Use the arrays produced by the **.csv** file by including *<filename>*.h into your test script or source code.

The separator options for the **.csv** file are defined in two locations:

- Data tables preferences: These specify the default behavior for HCL OneTest™ Embedded.
- Data tables section in the General Configuration settings: These allow you to override the default settings for a particular project of test node.

Example

This is an example of a valid **table.csv** data table:

```
var_A;var_B;var_C
```

```
int;signed int;float
```

```
12;34;45.2345
```

```
14;2;3.142
```

```
;-5;0
```

This produces the following corresponding **table.h** file:

```
int var_A[]={12,14};
```

```
signed int var_B[]={34,2,-5};
```

```
float var_C[]={45.2345,3.142,0};
```

Related Topics

[General Settings on page 987](#) | [Data table preferences on page 1011](#)

Editing code and test scripts

Editing code and test scripts

The product GUI provides its own text editor for editing and browsing script files and source code.

The Text Editor is a fully-featured text editor with the following capabilities:

- Syntax Coloring
- Find and Replace functions
- Go to line or column

The main advantage of the Text Editor included with HCL OneTest™ Embedded is its tight integration with the rest of the GUI. You can click items within the **Project Explorer**, **Output Window**, or any Test and Runtime Analysis report to immediately highlight and edit the corresponding line of code in the Editor.

To learn about

See

Creating a new text file

[Creating a text file on page 694](#)

Opening an existing text file in the Text Editor

[Opening a text file on page 694](#)

Locating a text string in the Text Editor	Finding text in the text editor on page 695
Replacing a text string with another string	Replacing text in the text editor on page 696
Going to a specific line or column in a text file	Locating a line and column in the text editor on page 697
Adjusting the syntax coloring to the current language	Text editor syntax coloring on page 697
Customizing the Text Editor	Text editor preferences on page 1012

Related Topics

[Using the Graphical User Interface on page 657](#) | [GUI elements on page 1019](#)

Creating a text file

To create a new text file:

1. Click the **New Text File**  toolbar button,
2. From the **Editor** menu, use the **Syntax Color** submenu to select the language.

or

1. From the **File** menu, select **New...** and then open the **Text File** option
2. From the **Editor** menu, use the **Syntax Color** submenu to select the language.

Opening a text file

The Text Editor is tightly integrated with the HCL OneTest™ Embedded GUI. Because of the links between the various views of the GUI, there are many ways of opening a text file. The most common ones are described here.

Using the Open command:

1. From the **File** menu, select **Open...** or click the **Open**  button from the standard toolbar.
2. Use the file selector to select the file type and to locate the file.
3. Select the file you want to open.
4. Click **OK**.

Using the File Explorer:

1. Select a file in the [Project Explorer on page 1021](#). If there are recognized components in the file, a '+' symbol appears next to it.
2. Click the '+' symbol to expand the list of references in the file.
3. Double-click a reference to open the **Text Editor** at the corresponding line.

You can also navigate through the source file by double-clicking other reference points in the Project Explorer.

Using a Test or Report Viewer:

1. With the **Report Viewer** open, locate an element inside the report.
2. Double-click the item to open the **Text Editor** at the corresponding line.

Related Topics

[About the Text Editor on page 693](#) | [Finding Text in the Text Editor on page 695](#) | [Locating a Line and Column in the Text Editor on page 697](#)

Finding text in the text editor

To locate a particular text string within the text editor, use the **Find** command.

Search Options

The **Search** box allows you to select the search mode:

- **All** searches for the first occurrence from the beginning of the file.
- **Selected** searches through selected text only.
- **Forward** and **Backward** specify the direction of the search, starting at the current cursor position.

Match case restricts search criteria to the exact same case.

Match whole word only restricts the search to complete words.

Use regular expression allows you to specify UNIX-like regular expressions as search criteria.

To find a text string in the Text Editor:

1. From the **Edit** menu, select **Find...**
2. The editor **Find and Replace** dialog appears with the **Find** tab selected.
3. Type the text that you want to find in the **Find what:** section. A history of previously searched words is available by clicking the **Find List** button.
4. Change search options if required.
5. Click **Find**.

Related Topics

[About the Text Editor on page 693](#) | [Replacing Text in the Text Editor on page 696](#) | [Locating a Line and Column in the Text Editor on page 697](#)

Replacing text in the text editor

To replace a text string with another string, you use the **Find and Replace** command.

To replace a text string:

1. From the **Edit** menu, select **Replace...**
2. The editor **Find and Replace** dialog appears with the **Replace** tab selected.
3. Type the text that you want to change in the **Find what** box. A history of previously searched words is available by clicking the **Find List** button.
4. Type the text that you want to replace it with in the **Replace with** box. A history of previously replaced words is available by clicking the **Replace List** button.
5. Change search options (see below) if required.
6. Click **Replace** to replace the first occurrence of the searched text, or **Replace All** to replace all occurrences.

Search Options

The **Search** box allows you to select the search mode:

- **All** searches for the first occurrence from the beginning of the file.
- **Selected** searches through selected text only.
- **Forward** and **Backward** specify the direction of the search, starting at the current cursor position.
- **Match case** restricts search criteria to the exact same case.
- **Match whole word only** restricts the search to complete words.
- **Use regular expression** allows you to specify UNIX-like regular expressions as search criteria.

Related Topics

[About the Text Editor on page 693](#) | [Finding Text in the Text Editor on page 695](#) | [Locating a Line and Column in the Text Editor on page 697](#)

Locating a line and column in the text editor

The **Go To** command allows you to move the cursor to a specified line and column within the Text Editor.

To use the Go To feature:

1. From the **Edit** menu, select **Go To...**
2. The Text Editor's **Find and Replace** dialog appears with the **Go To** tab selected.
3. Enter the number of the line or column or both.
4. Click **Go** to close the dialog box and to move the cursor to the specified position.

Related Topics

[About the Text Editor on page 693](#) | [Replacing Text in the Text Editor on page 696](#) | [Finding Text in the Text Editor on page 695](#)

Text editor syntax coloring

The Text Editor provides automatic syntax coloring for C, Ada and C++ source code as well for the C and Ada, C++ test script languages, and System Testing Script Language. The Text Editor automatically detects the language from the filename extension.

If the filename does not have a standard extension, you must select the language from the **Syntax Color** submenu.

To manually set the syntax coloring mode:

1. From the **Editor** menu, select the desired language through the **Syntax Color** submenu.

Related Topics

[Text Editor Preferences on page 1012](#)

Commenting code in the text editor

The text editor allows you simply to comment and uncomment blocks of source code or test script. The same principle also applies to declaring native C code in a C test script by prefixing each with a dash (#) character.

To comment a block of source code

1. In the text editor, select a block of code.
2. Click the **Comment** (-- or // depending on the language) button in the toolbar.

To uncomment a block of commented source code

1. In the text editor, select a block of commented code.
2. Click the **Uncomment** (-- or // depending on the language) button in the toolbar.

To declare native code in a .ptu test script

1. In the **.ptu** test script, select a block of native C code.
2. Click the **Native #** button in the toolbar.

Related Topics

[Editing code and test scripts on page 693](#) | [Text editor syntax coloring on page 697](#)

Running tests and applications

Building and Running a Node

You build and execute workspace nodes by using the **Build**  button on the Build toolbar. The build process compiles, links, deploys, executes, and then retrieves results. However, you first have to specify the various build options.

You can use the **Build**  command to execute any application node, as well as a single specific source file, a group node or even the whole project.

Note When you run the **Build**  command, all open files are saved. This means that any unsaved changes will actually be taken into account for the build.

Before building a node:

1. Select the correct Configuration for your target in the build toolbar.
2. Exclude any temporarily unwanted nodes from the build.
3. Select the build options for each particular node.
4. If necessary, clean up files left by any previous executions by clicking the **Clean**  button.

To build and execute the node:

1. From the Build toolbar, click the **Build**  button.
2. During run-time, the Build Clock indicates the execution time and the green LED flashes. The Project Explorer displays a check mark next to each item to mark progression of the build process.
3. When the build process is finished, you can view the related test reports.

Note If you are running a component test node containing multiple test scripts on the same source file, the test will run correctly but the results from the last compiled test will overwrite the previous ones. Only the results from the last test will be available.

To stop the execution:

1. If you want to stop the execution of a node before it finishes, or if the application does not stop by itself, click the **Stop Build/Execution** button.

Note You can save the content of your build log (compilation, command options, all traces...) in a file so that you can send it to the support if you encounter any problem. To do so, in the main toolbar, click **Project > Save build log**. Then, enter a file name in the window that opens to save the content of the log. To get more details in the log file, it is recommended to enable the verbose output preference. From the main toolbar, click **Edit > Preferences** and click to enable the **Verbose output** option.

Related Topics

[Selecting Configurations on page 210](#) | [Selecting Build Options on page 699](#) | [Excluding a Node from a Build on page 700](#) | [Cleaning Up Generated Files on page 702](#)

Selecting Build Options for a Node

The HCL OneTest™ Embedded Graphical User Interface allows you to specify the actions that will be performed during a build for each node in the test project.

Build options contain two sections:

- **Stages** contains the compilation options. In most cases, you will need to select the **All** option to ensure the test is up to date.
- **Runtime Analysis** allows you to enable debugging and Runtime Analysis tools.

Build options are linked to each node through the Configuration Settings mechanism. For example, you can decide to only apply Code Coverage to one node in the project. If you want your changes to apply to the entire project, set the build options on the project node.

By default, the build options of each node are inherited from those of the parent node. When you override the settings of a parent node, changes are propagated to all child nodes within the same Configuration.

To set the build options of a node:

1. In the **Project Explorer**, click the **Settings**  button.
2. Select a node in the **Project Explorer** pane.
3. Select the **Build** node.
4. Click the **Value** column the ... button.

5. Select the Runtime Analysis features (Memory Profiling, Performance Profiling, Code Coverage and Runtime Tracing) and build options to use them on the current node.
6. Click **OK** and **Apply**.

Related Topics

[Building and Running a Node on page 698](#) | [Excluding a Node from a Build on page 700](#) | [Cleaning Up Generated Files on page 702](#)

Excluding a Node from a Build

In some cases, you might want to exclude one or several nodes from the build process. This can be done by changing the Build state of the node directly in the Project Explorer, as described below, or through the Properties window.

Note If you exclude a node that contains child nodes, such as an application node, a group or even a project, none of the contents of the node are executed.

In the Project Explorer, there are three possible build states:

Build state	Symbol	Description
Build	◆	The node is normally built and executed.
Report only	R	The node is not built, but is used to produce the report.
Exclude from Build	X	The node is not built and ignored.

The **Report only** option means that only static result files (**.tsf** and **.fdc**) are used to generate the report, but the node is not built and does not produce any dynamic results.

To change the Build state of a node:

1. In the **Project Explorer**, click the Build state symbol to toggle the three different states.
2. In the **Properties** window set the Build property to No.

Related Topics

[Building and Running a Node on page 698](#) | [Excluding a Node from Instrumentation on page 700](#) | [Selecting Build Options on page 699](#) | [Properties Window on page 1023](#)

Excluding a Node from Instrumentation

In some cases, you might want to exclude one or several source files from the instrumentation process. This can be done directly in the Project Explorer, as described below, or through the Properties window.

- Instrumented files are displayed with a blue icon 
- Non-instrumented files are displayed with a white icon 

You can combine both of the following methods to exclude or include a large number of files from the instrumentation process.

To exclude entire directories from instrumentation:

1. In the **Project Explorer**, click the **Settings**  button.
2. Select Runtime Analysis, General Runtime Analysis and Selective Instrumentation.
3. In **Directories excluded from Instrumentation**, add the directories to be excluded.
4. Click **Ok**.

To turn off instrumentation for an individual node:

1. In the **Project Explorer**, select the node that you want to exclude from the build.
2. In the **Properties** window set the **Instrumented** property to **No**.

Related Topics

[Excluding a Node from a Build on page 700](#) | [General Runtime Analysis Settings on page 989](#)

Enabling and disabling tests, services and families

The Test Selection window enables you to enable or disable particular SERVICE, TEST or FAMILY blocks in the generated test driver. By default, all elements of a test script are enabled. During the run, the Component Test will only execute elements that are enabled.

To open the Test Selection window:

1. In the **Project Explorer**, right-click a **.ptu** test script.
2. From the pop-up menu, select **Test Selection**. Select the Service or Family tabs to specify the elements that you want to enable or disable.
 - Use this **Service** tab to enable or disable one or several SERVICE or TEST blocks defined in the test driver.
 - Use this **Family** tab to enable or disable one or several FAMILY blocks defined in the test driver.

3. Select **Use selection below** to modify the default list of tests, services or families that you want to include in the test. If the option **Use selection below** is not selected, then all listed items are enabled, regardless of whether they appear selected or not.
4. Click **Close**.

Related Topics

[About Component Testing for C and Ada on page 447](#) | [Test Script Structure on page 451](#)

Cleaning Up Generated Files

In some cases, you might want to delete any files created by a build execution, such as to perform the build process in a clean environment or when you are running short of disk space.

Use the **Clean All Generated Files** command to do this.

To clean your workspace:

1. From the Build toolbar, click the **Clean All Generated Files**  button.

Related Topics

[Building and Running a Node on page 698](#) | [Selecting Build Options on page 699](#)

Debug mode

The Debug option allows you to build and execute your application under a debugger.

The debugger must be configured in the Target Deployment Port. See the **The Target Deployment section** for further information.

Note Before running in Debug mode you must change the Compilation and Link Configuration Settings to support Debug mode. For example set the **-g** option with most Linux compilers.

Related Topics

[Configuration Settings on page 658](#) | [Selecting Build Options on page 699](#)

Setting Environment Variables for CLI tools

The command line interface (CLI) tools require several environment variables to be set.

These variables determine, for example, the Target Deployment Port (TDP) that you are going to use. The available TDPs are located in the product installation directory, under **targets**. Each TDP is contained in its own sub-directory.

Prior to running any of the CLI tools, the following environment variables must be set:

- **TESTRTDIR** indicates the installation directory of the product
- **ATLTGT** and **ATUTGT** specify the location of the current TDP: **\$TESTRTDIR/targets/ <tdp>**, where *<tdp>* is the name of the TDP.
- **PATH** must include an entry to **\$TESTRTDIR/bin/ <platform> / <os>**, where *<platform>* is the hardware platform and *<os>* is the current operating system.

You must also add the product installation **bin** directory to your **PATH**.

Note Some command-line tools may require additional environment variables. See the pages dedicated to each command in **Reference** section of the help.

Most of these environment variables are set during installation of the product. Under Linux, use the `testrtinit.sh` script to set these variables. See the [Reference on page 981](#) section for more information about these scripts.

Automated Testing

If you are using Component Testing or System Testing features, the following additional environment variables must be set:

- **ATUDIR** for Component Testing, points to **\$TESTRTDIR/lib**
- **ATS_DIR**, for System Testing, points to **\$TESTRTDIR/bin/ <platform> / <os>**, where *<platform>* is the hardware platform and *<os>* is the current operating system.

Library Paths

UNIX platforms require the following additional environment variable:

- On Solaris and Linux platforms: **LD_LIBRARY_PATH** points to **\$TESTRTDIR/lib/ <platform> / <os>**
- On HP-UX platforms: **SHLIB_PATH** points to **\$TESTRTDIR/lib/ <platform> / <os>**
- **On AIX platforms: LIB_PATH** points to **\$TESTRTDIR/lib/ <platform> / <os>**

where *<platform>* is the hardware platform and *<os>* is the current operating system.

Example

The following example shows how to set these variables for HCL OneTest™ Embedded with a **sh** shell on a Suse Linux system. The selected Target Deployment Port is **clinuxgnu** .

```
TESTRTDIR=/opt/HCL OneTest™ Embedded/TestRealTime.v2002R2
```

```
ATCDIR=$TESTRTDIR/bin/intel/linux_suse
```

```
ATUDIR=$TESTRTDIR/lib
```

```
ATS_DIR=$TESTRTDIR/bin/intel/linux_suse
ATLTGT=$TESTRTDIR/targets/clinugnu
ATUTGT=$TESTRTDIR/targets/clinugnu
LD_LIBRARY_PATH=$TESTRTDIR/lib/intel/linux_suse
PATH=$TESTRTDIR/bin/intel/linux_suse:$PATH

export TESTRTDIR

export ATCDIR

export ATUDIR

export ATS_DIR

export ATLTGT

export ATUTGT

export LD_LIBRARY_PATH

export PATH
```

Report Viewer

The Report Viewer allows you to view Test or Runtime Analysis reports from Component Testing, System Testing and any of the Runtime Analysis tools

To learn about	See
Opening and browsing Test or Runtime Analysis reports.	Using the Report Viewer on page 705
Exporting Test or Runtime Analysis reports in HTML.	Exporting reports on page 705
Interpreting results of Test or Runtime Analysis reports.	Understanding Reports on page 706
Changing and customizing the zoom level on a report.	Setting a Zoom Level on page 707
Obtaining a summery report.	Displaying a Report Summary Header on page 707
Understanding Report Viewer toolbar buttons.	Report Viewer Toolbar
Customizing the Test or Runtime Analysis reports.	Report Viewer Style Preferences on page 1018

Using the report viewer

Most reports are produced as **.xrd** files, which are generated during the execution of the test or application node.

To navigate through the report:

1. You can use the Report Explorer to navigate through the report. Click an element in the **Report Explorer** to go to the corresponding line in the **Report Viewer**.
2. You can also jump directly to the next or previous *Failed* test in the report by using the **Next Failed Test** or **Previous Failed Test** buttons.

To filter out passed tests:

You can choose to only display the Failed tests in the report.

1. From the **Report Viewer** menu, select **Failed Tests Only** or click the **Failed Tests Only** button in the Report Viewer toolbar.
2. To switch back to a complete view of the report, from the **Report Viewer** menu, select **All Tests** or click the **All Tests** button in the Report Viewer toolbar.

To hide or show report nodes:

The Report Viewer can filter out certain elements of a report.

1. From the **Report Viewer** menu, select the elements that you want to hide or show.

Related Topics

[Openin a Report on page 683](#) | [Report Explorer on page 1024](#) | [Understanding Reports on page 706](#) | [Report Viewer preferences on page 1018](#) | [Viewing UML sequence diagrams on page 402](#)

Exporting reports to HTML

You can export the following Test and Runtime Analysis reports to HTML.

- Memory Profiling
- Performance Profiling
- Code Coverage
- Static Metrics
- Component Testing for C and Ada
- Component Testing for C++
- System Testing for C

There are two methods of exporting to HTML, depending on whether you are viewing the report in a loaded project or you are viewing the report as a standalone document.

To export a report to HTML:

1. Open the report:
 - If the report is in a project, open the project in _HCL OneTest™ Embedded and select a report in the Project Browser.
 - If the report is not in a project, open the report with the **studioreport** command line. This automatically creates a project.
2. Select **File > Export** project report in HTML file format.
3. Choose between exporting the entire project (all the report files contained in the project) or only the selected report.
4. Select the type of report to export (only if you have selected the entire project) and the directory where you want the HTML files to be generated.
5. Click **Export**.

Note The **Generate HTML** menu option in the report viewer menu is no longer supported.

Related Topics

[Viewing reports on page 704](#)

[Studio Report - studioreport on page 1062](#)

Understanding Reports

HCL OneTest™ Embedded generates Test and Runtime Analysis reports based on the execution of your application.

Runtime analysis reports

- [Memory Profiling on page 363](#)
- [Performance Profiling on page 384](#)
- [Code Coverage on page 349](#)
- [Runtime Tracing on page 394](#)

Static analysis reports

- [Static metrics on page 226](#)
- [Code review on page 299](#)

Test verdict reports

- [Component Testing for C and Ada on page 508](#)
- [Component Testing for C++ on page 522](#)
- [System Testing for C on page 639](#)

Setting the zoom level

UML sequence diagrams and other reports can be viewed with different zoom levels.

To set the zoom level:

1. You can directly change the zoom level in the View Toolbar by using the **Zoom In**  and **Zoom Out**  buttons

Or

1. Select one of the pre-defined or custom levels from the **Choose Zoom Level** box of the View Toolbar.

Related Topics

[Toolbars on page 1025](#)

Displaying a report summary header

In some cases, test reports can be quite large and complicated when all you want is a quick summary. The report viewer can display a short summary header at the top of a Component Testing test report.

The summary header contains:

- The name of the report
- The number of failed and passed tests
- The total number of tests

To display the summary header for the current test:

1. Open a test report
2. From the **Test Report** menu, select **Show Header**.

To display a full summary for the entire project:

1. Right-click the main project node
2. Select **View Report** and **Test**.
3. From the **Test Report** menu, select **Show Header**.

Related Topics

Report Viewer Toolbar | [Using the Report Viewer on page 705](#)

Viewing graphical results

HCL OneTest™ Embedded can produce graphs that display the execution results of Component Testing and System Testing tests that involve loops and **FOR** loops. Graphs can be displayed and manipulated in the **Graphic Viewer**.

To enable graph output:

1. In the **Project Explorer**, click the **Settings**  button.
2. Select a node in the **Project Explorer** pane.
3. In the **Configuration Settings** list, expand **Build** and **Build Options**.
4. Click '...' to edit **Environment Variables** and set the variable **ATURTX** to **ACTIV**.
5. Click **OK** to validate the changes.

To open the Graphic Viewer:

1. Execute your test with the **Build** command.
2. Right-click the application or test node.
3. From the pop-up menu, select **View Report**, and then click **Graphic**.

Note Reports cannot be viewed before the application or test has been executed.

To change the view displayed in the Graphic Viewer:

1. With the Graphic Viewer open, open the **Graphics** menu (or **Command** menu in Eclipse).
 - a. **Display Curves**, **Display XY Curves**, or **Display XYZ Curves** enable you to change the axis of view of the graph. In 3D mode (XYZ curves) you can change the angle of view by clicking and moving the mouse.
 - b. **Configure** enables you to change the scale and display options for each axis.
 - c. **Reload** enables you to reload the **.rtx** graph file.
 - d. **Generate HTML** produces an html file with a **.png** image output of the current view.

Related Topics

[Graphic viewer preferences on page 1018](#) | [Build settings on page 983](#)

Monitoring the test process

About the test process monitor

The test process monitor provides an integrated monitoring feature that helps project managers and test engineers obtain a statistical analysis of the progress of their development effort.

Each generated metric is stored in its own file and consists of one or more fields.

The test process monitor works by gathering the statistical data from these files and then generating a graphical chart based on each field.

The preexistence of a file is required before running the test process monitor. Files are created either by running a runtime analysis feature that generates test process data, or by creating and updating your own file.

Note Only the Code Coverage tool provides data for the test process monitor. You can, however, build your own files with the Test Process Monitor tool (**tpmadd**).

Related Topics

[Test Process Monitor Tool on page 1059](#)

Changing Curve Properties

The **Curve Properties** menu allows you to change the way a particular graph is displayed.

To change the curve color:

1. Right-click a curve.
2. From the pop-up menu, select **Change Curve Color**.
3. Use the **Color Palette** to select a new color, and click **OK**.

To hide a curve:

1. Right-click a curve.
2. From the pop-up menu, select **Hide Curve**.

To set a maximum value:

Changing the maximum displayed value for a curve actually changes the scale at which it is displayed. For instance, when a curve only reaches 100, there is no point in displaying it at on a scale of 1000, unless you want to compare it with another curve that uses that scale.

1. Right-click a curve.
2. From the pop-up menu, select **Set Max Value**.
3. Enter the scale value, and click **OK**.

Note Setting a maximum value lower than the actual maximum value of a curve can result in erratic results.

To display a scale:

For any curve, you can display a scale on the right or left-hand side of the graph. When you display a new scale, it replaces any previously displayed one.

1. Right-click a curve.
2. From the pop-up menu, select **Right Scale** or **Left Scale**.

Custom Curves

In some cases, you may want to remove certain figures from a chart to make it more relevant. The custom curves capability allows you to alter the chart by selecting the records that you want to include.

Note Using the custom curves capability does not impact the actual database. If you remove a record from the chart by using the custom curves function, the actual record remains in the database and may impact other figures.

Custom curves create a new metric, using the name of the base metric, with a Custom prefix.

To create a custom curve:

1. Make sure a user is selected in the **Report Explorer** pane. If not, select a user.
2. From the **Project** menu, select **Test Process Monitor** and **Custom Curves**.
3. In the **Custom Curves** dialog box, select a metric and the start and end date of your chart.
4. The record list displays all the records contained in the database of that metric. Select the records that you want to use for your custom curve. Clear the records that you do not want to use.
5. Click **OK**. A new metric is created.

To change a custom curve:

1. From the **Project** menu, select **Test Process Monitor** and **Custom Curves**.
2. In the **Custom Curves** dialog box, select the Custom metric that you want to modify.
3. Select the records that you want to use for your custom curve. Clear the records that you do not want to use.
4. Click **OK**.

Event markers

Use event markers to identify milestones or special events within your Test Process Monitor chart. An event marker is identified by the date of the event and a marker label.

Event markers appear as bold vertical lines in a Test Process Monitor chart.

To create an event marker:

1. Right-click the location where you want to put the chart
2. From the pop-up menu, select **Event Properties** and **New Event**.
3. Enter the date of the event, and a marker label, and click **OK**.

To remove an event marker:

1. Right-click the event marker that you want to hide.
2. From the pop-up menu, select **Delete Event**.

To hide a specific event marker:

Hiding a marker does not remove it. You can still make the marker reappear.

1. Right-click the event marker that you want to hide.
2. From the pop-up menu, select **Hide Event**.

To hide or show all event markers:

1. In the **Test Process Monitor** toolbar, click the **Events** button to hide all event markers.
2. Click again to show all hidden event markers.

Setting the time scale

The scale capability defines the period that you want to view in the **Test Process Monitor** window. This option allows you to select an annual, monthly or daily view, as well as a user-definable time period.

To set the time scale:

1. Select a user in the **Report Explorer** pane.
2. From the **Project** menu, select **Test Process Monitor, Scale** and the desired time scale.
3. If you chose **Customize**, enter the start and end date of the period that you want to monitor, and click **OK**.

Adding a metric

Metrics generated Code Coverage or other tools are directly available through the Test Process Monitor. Each metric file contains one or several fields.

To open a metric database a metric chart:

1. From the **Project** menu, select **Test Process Monitor** and either **Project** or **Current Workspace**. **Current Workspace** applies to the user of the current workspace. **Project** applies to all workspace users in the project.
2. If a new metric database is detected, you need to provide a name for the metric, as well as a label for each field of the database.
3. In the **Report Explorer**, select a user.
4. From the **Project** menu, select **Test Process Monitor**, the metric and the field that you want to display.

You can add as many curves as you want to the chart.

To hide a curve:

1. Right-click a curve.
2. From the pop-up menu, select **Hide Curve**.

Customizing tools

Custom tools overview

The **Tools** menu is a user-configurable menu that allows you to access personal tools from the HCL OneTest™ Embedded graphical user interface (GUI). You can customize the Tools menu to meet your own requirements.

Custom tools can be applied to a selection of nodes in the Project Explorer. Selected nodes can be sent as a parameter to a user-defined tool application. A series of macro variables is available to pass parameters on to your tool's command line.

The **Tool Configuration** dialog allows you to configure a new or existing tool.

In the **Tools** menu, each tool appears as a sub-menu item, or **Name**, with one or several associated actions or **Captions**.

Identification

In this tab, you describe how the tool will appear in the **Tools** menu.

- Enter the **Name** of the tool sub-menu as it will appear in the Tools menu and a **Comment** that is displayed in the lower section of the Toolbox dialog box.
- Select the type of tool:
 - Select **Change Management System** if the tool is used to send and retrieve from a change management system. When **Change Management System** is selected, **Check In** and **Check Out** actions are automatically added to the Action tab (see below) and a **Change Management System** toolbar is activated.
 - Select **External Editor** if the tool is an editor. When **External Editor** is selected, you can select **Automatic Launch** if you want this editor to replace the HCL OneTest™ Embedded editor for file extensions specified in the **Files Filter** list. (for example: "*.c;*.cpp;*.txt").
 - Select **Other** if the tool is neither a configuration management tool nor an editor.
- Clear the **Add to Tools menu** check box if you do not want the tool to be added to the Tools menu.
- Select **Send messages to custom tab** if you want to view the tool's text output to be sent to a specific tab in the **Output Window**.
- Use the **Icon** button to attach a custom icon to the tool that will appear in the **Tools** menu. Icons must be either **.xpm** or **.png** graphic files and have a size of 22x22 pixels.

Actions

This tab allows you to describe one or several actions for the tool.

- The **Actions** list displays the list of actions associated with the tool. If **Change Management System** is selected on the **Identification** tab, **Check In** and **Check Out** tool commands will be listed here. These cannot be renamed or removed.
- **Menu text** is the name of the action that will appear in the **Tools** sub-menu.
- **Command** is a shell command line that will be executed when the tool action is selected from **Tools** menu. Command lines can include GUI macro variables and functions.

A series of macro variables is available to pass parameters on to your tool's command line. See **GUI Macro Variables** in the **Reference** section for detailed information about using the macro command language.

Click **OK** to validate any changes made to the Tool Edit dialog box.

Examples

IBM Rational ClearCase is pre-configured in the Tools menu as the default configuration management tool. If you are using another tool you can simply add it to the Tools menu. For example, to add CVS to the Tools menu:

1. Select **Tools > Configure Tools** and click **Add**.
2. On the **Identification** page, enter **CVS** in the **Name** field, and select **Change Management System**.
3. On the **Actions** page, enter the following command lines:
 - Add to Source Control: **cvcs -add \$\$VCSITEMS**
 - Check Out: **cvcs -co \$\$VCSITEMS**
 - Check In: **cvcs -ci \$\$VCSITEMS**
4. Click **OK**.

To add, for example, the Windows Notepad editor to the Tools menu:

1. Select **Tools > Configure Tools** and click **Add**.
2. On the **Identification** page, enter **Notepad** in the name field, and select **External Editor**.
3. If you want Notepad to replace the default editor for **.c** files for example, then select Automatic launch and enter:

*.c

1. On the **Actions** page, enter:

notepad.exe \$\$NODEPATH

1. Click **OK**.

Related Topics

[Configuring the Tools menu on page 714](#) | [GUI macro variables on page 1029](#)

Customizing the Tools menu

The **Tools** menu is a user-configurable menu that allows you to access personal tools from the HCL OneTest™ Embedded graphical user interface (GUI). You can customize the Tools menu to meet your own requirements.

In the **Tools** menu, each tool appears as a sub-menu item, or **Name**, with one or several associated actions or **Captions**.

The **Tool Configuration** dialog allows you to configure a new or existing tool.

Using the Tools Menu

To use a user-defined tool:

1. Select an icon from the **Project Explorer** pane.
2. Click the **Tools** menu and select the tool you want to use.

To add a new tool to the Tools menu:

1. Select **Tools > Configure Tools**.
2. To add a new tool, click **Add...** If you want to create and modify a copy of an existing tool, select the existing tool, click **Copy** and click **Edit...**
3. Edit the tool in the **Tool Edit** box. See [Custom tools overview on page 712](#).
4. Click **OK** and **Close**.

To edit a user-defined tool:

1. Select **Tools > Configure Tools**.
2. Select the tool that you want to modify and click **Edit...**
3. Edit the tool in the **Tool Edit** box. See [Custom tools overview on page 712](#).
4. Click **OK** and **Close**.

To remove a tool from the Tools menu:

1. Select **Tools > Configure Tools**.
2. Select an existing tool from the tool list.
3. Click **Remove** and **Close**.

Related Topics

[About the Tools Menu on page 712](#)

Test script languages

This section contains advanced information for using HCL OneTest™ Embedded test script languages, component test and system test command line tools

To learn about	See
Component Testing for C test scripts	C test script language on page 716
Component Testing for C++ test scripts	C++ test driver script (.otd) on page 751

Component Testing for C++ contract check scripts	C++ contract check script (.otc) on page 786
Component Testing for Ada test scripts	Ada test script language on page 800
System Testing for C test scripts	System Testing driver script (.pts) on page 833
System Testing for C supervisor scripts	System Testing supervisor script (.spv) on page 893

Component Testing for C

C test script language reference

Component Testing for C uses its own simple language for test scripting.

This section describes each keyword of the C test script language, including:

- Syntax
- Functionality and rules governing its usage
- Examples of use

Notation conventions

Throughout this guide, command notation and argument parameters use the following standard convention:

Notation	Example	Meaning
BOLD	BEGIN	Language keyword
<italic>	<filename>	Symbolic variables
[]	[<option>]	Optional items
{ }	{ <filenames> }	Series of values
{ { }	{ { <filenames> } }	Optional series of variables
	on off	OR operator

C test script keywords are case insensitive. This means that **STUB**, **stub**, and **Stub** are interpreted the same way. The keyword **others** is an exception, and must always be expressed in lower case.

For conventional purposes however, this document uses upper-case notation for the C test script keywords in order to differentiate from native source code.

Split statements

C test script statements may be split over several lines in a **.ptu** test script. Continued lines must start with the ampersand (&) symbol to be recognized as a continuation of the previous line. No tabs or spaces should precede the ampersand.

Statements must be shorter than 2048 characters, although this limit may be lower on some platforms.

Language identifiers

A C test script identifier is a text string used as a label, such as the name of a **TEST** or a **STUB** in a **.ptu** test script.

Identifiers are made of an unlimited sequence of the following characters:

- a-z
- A-Z
- 0-9
- _ (underscore)

Spaces are not valid identifier characters.

Note that identifiers starting with a numeric character are allowed. The following statement, for example, is syntactically correct:

```
TEST 1
```

```
...
```

```
END TEST
```

C test script identifiers are case sensitive. This means that **LABEL**, **label**, and **Label** are three different identifiers.

C test script structure

The C test script language allows you to structure tests to:

- Describe several test cases in a single test script,
- Select a subset of test cases according to different Target Deployment Port criteria.

Test script filenames must contain only plain alphanumeric characters.

Basic structure

A typical C Component Testing **.ptu** test script looks like this:

```
HEADER add, 1, 1

<variable declarations for the test script>

BEGIN

SERVICE add

<local variable declarations for the service>

TEST 1

FAMILY nominal

ELEMENT

VAR variable1, INIT=0, EV=0

VAR variable2, INIT=0, EV=0

#<call to the procedure under test>

END ELEMENT

END TEST

END SERVICE
```

All instructions in a test script have the following characteristics:

- All statements begin with a keyword.
- Statements are not case sensitive (except when C expressions are used).
- Statements start at the beginning of a line and end at the end of a line. You can, however, write an instruction over several lines using the ampersand (&) continuation character at the beginning of additional lines. In this case, the ampersand must be the very first character on that line; no spaces or tabs should precede it.
- Statements must be shorter than 2048 characters, although this limit may be lower on some platforms.

Structure statements

The following statements allow you to describe the structure of a test.

- **HEADER:** For documentation purposes, specifies the name and version number of the module being tested, as well as the version number of the tested source file. This information is displayed in the test report.
- **BEGIN:** Marks the beginning of the generation of the actual test program.

- **SERVICE:** Contains the test cases related to a given service. A service usually refers to a procedure or function. Each service has a unique name (in this case add). A **SERVICE** block terminates with the instruction **END SERVICE**.
- **TEST:** Each test case has a number or identifier that is unique within the block **SERVICE**. The test case is terminated by the instruction **END TEST**.
- **FAMILY:** Qualifies the test case to which it is attached. The qualification is free (in this case **nominal**). A list of qualifications can be specified (for example: **family, nominal, structure**) in the Tester Configuration dialog box.
- **ELEMENT:** Describes a test phase in the current test case. The phase is terminated by the instruction **END ELEMENT**. The different phases of the same test case cannot be dissociated after the tests are run, unlike the test cases introduced by the instruction **NEXT_TEST**. However, the test phases introduced by the instruction **ELEMENT** are included in the loops created by the instruction **LOOP**.

The three-level structure of the test scripts has been deliberately kept simple. This structure allows:

- A clear and structured presentation of the test script and report
- Tests to be run selectively on the basis of the service name, the test number, or the test family.

Related Topics

[C test script keywords on page 719](#) | [C test script language on page 716](#) | [Writing a Test Script on page 450](#)

C test script keywords

The C Test Script Language keywords are not case sensitive. This means that *STUB*, *stub*, and *Stub* are equivalent. For conventional purposes however, this document uses upper-case notation for the C Test Script Language keywords in order to differentiate from native source code.

Block Keywords

- [ELEMENT...END ELEMENT on page 723](#)
- [ENVIRONMENT...END ENVIRONMENT on page 725](#)
- [INITIALIZATION...END INITIALIZATION on page 731](#)
- [SERVICE...END SERVICE on page 733](#)
- [SIMUL...ELSE_SIMUL...END SIMUL on page 734](#)
- [TERMINATION...END TERMINATION on page 738](#)
- [TEST...END TEST on page 738](#)

Other Keywords

- [BEGIN on page 720](#)
- [COMMENT on page 721](#)
- [DEFINE STUB on page 722](#)
- [FAMILY on page 726](#)
- [FORMAT on page 727](#)
- [HEADER on page 728](#)
- [IF...ELSE...END IF on page 729](#)
- [INCLUDE on page 730](#)
- [NEXT_TEST on page 732](#)
- [STUB on page 735](#)
- [USE on page 739](#)
- [VAR, ARRAY and STR on page 741](#)
 - [<initialization> Parameter on page 744](#)
 - [<expected> Parameter on page 747](#)
 - [<variable> Parameter on page 743](#)

BEGIN

C Test Script Language

Purpose

The **BEGIN** instruction marks the beginning of the test program.

Syntax

BEGIN

Description

BEGIN marks the beginning of the C code generation.

The **BEGIN** instruction is mandatory and must be located before any other Component Testing instruction for C, except a **HEADER** instruction.

If the **BEGIN** keyword is not found, a warning message is generated and a **BEGIN** instruction is implicitly created before the first occurrence of a **SERVICE** instruction.

Related Topics

[HEADER on page 728](#) | [SERVICE on page 733](#)

COMMENT

C Test Script Language

Purpose

The **COMMENT** instruction adds a textual comment to the test report.

Syntax

COMMENT [*<text>*]

Argument

<text> is an optional text string to be displayed.

Description

The **COMMENT** instruction is optional and can be used anywhere in the test script.

The position of the **COMMENT** instruction in the test script determines the position where the comment is displayed in the test report:

- Before the first **SERVICE** block: the comment is displayed after the report information header and before the first service.
- Inside a **SERVICE** block: the comment is displayed in the service header, before the test descriptions.
- Outside a **SERVICE** block: the comment is displayed in the following service header, before the test descriptions.
- After the last **SERVICE** block: the comment is ignored.
- Inside an **ELEMENT** block: the comment is displayed before the variable state descriptions.
- After a **TEST** instruction: the comment is displayed in the test header, before the variable descriptions.

Example

TEST 1

FAMILY nominal

COMMENT histogram computation for a black image

ELEMENT

Related Topics

[ELEMENT on page 723](#) | [TEST on page 738](#) | [SERVICE on page 733](#)

DEFINE STUB ... END DEFINE

The **DEFINE STUB** and **END DEFINE** instructions delimit a simulation block consisting of stub definition functions, variables or procedure declarations.

This instruction applies to C Test Script Language.

Syntax

```
DEFINE STUB <stub_name> [ <stub_dim> ]  
END DEFINE
```

<stub_name> is the mandatory name of a simulation block.

<stub_dim> is an optional maximum number of stub calls errors that will be displayed in the report.

Description

Defining stubs in a test script is optional.

By using the stub definitions, the C Test Script Compiler generates simulation variables and functions with the same interface for the stubbed variables and functions.

The purpose of these simulation variables and functions is to store and test input parameters, assign values to output parameters, and if necessary, return appropriate values.

Definitions of functions must be in the form of ANSI prototypes for C.

Stub parameters describe both the type of item used by the calling function and the passing mode. The parameter passing mode is specified by adding the following parameters before the parameter name:

- **_in** for input parameters
- **_out** for output parameters
- **_inout** for input/output parameters
- **_no** for parameters that you do not want to test

Additionally, when using the **_in** or **_inout** parameter, you can add an optional **_nocheck** parameter before the **_in** or **_inout** parameter (see the Example). This allows the parameters to be sent to the stub without being checked.

You can also add **_atcc_const** before the **_in** or **_inout** parameter when the parameter is declared as a **const** type. By default the **const** type modifier is ignored to allow better use of the parameters during the test, but this can lead to compilation issues. If you use **_atcc_const**, the parameter will be considered as a **const** type.

The parameter mode is optional. If no parameter mode is specified, the **_in** mode is assumed by default.

A return parameter is always deemed to be an output parameter.

Global variables defined in **DEFINE STUB** blocks replace the real global variables.

By default, only the first 10 errors are shown in the report. Additional errors are not recorded. The number of calls should be customized if necessary by using the `<stub_dim>` parameter.

DEFINE STUB / END DEFINE blocks must be located after the **BEGIN** instruction and outside any **SERVICE** block.

Example

An example of the use of stubs is available in the **StubC** example project installed with the application.

```
BEGIN
DEFINE STUB Example
  #int open_file(char _in f[100]);
  #int create_file(char _in f[100]);
  #int read_file(int _in fd, char _out l[100]);
  #int write_file(int fd, char _in l[100]);
  #int write(int fd, char _nocheck _in l[100]);
  #int close_file(int fd);
END DEFINE
```

```
DEFINE STUB Example
#int foo1 (int _in param1)
#{
# {int foo1_b ;
# foo1_b = 10 ;}
#}
END DEFINE
```

Related Topics

[STUB on page 735](#) instruction

ELEMENT ... END ELEMENT

C Test Script Language

Purpose

The **ELEMENT** and **END ELEMENT** instructions delimit a test phase or **ELEMENT** block.

Syntax

ELEMENT

END ELEMENT

Description

The **ELEMENT** instruction is mandatory and can only be located within a **TEST** block. If absent, a warning message is generated and the **ELEMENT** block is implicitly declared before the first occurrence of a **VAR**, **ARRAY**, **STR**, or **STUB** instruction.

The block must end with the instruction **END ELEMENT**. If absent, a warning message is generated and it is implicitly declared before the next **ELEMENT** instruction, or the **END TEST** instruction.

The **ELEMENT** block contains a call to the service under test as well as instructions describing the initializations and checks on test variables.

Positioning of **VAR**, **ARRAY**, **STR** or **STUB** instructions is irrelevant, with respect to the test procedure call since the Test Script Compiler separates these instructions into two parts:

1. The test initializer (described by **INIT**) is generated with the **ELEMENT** instruction.

The test of the expected value (described by **EV**) is generated with the **END ELEMENT** instruction.

Example

```
TEST 1
```

```
FAMILY nominal
```

```
ELEMENT
```

```
VAR x1, init = 0, ev = init
```

```
VAR x2, init = SIZE_IMAGE-1, ev = init
```

```
VAR y1, init = 0, ev = init
```

```
VAR y2, init = SIZE_IMAGE-1, ev = init
```

```
ARRAY image, init = 0, ev = init
```

```
VAR histo[0], init = 0, ev = SIZE_IMAGE*SIZE_IMAGE
```

```
ARRAY histo[1..SIZE_HISTO-1], init = 0, ev = 0
```

```
VAR status, init ==, ev = 0
```

```
#status = compute_histo(x1,y1,x2,y2,histo);
```

```
END ELEMENT
```

```
END TEST
```

Related Topics

[VAR on page 741](#) | [ARRAY on page 741](#) | [STR on page 741](#) | [STUB on page 735](#) | [NEXT_TEST on page 732](#) | [Initialization Expressions for C on page 744](#) | [Expected Value Expression for C on page 747](#)

ENVIRONMENT ... END ENVIRONMENT

The **ENVIRONMENT** instruction defines a test environment declaration, that is, a default set of test specifications. It applies to C Test Script Language.

Syntax

```
ENVIRONMENT <name> [ ( <param> { , <param> } ) ]
END ENVIRONMENT
```

<name> is a mandatory identifier that provides a unique environment name.

<param> is an optional identifier.

Description

The test environment defines a general context. Variables that are declared within a context can be overwritten by a **TEST** statement.

Every environment can contain parameters. The declared parameters can be used in initialization and expected value expressions. These parameters are initiated by the **USE** instruction.

The **END ENVIRONMENT** instruction marks the end of an environment declaration.

<name> specifies an environment name that is referenced in the **USE** instruction.

An environment must be defined after the **BEGIN** instruction.

Each environment is visible in the block in which it has been declared and in any blocks included in this block, after its declaration.

An environment can only contain **VAR**, **ARRAY**, **STR**, **FORMAT** or **STUB** instructions and conditional generation instructions. If it is empty, a warning message is generated.

An environment is activated by the **USE** instruction that defines its scope and its priority. **ENVIRONMENT** blocks are executed in the reverse order of their respective **USE** instruction.

After generating the initializations and the tests of an **ELEMENT** block, visible environments are included in order of priority, at every **END ELEMENT** instruction, in order to complete the initializations and tests.

The scope of an **ENVIRONMENT** block is important insofar as only "visible" environment blocks apply, and use clauses can be out of scope.

Example

```

ENVIRONMENT compute_histo
  VAR x1, init = 0, ev = init
  VAR x2, init = SIZE_IMAGE-1, ev = init
  VAR y1, init = 0, ev = init
  VAR y2, init = SIZE_IMAGE-1, ev = init
  ARRAY histo, init = 0, ev = 0
  VAR status, init ==, ev = 0
END ENVIRONMENT

```

Related Topics

[USE on page 739](#) | [VAR on page 741](#) | [ARRAY on page 741](#) | [STR on page 741](#) | [FORMAT on page 727](#)

instructions

FAMILY

C Test Script Language

Purpose

The **FAMILY** instruction groups tests by families or classes.

Syntax

FAMILY <family_name> { , <family_name> }

Argument

<family_name> is a mandatory identifier indicating the name of the test family. Typically, you could specify nominal, structural, or robustness families.

Description

The **FAMILY** instruction appears within **TEST** blocks, where it defines the families to which the test belongs.

When you run the test sequence, you can request that only tests of a given *family* are executed.

A test can belong to several families. In this case, the **FAMILY** instruction contains a <family_name> list, separated by commas.

The **FAMILY** instruction must be located before the first **ELEMENT** block of the **TEST** block and must be unique in the **TEST** block.

The **FAMILY** instruction is optional. If it is omitted, a warning message is generated and the test belongs to every family.

Example

TEST 1

FAMILY nominal

COMMENT histogram computation on a black image

ELEMENT

Related Topics

[ELEMENT on page 723](#) | [TEST on page 738](#)

FORMAT

C Test Script Language

Syntax

FORMAT <variable> = [<new type>[#<display directive> [<size>]]

FORMAT <type> = <new type>[#<display directive> [<size>]]

FORMAT <field> = <new type>[#<display directive> [<size>]]

Description

The **FORMAT** instruction allows you to modify the type of the tested element, where:

- <variable> is a variable.
- <type> is a simple C type declared by typedef; in this case, the new type will be applied to all variables of this type.
- <field> is a member of a structure or a C union; in this case, the new type will be applied to all the members of this field.

The <new type> is an abstract C type.

The optional <display directive> is one of the following suffixes for integers only:

- **#h** for hexadecimal display,
- **#b** for binary display,
- **#u** for unsigned decimal display,
- **#d** for signed decimal display,

With the following possibilities for floating variables:

- **#f** to display without an exponent,
- **#e** to display with an exponent.

For integers, <size> is the number of bits to be displayed. For floating variables, <size> is the number of the number of digits after the decimal point.

Associated Rules

The **FORMAT** statement is optional and must be located after the **BEGIN** statement.

The **FORMAT** definition can be replaced by an optional <format> parameter in a **VAR** statement.

It is applicable immediately, only in the block in which it is declared.

<variable> follows standard C syntax rules. <type> is a C identifier used in *typedef*, *struct* or *union* instructions.

<format> is an abstract C type.

If the change is to be applied to array elements, you can use an abstract C type to describe the new modified variable, field, or type.

A format cannot be empty. It must contain either the abstract C type or the display directive.

In the display directive, the size is optional. The size must be a multiple of 8 for the integers. The default values for this size are the following ones:

- For integers, the number of bits of the abstract type if it is given, or if it is not, the number of bits of the type or the variable whose printing format is modified

For **#f**, 6 digits after the decimal point and for **#e**, 2 digits after the decimal point

Example

```
#char x;
#char t[10];
FORMAT t = int -- t is an array of integers
FORMAT x = int#h8 -- display in hexa, only 8 bits
FORMAT y = #b -- display in binary without modifying the type
FORMAT z = short#u -- display in unsigned decimal
FORMAT f1 = #f -- displays for example 3.670000
FORMAT f1 = #f4 -- displays for example 3.6700
FORMAT f1 = #e4 -- displays for example 0.36700E1
```

Related Topics

[VAR, ARRAY and STR on page 741](#)

HEADER

C Test Script Language

Purpose

The **HEADER** instruction specifies the name and version of the module under test as well as the version number of the test script.

Syntax

HEADER <module_name> , <module_version> , <test_plan_version>

<module_name>, <module_version> and <test_plan_version> are character strings with no restrictions, except for versions beginning with a dollar sign ('\$'). These instructions must be followed by an identifier.

Description

This information contained in the **HEADER** keyword is reproduced in the test report header to identify the test sequence.

The module and test script versions can be read from the environment variables if they are identifiers beginning with a dollar sign (\$).

The **HEADER** instruction is mandatory, but its arguments are optional. It must be the first instruction in the test program. If it is absent, a warning message is generated.

Example

```
HEADER histo, 01a, 01a
```

```
BEGIN
```

```
IF ... ELSE ... END IF
```

C Test Script Language

Syntax

IF <condition> { , <condition> }

ELSE

END IF

Description

The **IF**, **ELSE** and **END IF** statements allow conditional generation of the test program.

These statements enclose portions of script that are included depending on the presence of one of the conditions in the list provided to the C Test Script Compiler by the **-define** option.

The <condition> list forms a series of conditions that is equivalent to using an expression of logical **ORs**.

The **IF** instruction starts the conditional generation block.

The **END IF** instruction terminates this block.

The **ELSE** instruction separates the condition block into 2 parts, one being included when the other is not.

Associated Rules

<condition> is any identifier. You must have at least one condition in an **IF** instruction.

This block can contain any scripting or native language.

IF and **END IF** instructions must appear simultaneously.

The **ELSE** instruction is optional.

The generating rules are as follows:

- If at least one of the conditions specified in the **IF** instruction's list of conditions appears in the list associated with the -define option, the first part of the block is included.

If none of the conditions specified in the **IF** instruction appears in the list associated with the -define option, then the second part of the block is included (if **ELSE** is present).

The **IF...ELSE...END IF** block is equivalent to the following block in C:

```
#if defined(<condition>) { || defined(<condition>) } ...
```

```
...
```

```
#else
```

```
...
```

```
#endif
```

Example

```
IF test_on_target
```

```
VAR register, init == , ev = 0
```

```
ELSE
```

```
VAR register, init = 0 , ev = 0
```

```
END IF
```

INCLUDE

C Test Script Language

Syntax

```
INCLUDE CODE <file>
```

```
INCLUDE PTU <file>
```

Description

The **INCLUDE** specifies an external file for the C Test Script Compiler to process.

When an **INCLUDE** instruction is encountered, the C Test Script Compiler leaves the current file, and starts pre-processing the specified file. When this is done, the C Test Script Compiler returns to the current file at the point where it left.

Including a file with the additional keyword **CODE** lets you include a source file without having to start each line with a hash character (`#`).

Including a file with the additional keyword **PTU** lets you include a test script within a test script. In this case, included **.ptu** test scripts must not contain **BEGIN** or **HEADER** statements.

Associated Rules

The name of the included file can be specified with an absolute path or a path relative to the current directory.

If the file is not found in the current directory, all directories specified by the **-incl** option are searched when the preprocessor is started.

If it is still not found or if access is denied, an error is generated.

The instruction **INCLUDE CODE <file>** inserts the entire file into the generated source code. A workaround to this is to use the following line in C:

```
##include "<file>"
```

Example

```
INCLUDE CODE file1.c
```

```
INCLUDE CODE ../file2.c
```

```
INCLUDE PTU /usr/foo/test/file3.ptu
```

INITIALIZATION ... END INITIALIZATION

C Test Script Language

Syntax

```
INITIALIZATION
```

```
END INITIALIZATION
```

Description

The **INITIALIZATION** and **END INITIALIZATION** statements let you provide native code that is integrated into the generation as the first native instructions of the test program (first lines of main).

In some environments, such as if you are using a different target machine, this provides a way to initialize the target.

Associated Rules

An **INITIALIZATION** block must appear after the **BEGIN** instruction or between two **SERVICE** blocks.

This block can only contain native code. This code must begin with '#' or '@'.

There is no limit to the number of **INITIALIZATION** blocks. During the run process, they are concatenated in the order in which they appeared in the test script.

Related Topics

[TERMINATION on page 738](#)

NEXT_TEST

C Test Script Language

Syntax

NEXT_TEST [**LOOP** <nb>]

where:

- <nb> is an integer expression strictly greater than 1.

Description

The **NEXT_TEST** instruction allows you to repeat a series of test contained within a previously defined **TEST** block.

It contains one more **ELEMENT** block. It does not contain the **FAMILY** instruction.

For this new test, a number of iterations can be specified by the keyword **LOOP**.

The **NEXT_TEST** instructions can only appear in a **TEST ... END TEST** block.

The main difference between a **NEXT_TEST** block and an **ELEMENT** block is when you use an **INIT IN** statement within a test block:

- If the **INIT IN** is in a **TEST** block, there will be a loop over the entire **TEST** block, without consideration of the **ELEMENT** blocks that it might contain.

If the **INIT IN** is inside a **NEXT_TEST** block however, the loop will not affect the **ELEMENT** blocks within other **TEST** blocks

Example

```
SERVICE COMPUTE_HISTO
```

```
# int x1, x2, y1, y2 ;
```

```
# int status ;
```

```
# T_HISTO histo;
```

```
TEST 1
```

```
FAMILY nominal
```

```
ELEMENT
```

```
...
```

```
END ELEMENT
```

```
NEXT_TEST LOOP 2
```

```
ELEMENT
```

Related Topics

[TEST on page 738](#) | [ELEMENT ... END ELEMENT on page 723](#)

SERVICE ... END SERVICE

Applies to C Test Script Language

Syntax

```
SERVICE <service_name>
```

```
END SERVICE
```

Description

The **SERVICE** instruction starts a **SERVICE** block. This block contains the description of all the tests relating to a given service of the module to be tested.

The <service_name> parameter flags the tested service in the test report, and is therefore usually the name of this service (although this is not obligatory).

The **END SERVICE** instruction indicates the end of the service block.

Associated Rules

The **SERVICE** instruction must appear after the **BEGIN** instruction.

The <service_name> parameter can be any identifier. It is obligatory. It must be unique in the PTU.

Example

```
BEGIN  
  
SERVICE COMPUTE_HISTO  
  
# int x1, x2, y1, y2 ;  
  
# int status ;  
  
# T_HISTO histo;  
  
TEST 1  
  
FAMILY nominal  
  
SIMUL ... ELSE_SIMUL ... END SIMUL
```

C Test Script Language

Purpose

The **SIMUL**, **ELSE_SIMUL**, and **END SIMUL** instructions allow conditional generation of test script program.

Syntax

SIMUL

ELSE_SIMUL

END SIMUL

Description

Code enclosed within a **SIMUL** block is conditionally generated depending on the status of the **Simulation** setting in HCL OneTest™ Embedded.

The **SIMUL** instruction starts the conditional generation block.

The **END SIMUL** instruction terminates this block.

The **ELSE_SIMUL** instruction separates this block into two parts, one being included when the other is not, and vice versa.

This block of instructions can appear anywhere in the test program and can contain both scripting instructions or native code.

The **SIMUL** and **END SIMUL** instructions must appear as a pair. One cannot be used without the other.

The **ELSE_SIMUL** instruction is optional.

When using HCL OneTest™ Embedded in the command line interface, use the **-nosimulation** option to deactivate the simulation setting in the [C Test Script Compiler on page 1121](#).

When using the HCL OneTest™ Embedded user interface, select or clear the **Simulation** option in the **Component Testing for C** tab of the **Configuration Settings** dialog box.

The generating rules are as follows:

1. If **Simulation** is enabled => the first part of the **SIMUL** block is included.
2. If **Simulation** is disabled => the second part of the block (**ELSE_SIMUL**) is included if it exists. If there is no **ELSE_SIMUL** statement, then the **SIMUL** block is ignored.

Example

SIMUL

#x = 0;

ELSE_SIMUL

#x = (type_x *) malloc (sizeof(*x));

END SIMUL

...

SIMUL

VAR x , INIT = 0 , EV = 1

VAR p , INIT = NIL , EV = NONIL

ELSE_SIMUL

VAR x , INIT = 0 , EV = 0

VAR p , INIT = NIL , EV = NIL

END SIMUL

STUB

C Test Script Language

Purpose

The STUB instruction for C describes all calls to a simulated function in a test script.

Syntax

```
STUB [<stub_name>.] <function> [<call_range> =>] ([<param_val> {, <param_val> }]) [<return_val>] {, [<call_range> =>]
([<param_val> {, <param_val> }]) [<return_val>] }
```

Description

The following is described for every parameter of this function and for every expected call:

1. For **_in** parameters, the values passed to the function; these values will be stored and then tested during execution,

For **_out** parameters and, where appropriate, the return value, the values returned by the function; these values will be stored in order to be returned during execution,

For **_inout** parameters, both the previous two values are required,

For **_no** parameters, any parameter is ignored.

The optional *<call_range>* describes one or several successive calls as follows:

<call_num> =>

<call_num> .. *<call_num>* =>

others =>

where *<call_num>* is the number of the stub call. The keyword **others** specifies the behavior of any further calls that have not been described. A *<call_num>* value of 0 means that no calls are expected to the stub. For example, the following line specifies that test will pass if there are 0 or more calls to the stub:

```
STUB close_file others=>(5)1
```

Moreover, you can use **others** to specify that the calls are optional. Combining others with a list of call numbers, enables you to check the minimum number of calls. For example, the following line specifies that test will pass if there are at least 4 calls to the stub:

```
STUB close_file 1=>(3)1, 2..4=>(4)1, others=>(5)1
```

If *<call_range>* is not specified, then the next call number is assumed. For example, the following lines specify that the test will pass if there are 2 calls to the stub:

```
STUB open_file ("file1")3
```

```
STUB open_file ("file2")4
```

<function> is the name of the simulated function. It is obligatory. You must previously have described this function in a **DEFINE STUB ... END DEFINE STUB** block. You can specify in which stub (*<stub_name>*) the declaration was made.

<param_val> is an expression describing the test values for **_in** parameters and the returned values for **_out** parameters. For **_inout** parameters, *<param_val>* is expressed in the following way:

(<in_param_val> , <out_param_val>)

<return_val> is an expression describing the value returned by the function if its type is not void. Otherwise, no value is provided.

You must give values for every **_in**, **_out** and **_inout** parameter; otherwise, a warning message is generated. You must not give a value for any **_no** parameters; otherwise, a warning message is generated.

<param_val> and <return_val> are expressions that can contain:

- Numeric (integer or floating-point), character, or character string literal values. Strings can be delimited by single or double inverted commas
- Constants which can be numeric, characters, or character strings
- Constants defined in the test script
- Variables belonging to the test program or the module to be tested
- C functions
- The keyword **NIL** to designate a null pointer
- Pseudo-variables **I**, **I1**, **I2** ..., **J**, **J1**, **J2** ..., where **I** *n* is the current index of the *n*th dimension of the parameter and **J** *m* the current number of the subtest generated by the test scenario's *m*th **INIT IN**, **INIT FROM** or **LOOP**; the **I** and **I1** variables are therefore equivalent as are **J** and **J1**; the subtest numbers begin at **1** and are incremented by **1** at each iteration
- An expression with one or more of the above elements combined using any of the C operators (**+**, **-**, *****, **/**, **%**, **&**, **|**, **^**, **&&**, **||**, **<<**, **>>**) and casting, with all required levels of parentheses, and conforming to C rules of syntax and semantics, the **+** operator being allowed to concatenate character string variables
- For arrays and structures, a list of expressions between braces (**{** and **}**) or brackets (**[** and **]**) with, where appropriate:
 - For an array element, part of an array or a structure field, its index, interval or name followed by **=>** and by the value of the array element, common to all elements of the array portion or structure field
 - The keyword **others** (written in lower case) followed by **=>** and the default value of any array elements or structure fields not yet mentioned.

You must describe at least one call in the **STUB** instruction. There can be several descriptions, separated by commas (,). **STUB** instructions can appear in **ELEMENT** or **ENVIRONMENT** blocks.

Type Modifier '@' Syntax

In a STUB definition you can use a **@** before a type modifier to indicate that this type modifier should not be used when generating variable that test the correct execution of STUBs.

Without the @ symbol, the variables are of *const int* type and therefore are not modified by the test harness.

Example

```
STUB read_file (3,"line 1")1, (3,"line 2")1, (3,"")0
```

```
STUB write_file (4,"line 1")1, (4,"line 2")1
```

```
STUB close_file 1=>(3)1, 2..4=>(4)1, others=>(5)1
```

TERMINATION ... END TERMINATION

C Test Script Language

Syntax

```
TERMINATION
```

```
END TERMINATION
```

Description

The **TERMINATION** and **END TERMINATION** instructions delimit a block of native code that is integrated into the generation process as the last instructions to be executed (last lines of main).

In certain environments (for example, a different target machine), these instructions terminate execution on the target machine.

Associated Rules

A **TERMINATION/END TERMINATION** block must appear after the **BEGIN** instruction and outside any **SERVICE** block.

This block can only contain native code. This code must begin with '#' or '@'.

There is no limit to the number of **TERMINATION** blocks. They are concatenated at generation.

Related Topics

[INITIALIZATION on page 731](#)

TEST ... END TEST

C Test Script Language

Syntax

```
TEST <test_name> [ LOOP <nb>]
```

```
END TEST
```

Description

The **TEST** instruction starts a **TEST** block. This block describes the test case for a service. It contains one more **ELEMENT** blocks specifying the test.

In the test report, the `<test_name>` parameter flags the test within the **SERVICE** block. Tests are usually given numbers in ascending order.

A number of iterations can be specified for each test with the optional **LOOP** keyword.

The **TEST LOOP** statement can generate graph metric results in a `.rtx` file. To do this, you must set the environment variable **ATURTX** to `True` . The produced `.rtx` graph can be viewed in the HCL OneTest™ Embedded Graphic Viewer.

The END TEST instruction marks the end of the **TEST** block.

Associated Rules

The TEST and END TEST instructions can only appear in a SERVICE block.

`<test_name>` is obligatory. If it is absent, the Test Script Compiler generates an error message.

`<nb>` is an integer expression strictly greater than 1.

Example

```
SERVICE COMPUTE_HISTO
```

```
# int x1, x2, y1, y2 ;
```

```
# int status ;
```

```
# T_HISTO histo;
```

```
TEST 1
```

```
FAMILY nominal
```

```
ELEMENT
```

Related Topics

[ELEMENT](#) on page 723 | [SERVICE](#) on page 733

USE

C Test Script Language

Purpose

The **USE** instruction activates a test environment that is defined using the **ENVIRONMENT** instruction.

Syntax

USE <name> [(<expression> { , <expression> })]

Description

The position of the **USE** instruction determines which tests are affected by the environment used:

1. If **USE** occurs outside a **SERVICE** block, the instructions contained in this environment are applied to all subsequent **ELEMENT** blocks.

If **USE** occurs within a **SERVICE** block and outside a **TEST** block, the instructions contained in this environment are applied to all subsequent **ELEMENT** blocks of this **SERVICE** block.

If **USE** occurs within a **TEST** block and outside an **ELEMENT** block, the instructions contained in this environment are applied to all subsequent **ELEMENT** blocks of this **TEST** block.

If **USE** occurs within an **ELEMENT** block, the instructions contained in this environment will only be applied to this block.

Because the **USE** instruction can appear at these four different levels, four priority levels are created from "outside a **SERVICE** block" (the lowest priority) to "inside an **ELEMENT** block" (the highest priority).

Within the same priority level, the last **USE** instruction is the one with the highest priority.

Testing is completed according to these priority rules, and on the basis that variables tested several times are included in the environment with the highest priority.

This is also true for every element of arrays described in extended mode.

If the environment it references takes parameters, the **USE** instruction must initialize these parameters using C expressions.

Associated Rules

The **USE** instruction can appear after **BEGIN** and outside an **ENVIRONMENT** block, after the definition of the environment it references.

<name> is the name of an environment declared by the **ENVIRONMENT** instruction.

<expression> must be an expression that conforms to C syntax and semantics.

Example

```
ENVIRONMENT compute_histo
```

```
VAR x1, init = 0, ev = init
```

```
VAR x2, init = SIZE_IMAGE-1, ev = init
```

```

VAR y1, init = 0, ev = init
VAR y2, init = SIZE_IMAGE-1, ev = init
ARRAY histo, init = 0, ev = 0
VAR status, init ==, ev = 0
END ENVIRONMENT
USE compute_histo

```

Related Topics

[ENVIRONMENT](#) on page 725

VAR, ARRAY and STR

Purpose

The **VAR**, **ARRAY**, and **STR** instructions declare the test of a simple variable, a variable array or a variable structure. It applies to C Test Script Language.

Syntax

```

VAR <variable>, [<format>], <initialization>, <expected_value>
ARRAY <variable>, [<format>], <initialization>, <expected_value>
STR <variable>, [<format>], <initialization>, <expected_value>

```

where:

- <variable> is a [variable on page 743](#)
- <format> optionally defines the format of the variable.
- <initialization> is a Component Testing [initialization on page 744](#) parameter for C
- <expected value> is a Component Testing [expected_value on page 747](#) parameter for C

Description

Use the **VAR**, **ARRAY**, and **STR** instructions to declare a variable test. During test execution, if the value of the variable is out of the bounds specified in the <expected_value> expression, the test is *Failed*.

The usage of **VAR**, **ARRAY** or **STR** does not change the behavior of the test, but each keyword specify how the result is displayed in the test report. Use:

- **VAR**: for simple variables.
- **ARRAY**: for variable arrays.
- **STR**: for variable structures.

If you use a **VAR** statement to test an array or structure, the report lists each element of the array or structure.

If you use a **STR** in an **ENVIRONMENT** block, then all elements of the structure are shown in the report, regardless of whether the test passes or fails. If the **STR** is in an **ELEMENT** block, then only the failed elements of the structure are displayed.

The **VAR**, **ARRAY**, and **STR** instructions must be located in an **ELEMENT** or an **ENVIRONMENT** block.



Note:

- The initialization expressions must not use '--' or '++' operators, as these may be interpreted as comments in some environments.

The optional *<format>* parameter allows you to modify the type of the tested element. This parameter uses the same syntax as a the **FORMAT** statement. See [FORMAT on page 727](#) for more information.

In addition the following formats are available in a **VAR**, **ARRAY**, or **STR** statement only:

- **pointer**: Initialize and test as a pointer, with pointer cast (void*).
- **string_ptr**: Initialize as a pointer and test as a string.
- **string**: Initialize and test as a string.

Example

```
#char *ar;
```

```
#char *ar1;
```

```
#char ar2[50];
```

```
#unsigned char t;
```

```
VAR t, #h, init=200, ev =init -- display as hexadecimal
```

```
VAR ar1,string_ptr, init ="defg", ev =INIT -- init as a pointer and test as a string
```

```
VAR ar2,string, init ="defgh", ev =INIT -- init and test as a string
```

```
VAR ar,pointer#b, init =0x12345678, ev =NONIL -- init and test as a pointer and display in binary
```

Related Topics

[Initialization Expressions for C on page 744](#) | [Expected Value Expression for C on page 747](#) | [C Variables on page 743](#) | [FORMAT on page 727](#)

VAR, ARRAY and STR variable Parameter

VAR, ARRAY and STR <variable> Parameter

C Test Script Language

Description

In the current documentation, the Component Testing <variable> parameter for C is a conventional notation name for a C variable under test. The syntax of the <variable> parameter allows you to specify the upper and lower boundaries of the range of the test for each dimension of the array:

[<lower> .. <upper>]

where:

<lower> is lower boundary for acceptable values of <variable>

<upper> is the upper boundary for acceptable values of <variable>

Associated Rules

<variable> can be a simple variable (integer, floating-point number, character, pointer or character string), an element of an array or structure, part of an array, an entire array, or a complete structure.

If no test boundaries have been specified for a variable array, all array elements are tested. Similarly, if one of the fields of a variable structure is an array, all elements of this field are tested.

The variable must have been declared in advance.

Example

VAR x, ...

VAR y[4], ...

VAR z.field, ...

VAR p->value, ...

ARRAY y[0..100], ...

ARRAY y, ...

STR z, ...

STR *p, ...

Related Topics

[VAR, ARRAY and STR on page 741](#) | [Initialization Expressions on page 744](#) | [Expected Value Expressions on page 747](#)

VAR, ARRAY and STR <initialization> Parameter

In this documentation, the Component Testing <initialization> parameters for **C Test Script Language** specify the initial value of the variable.

Syntax

```
INIT = <exp>
INIT IN { <exp>, <exp>, ... }
INIT ( <variable> ) WITH { <exp>, <exp>, ... }
INIT FROM <exp> TO <exp> [STEP <exp> | NB_TIMES <nb> | NB_RANDOM <nb>[+ BOUNDS]]
INIT FROM <exp> TO <exp> [STEP <exp> | NB_VALUE <nb> | NB_RANDOM <nb>[+ BOUNDS]]
INIT ==
```

where:

- <exp> is an expression as described below.
- <nb> is an integer constant that is either literal or derived from an expression containing native constants.
- <variable> is a C variable.

Description

The <initialization> expressions are used to assign an initial value to a variable. The initial value is displayed in the Component Testing report for C.

The **INIT** value is calculated during the pre-processing phase, not dynamically during test execution.

Initializations can be expressed in the following ways:

- **INIT = <exp>** initializes a variable before the test with the value <expression>.
- **INIT IN { <exp>, <exp>, ... }** declares a list of initial values. This is a condensed form of writing that enables several tests to be contained within a single instruction.
- **INIT (<variable>) WITH { <exp>, <exp>, ... }** declares a list of initial values that is assigned in correlation with those of the variable initialized by an **INIT IN** instruction. There must be the same number of initial values.
- **INIT FROM <lower> TO <upper>** allows the initial value of a numeric variable (integer or floating-point) to vary between lower and upper boundary limits:
- **STEP**: the value varies by successive steps.
- **NB_TIMES <nb>** or **NB_VALUE <nb>**: The value varies by a number <nb> of values that are equidistant between the two boundaries, where <nb> >= 2 (**NB_TIMES** and **NB_VALUE** are equivalent keywords). This option requires that the target platform supports floating point numbers.

- **NB_RANDOM** <nb>: The value varies by generating random values between the two boundaries, including, when appropriate, the boundaries, where <nb> >= 1.
- **BOUNDS**: When you enter the '+ BOUNDS' instruction after 'NB_RANDOM nb', two numerical values are added to the nb (number) values.



Important:

- The **INIT IN** and **INIT** (<variable>.) **WITH** expressions cannot be used for **ARRAYS** that were initialized in extended mode or for structures.
- The **INIT FROM** expression can only be used for numeric variables.
- The **STEP** syntax cannot be used when the same variable is tested by another **VAR**, **ARRAY** or **STR** statement.
- The **NB_TIMES**, **NB_VALUE**, and **NB_RANDOM** keywords require that the target platform supports floating point numbers.
- The **NB_TIMES**, **NB_VALUE**, and **NB_RANDOM** keywords
- **INIT ==** allows the variable to be left uninitialized. You can thus control the values of variables that are dynamically created by the service under test. The initial value is displayed in the test report as a question mark (?).
- An initialization expression can still be used (**INIT == <expression>**) to include of expected value expression when using the **INIT** pseudo-variable is used. See [Expected_Value Expressions on page 747](#).
- The following syntaxes cannot be used in an **ARRAY** instruction:
 - **INIT FROM** <exp> **TO** <exp> **STEP** <exp>>,
 - **INIT FROM** <exp> **TO** <exp> **NB_TIMES** <nb>>,
 - **INIT FROM** <exp> **TO** <exp> **NB_VALUE** <nb>>,
 - **INIT FROM** <exp> **TO** <exp>**NB_RANDOM** <nb>>,
 - **INIT FROM** <exp> **TO** <exp>**NB_RANDOM** <nb>[+ **BOUNDS**]

Expressions

The initialization expressions <exp> can be among any of the following values:

- Numeric (integer or floating-point), character, or character string literal values. Strings can be delimited by single or double quotes.
- Native constants, which can be numeric, characters, or character strings.
- Variables belonging to the test program or the module to be tested.
- C or Ada functions.
- The keyword **NIL** to designate a null pointer.
- Pseudo-variables **I**, **I1**, **I2** ..., **J**, **J1**, **J2** ..., where **I** *n* is the current index of the *n*th dimension of the parameter and **J** *m* the current number of the sub-test generated by the test scenario's *m*th **INIT IN**, **INIT FROM** or **LOOP**; the **I** and **I1** variables are therefore equivalent as are **J** and **J1**; the subtest numbers begin at **1** and are

- incremented by **1** at each iteration. These pseudo-variables **I**, **I n**, **J**, and **J n** must not be declared as typedefs or variables in the source code.
- A C expression with one or more of the above elements combined using any operators and casting, with all required levels of parentheses, the + operator being allowed to concatenate character string variables.
- For arrays and structures, any of the above-mentioned expressions between braces (‘{’) for C or brackets (‘[]’) for Ada, including when appropriate:
 - For an array element, part of an array or a structure field, its index, interval or name followed by ‘=>’ and by the value of the array element, common to all elements of the array portion or structure field.
 - For structures you can test some fields only, by using the following syntax:
 - For C: { <value>,,<value> }
 - For all languages: [<fieldname>=><value>, <fieldname>=><value>]
- The keyword **others** (written in lower case) followed by ‘=>’ and the default value of any array elements or structure fields not yet mentioned.
- For **INIT IN** and **INIT WITH** only, a list of values delimited by braces (‘{’) for C composed of any of the previously defined expressions.

Additional Rules

Any integers contained in an expression must be written either in accordance with native lexical rules, or under the form:

- <hex_integer> **H** for hexadecimal values. In this case, the integer must be preceded by **0** if it begins with a letter.
- <binary_integer> **B** for binary values.



Note: Because of the way hexadecimal values are handled, the value range should not exceed half of the maximum range when the initialization is expressed in hexadecimal.

- The number of values inside an **INIT IN** parameter is limited to 100 elements in a single **VAR** statement.
- The number of **INIT IN** parameters per **TEST LOOP** block is limited to 7.
- The number of **INIT IN** parameters per **TEST** block is limited to 8.
- In Component Testing for C, if variables are used in the expression, then the test evaluates the the **INIT** value with variable values from after the execution.
- All Euclidean divisions performed by the Test Script Compiler round to the inferior integer. Therefore, writing **-a/b** returns a different result than **-(a/b)**, as in the following examples:

```
- (9/2) returns -4
```

```
-9/2 returns -5
```

Examples

```
VAR x, INIT = pi/4-1, ...
VAR y[4], INIT IN { 0, 1, 2, 3 }, ...
VAR y[5], INIT(y[4]) WITH { 10, 11, 12, 13 }, ...
VAR z.field, INIT FROM 0 TO 100 NB_RANDOM 3, ...
VAR z.field, INIT FROM 0 TO 100 NB_RANDOM 3 + BOUNDS, ..
VAR p->value, INIT ==, ...
ARRAY y[0..100], INIT = sin(I), ...
ARRAY y, INIT = {50=>10,others=>0}, ...
STR z, INIT = {0, "", NIL}, ...
STR *p, INIT = {value=>4.9, valid=>1}, ...
```

In the following example, the C test Script Compiler generates code that tests *x* against *a* then *b* after the execution of the code under test:

```
VAR y, init in (1,2), ev = init
VAR a, init(y) with ( 10, 20 ), ev = 50
VAR b, init(y) with ( 30, 40 ), ev = 70
VAR x, init(y) with (a, b), ev = init
#a := 50;
#b := 70;
```

Additional Ex

```
VAR z.field, INIT FROM 0 TO 100 NB_RANDOM 3 + BOUNDS, ...
```

Related Topics

[<expression> parameter on page 731](#) | [Expected_Value Expressions on page 747](#) | [<variable> parameter \(C\) on page 743](#) | [VAR, ARRAY and STR on page 741](#)

VAR, ARRAY and STR expected Parameter

Purpose

In this documentation, the Component Testing *<expected value>* parameters for C Test Script Language specify the expected value of a variable.

```
EV = <exp>
EV = <exp> , DELTA = <delta>
MIN = <exp>, MAX = <exp>
EV IN { <exp>, <exp>, ... }
EV ( <variable> ) IN { <exp>, <exp>, ... }
EV ==
```

Where *<exp>* can be any of the expressions of the [Initialization Parameters on page 744](#), and additionally the following expressions:

- *<delta>* is the acceptable tolerance of the expected value and can be expressed.
- *<variable>* is a C variable.

Description

The *<expected value>* expressions are used to specify a test criteria by comparison with the value of a variable. The test is considered as Passed when the actual value matches the *<expected value>* expression.

The **EV** value is calculated during the preprocessing phase, and not dynamically during test execution.

An acceptable tolerance *<delta>* can be expressed:

- As an absolute value, by a numerical expression in the form described above.
- As a percentage of the expected value. Tolerance is then written as follows: *<exp> %*.

Expected values can be expressed in the following ways:

- **EV** = *<exp>* specifies the expected value of the variable when it is known in advance. The value of variable is considered correct if it is equal to *<exp>*.
- **EV** = *<exp>*, **DELTA** = *<tolerance>* allows a tolerance for the expected value. The value of variable is considered correct if it lies between *<exp> - <tolerance>* and *<exp> + <tolerance>*.
- **MIN** = *<exp>* and **MAX** = *<exp>* specify an interval delimited by an upper and lower limit. The value of the variable is considered correct if it lies between the two expressions. Characters and character strings are processed in dictionary order.
- **EV IN** { *<exp>*, *<exp>*, ... } specifies the values expected successively, in accordance with the initial values, for a variable that is declared in **INIT IN**. It is therefore essential that the two lists have an identical number of values.
- **EV (<variable>) IN** is identical to **EV IN**, but the expected value is a function of other variable that has previously been declared in **INIT IN**. As for **EV IN**, the two lists must have an identical number of values.
- **EV ==** allows the value of *<variable>* not to be checked at the end of the test. Instead, this value is read and displayed. The value of *<variable>* is always considered correct.

Expressions

The initialization expressions *<exp>* can be among any of the following values:

- Numeric (integer or floating-point), character, or character string literal values. Strings can be delimited by single or double quotes.
- Native constants, which can be numeric, characters, or character strings.
- Variables belonging to the test program or the module to be tested.
- C or Ada functions.
- The keyword **NIL** to designate a null pointer.

- The keyword **NONIL**, which tests if a pointer is non-null.
- Pseudo-variables **I**, **I1**, **I2** ..., **J**, **J1**, **J2** ..., where **I** *n* is the current index of the *n*th dimension of the parameter and **J** *m* the current number of the subtest generated by the test scenario's *m*th **INIT IN**, **INIT FROM** or **LOOP**; the **I** and **I1** variables are therefore equivalent as are **J** and **J1**; the sub-test numbers begin at **1** and are incremented by **1** at each iteration.
- A C or Ada expression with one or more of the above elements combined using any operators and casting, with all required levels of parentheses, the + operator being allowed to concatenate character string variables.
- For arrays and structures, any of the above-mentioned expressions between braces (**{}**) for C, including when appropriate:
 - For an array element, part of an array or a structure field, its index, interval or name followed by **'=>'** and by the value of the array element, common to all elements of the array portion or structure field.
 - For structures you can test some fields only, by using the following syntax:


```
{ <value>, , <value> }
```
- The keyword **others**(written in lower case) followed by **'=>'** and the default value of any array elements or structure fields not yet mentioned.
- The pseudo-variable **INIT**, which copies the initialization expression. You cannot use the pseudo-variable **INIT** inside an array or structure. The keyword **INIT** applies to the entire expression.



Note: The following syntaxes cannot be used in an **ARRAY** instruction:

```
EV IN ( <exp>, <exp>, ... )
EV ( <variable> ) IN ( <exp>, <exp>, ... )
```

Additional Rules

- **EV** with **DELTA** is only allowed for numeric variables. The **STR** statement does not support **DELTA**.
- **MIN** = **<exp>** and **MAX** = **<exp>** are only allowed for alphanumeric variables that use lexicographical order for characters and character strings.
- **MIN** = **<exp>** and **MAX** = **<exp>** are not allowed for pointers.
- Only **EV =** and **EV ==** are allowed for structured variables.
- In some cases, in order to avoid generated code compilation warnings, the word **CAST** must be inserted before the **NIL** or **NONIL** keywords.
- All Euclidian divisions performed by the Test Script Compiler round to the inferior integer. Therefore, writing **-a/b** returns a different result than **-(a/b)**, as in the following examples:

```
-(9/2) returns -4
-9/2 returns -5
```

- Not a number and infinite float values: Component testing can handle *not a number* and infinite float values as **MIN** and **MAX** parameters. Such values must be assigned through a variable. The test produces a verdict depending on the nature of the values.

The following table describes the verdict to be expected for each combination of **MIN** and **MAX** values. For example, if the **MIN** is a real float value and the **MAX** is *+infinite*, then the test will fail if the actual return value is not a number or *+infinite* and will pass if the value is *+infinite* or greater than **MIN**.

Expected values		Actual return values			
MIN	MAX	Not a number	-infinite	float value	+infinite
Not a number	Not a number	Pass	Fail	Fail	Fail
Not a number	-infinite	Pass	Pass	Fail	Fail
Not a number	float value	Pass	Fail	x==MAX	Fail
Not a number	+infinite	Pass	Fail	Fail	Pass
-infinite	Not a number	Pass	Pass	Fail	Fail
-infinite	-infinite	Fail	Pass	Fail	Fail
-infinite	float value	Fail	Pass	x<=MAX	Fail
-infinite	+infinite	Fail	Pass	Pass	Pass
float value	Not a number	Pass	Fail	x==MIN	Fail
float value	-infinite	Fail	Fail	Fail	Fail
float value	float value	Fail	Fail	MIN<=x<=MAX	Fail
float value	+infinite	Fail	Fail	x>=MIN	Pass
+infinite	Not a number	Pass	Fail	Fail	Pass
+infinite	-infinite	Fail	Fail	Fail	Fail
+infinite	float value	Fail	Fail	Fail	Fail
+infinite	+infinite	Fail	Fail	Fail	Pass

Example

```
VAR x, ..., EV = pi/4-1
VAR y[4], ..., EV IN { 0, 1, 2, 3 }
VAR y[5], ..., EV(y[4]) IN { 10, 11, 12, 13 }
VAR z.field, ..., MIN = 0, MAX = 100
VAR p->value, ..., EV ==
ARRAY y[0..100], ..., EV = cos(I)
ARRAY y, ..., EV = {50=>10,others=>0}
STR z, ..., EV = {0, "", NIL}
STR *p, ..., EV = {value=>4.9, valid=>1}
```

Related Topics

[Initialization Expressions on page 744](#) | [VAR, ARRAY and STR on page 741](#) | [C Variables on page 743](#)

Requirement

Purpose

The **Requirement** instruction allows the testers to link a test or a set of tests to one or a set of requirements. **Requirement** is optional.

Syntax

```
REQUIREMENT <requirement_name> {, [<attribute_name> =|:] <attribute_value>}
```

Argument

<requirement_name> is a mandatory identifier indicating the name of the requirement.

<attribute_name> is the name of one attribute of the requirement. It is an identifier.

<attribute_value> is the value of the attribute. The syntax may be: \$<identifier>. In this case, the attribute value is substituted with the content of an environment variable whose name is \$<identifier>.

Description

The **REQUIREMENT** instruction appears within **TEST** blocks, where it defines the requirements for this test or within **SERVICE** blocks where it defines the requirements for the tests including in this service or before the first **SERVICE** block where it defines the requirements for the all the tests in the file.

Requirements are cumulative between test and service.

rod2req is a binary that generates an XML file analyzing the rod files and describing the tracability matrix between tests and requirements with pass/failed status.

Example

```
TEST1
FAMILY nominal
REQUIREMENT req1, req2
COMMENT histogram computation on a black image
ELEMENT
```

Component Testing for C++

C++ test driver script (.otd)

Component Testing for C++ uses its own simple language for test driver scripting.

This section describes each keyword of the C++ test driver script language, including:

- Syntax
- Functionality and rules governing its usage
- Examples of use

Notation conventions

Throughout this section, command notation and argument parameters use the following standard convention:

Notation	Example	Meaning
BOLD	BEGIN	Language keyword
<i><italic></i>	<i><filename></i>	Symbolic variables
[]	[<i><option></i>]	Optional items
{ }	{ <i><filenames></i> }	Series of values
[{ }]	[{ <i><filenames></i> }]	Optional series of variables
	on off	OR operator

C++ test driver script keywords are case insensitive. This means that **STUB**, **stub**, and **Stub** are interpreted the same way.

For conventional purposes however, this document uses upper-case notation for the C++ test driver script keywords in order to differentiate from native source code.

Split statements

C++ test driver script statements may be split over several lines in an **.otd** test script. Continued lines must start with the ampersand ('&') symbol to be recognized as a continuation of the previous line. No tabs or spaces should precede the ampersand.

Statements must be shorter than 2048 characters, although this limit may be lower on some platforms.

Language identifiers

A C++ test script identifier is a text string used as a label, such as the name of a **TEST** or a **STUB** in an **.otd** test script.

Identifiers are made of an unlimited sequence of the following characters:

- a-z
- A-Z

- 0-9
- _ (underscore)

Spaces are not valid identifier characters.

Note that identifiers starting with a numeric character are allowed. The following statement, for example, is syntactically correct:

```
TEST CASE 1
```

```
{
..
}
```

C++ test driver script identifiers are case sensitive. This means that **LABEL**, **label**, and **Label** are three different identifiers.

Related Topics

[C on page 753](#) [++ test driver script structure on page 753](#) | [C++ test driver script keywords on page 755](#) | [C++ contract check scripts \(.otc\) on page 786](#)

C++ test driver script structure

A Component Testing for C++ test driver script (**.otd** script) describes a test driver. Its purpose is to stimulate the tested classes by creating objects and calling their methods. It provides different ways to check that the objects behavior is the one that was expected.

When executed, the script is translated into a C++ source by Component Testing for C++. Furthermore, it instruments the source code under test whenever the STUB, CHECK STUB, or CHECK METHOD statements are used.

Order is meaningful for INCLUDE and native statements. RUN may appear only once in a C++ Test Driver script. Other entities are not ordered: for instance, a TEST CLASS can forward-reference a STUB.

Note A C++ Test Driver script is made both of statements and instructions. Instructions are ordered: their relative position is meaningful. Statements have no order: they have a declarative nature.

Basic structure

A typical Component Testing **.otd** test script structure could look like this:

```
TEST CLASS TestAnyPhilosopher (Philosopher_type)
{
```

```
TEST CLASS TestNominal (Philosopher_type)
```

```
{
```

```
PROLOGUE
```

```
{
```

```
// Actions to be performed when entering this test class.
```

```
}
```

```
TEST CASE AssignForks
```

```
{
```

```
// CHECK statements
```

```
}
```

```
EPILOGUE
```

```
{
```

```
// Actions to be performed when leaving this test class.
```

```
}
```

```
RUN
```

```
{
```

```
// Runs the test cases
```

```
}
```

```
}
```

```
RUN
```

```
{
```

```
// Runs the test class
```

```
}
```

All instructions in a test script have the following characteristics:

- All statements begin with a keyword.
- Statements are not case sensitive (except when C expressions are used).
- Statements start at the beginning of a line and end at the end of a line. You can, however, write an instruction over several lines using the ampersand (&) continuation character at the beginning of additional lines. In this case, the ampersand must be the very first character on that line; no spaces or tabs should precede it.
- Statements must be shorter than 2048 characters, although this limit may be lower on some platforms.

Structure statements

The following statements allow you to describe the structure of a test.

- **TEST CLASS:** Describes an object test class, which is one of the structuring entities of a C++ test driver script. Test classes can appear at the root-level of a C++ Test Driver Script and in test classes.
- **PROLOGUE:** Defines native code that is to be executed whenever the surrounding test class execution begins. This code is executed before any other of the test class' components.
- **TEST CASE:** Describes an object test case, which is the smallest testing structure in a hierarchical C++ test driver script. Test cases appear in test classes and test suites.
- **EPILOGUE:** Defines native code that is to be executed whenever the execution of the surrounding test class ends. This code is executed after other test class components.
- **RUN:** Defines the behavior of the surrounding test class.

Related Topics

[TEST CLASS on page 779](#) | [PROLOGUE on page 772](#) | [TEST CASE on page 777](#) | [EPILOGUE on page 766](#) | [RUN on page 774](#)

C++ test driver script keywords

Structure-related Keywords

- EPILOGUE
- INCLUDE
- TEST CLASS
- TEST SUITE
- TEST CASE
- PROLOGUE

- RUN
- PROPERTY
- PROC
- REQUIRE
- ENSURE

Verification Keywords

- CHECK
- CHECK EXCEPTION
- CHECK METHOD
- CHECK PROPERTY
- [CHECK STUB on page 761](#)

Error-handling Keyword

- [ON ERROR on page 767](#)

Stubbing Keyword

- STUB
- REQUIRE
- ENSURE

Instructions

- COMMENT
- PRINT
- CALL

CALL

C++ Test Script Language

Purpose

The CALL instruction calls a procedure defined with a PROC statement.

Syntax

```
CALL <procedure name> [( <actual parameter> [( , <actual parameter> ) ] )];
```

Arguments

<procedure_name> is a valid procedure name, defined within a test class or test suite, or in an inherited test class.

<actual_parameter> is an optional list of parameters that must conform to the expected procedure parameter list.

Description

A CALL instruction can be located within a TEST CASE or PROC block.

Example

```
TEST CLASS TestA {
  PROC InitArray (array, length)
  {
    #{
    for (int i = 0; i<length; i++)
    array[i].init ();
    }#
  }
  TEST CASE tc1 {
    #Array<int> ia (50);
    CALL InitArray (ia, 49);
  }
}
```

CHECK

C++ Test Script Language

Purpose

The CHECK instruction evaluates the Boolean value of a *native expression*.

Syntax

CHECK [*<comment>*] (*<native expression>*);

Arguments

<comment> is an optional string that appears in the test results.

<native expression> is a valid C++ expression, which may be converted into a Boolean.

Location

TEST CASE, PROC, STUB, [CHECK STUB on page 761](#)

Description

The CHECK instruction evaluates the *native expression*. If the result of the check is TRUE, the result of the corresponding test is Passed. Otherwise, an error is generated. The result of the error handling is specified with the [ON ERROR on page 767](#) keyword.

Example

```
TEST CLASS TestA {
TEST CASE tc1 {
CHECK (s.empty ());
}
RUN { tc1; }
}
```

CHECK EXCEPTION

C++ Test Script Language

Purpose

The CHECK EXCEPTION statement checks that an exception is raised within a block.

Syntax

CHECK EXCEPTION (*<native type>* [*<native parameter name>*]) (({ *<on exception item>* }) | ;)

Arguments

<native type> is the C++ type of the expected exception.

<native parameter name> is the optional name of the exception. It may also be used in *<on exception item>*.

<on exception item> may be a **COMMENT**, a **PRINT** or a native-code statement.

Description

The **CHECK EXCEPTION** statement specifies that the exception of type *<native type>* is expected to be raised in the current C++ Text Script Language block (test case or proc). If this exception is not raised in the block, an [error on page 520](#) is generated.

Only one **CHECK EXCEPTION** may occur per block. A **CHECK EXCEPTION** can be located in a **TEST CASE** or **PROC** block.

Example

```
TEST CASE TC1 {
CHECK EXCEPTION (DivideByZeroException) {
PRINT "ok";
}
#b = 1; c = 0;
#a = b / c;
}
```

Related Topics

[COMMENT on page 763](#) | [PRINT on page 770](#) | [TEST CASE on page 777](#) | [PROC on page 771](#) | [Error handling on page 520](#) | [Native code on page 783](#)

CHECK METHOD

C++ Test Script Language

Syntax

CHECK METHOD *<native routine signature>*;

Location

TEST CASE

Description

The **CHECK METHOD** checks that a routine (function or class member) is called during the execution of the surrounding test case. If the routine is not called, an error is generated. Error-handling behavior is specified with the [ON ERROR on page 767](#) keyword.

Note The use of **CHECK METHOD** requires instrumentation of the source code under test.

<*native routine signature*> refers to an existing routine.

- If it is a class or namespace member, its name must be qualified but the return type may be omitted.
- If it is a class member, and if it not overloaded, the parameters may be omitted.

If parameters are specified, their names may be omitted.

Only one **CHECK METHOD** referring to each routine may occur in each **TEST CASE**.

Note When the **CHECK METHOD** statement is used in an **.otd** test script, the related source files are always instrumented even if they are displayed as not instrumented in Project Explorer.

Example

```
TEST SUITE A {
TEST CASE 1 {
CHECK METHOD IntArray::ModifyCell (int);

#IntArray ia;

#InitializeArray (ia); // this function calls IntArray::ModifyCell

// ia is filled with random numbers

}

TEST CASE 2 {

CHECK METHOD IntArray::ModifyCell; // you can omit parameters

#IntArray ia;

// IntArray::ModifyCell was not called => error

}

}

RUN { A; }
```

CHECK PROPERTY

C++ Test Script Language

Syntax

```
CHECK PROPERTY [ "<comment>" ] <property name> [ ( <actual parameter> [ ( , <actual parameter> )* ] ) ]
```

Location

TEST CASE, PROC, STUB, [CHECK STUB on page 761](#)

Description

The **CHECK PROPERTY** instruction evaluates the property *<property name>*. If the result is TRUE, the test is passed. Otherwise, it generates an [error on page 520](#). Error handling behavior is specified with the [ON ERROR on page 767](#) keyword.

<comment> is an optional string that appears in test results.

<property name> is a valid property defined in the current test class, in a nesting test class or in an inherited test class.

Note: properties are defined with the keyword [PROPERTY](#).

Example

```
TEST CLASS TestA {
PROPERTY Empty { ( s.count() == 0) }
TEST CASE tc1 {
CHECK PROPERTY Empty;
}
RUN { tc1; }
}
```

CHECK STUB

C++ Test Script Language

Syntax

CHECK STUB *<stub name>* { *<stub item>* }

CHECK STUB *<stub name>*;

Location

TEST CASE

Description

The CHECK STUB instruction checks that a stub is called at least once during the TEST CASE execution.

If a block is provided, it specifies that *<stub item>* should be executed instead of the stub's "..." zone. If no block is provided, the execution of the stub's "..." does nothing.

If the stub is not called, an error is generated. Error handling behavior is specified with the [ON ERROR on page 767](#) keyword.

Note: The use of stubs requires instrumentation of the source code under test.

<stub name> is a valid stub identifier.

<stub item> may be one the following entities:

1. CHECK
COMMENT
PRINT
Native statement

Only one CHECK STUB may refer to the same STUB in a TEST CASE.

Note The CHECK STUB statement may be used before the corresponding STUB is defined.

Example

```
STUB ModifyCell : int IntArray::Modify (int Cell)
REQUIRE (Cell != 128)
{
#int Nb = random(10000);
... // this part is completed by the code of CHECK STUB
this.array[Cell] = Nb;
#return (Nb);
}
TEST SUITE A {
TEST CASE 1 {
CHECK STUB ModifyCell;
#IntArray ia;
```

```

#InitializeArray (ia); // this function calls IntArray::ModifyCell

// ia is filled with random numbers

}

TEST CASE 2 {

CHECK STUB ModifyCell {

#Nb = 0;

}

#IntArray ia;

#InitializeArray (ia); // this function calls IntArray::ModifyCell

// ia is filled with 0

}

}

RUN { A; }

```

COMMENT

C++ Test Script Language

Syntax

COMMENT <*one-line text*>

COMMENT { <*multiple-line text*> }

Location

TEST CASE, PROC, STUB, [CHECK STUB on page 761](#), PROLOGUE, EPILOGUE, [ON ERROR on page 767](#), CHECK EXCEPTION

Description

The **COMMENT** instruction allows the output of static comments to a trace file. These comments can be visualized through the UML/SD Viewer in the GUI.

Example

```

TEST CASE tc1 {

#s.push (i);

```

COMMENT An element was added to the stack.

```
CHECK (!s.full ());
```

```
}
```

ENSURE

C++ Test Script Language

Syntax

ENSURE <native expression>

Location

[WRAP](#), [STUB on page 776](#), [PROC on page 771](#)

Description

The ENSURE statement describes a method post-condition. It can be used in a **WRAP**, **STUB** or **PROC** block.

Note The information below pertains to the use of **ENSURE** within a **WRAP** block. For more information about using the **REQUIRE** and **ENSURE** statement within a **STUB** or **PROC** block, please refer to the [STUB on page 776](#) and [PROC on page 771](#).

<native expression> is a C++ Boolean expression (or an expression that can be converted into a Boolean), which can use:

- Any of the public or protected class members.
- The method parameters (with the names used in the signature or in the method definition).
- Any of the global variables declared in the file where the method is defined.
- The **_ATO_result**, **_ATO_old** and **_ATO_in_exception** variables (see below).

The following symbols cannot be used in the <native expression> parameter of the **ENSURE** statement:

- Local variables
- Macros

Variables

- **_ATO_result**: This variable contains the method return value, if any. Its type is that of the method return type. Its value may be undefined if no value is returned (because an exception was thrown, or a return without a value is executed, or the function implicitly returns). **_ATO_result** is not available when the option--**postcondition_before_return** is activated (see [Target Deployment Port options on page 798](#)).

- **_ATO_old**: This variable contains a copy of the object as it was before the method call. The **_ATO_old** object is generated by the class copy constructor. If the class copy constructor is explicitly defined, you should remember that **_ATO_old** is not a raw copy of the current object, but a copy as defined by the copy constructor. **_ATO_old** is not available in constructors.

For performance purposes, the **_ATO_old** variable is generated only if it is used in the **ENSURE** expression.

- **_ATO_in_exception**: This is a Boolean variable that is **TRUE** if the post-condition is executed because an exception has been thrown. This variable is available only if the Target Deployment Package is configured to support exceptions.

Evaluation

When `--postcondition_before_return` option is set in the Target Deployment Package **.opp** file, `<native expression>` is evaluated before the return expression. If the return expression evaluation causes side-effects, they are not taken into account at the time the post-condition is checked. This option is provided for compatibility with limited C++ compilers, and its use should be avoided as much as possible.

Otherwise, `<native expression>` is evaluated after any code of the method (local variables are already popped).

Warning: you can call methods in `<native expression>`, but you must make sure that these calls do not modify the object's state (that is, they do not write to any field). You can ensure this by calling `const` methods only. If you want the compiler to check this, use the **ATO_AC_STRICT_CHECKING** Target Deployment Port option.

Example

C++ source code example:

```
class Stack {
public:
int count;
Stack () : count(0) {}
void push (void *);
void *pop ();
};
```

C++ Contract Check Script code example:

```
CLASS Stack {
WRAP push
```

```
ENSURE (count == _ATO_old.count + 1)
}
```

EPILOGUE

C++ Test Script Language

Syntax

```
EPILOGUE { <epilogue item>* }
```

Location

TEST CLASS | TEST SUITE

Description

The EPILOGUE structure defines native code that is to be executed whenever the execution of the surrounding test class ends. This code is executed after other test class components.

An EPILOGUE statement may appear at most once in a test class. In an object-context, an EPILOGUE can be compared to a destructor.

<epilogue item> may be one of the following entities:

- COMMENT
- PRINT
- Native statement

Order is meaningful.

Example

```
TEST CLASS ATest
{
PROLOGUE {

#Stack *s = new Stack(20);

#s->fill ();

}

EPILOGUE {
```

```
#delete s;

}

TEST CASE tc1 {

CHECK (!s->full ());

}

RUN {

tc1;

}

}
```

INCLUDE

C++ Test Script Language

Syntax

```
INCLUDE " <file name> ";
```

Location

C++ Test Driver Script

Description

The INCLUDE statement opens the file <file name> and inserts its code into the C++ Test Driver Script.

A file cannot include itself, directly or indirectly.

An included file must not have a RUN statement at the script level. A RUN statement at script level is only allowed in the main test script.

Example

```
INCLUDE "test1.otd";
```

ON ERROR

C++ Test Script Language

Syntax

```
ON ERROR [ { <error item> } ] <error action>;
```

Location

C++ Test Driver Script, TEST CLASS, TEST SUITE, TEST CASE, PROC

Description

The ON ERROR statement defines the behavior of the test driver when an error occurs.

ON ERROR applies to the current scope level, and to nested scopes, unless another ON ERROR statement has been defined. The general rule is that the most nested ON ERROR statement is applied.

Note An error can be raised by any instruction of a TEST CASE or a PROC, and by native code from a PROLOGUE or EPILOGUE.

ON ERROR does not apply to stubs. There is always an implicit ON ERROR CONTINUE behavior in stubs

<error item> may be one of the following entities:

1. COMMENT

PRINT

Native statement

This block is executed when an error occurs.

<error action> is a keyword which defines the behavior of the test driver when an error occurs:

1. **CONTINUE**: The execution continues just as if no error occurred. If the error comes from an unexpected exception raised by native code, the execution continues after the native code, except for an error in a PROLOGUE block. Since it is the default behavior, this on-error action should only be specified to override another on-error action.

EXIT : The execution of the test driver stops at the error point.

BYPASS: The execution of the rest of the current test case or procedure is skipped.

BYPASS <test class name> | <test suite name> | <test case name> | <proc name> : The execution of the rest of the referred entity is skipped.

Example

```
ON ERROR CONTINUE;
```

```
TEST CLASS A {
```

```
ON ERROR EXIT;
```

```
TEST SUITE A1 {
```

```
ON ERROR BYPASS A;

TEST CASE A1a {

ON ERROR CONTINUE;

CHECK (false); // this leads to an error but execution continues

PRINT "ok"; // this instruction is executed

}

TEST CASE A1b {

CHECK (false); // this leads to an error

// execution resumes after TEST CLASS A

PRINT "ko"; // this instruction is never executed

}

}

TEST CASE A2 {

CHECK (false); // this leads to an error -- the test driver exits

PRINT "ko"; // this instruction is never executed

}

RUN { A1; A2; }

}

TEST CLASS B {

TEST CASE B1 {

ON ERROR BYPASS;

CHECK (false); // this leads to an error -- execution resumes after B1

PRINT "ko"; // this instruction is never executed

}

TEST CASE B2 {

CHECK (false); // this leads to an error but execution continues
```

```

PRINT "ok"; // this instruction is executed

}

RUN { B1; B2; B1; }

}

RUN { B; A; A.A2; }

```

In this example, the execution is: B1 (aborted), B2, B1 (aborted), A1a, A1b (A is aborted), A2 (exited).

PRINT

C++ Test Script Language

Syntax

```
PRINT (<expression> [ ( , <expression> ) ] );
```

Location

C++ Test Driver Script, TEST CASE, PROC, STUB, [CHECK STUB on page 761](#), PROLOGUE, EPILOGUE, [ON ERROR on page 767](#), CHECK EXCEPTION

Description

The PRINT instruction outputs dynamic comments to the traces file. These comments are displayed in the UML sequence diagrams and test reports produced by Component Testing for C++.

<expression> is a valid C++ expression whose type must be handled by the Target Deployment Port (handled types are scalar types, floating types, strings and pointers). <expression> is evaluated on execution.

A PRINT instruction may generate an error when the evaluation of one of the arguments raises an unexpected exception. Use the [CHECK EXCEPTION](#) statement to specify exceptions.

Example

Execution of the following test displays the string: "**Adding element 12 at position 3**". This string is displayed as a note in the sequence diagram and in a tab in the test report.

```

TEST CASE tc1 {

# pos = s.push(l);

PRINT ("Adding element ", l, " at position ", pos);

CHECK (!s.full ());

}

```

PROC

C++ Test Script Language

Syntax

```
PROC <procedure name> [ ( <formal parameter> [ ( , <formal parameter> ) ] ) ]
```

```
[ REQUIRE ( <native expression> ) ]
```

```
<procedure item>
```

```
[ ENSURE ( <native expression> ) ]
```

Location

TEST CLASS, TEST SUITE

Description

The PROC statement defines a procedure.

Note Procedures can be called with the CALL statement.

<procedure name> is a C++ Test Script Language identifier. It is visible in the surrounding test class or test suite, in sub-test classes or sub-test suites, and in inheriting test classes.

<formal parameter> is a C++ Test Script Language identifier. It has no type: it is replaced into the procedure by an actual parameter. Thus it can refer to a C++ type as well a C++ constant or a C++ variable.

<native expression> is a C++ expression that can be evaluated to a Boolean. The REQUIRE expression is evaluated before execution of the procedure. The ENSURE expression is evaluated after execution of the procedure. If any of these optional expressions is *False*, the evaluation leads to an error in the caller's context.

<procedure item> may be one the following entities:

- CHECK EXCEPTION
- CHECK
- CHECK PROPERTY
- CALL
- [CHECK STUB on page 761](#)
- CHECK METHOD
- [ON ERROR on page 767](#)

- COMMENT
- PRINT
- Native statement

Order is meaningful, except for CHECK STUB, CHECK METHOD, and ON ERROR statements.

The ON ERROR statement may only appear once.

Example

```
TEST CLASS TestA {  
    PROC InitArray (array, length)  
    REQUIRE (length>30 && length<array.length())  
  
    {  
        #{  
        for (int i = 0; i<length; i++)  
        array[i].init ();  
        }#  
    }  
  
    ENSURE (array[0].initialized())  
}
```

PROLOGUE

C++ Test Script Language

Syntax

PROLOGUE { <prologue item>* }

Location

TEST CLASS | TEST SUITE

Description

The PROLOGUE statement defines native code that is to be executed whenever the surrounding test class execution begins. This code is executed before any other of the test class' components.

The PROLOGUE statement may appear at most once in a test class. In an object-context, a prologue can be compared to a constructor.

<prologue item> may be one of the following entities:

- COMMENT
- PRINT
- Native statement

Order is meaningful. The native code can be made of declarations and instructions. Variables declared in prologue are visible from every component of the surrounding test class.

Note If the native code raises an exception, the prologue generates an error, handled by the ON ERROR local block. Even if the ON ERROR statement is CONTINUE, the whole TEST CLASS or TEST SUITE is skipped, including its EPILOGUE.

Example

```
TEST CLASS ATest
{
PROLOGUE {
#Stack s(20);
#s.fill ();
}
TEST CASE tc1 {
CHECK (!s.full ());
}
RUN {
tc1;
}
}
```

PROPERTY

C++ Test Script Language

Syntax

PROPERTY <property name> [(<parameter> [(, <parameter>)])]

{ ((<native expression>)) }

Location

TEST CLASS, TEST SUITE

Description

The PROPERTY statement associates a global state, defined by the conjunction of <native expression>, to a name. This name is visible in the TEST CLASS where the property is defined.

Note The occurrence of a property may be checked with the keyword CHECK PROPERTY.

<native expression> is a valid C++ expression that may be evaluated to a Boolean.

Example

```
TEST CLASS TestA {
PROPERTY Empty { ( s.count() == 0 ) }

TEST CASE tc1 {

CHECK PROPERTY Empty;

}

RUN { tc1; }

}
```

RUN

C++ Test Script Language

Syntax

RUN { <run item> }

Location

TEST CLASS, OTD Script

Description

The RUN statement defines the behavior of the surrounding test class.

<run item> may be one of the following entities:

- Test class name

Test suite name

Test case name

These names refer to a component defined in the surrounding test class or in an inherited test class. Order is meaningful. They can refer to a nested item (the nesting sequence is specified with the list of identifiers, from the most-surrounding to the most-nested one, separated by a dot).

The RUN statement can be located either within a **TEST CLASS** or at the root level of a C++ Test Driver Script:

- When used in a **TEST CLASS**, the **RUN** statement defines the behavior of the surrounding **TEST CLASS**.
- When used at the root level of a script, the **RUN** statement defines which entities are to be run when the script is executed. The **RUN** items can refer to any entity of the script.

Only one **RUN** statement is allowed at the root of a script or within each **TEST CLASS**.

The **RUN** statement is not allowed in included scripts.

RUN items are executed sequentially.

Example

```
TEST CLASS ATest
{
TEST CASE tc1 {
#s.push (i);
CHECK (!s.full ());
}
TEST CASE tc2 {
#s.pop ();
CHECK (!s.empty ());
RUN {
tc1; tc2; tc1;
}
}
```

STUB

The STUB statement defines a stub for a function or method. A stub defines or replaces the initial routine.

C++ Test Script Language

Syntax

```
STUB <stub name> : <native routine signature>
[ REQUIRE ( <require native expression> ) ]
{<stub item>... <stub item>}
[ ENSURE ( <ensure native expression> ) ]
```

Location

[C++ contract check scripts \(.otc\) on page 786](#)

Note The use of stubs requires instrumentation.

<stub name> is a unique C++ Test Script Language identifier.

<native routine signature> is a C++ signature matching the routine to stub. Unlike WRAP signatures, the signature must be complete; the return type and parameters (type and name) must be specified. If the routine is a class member or belongs to a namespace, its name must be qualified. If the routine is a template function or a template class member, the usual template<...> prefix must be used. If it is a generic template, any instance of this template is stubbed. If it is a template specialization, only the corresponding instance is stubbed.

<require native expression> is a C++ expression that can be evaluated to a Boolean. It is evaluated before the stub execution. It can refer to:

- The global variables defined in the test script.
- The stubbed routine's parameters.

<ensure native expression> is a C++ expression which can be evaluated to a Boolean. It is evaluated after the stub execution. It can refer to:

- The global variables defined in the test script.
- The stubbed routine's parameters.
- The **_ATO_result** variable that contains the routine return value, if any. Its type is that of the routine return type. Its value may be undefined if no value is returned (because an exception was thrown, or a return without a value is executed, or the function implicitly returns).
- The **_ATO_in_exception** Boolean variable, which is *True* if the post-condition is executed because an exception has been thrown. This variable is available only if the Target Deployment Package is configured to support exceptions.

If one of these expressions is *False*, the stub is failed but not the **CHECK STUB**, which could still have been defined to ensure the stub is called.

<stub item> may be one the following entities:

- CHECK
- COMMENT
- PRINT
- Native statement

The "..." zone is optional and is replaced by the code provided through the [CHECK STUB on page 761](#) statement. If not specified, it is implicitly defined at the end of the **STUB** block.

You cannot define several stubs for the same method. However you can define a stub for each instance of a template function or a template class member.

If a statement of the **STUB** generates an error, the stub is declared failed, but its execution continues (there is always an implicit **ON ERROR CONTINUE** in stubs).

An error in a **STUB** does not imply an error in the **TEST CASE** containing the corresponding **CHECK STUB**. The **CHECK STUB** statement only checks that the stub is called, not that its execution is correct.

Example : STUB ModifyCell : int IntArray::Modify (int Cell)

```
REQUIRE (Cell != 128)
```

```
{
```

```
#int Nb = random(10000);
```

```
... // this part is completed by the code of CHECK STUB
```

```
#return (Nb);
```

```
}
```

In this example, a number *Nb* is randomly chosen. If no additional code is provided by a **CHECK STUB**, then this number is returned. If a **CHECK STUB** is provided, assign the expected return value to *Nb* on a case-by-case basis.

TEST CASE

C++ Test Script Language

Syntax

```
TEST CASE <test case name> { <test case item> }
```

Location

TEST CLASS, TEST SUITE

Description

The TEST CASE statement describes an object test case, which is the smallest testing structure in a hierarchical C++ Test Driver Script. Test cases appear in test classes and test suites.

<test case name> is a C++ Test Script Language identifier.

<test case item> may be one of the following entities:

- [ON ERROR on page 767](#)
- CHECK EXCEPTION
- CHECK
- CHECK PROPERTY
- CHECK METHOD
- [CHECK STUB on page 761](#)
- CALL
- COMMENT
- PRINT
- Native statement

CALL, CHECK, CHECK PROPERTY, COMMENT, PRINT as well as Native statements are ordered (they are executed sequentially). Other entities are not (they have a global effect on the test case).

ON ERROR and CHECK EXCEPTION may appear only once.

Example

```
TEST CLASS A {  
  
TEST CASE 1 {  
  
CHECK (x == 1);  
  
#do_something ();  
  
CHECK PROPERTY ok;  
  
}
```

```

RUN {
1;
}
}

```

TEST CLASS

C++ Test Script Language

Syntax

```

TEST CLASS <test_class_name> [<formal_parameter> [ , <formal_parameter> ]] [: <parent_class>]
[<actual_parameter> [ , <actual_parameter>]] { <test_class_item>}

```

Location

C++ Text Driver Script, TEST CLASS

Description

The TEST CLASS statement describes an object test class, which is one of the structuring entities of a C++ Test Driver Script. Test classes can appear at the root-level of a C++ Test Driver Script and in test classes.

<test class name> is a C++ Test Script Language identifier.

<formal parameter> is a C++ Test Script Language identifier. It has no type: it is replaced into the test class by an actual parameter. Thus it can refer to a C++ type as well a C++ constant or a C++ variable.

<actual parameter> is a C++ actual parameter.

<parent class> is a valid test class that is defined in the same scope that contains the TEST CLASS. All entities of a parent class are inherited. This means that they are available just as if they were defined in <test class name> itself. The entities defined in the current test class with the same name as in the parent class are said to override, or replace, the entities defined in the parent class.

<test class item> may be one of the following entities:

- TEST CLASS
- TEST SUITE
- TEST CASE
- [ON ERROR on page 767](#)
- PROPERTY

- PROC
- PROLOGUE
- EPILOGUE
- RUN

A test class scope has no order, so these entities can appear in any order. However ON ERROR, EPILOGUE, PROLOGUE, and RUN may appear only once. The execution of a [TEST CLASS on page 779](#) without a RUN statement will execute the class' PROLOGUE and EPILOGUE only.

Example

```
TEST CLASS AdvancedTest (T) : BasicTest
```

```
{  
PROLOGUE {  
#Stack s (20);  
}  
PROPERTY Initial { (s.count == 0) }  
PROPERTY Final { (s.count == 1) }  
TEST CASE tc1 {  
CHECK PROPERTY Initial;  
#s.push (1);  
CHECK PROPERTY Final;  
}  
RUN {  
tc1;  
}  
}
```

TEST SUITE

C++ Test Script Language

Syntax

TEST SUITE *<test suite name>* { *<test suite item>*}

Location

OTD script, TEST CLASS, TEST SUITE

Description

The TEST SUITE statement describes an Object test suite, which is one of the structuring entities of an C++ Test Driver Script. Test suites can appear at the root-level of a C++ Test Driver Script, in test classes, and in test suites.

<test suite name> is a C++ Test Script Language identifier.

<test suite item> may be one of the following entities:

- TEST SUITE
- TEST CASE
- [ON ERROR on page 767](#)
- PROPERTY
- PROC
- PROLOGUE
- EPILOGUE

All entities but TEST CASE are not ordered in a test suite scope. However, ON ERROR, EPILOGUE, and PROLOGUE may appear only once. The test cases and test suites of a test suite are executed sequentially.

Example

```
TEST SUITE ChargeTest {
```

```
TEST CASE Test1
```

```
{
/... */
```

```
}
```

```
TEST SUITE Test2
```

```
{
```

```
TEST CASE SubTest2a
```

```
{
```

```
/*  
*/  
}  
TEST CASE SubTest2b  
  
{  
/*  
*/  
}  
}  
}
```

REQUIRE

C++ Test Script Language

Syntax

REQUIRE <*native expression*>

Location

[WRAP](#), [STUB on page 776](#), [PROC on page 771](#)

Description

The REQUIRE statement describes a method pre-condition. It can be used in a **WRAP**, **STUB** or **PROC** block.

Note The information below pertains to the use of **REQUIRE** within a **WRAP** block. For more information about using the **REQUIRE** and **ENSURE** statement within a **STUB** or **PROC** block, please refer to the [STUB on page 776](#) and [PROC on page 771](#).

<*native expression*> is a C++ Boolean expression (or an expression that can be converted into a Boolean), which can use:

- Any of the public or protected class members.
- The method parameters (with the names used in the signature or in the method definition).
- Any of the global variables declared in the file where the method is defined.

The following symbols cannot be used in the <*native expression*> parameter of the **REQUIRE** statement:

- Local variables
- Macros

Evaluation

The *<native expression>* parameter of the **REQUIRE** statement is evaluated before any code of the method is executed (local variables are not pushed yet).

Warning: you can call methods in *<native expression>*, but you must ensure that these calls do not modify the object's state by writing to any field. You can ensure this by calling *const* methods only.

Example

C++ source code example:

```
class Stack {
    int count;
    Stack () : count(0) {}
    void push (void *);
    void *pop ();
};
```

OTC code example:

```
CLASS Stack {
    WRAP pop
    REQUIRE (count > 0)
}
```

Native Code

Syntax

```
# <single-line C++ code>
```

```
C++# <single-line C++ code>
```

```
#{ <multiple-line C++ code> }#
```

```
C++#{ <multiple-line C++ code> }#
```

Location

C++ Test Driver Script, PROLOGUE, EPILOGUE, TEST CASE, PROC, STUB, CHECK EXCEPTION

Description

<*single-line C++ code*> and <*multiple-line C++ code*> are made of one or several C++ statements. They must conform to the syntax expected by the host compiler and must be relevant to the current context.

Macros may be used, but it is recommended to define them only at the root-level of the C++ Test Driver Script.

Native code is copied as is in the generated test driver source.

Only global declarations are allowed inside the C++ Test Driver Script.

Inside a PROLOGUE statement, the declaration's scope is that of the surrounding structure (TEST CLASS or TEST SUITE). Elsewhere, the scope is local (visible from the declaration to the end of the C++ Test Script Language block).

the sequence `#}` is different from `}#`:

- `#}` ends a multiple-line native code block started by `#{`.
- `#}` is a single-line native code made with the character "closing brace."

Warning: The use of `return`, `goto`, or any other jump instruction is not allowed in native code. If jump instructions are used, unexpected results will occur.

Native code may generate an error when it raises an unexpected exception. Use the CHECK EXCEPTION statement to specify exceptions.

Example

```
##include <myclass.h>

#{

static int counter;

extern void initialize (MyClass &);

static const int MAX=200;

}#

TEST CLASS A {

ON ERROR EXIT;

PROLOGUE {

#MyClass mc;

#initialize (mc);

#for (counter=0; counter < MAX; counter++) {
```

```
}  
  
TEST CASE 1 {  
  
#void *temp;  
  
#temp = mc.create ();  
  
#mc.unref (temp);  
  
CHECK mc.empty ();  
  
}  
  
TEST CASE 2 {  
  
#{  
  
void *temp[MAX];  
  
for (int i = 0; i<counter; i++)  
  
{  
  
temp[i] = mc.create ();  
  
}  
  
for (int i = 0; i<counter; i++)  
  
{  
  
mc.unref (temp[i]);  
  
}  
  
CHECK mc.empty ();  
  
}#  
  
}  
  
EPILOGUE {  
  
#} //end of for  
  
}  
  
}  
  
RUN {
```

```

A.1;
A.2;
}

```

In this example, a loop is defined around the components of the test class A. The loop starts in **PROLOGUE**, and ends in **EPILOGUE**. The execution of test class A will run nothing, because there is no **RUN** statement in this test class. However, the two test cases may be run separately, as it is shown in the above example.

The execution sequence is: *A.PROLOGUE, A.1 (200 times), A.EPILOGUE, A.PROLOGUE, A.2 (200 times), A.EPILOGUE*.

C++ contract check scripts (.otc)

Component Testing for C++ uses its own simple language for describing contracts.

This section describes each keyword of the C++ contract check language, including:

- Syntax
- Functionality and rules governing its usage
- Examples of use

Notation conventions

Throughout this section, command notation and argument parameters use the following standard convention:

Notation	Example	Meaning
BOLD	BEGIN	Language keyword
<i><italic></i>	<i><filename></i>	Symbolic variables
[]	[<i><option></i>]	Optional items
{ }	{ <i><filenames></i> }	Series of values
[{ }]	[{ <i><filenames></i> }]	Optional series of variables
	on off	OR operator

C++ test driver script keywords are case insensitive. This means that **STUB**, **stub**, and **Stub** are interpreted the same way.

For conventional purposes however, this document uses upper-case notation for the C++ contract check script keywords in order to differentiate from native source code.

Split statements

C++ contract check script statements may be split over several lines in an **.otc** contract check script. Continued lines must start with the ampersand (&) symbol to be recognized as a continuation of the previous line. No tabs or spaces should precede the ampersand.

Statements must be shorter than 2048 characters, although this limit may be lower on some platforms.

Language identifiers

A C++ contract check identifier is a text string used as a label, such as the name of a **TEST** or a **STUB** in an **.otc** contract check script.

Identifiers are made of an unlimited sequence of the following characters:

- a-z
- A-Z
- 0-9
- _ (underscore)

Spaces are not valid identifier characters.

Note that identifiers starting with a numeric character are allowed. The following statement, for example, is syntactically correct:

```
CLASS 1
{
..
}
```

C++ test driver script identifiers are case sensitive. This means that **LABEL**, **label**, and **Label** are three different identifiers.

Related Topics

[C on page 753](#) [++ test driver script structure on page 753](#) | [C++ test driver script keywords on page 755](#) | [C++ contract check scripts \(.otc\) on page 786](#)

C++ contract check script structure

A C++ Contract Check Script (**.otc** script) describes assertions for a set of classes. Each C++ class can be associated to a Contract Check CLASS block. When built, the script instruments the source code under test.

The scripts, being descriptions, are made of statements only. As a consequence, the order of execution is irrelevant. There is no hierarchical structure.

The Contract Check CLASS block describes assertions for a C++ class.

Note The evaluation of the contract should not have any side effects. The contract evaluation does not alter the state of the corresponding system. For more specific information refer to REQUIRE, ENSURE, INVARIANT and STATE sections.

Basic structure

A typical Component Testing **.otc** contract check script structure looks like this:

```

CLASS VCR {
  STATE Empty {
    (media_present() == false)
  }
  STATE Loaded {
    (media_present() == true)
    (mode () == m_stop;)
  }
  STATE Playing {
    (media_present() == true)
    (mode() == m_play || mode() == m_pause)
  }

  TRANSITION Empty TO Loaded;
  TRANSITION Loaded TO Playing;
  TRANSITION Playing TO Loaded;
  TRANSITION Playing TO Empty;
  TRANSITION Loaded TO Empty;
}

```

All instructions in a test script have the following characteristics:

- A **CLASS** block contains all the assertions for a C++ class.
- All statements begin with a keyword.
- Statements are not case sensitive (except when C expressions are used).
- Statements start at the beginning of a line and end at the end of a line. You can, however, write an instruction over several lines using the ampersand (&) continuation character at the beginning of additional lines. In this case, the ampersand must be the very first character on that line; no spaces or tabs should precede it.
- Statements must be shorter than 2048 characters, although this limit may be lower on some platforms.

Related Topics

[C++ contract check script \(.otc\) on page 786](#) | [C++ test script keywords on page 789](#)

C++ contract check script keywords

- CLASS and SINGLE CLASS
- WRAP
- REQUIRE
- ENSURE
- INVARIANT
- STATE
- TRANSITION ... TO

Related Topics

[C++ contract check script \(.otc\) on page 786](#) | [C++ contract check script structure on page 787](#)

Inheritance

Contracts are divided into several semantic parts:

- State machine
- List of invariants
- Pre-conditions
- Post-conditions

Each of these parts can be inherited in separate ways in derived classes, unless a matching part has been found in the derived class.

If you specify invariants for a class, they override the invariants defined for any base class. Similarly, a state machine description for a class overrides any state machine definition inherited from a base class.

If a class inherits from several base classes for which a class contract is defined, but does not define any invariant, the base class's invariants are merged. Similarly, if no state transition is defined, a state-transition is maintained for every sub-object inheriting a tested base class.

If you want to define a contract for a class, but not all of its base classes are associated to a contract, then you should use invariants and state transitions with care, because the methods inherited from the non-tested classes are not instrumented. In this situation, define a contract, even empty, for every base class of the class you want to test. A warning is generated during the instrumentation if such a case is encountered.

CLASS and SINGLE CLASS

C++ Test Script Language

Syntax

```
[SINGLE] CLASS <native class signature> { <class assertions>* }
```

Location

C++ Contract Check Script

Description

The CLASS statement introduces a block describing assertions for a specific C++ class. This block is called a *class contract*. Assertions described in a CLASS statement also apply to derived classes, unless the SINGLE keyword is used.

Use SINGLE CLASS to describe assertions only for the <native class signature> class.

<native class signature> is a qualified C++ class name. It can refer to template classes.

For instances of [template classes on page 520](#), the signature must follow the pattern:

```
template<> class_name<actual_parameters>
```

Note: The template<> sequence may be omitted in order to comply with deprecated usage, but it is best to specify it.

For template classes with generic parameters, the signature must follow the pattern:

```
template<formal_parameters> class_name
```

Note: The template<...> sequence may be omitted, but in that case it is not possible to use the formal parameters in the nested WRAP signatures).

<class assertions> can be one of the following:

- WRAP
- INVARIANT
- STATE
- TRANSITION

Examples

C++ source code example:

```
class A {
class B {
// ...
};
// ...
};
template<class T,int N> class C
{
// ...
};
```

C++ Contract Check Script example:

```
CLASS A
{
INVARIANT (/*...*/);
// ...
}
CLASS A::B
{
// ...
}
```

```
CLASS template<class T,int N> C
```

```
{
```

```
// ...
```

```
}
```

```
CLASS template<> C<char*,255>
```

```
{
```

```
// ...
```

```
}
```

```
}
```

Related Topics

Inheritance | Template classes

INVARIANT

C++ Test Script Language

Syntax

INVARIANT <*native expression*>;

Location

CLASS

Description

The INVARIANT statement describes a condition that should be always true in an object life, that is, whenever one of its method can be called. It appears in a CLASS block.

<*native expression*> is a C++ Boolean expression (or an expression that can be converted to a Boolean).

The following symbols can be used in <*native expression*>:

- Any of the class members.
- Any of the global variables declared in every file where a method of the class (or a method of descendant if it is not a single class contract) is defined.

The following symbols cannot be accessed in <*native expression*>:

- Local variables of any methods.
- Macros: Global variables that are not defined in at least one file where a method of the class (or one of its descendants, if it is not a single class contract) is defined.

Evaluation:

<native expression> is evaluated at the end of the execution of the class constructors (except the implicitly defined copy constructor), at the beginning of the class destructors, and both at the beginning and the end of other non-static non implicitly defined methods.

Warning: You can call methods in <expr>, but you must ensure that these calls do not modify the object's state (that is, they do not write to any field). You can ensure this by calling *const* methods only. If you want the compiler to check this, see the **ATO_AC_STRICT_CHECKING** Target Package option.

Example

C++ source code example:

```
class Stack {
int count;
Stack () : count(0) {}
void push (void *);
void *pop ();
};
```

C++ Contract Check Script code example:

```
CLASS Stack {
INVARIANT (count >= 0);
}
```

STATE

C++ Test Script Language

Syntax

STATE <state name> { (<native expression>)}

Location

CLASS

Description

The STATE statement describes a state for the current class, which is defined by the conjunction of one or several Boolean expression.

Notes STATE by itself does not generate any source code instrumentation. STATE should be used along with the TRANSITION statement.

<state name> is a C++ Test Script Language identifier.

<native expression> is a C++ Boolean expression (or an expression that can be converted to a Boolean).

The following symbols can be used in <native expression> :

- Any of the class members.
- Any of the global variables declared in every file where a method of the class (or a method of descendant if it is not a single class contract) is defined.

The following symbols cannot be accessed in <native expression>:

- Local variables of any methods.
- Macros.
- Global variables that are not defined in at least one file where a method of the class (or one of its descendants if it is not a single class contract) is defined.

Evaluation

<native expression> may be evaluated at the end of the execution of class constructors (except for implicitly defined copy constructors), at the beginning of class destructors, and both at the beginning and the end of other non-static non-implicitly defined methods.

Warning: You can call methods in <native expression>, but you must ensure that these calls do not modify the object's state by writing to any fields. You can ensure this by using *const* methods only. If you want the compiler to check this, see the ATO_AC_STRICT_CHECKING Target Package [option](#) .

Example

C++ source code example:

```
class Stack {  
  
    int count;  
  
    Stack () : count(0) {}  
  
    void push (void *);  
};
```

```
void *pop ();
};
```

C++ Contract Check Script code example:

```
CLASS Stack {
STATE Empty { (count == 0) }
STATE NotEmpty { (count > 0) }
}
```

TRANSITION ... TO

C++ Test Script Language

Syntax

TRANSITION <state name> **TO** <state name>;

Location

CLASS

Description

The TRANSITION statement describes a transition between two states an object can execute during its life.

<state name> is a valid state name defined with the STATE keyword.

Transitions are checked between two state evaluations. States are evaluated at the end of the execution of class constructors (except for implicitly-defined copy constructor), at the beginning of class destructors, and both at the beginning and the end of other non-static non-implicitly defined methods.

All states are evaluated after an object has been created (when leaving a constructor). Consequently, the initial state must be described in a non-ambiguous way: one - and one only - state must occur when leaving a constructor.

Once a state has been determined, only authorized states (according to the defined transitions) are checked. Ambiguity must not occur when choosing the next state.

States are always reflexive. This means that a transition from a state to itself is implicitly defined. There must be no ambiguity between one state and any other state that can be reached through a single transition.

Example

C++ source code example:

```
class Stack {
```

```
int count;

int capacity;

Stack () : count(0) {}

void push (void *);

void *pop ();

};
```

C++ Contract Check Script code example:

```
CLASS Stack {

STATE Empty { (count == 0) }

STATE NotEmpty { (count > 0) (count < capacity) }

STATE Full { (count == capacity) }

TRANSITION Empty TO NotEmpty;

TRANSITION NotEmpty TO Full;

TRANSITION Full TO NotEmpty;

TRANSITION NotEmpty TO Empty;

}
```

WRAP

C++ Test Script Language

Syntax

WRAP <native method signature> <WRAP assertions>

Location

CLASS

Description

The WRAP statement describes pre- and post-conditions for a method.

<native method signature> refers to an existing method within the class.

The return type may be omitted.

The parameters names may be omitted if the WRAP assertions do not refer to the parameters.

The parameters list may be omitted if the method is not overloaded and if the WRAP assertions do not refer to the parameters.

<WRAP assertions> is made of either one or several [REQUIRE on page 782](#) or [ENSURE on page 764](#) statements.

Wraps can be defined for any method of the class, whatever its access specifiers may be.

Wraps cannot be associated to an inherited method, defined in a base class. If you want to do so, define a WRAP in a contract associated with the base class.

If the method is virtual, and the WRAP does not belong to a SINGLE CLASS, the wrap definition also applies to any redefinition of the method, unless a specific wrap has been defined for the redefinition in a daughter class.

Example

C++ source code:

```
class A {
    int f ();
    char *g (int); // overloaded method
    void g(void *); // overloaded method
};
```

C++ Contract Check Scriptcode:

```
CLASS A {
    WRAP f /OK */
    REQUIRE ( /*... */ )
    ENSURE ( /*... */ )

    WRAP g /ambiguous -> error */
    /*... */

    WRAP g(int) / OK */
    /*... */

    WRAP g(void *p) / OK */
```

```

/... */
}

```

Target Deployment Port options

Common Options

The following options pertain to the Component Testing for C++ feature.

Option	Description
ATO_ CAST_ PRINT_ BUFFER_ SIZE	This macro defines the size of the buffer devoted to the PRINT instruction. This buffer must be large enough to contain the output of a single PRINT instruction. If memory is an issue, you can set this value to 0. In that case, a single PRINT instruction will result in several notes in the graphical report, one per argument.

C++ Test Driver Script Options

The following options pertain only to C++ Test Driver Scripts.

Option	Description
ATO_ USE_ CAST	Usually ATL_YES , this macro can be set to ATL_NO if you are not using C++ Test Driver scripting. In this case, the Target Deployment Package object is smaller, and the compiler requires less memory to compile instrumented files.
ATO_ CAST_ STOP_ ON_ER- ROR	When this macro is set to ATL_YES , a function named ATL_Breakpoint is called whenever an error occurs in the C++ Test Driver Script. In this case, you must provide this function, either by defining it in custom.hor by defining a macro naming your own breakpoint function in custom.h . You can thus set a breakpoint on this function and debug your test application when an unexpected result is encountered.
ATO_ CAST_ DUMP_ SUC- CESS	By default the value is ATL_YES . This macro can be set to ATL_NO if you do not want passed checks of your C++ Test Driver Script to be added to the trace file. This may be important if trace file size is an issue.

ATO_-
CAST_-
MAX_-
INS-
TANCES

This macro defines the maximum number of instances you expect to be used at the same time when running a C++ Test Driver Script. An instance is pushed in a stack when a **TEST CLASS**, **TEST SUITE**, or **TEST CASE** is entered and when a **PROC** or a **STUB** is called. Note that stubs can be recursive. The default value is 256. You can lower this value if memory is an issue and you know how many instances are used at the same time. You can increase it if your script is complex or if you use many stubs that call themselves or each other.

C++ Contract Check Script Options

The following options pertain only to the C++ Contract Check Scripts.

Option Description

ATO_-
USE_-
AC

Usually **ATL_YES**, this macro can be defined to **ATL_NO** if you are not using C++ Contract Check scripting. In this case, the Target Deployment Package object is smaller, and the compiler requires less memory to compile instrumented files and generated files.

ATO_-
AC_S-
TOP_-
ON_-
ERROR

When this macro is set to **ATL_YES**, a function named **ATL_Breakpoint** is called whenever an error occurs in the C++ Contract Check Script. In this case, you must provide this function, either by defining it in **incustom.h**, or by defining a macro naming your own breakpoint function **incustom.h**. You can thus set a breakpoint on this function and debug your test application when an unexpected result is encountered.

ATO_-
AC_-
DUMP_-
SUC-
CESS

Usually **ATL_YES**, this macro can be defined to **ATL_NO** if you do not want passed checks of your C++ Contract Check Script to be added to the trace file. This may be important if the trace file size is an issue.

ATO_-
AC_-
FILE_-
NAME

This macro defines the default trace file name when executing the C++ Contract Check Script instrumented application. This name is used if you have not provided the **GetEnvironment** macro or **\$ATO_TRACES** or **\$ATT_TRACES** (**%ATO_TRACES%** or **%ATT_TRACES%** on Win32 platforms) environment variables, and if you are not using C++ Contract Check scripting.

ATO_-
AC_-
STRIC-

When this macro is set to **ATL_YES**, the invariants and states defined in C++ Contract Check Scripts are enforced to be *const*. This implies that the compiler ensures that they do not modify any field of the object, and that they call only *const* methods. The default is **ATL_NO** because users often omit to specify the *const*

T_- stqualifier for methods that are actually *const*. If **ATL_NO** is chosen, you must make sure that your invariants CHECK- and state evaluations do not modify your objects.
ING

Component Testing for Ada

Ada test script language reference

Component Testing for Ada uses its own simple language for test scripting.

This section describes each keyword of the Ada test script language, including:

- Syntax
- Functionality and rules governing its usage
- Examples of use

Notation conventions

Throughout this section, command notation and argument parameters use the following standard convention:

Notation	Example	Meaning
BOLD	BEGIN	Language keyword
<italic>	<filename>	Symbolic variables
[]	[<option>]	Optional items
{ }	{ <filenames> }	Series of values
[{ }]	[{ <filenames> }]	Optional series of variables
	on off	OR operator

Ada test script keywords are case insensitive. This means that **STUB**, **stub**, and **Stub** are interpreted the same way. The keyword **others** is an exception, and must always be expressed in lower case.

For conventional purposes however, this document uses upper-case notation for the Ada test script keywords in order to differentiate from native source code.

Split statements

Ada test script statements may be split over several lines in a **.ptu** test script. Continued lines must start with the ampersand (&) symbol to be recognized as a continuation of the previous line. No tabs or spaces should precede the ampersand.

Statements must be shorter than 2048 characters, although this limit may be lower on some platforms.

Language identifiers

An Ada test script identifier is a text string used as a label, such as the name of a **TEST** or a **STUB** in a **.ptu** test script.

Identifiers are made of an unlimited sequence of the following characters:

- a-z
- A-Z
- 0-9
- _ (underscore)

Spaces are not valid identifier characters.

Note that identifiers starting with a numeric character are allowed. The following statement, for example, is syntactically correct:

```
TEST 1
```

```
...
```

```
END TEST
```

Ada test script identifiers are case sensitive. This means that LABEL, label, and Label are three different identifiers.

Ada test script structure

The Ada test script language allows you to structure tests to:

- Describe several test cases in a single test script,
- Select a subset of test cases according to different Target Deployment Port criteria.

Test script filenames must contain only plain alphanumerical characters.

Basic structure

A typical Ada Component Testing **.ptu** test script looks like this:

```
HEADER add, 1, 1

<variable declarations for the test script>

BEGIN

SERVICE add

<local variable declarations for the service>

TEST 1

FAMILY nominal

ELEMENT

VAR variable1, INIT=0, EV=0

VAR variable2, INIT=0, EV=0

#<call to the procedure under test>

END ELEMENT

END TEST

END SERVICE
```

All instructions in a test script have the following characteristics:

- All statements begin with a keyword.
- Statements are not case sensitive (except when Ada expressions are used).
- Statements start at the beginning of a line and end at the end of a line. You can, however, write an instruction over several lines using the ampersand (&) continuation character at the beginning of additional lines. In this case, the ampersand must be the very first character on that line; no spaces or tabs should precede it.
- Statements must be shorter than 2048 characters, although this limit may be lower on some platforms.

Structure statements

The following statements allow you to describe the structure of a test.

- **HEADER:** For documentation purposes, specifies the name and version number of the module being tested, as well as the version number of the tested source file. This information is displayed in the test report.
- **BEGIN:** Marks the beginning of the generation of the actual test program.

- **SERVICE:** Contains the test cases related to a given service. A service usually refers to a procedure or function. Each service has a unique name (in this case add). A **SERVICE** block terminates with the instruction **END SERVICE**.
- **TEST:** Each test case has a number or identifier that is unique within the block **SERVICE**. The test case is terminated by the instruction **END TEST**.
- **FAMILY:** Qualifies the test case to which it is attached. The qualification is free (in this case **nominal**). A list of qualifications can be specified (for example: **family, nominal, structure**) in the Tester Configuration dialog box.
- **ELEMENT:** Describes a test phase in the current test case. The phase is terminated by the instruction **END ELEMENT**. The different phases of the same test case cannot be dissociated after the tests are run, unlike the test cases introduced by the instruction **NEXT_TEST**. However, the test phases introduced by the instruction **ELEMENT** are included in the loops created by the instruction **LOOP**.

The three-level structure of the test scripts has been deliberately kept simple. This structure allows:

- A clear and structured presentation of the test script and report
- Tests to be run selectively on the basis of the service name, the test number, or the test family.

Related Topics

[Ada test script language reference on page 800](#) | [Ada test script keywords on page 803](#) | [Writing a test script on page 532](#)

Ada test script keywords

Block Keywords

- [ELEMENT...END ELEMENT on page 807](#)
- [ENVIRONMENT ... END ENVIRONMENT on page 809](#)
- [INITIALIZATION...END INITIALIZATION on page 815](#)
- [SERVICE...END SERVICE on page 817](#)
- [SIMUL...ELSE_SIMUL...END SIMUL on page 818](#)
- [TERMINATION...END TERMINATION on page 823](#)
- [TEST...END TEST on page 823](#)

Other Keywords

- [BEGIN on page 804](#)
- [COMMENT on page 805](#)
- [DEFINE STUB on page 806](#)
- [FAMILY on page 811](#)
- [HEADER on page 812](#)
- [IF...ELSE...END IF on page 812](#)
- [INCLUDE on page 814](#)
- [NEXT_TEST on page 815](#)
- [STUB on page 820](#)
- [VAR, ARRAY and STR on page 824](#)
 - [<initialization> parameter on page 826](#)
 - [<expected> parameter on page 830](#)
 - [<variable> parameter on page 825](#)

BEGIN

Ada Test Script Language

Purpose

The **BEGIN** instruction marks the beginning of the Ada code generation. The **BEGIN GENERIC** option is specifically for testing Ada generic packages.

Syntax

```
BEGIN [ <parent_unit> [ , <procedure> ] ]
```

```
BEGIN GENERIC( <generic_package>, <instance> ) [ , <procedure> ]
```

where:

- *<parent_unit>* is the full name of the unit under test.
- <procedure>* is the name of the generated separate procedure, by default **ATTOL_TEST**.
- <generic_package>* is the name of a generic unit under test.
- <instance>* is the name of the instantiated unit from the generic.

Description

The **BEGIN** instruction is mandatory and must be located after a **HEADER** statement, and before any other Ada Test Script instruction.

By default, the Ada Test Script Compiler creates an independent compilation unit. To test private elements of a package you must first generate a procedure.

The reference body to the separate procedure must be written in the parent unit package.

If a **BEGIN** keyword is not found in the test script, a warning message is generated and a **BEGIN** instruction is implicitly created before the first occurrence of a **SERVICE** instruction.

To test a generic package, you need to generate the test driver separately and call it as a procedure of the instance. Use the **BEGIN GENERIC** syntax to automatically generate a separate procedure *<procedure>* of *<generic_package>*. This allows you to access the procedure *<instance>* . *<procedure_name>*, which is generated by the Ada Test Script Compiler.

Note This technique also allows testing of private types within the generic package.

Related Topics

[HEADER on page 812](#) | [SERVICE on page 817](#)

COMMENT

Ada Test Script Language

Purpose

The **COMMENT** instruction adds a textual comment to the test report.

Syntax

COMMENT [*<text>*]

where:

- *<text>* is an optional comment text string to be displayed.

Description

The **COMMENT** instruction is optional and can be used anywhere in the test script.

The position of the **COMMENT** instruction in the test script determines the position where the comment is displayed in the test report:

- Before the first **SERVICE** block: the comment is displayed after the report information header and before the first service.
- Inside a **SERVICE** block: the comment is displayed in the service header, before the test descriptions.
- Outside a **SERVICE** block: the comment is displayed in the following service header, before the test descriptions.
- After the last **SERVICE** block: the comment is ignored.
- Inside an **ELEMENT** block: the comment is displayed before the variable state descriptions.
- After a **TEST** instruction: the comment is displayed in the test header, before the variable descriptions.

Example

```
TEST 1
```

```
FAMILY nominal
```

```
COMMENT histogram computation for a black image
```

```
ELEMENT
```

Related Topics

[ELEMENT on page 807](#) | [TEST on page 823](#) | [SERVICE on page 817](#)

DEFINE STUB ... END DEFINE

Ada Test Script Language

Purpose

The **DEFINE STUB** and **END DEFINE** instructions delimit a simulation block consisting of stub definition declarations written in Ada.

Syntax

```
DEFINE STUB <stub_name> [ <stub_dim> ]
```

```
[# <Ada statement>]
```

```
END DEFINE
```

where:

1. <stub_name> is the mandatory name of a simulation block.

<stub_dim> is an optional maximum number of stub calls errors that will be displayed in the report.

Description

Defining stubs in a test script is optional.

DEFINE STUB blocks must be located after the **BEGIN** instruction and outside any **SERVICE** block.

By default, all functions and procedures of the *<stub_name>* package are simulated. The **DEFINE STUB** block can also contain Ada function, procedure and assignment declarations preceded with the # character.

Using the stub definitions, the Ada Test Script Compiler generates simulation variables and functions for which the interface is identical to that of the stubbed variables and functions.

The purpose of these simulation variables and functions is to store and test input parameters, assign values to output parameters, and if necessary, return appropriate values.

Stub parameters describe both the type of item used by the calling function and the mode of passing. The mode of passing the parameter is specified by adding the following before the parameter name:

1. **in** for input parameters
 - out** for output parameters
 - in out** for input/output parameters
 - _no** for parameters that you do not want to test

Additionally, when using the **in** or **in out** parameters, you can add an optional **_nocheck** parameter before the **in** or **in out** parameter (see the Example paragraph). This allows the parameters to be sent to the stub without being checked.

The parameter mode is optional. If no parameter mode is specified, the **in** mode is assumed by default.

A return parameter is always deemed to be an output parameter.

Global variables defined in **DEFINE STUB** blocks replace the real global variables.

By default, only the first 10 errors are shown in the report. Any more errors are not recorded. The number of calls should be customized if necessary by using the *<stub_dim>* parameter.

Example

An example of the use of stubs is available in the **StubAda** example project installed with the application.

Related Topics

[STUB on page 820](#)

ELEMENT ... END ELEMENT

Ada Test Script Language

Purpose

The **ELEMENT** and **END ELEMENT** instructions delimit a test phase or **ELEMENT** block.

Syntax

ELEMENT

END ELEMENT

Description

The **ELEMENT** instruction is mandatory and can only be located within a **TEST** block. If absent, a warning message is generated and the **ELEMENT** block is implicitly declared before the first occurrence of a **VAR**, **ARRAY**, **STR**, or **STUB** instruction.

The block must end with the instruction **END ELEMENT**. If absent, a warning message is generated and it is implicitly declared before the next **ELEMENT** instruction, or the **END TEST** instruction.

The **ELEMENT** block contains a call to the service under test as well as instructions describing the initializations and checks on test variables.

Positioning of **VAR**, **ARRAY**, **STR** or **STUB** related to the actual test procedure is irrelevant as the Test Script Compiler separates these instructions into two parts:

1. The test initialization (described by **INIT**) is generated with the **ELEMENT** instruction

The test of the expected value (described by **EV**) is generated with the **END ELEMENT** instruction

Example

TEST 1

FAMILY nominal

ELEMENT

VAR x1, init = 0, ev = init

VAR x2, init = SIZE_IMAGE-1, ev = init

VAR y1, init = 0, ev = init

VAR y2, init = SIZE_IMAGE-1, ev = init

ARRAY image, init = 0, ev = init

VAR histo(0), init = 0, ev = SIZE_IMAGE*SIZE_IMAGE

ARRAY histo(1..SIZE_HISTO-1), init = 0, ev = 0

```
#compute_histo(x1,y1,x2,y2,histo);
```

```
END ELEMENT
```

```
END TEST
```

Related Topics

[VAR on page 824](#) | [ARRAY on page 824](#) | [STR on page 824](#) | [STUB on page 820](#) | [NEXT_TEST on page 815](#) | [Initialization Expressions for Ada on page 826](#) | [Expected Value Expression for Ada on page 830](#)

ENVIRONMENT ... END ENVIRONMENT

Ada Test Script Language

Purpose

The **ENVIRONMENT** instruction defines a test environment declaration, that is, a default set of test specifications.

Syntax

```
ENVIRONMENT <name>
```

```
END ENVIRONMENT
```

<name> is a mandatory identifier that provides a unique environment name.

Description

The test environment defines a general context. Variables which are declared within a context can be overwritten by a **TEST** statement.

The **END ENVIRONMENT** instruction marks the end of an environment declaration.

<name> specifies an environment name that is referenced in the **USE** instruction.

An environment must be defined after the **BEGIN** instruction.

Each environment is visible in the block in which it has been declared and in any blocks included in this block, after its declaration.

An environment can only contain **VAR**, **ARRAY**, **STR**, **FORMAT** or **STUB** instructions and conditional generation instructions. If it is empty, a warning message is generated.

An environment is activated by the **USE** instruction that defines its scope and its priority. **ENVIRONMENT** blocks are executed in the reverse order of their respective **USE** instruction.

Note If the **USE** instruction follows directly the **ENVIRONMENT** block, the first occurrence of the **ENVIRONMENT** overrides the later, or local ones.

After generating the initializations and the tests of an **ELEMENT** block, visible environments are included in order of priority, at every **END ELEMENT** instruction, in order to complete the initializations and tests.

The scope of an **ENVIRONMENT** block is important insofar as only "visible" environment blocks apply, and use clauses can be out of scope.

Example

```
ENVIRONMENT compute_histo
```

```
VAR x1, init = 0, ev = init
```

```
VAR x2, init = SIZE_IMAGE-1, ev = init
```

```
ARRAY image, init = 0, ev = init
```

```
END ENVIRONMENT
```

Related Topics

[VAR on page 824](#) | [ARRAY on page 824](#) | [STR on page 824](#)

EXCEPTION

Ada Test Script Language

Purpose

The **EXCEPTION** instruction describes the behavior of the test script if any exceptions are raised during the execution.

Syntax

```
EXCEPTION <exception_name>
```

Description

This instruction can only appear in an **ELEMENT** block.

<exception_name> is the name of the exception under test.

This instruction must be unique in the block where it appears. If it is absent, the test shall not raise any exception, otherwise, an error is generated.

Only exceptions raised by the procedure under test can be tested. Exceptions raised during the initialization of the variables or during the test of the variables cannot be tested. They are nevertheless detected and written in the test report.

Note Do not use the **EXCEPTION** statement simultaneously with any native exception handling code, as this will create internal conflicts.

Example

In this example, the exception class is **overflow**.

ELEMENT

-- The test shall raise the overflow exception

EXCEPTION overflow

....

-- Using the 'exception' variable

VAR exception->ch1,

END ELEMENT

FAMILY

Ada Test Script Language

Purpose

The **FAMILY** instruction groups tests by families or classes.

Syntax

FAMILY <family_name> { , <family_name> }

Argument

<family_name> is a mandatory identifier indicating the name of the test family. Typically, you could specify nominal, structural, or robustness families.

Description

The **FAMILY** instruction appears within **TEST** blocks, where it defines the families to which the test belongs.

When you run the test sequence, you can request that only tests of a given *family* are executed.

A test can belong to several families. In this case, the **FAMILY** instruction contains a <family_name> list, separated by commas.

The **FAMILY** instruction must be located before the first **ELEMENT** block of the **TEST** block and must be unique in the **TEST** block.

The **FAMILY** instruction is optional. If it is omitted, a warning message is generated and the test belongs to every family.

Example

TEST 1

FAMILY nominal

COMMENT histogram computation on a black image

ELEMENT

Related Topics

[ELEMENT on page 807](#) | [TEST on page 823](#)

HEADER

Ada Test Script Language

Purpose

The **HEADER** instruction specifies the name and version of the module under test as well as the version number of the test script.

Syntax

HEADER *<module_name>* , *<module_version>* , *<test_plan_version>*

<module_name>, *<module_version>* and *<test_plan_version>* are character strings with no restrictions, except for versions beginning with a dollar sign (**\$**). These instructions must be followed by an identifier.

Description

This information contained in the **HEADER** keyword is reproduced in the test report header to identify the test sequence.

The module and test script versions can be read from the environment variables if they are identifiers beginning with a dollar sign (**\$**).

The **HEADER** instruction is mandatory, but its arguments are optional. It must be the first instruction in the test program. If it is absent, a warning message is generated.

Example

```
HEADER histo, 01a, 01a
```

```
BEGIN
```

```
IF ... ELSE ... END IF
```

Ada Test Script Language

Purpose

The **IF**, **ELSE** and **END IF** statements allow conditional generation of the test driver.

Syntax

IF *<condition>* { , *<condition>* }

ELSE

END IF

where:

- *<condition>* is an identifier sent by the **-define** option to the Ada Test Script Compiler.

Description

These statements enclose portions of script that are included depending on the presence of one of the conditions in the list provided to the Ada Test Script Compiler by the **-define** option.

The *<condition>* list forms a series of conditions that is equivalent to using an expression of logical **ORs**.

The **IF** instruction starts the conditional generation block.

The **END IF** instruction terminates this block.

The **ELSE** instruction separates the condition block into 2 parts, one being included when the other is not.

Associated Rules

<condition> is any identifier. You must have at least one condition in an **IF** instruction.

This block can contain any code written in Ada Test Script Language or native Ada.

IF and **END IF** instructions must appear simultaneously.

The **ELSE** instruction is optional.

The generation rules are as follows:

- If at least one of the conditions specified in the **IF** instruction's list of conditions appears in the list associated with the **-define** option, the first part of the block is included.

If none of the conditions specified in the **IF** instruction appears in the list associated with the **-define** option, then the second part of the block is included (if **ELSE** is present).

Example

```
IF test_on_target
```

```
VAR register, init == , ev = 0
```

ELSE

VAR register, init = 0 , ev = 0

END IF

Related Topics

[SIMUL ... ELSE_SIMUL ... END SIMUL on page 818](#)

INCLUDE

Ada Test Script Language

Purpose

The **INCLUDE** statement specifies an external file for the Ada Test Script Compiler to process.

Syntax

INCLUDE CODE <file.ada>

INCLUDE PTU <file.ptu>

where:

1. <file.ada> is the file name of an external Ada source file

<file.ptu> is the file name of an Ada test script

Description

When an **INCLUDE** instruction is encountered, the Ada Test Script Compiler leaves the current file, and starts pre-processing the specified file. When this is done, the Ada Test Script Compiler returns to the current file at the point where it left.

Including a file with the additional keyword **CODE** lets you include a source file without having to start every line with a hash character ('#').

Including a file with the additional keyword **PTU** lets you include an Ada test script within another Ada test script. In this case, included **.ptu** test scripts must not contain **BEGIN** or **HEADER** statements.

The name of the included file can be specified with an absolute path or a path relative to the current directory.

If the file is not found in the current directory, all directories specified by the **-incl** option are searched when the preprocessor is started.

If it is still not found or if access is denied, an error is generated.

Example

```
INCLUDE CODE file1.ada
```

```
INCLUDE CODE ../file2.ada
```

```
INCLUDE PTU /usr/tests/file3.ptu
```

INITIALIZATION ... END INITIALIZATION

Ada Test Script Language

Purpose

Specifies native Ada code to initialize the test driver

Syntax

```
INITIALIZATION
```

```
END INITIALIZATION
```

Description

The **INITIALIZATION** and **END INITIALIZATION** statements let you provide native Ada code that is integrated as the first *main* statements of the test driver.

In some environments, such as when using a different target machine, this is a convenient way to initialize the target.

An **INITIALIZATION** block must appear after the **BEGIN** instruction or between two **SERVICE** blocks.

This block can only contain native Ada code. Each line of native code must be preceded with '#' or '@'.

There is no limit to the number of **INITIALIZATION** blocks. Upon test driver generation, they are concatenated in the order in which they appeared in the test script.

Related Topics

[TERMINATION on page 823](#)

NEXT_TEST

Ada Test Script Language

Purpose

The **NEXT_TEST** instruction starts a **TEST** block that is linked to the previous test block.

Syntax

NEXT_TEST [LOOP <nb>]

where:

1. <nb> is an integer expression strictly greater than 1.

Description

The **NEXT_TEST** instruction allows you to repeat a series of test contained within a previously defined **TEST** block.

It contains one more **ELEMENT** block. It does not contain the **FAMILY** instruction.

For this new test, a number of iterations can be specified by the keyword **LOOP**.

The **NEXT_TEST** instructions can only appear in a **TEST ... END TEST** block.

The main difference between a **NEXT_TEST** block and an **ELEMENT** block is when you use an **INIT IN** statement within a test block:

1. If the **INIT IN** is in a **TEST** block, there will be a loop over the entire **TEST** block, without consideration of the **ELEMENT** blocks that it might contain.

If the **INIT IN** is inside a **NEXT_TEST** block however, the loop will not affect the **ELEMENT** blocks within other **TEST** blocks

Example

```
SERVICE COMPUTE_HISTO
```

```
# x1, x2, y1, y2 : integer ;
```

```
# histo : T_HISTO ;
```

```
TEST 1
```

```
FAMILY nominal
```

```
ELEMENT
```

```
...
```

```
END ELEMENT
```

```
NEXT_TEST LOOP 2
```

```
ELEMENT
```

Related Topics

[TEST on page 823](#) | [ELEMENT ... END ELEMENT on page 807](#)

SERVICE ... END SERVICE

Ada Test Script Language

Purpose

A **SERVICE** block contains a common description for all tests related to a given service of the module under test.

Syntax

SERVICE <service_name>

END SERVICE

where:

- <service_name> specified the tested service in the test report

Description

The **SERVICE** instruction starts a **SERVICE** block. This block contains the description of all the tests relating to a given service of the module to be tested.

The <service_name> is usually the name of the service under test, although this is not mandatory.

The **END SERVICE** instruction indicates the end of the service block.

Associated Rules

The **SERVICE** instruction must appear after the **BEGIN** instruction.

The <service_name> parameter can be any identifier. It is obligatory.

Example

BEGIN

SERVICE COMPUTE_HISTO

x1, x2, y1, y2 : integer ;

histo : T_HISTO ;

TEST 1

FAMILY nominal

SERVICE_TYPE

Ada Test Script Language

Purpose

The **SERVICE_TYPE** statement indicates the type of service tested.

Syntax

SERVICE_TYPE <type> {, <type>}

where:

1. <type> is a user-defined service type identifier

Description

The **SERVICE_TYPE** instruction allows you to specify an identifier indicating the type of service tested. This identifier is included in the test report.

You can use this functionality to specify whether a service is internal or external.

If **SERVICE_TYPE** is placed within a **SERVICE ... END SERVICE** block, it indicates the type of the current **SERVICE** block.

If the **SERVICE_TYPE** statement is placed outside a **SERVICE** block, then it indicates the default service type for all **SERVICE** blocks that do not contain a **SERVICE_TYPE** statement.

Example

```
SERVICE_TYPE internal, external
```

```
SERVICE count
```

```
SERVICE_TYPE internal
```

```
...
```

```
END SERVICE
```

```
SIMUL ... ELSE_SIMUL ... END SIMUL
```

Ada Test Script Language

Purpose

The **SIMUL**, **ELSE_SIMUL**, and **END SIMUL** instructions allow conditional generation of test driver.

Syntax

```
SIMUL
```

```
ELSE_SIMUL
```

END SIMUL

Description

Code enclosed within a **SIMUL** block is conditionally generated depending on the status of the **Simulation** configuration setting in the HCL OneTest Embedded GUI, or the **-nosimulation** command line option of the Ada Test Script Compiler.

The **SIMUL** instruction starts the conditional generation block.

The **END SIMUL** instruction marks the end of the conditional block.

The **ELSE_SIMUL** instruction separates this block into two parts, one being included when the other is not, and vice versa.

This block of instructions can appear anywhere in the test program and can contain both Ada Test Script Language or native Ada code.

The **SIMUL** and **END SIMUL** instructions must appear as a pair. One cannot be used without the other.

The **ELSE_SIMUL** instruction is optional.

When using the HCL OneTest™ Embedded user interface, select or clear the **Simulation** option in the **Component Testing for Ada** tab of the **Configuration Settings** dialog box.

The code generation rules are as follows:

1. If **Simulation** is enabled => the first part of the **SIMUL** block is included.
2. If **Simulation** is disabled => the second part of the block (**ELSE_SIMUL**) is included if it exists. If there is no **ELSE_SIMUL** statement, then the **SIMUL** block is ignored.

Example

SIMUL

#x := 0;

ELSE_SIMUL

#x := 1;

END SIMUL

...

SIMUL

VAR x , INIT = 0 , EV = 1

VAR p , INIT = NIL , EV = NONIL

```
ELSE_SIMUL
```

```
VAR x , INIT = 0 , EV = 0
```

```
VAR p , INIT = NIL , EV = NIL
```

```
END SIMUL
```

Related Topics

[Ada Test Script Compiler on page 1140](#)

STUB

Ada Test Script Language

Purpose

The STUB instruction for Ada describes all calls to a simulated function in a test script.

Syntax

```
STUB <stub_name> [<call_range> =>] ([<param_val> {, <param_val> }]) [<return_val>] {, [<call_range> =>] ([<param_val> {, <param_val> }]) [<return_val>] }
```

Description

The following is described for every parameter of this function and for every expected call:

- For **in** parameters, the values passed to the function. These values are stored and tested during execution.
- For **out** parameters and, where appropriate, the return value, the values returned by the function. These values are stored in order to be returned during execution.
- For **in out** and **in access** parameters, both the previous two values are required.
- For **no** parameters, no expression is required.

<stub_name> is the name of the simulated procedure or function. It is obligatory. You must previously have described this procedure or function in a **DEFINE STUB** block.

The optional <call_range> describes one or several successive calls as follows:

```
<call_num> =>
```

```
<call_num> .. <call_num> =>
```

```
others =>
```

where *<call_num>* is the number of the stub call. The keyword **others** specifies the behavior of any further calls that have not been described. A *<call_num>* value of 0 means that no calls are expected to the stub. For example, the following line specifies that test will pass if there are 0 or more calls to the stub:

```
STUB close_file others=>(5)1
```

Moreover, you can use **others** to specify that the calls are optional. Combining **others** with a list of call numbers, enables you to check the minimum number of calls. For example, the following line specifies that test will pass if there are at least 2 calls to the stub:

```
STUB close_file 1=>(3)1, 2=>(4)1, others=>(5)1
```

If *<call_range>* is not specified, then the next call number is assumed. For example, the following lines specify that the test will pass if there are 2 calls to the stub:

```
STUB open_file ("file1")3
```

```
STUB open_file ("file2")4
```

<param_val> is an expression describing the test values for in parameters and the returned values for out parameters. If named, parameters can be in any order. For in out parameters, *<param_val>* is expressed in the following way:

```
( [IN =>]<in_param_val> , [OUT =>]<out_param_val> )
```

If you use the optional **IN =>** and **OUT =>** specifiers, you can invert the order of the parameters.

<return_val> is an expression describing the value returned by the function if its type is not void. Otherwise, no value is provided.

You must give values for every in, out and in out parameter; otherwise, a warning message is generated. The no parameters are ignored.

<param_val> and *<return_val>* are Ada expressions that can contain:

- Numeric (integer or floating-point), character, or character string literal values. Strings can be delimited by single or double inverted commas.

Constants, in the Ada sense of the word, which can be numeric, characters, or character strings

Variables belonging to the test program or the module to be tested

Ada functions

The keyword **NIL** to designate a null pointer

Pseudo-variables **I**, **I1**, **I2** ..., **J**, **J1**, **J2** ..., where **I** *n* is the current index of the *n*th dimension of the parameter and **J** *m* the current number of the subtest generated by the test scenario's *m*th **INIT IN**, **INIT FROM** or **LOOP**; the **I** and **I1** variables are therefore equivalent as are **J** and **J1**; the subtest numbers begin at **1** and are incremented by **1** at each iteration

An Ada expression with one or more of the above elements combined using any of the Ada operators and casting, with all required levels of parentheses, and conforming to Ada rules of syntax and semantics

For arrays and structures, aggregates between parentheses ('()') or brackets ('[]').

<param_val> can contain for an **in** value:

- The **<->** expression to specify that the parameter should be ignored

The **<value> <-> <value>** expression to specify a range of values for the parameter

<param_val> can contain for an **out** value or **return** value:

- The **==** expression to specify that the parameter should not be set

If are using one of the above expressions, you can specify the type of parameter by using the **==: <type>** syntax for the **out** and **return** value or **<->: <type>** for the **in** value.

<return_val> can also refer to an Ada exception name introduced by the following syntax:

[: *<return_type>*] **RAISE** *<exception_name>*

where : *<return_type>* is used to specify the function returned type in case of overloading.

You must describe at least one call in the **STUB** instruction. Several descriptions can occur separated by commas.

STUB instructions can appear in **ELEMENT** blocks.

Example

```
STUB open_file ("file1")3
```

```
STUB create_file ("file2")4
```

```
STUB read_file 1..2 =>(3,"line 1",1)1,(3,"line 2",2<->3)1,
```

```
& 4..7 =>(3,"",0)0
```

```
STUB write_file (4,"line 1")1, (4,"line 2")1
```

```
STUB close_file (3)1,(4)1, (<->) RAISE DEVICE_ERROR
```

Related Topics

[DEFINE STUB ... END DEFINE on page 806](#)

TERMINATION ... END TERMINATION

Ada Test Script Language

Syntax

TERMINATION

END TERMINATION

Description

The **TERMINATION** and **END TERMINATION** instructions delimit a block of native code that is integrated into the generation process as the last *main* statements to be executed.

In some environments, such as when using a different target machine, this is a convenient way to exit the target.

Associated Rules

A **TERMINATION ... END TERMINATION** block must appear after the **BEGIN** instruction and outside any **SERVICE** block.

This block can only contain native Ada code. Each line of native code must be preceded with '#' or '@'.

There is no limit to the number of **TERMINATION** blocks. Upon test driver generation, they are concatenated in the order in which they appeared in the test script.

Related Topics

[INITIALIZATION on page 815](#)

TEST ... END TEST

Ada Test Script Language

Syntax

TEST <test_name> [**LOOP** <nb>]

END TEST

Description

The **TEST** instruction starts a **TEST** block. This block describes the test case for a service. It contains one more **ELEMENT** blocks specifying the test.

In the test report, the `<test_name>` parameter flags the test within the **SERVICE** block. Tests are usually given numbers in ascending order.

A number of iterations can be specified for each test with the optional **LOOP** keyword.

The **TEST LOOP** statement can generate graph metric results in a `.rtx` file. To do this, you must set the environment variable **ATURTX** to `True`. The produced `.rtx` graph can be viewed in the HCL OneTest Embedded Graphic Viewer.

The END TEST instruction marks the end of the **TEST** block.

Associated Rules

The **TEST** and **END TEST** instructions can only appear in a SERVICE block.

`<test_name>` is obligatory. If it is absent, the Test Script Compiler generates an error message.

`<nb>` is an integer expression strictly greater than 1.

Example

```
SERVICE COMPUTE_HISTO
```

```
# int x1, x2, y1, y2 : integer ;
```

```
# histo : T_HISTO ;
```

```
TEST 1
```

```
FAMILY nominal
```

```
ELEMENT
```

Related Topics

[ELEMENT on page 807](#) | [SERVICE on page 817](#)

VAR, ARRAY, and STR

Ada Test Script Language

Purpose

The **VAR**, **ARRAY**, and **STR** instructions declare the test of a simple variable, a variable array or a variable structure.

Syntax

```
VAR <variable>, <initialization>, <expected>
```

```
ARRAY <variable>, <initialization>, <expected>
```

```
STR <variable>, <initialization>, <expected>
```

where:

- `<variable>` is a [variable on page 825](#)
- `<initialization>` is an [initialization on page 826](#) parameter
- `<expected value>` is an [expected on page 830](#) parameter

Description

Use the **VAR**, **ARRAY**, and **STR** instructions to declare a variable test. During test execution, if the value of the variable is out of the bounds specified in the `<expected>` expression, the test is *Failed*.

VAR, **ARRAY** or **STR** are synonymous and do not change the way in which the result displayed in the test report.

- **VAR**: For simple variables.
- **ARRAY**: For variable arrays.
- **STR**: For variable structures.

If you use a **VAR** statement to test an array or structure, the report lists each element of the array or structure.

The **VAR**, **ARRAY**, and **STR** instructions must be located in an **ELEMENT** or an **ENVIRONMENT** block.

If a **TEST** block does not contain a **VAR**, **ARRAY**, or **STR** instruction, it is reported as an empty test. The **STUB** instruction is not considered as part of the the **TEST** as **STUBs** are always tested whether there is a **STUB** statement present or not.

Related Topics

[VAR, ARRAY and STR <variable> Parameter on page 825](#) | [VAR, ARRAY and STR <initialization> Parameter on page 826](#) | [VAR, ARRAY and STR <expected> Parameter on page 830](#)

VAR, ARRAY and STR <variable> Parameter

Description

In conjunction with the **VAR**, **ARRAY** and **STR** keywords, the `<variable>` parameter for Ada is a conventional notation name for an Ada variable under test.

Associated Rules

`<variable>` can be a simple variable (integer, floating-point number, character, pointer, character string, ...), an element of an array or record, part of an array, an entire array, or a complete record.

If the variable is an array for which no test boundaries have been specified, all the array elements are tested. Similarly, if the variable is a record of which one of the fields is an array, all elements of this field are tested.

Brackets or parentheses can be used to index array variables.

The variable must have been declared in Ada before it is used in the **.ptu** test script.

Example

VAR x, ...

VAR y(4), ...

VAR z.field, ...

VAR p.value, ...

ARRAY y(0..100), ...

ARRAY y, ...

STR z, ...

STR p.all, ...

Related Topics

[VAR, ARRAY and STR on page 824](#) | [VAR, ARRAY and STR <expected> Parameter on page 830](#) | [VAR, ARRAY and STR <initialization> Parameter on page 826](#)

VAR, ARRAY and STR <initialization> Parameter

In conjunction with the **VAR**, **ARRAY** and **STR** keywords, the *<initialization>* parameters for Ada Test Script Language specify the initial value of the variable.

Syntax

```
INIT = <exp>
INIT IN ( <exp>, <exp>, ... )
INIT ( <variable> ) WITH ( <exp>, <exp>, ... )
INIT FROM <exp> TO <exp> [STEP <exp> | NB_TIMES <nb> | NB_RANDOM <nb>[+ BOUNDS]]
INIT FROM <exp> TO <exp> [STEP <exp> | NB_VALUE <nb> | NB_RANDOM <nb>[+ BOUNDS]]
INIT ==
```

where:

- *<exp>* is an expression as described below.
- *<nb>* is an integer constant that is either literal or derived from an expression containing native constants.
- *<variable>* is an Ada variable.

Description

The *<initialization>* expressions are used to assign an initial value to a variable. The initial value is displayed in the Component Testing report for Ada.

The **INIT** value is calculated during the pre-processing phase, not dynamically during test execution.

Initializations can be expressed in the following ways:

- **INIT** = *<exp>* initializes a variable before the test with the value *<expression>*.
- **INIT IN** { *<exp>* , *<exp>* , ... } declares a list of initial values. This is a condensed form of writing that enables several tests to be contained within a single instruction.
- **INIT** (*<variable>*) **WITH** { *<exp>* , *<exp>* , ... } declares a list of initial values that is assigned in correlation with those of the variable initialized by an **INIT IN** instruction. There must be the same number of initial values.
- **INIT FROM** *<lower>* **TO** *<upper>* allows the initial value of a numeric variable (integer or floating-point) to vary between lower and upper boundary limits:
- **STEP**: the value varies by successive steps.
- **NB_TIMES** *<nb>* or **NB_VALUE** *<nb>*: the value varies by a number *<nb>* of values that are equidistant between the two boundaries, where *<nb>* \geq 2 (**NB_TIMES** and **NB_VALUE** are equivalent keywords). This option requires that the target platform supports floating point numbers.
- **NB_RANDOM** *<nb>*: the value varies by generating random values between the 2 boundaries, including, when appropriate, the boundaries, where *<nb>* \geq 1.
- **BOUNDS**: When you enter the '+ BOUNDS' instruction after 'NB_RANDOM nb', two numerical values are added to the nb (number) values.



Note:

- The **INIT IN** and **INIT** (*<variable>*) **WITH** expressions cannot be used with for arrays that were initialized in extended mode or for structures.
- The **INIT FROM** expression can only be used for numeric variables.
- The **STEP** syntax cannot be used when the same variable is tested by another **VAR**, **ARRAY** or **STR** statement.
- The **NB_TIMES**, **NB_VALUE**, and **NB_RANDOM** keywords require that the target platform supports floating point numbers.
- An initialization expression can still be used (**INIT** == *<expression>*) to include of expected value expression when using the **INIT** pseudo-variable is used. See [Expected_Value Expressions on page 830](#).
- The following syntaxes cannot be used in an **ARRAY** instruction:



- INIT FROM <exp> TO <exp> STEP <exp> ,
- INIT FROM <exp> TO <exp> NB_TIMES <nb> ,
- INIT FROM <exp> TO <exp> NB_VALUE <nb> ,
- INIT FROM <exp> TO <exp>NB_RANDOM <nb> ,
- INIT FROM <exp> TO <exp>NB_RANDOM <nb>[+ BOUNDS]

Expressions

The initialization expressions <exp> can be among any of the following values:

- Numeric (integer or floating-point), character, or character string literal values. Strings can be delimited by single or double quotes.
- Native constants, which can be numeric, characters, or character strings.
- Variables belonging to the test program or the module to be tested.
- Ada functions.
- The keyword **NIL** to designate a null pointer.
- Pseudo-variables **I**, **I1**, **I2** ..., **J**, **J1**, **J2** ..., where **I** *n* is the current index of the *n*th dimension of the parameter and **J** *m* the current number of the subtest generated by the test scenario's *m*th **INIT IN**, **INIT FROM** or **LOOP**; the **I** and **I1** variables are therefore equivalent as are **J** and **J1**; the subtest numbers begin at **1** and are incremented by **1** at each iteration.
- An Ada expression with one or more of the above elements combined using any operators and casting, with all required levels of parentheses, the + operator being allowed to concatenate character string variables.
- For arrays and structures, any of the above-mentioned expressions between brackets ([]) for Ada, including when appropriate:
 - For an array element, part of an array or a structure field, its index, interval or name followed by '=>' and by the value of the array element, common to all elements of the array portion or structure field.
 - For structures you can test some fields only, by using the following syntax:


```
[ <fieldname> => <value> , <fieldname> => <value> ]
```
- The keyword **others** (written in lower case) followed by '=>' and the default value of any array elements or structure fields not yet mentioned.
- For **INIT IN** and **INIT WITH** only, a list of values delimited by brackets ([]) for Ada composed of any of the previously defined expressions.

Additional Rules

- Any integers contained in an expression must be written either in accordance with native lexical rules, or under the form:
 - *<hex_integer>* **H** for hexadecimal values. In this case, the integer must be preceded by **0** if it begins with a letter.
 - *<binary_integer>* **B** for binary values.



Note: Because of the way hexadecimal values are handled, the value range should not exceed half of the maximum range when the initialization is expressed in hexadecimal.

- The number of values inside an **INIT IN** parameter is limited to 100 elements in a single VAR statement.
- The number of **INIT IN** parameters per TEST LOOP block is limited to 7.
- The number of **INIT IN** parameters per TEST block is limited to 8.
- In Component Testing for Ada, if variables are used in the expression, then the test evaluate the the **INIT** value with variable values from before the execution.

Examples

```
VAR x, INIT = pi/4-1, ...
VAR y[4], INIT IN ( 0, 1, 2, 3 ), ...
VAR y[5], INIT(y[4]) WITH ( 10, 11, 12, 13 ), ...
VAR z.field, INIT FROM 0 TO 100 NB_RANDOM 3, ...
VAR p->value, INIT ==, ...
ARRAY y[0..100], INIT = sin(I), ...
ARRAY y, INIT = (50=>10,others=>0), ...
STR z, INIT = (0, "", NIL), ...
STR *p, INIT = (value=>4.9, valid=>1), ...
```

In the following example, the Ada test Script Compiler generates code that tests *x* against *a* then *b* before the execution of the code under test:

```
VAR y, init in (1,2), ev = init
VAR a, init(y) with ( 10, 20 ), ev = 50
VAR b, init(y) with ( 30, 40 ), ev = 70
VAR x, init(y) with (a, b), ev = init
#a := 50;
#b := 70;
```

Related Topics

[VAR, ARRAY and STR on page 824](#) | [VAR, ARRAY and STR <variable> Parameter on page 825](#) | [VAR, ARRAY and STR <expected> Parameter on page 830](#)

VAR, ARRAY and STR <expected> Parameter

Purpose

In conjunction with the **VAR**, **ARRAY** and **STR** keywords, the <expected value> parameters for Ada Test Script Language specify the expected value of a variable.

Syntax

```
EV = <exp>
EV = <exp> , DELTA = <delta>
MIN = <exp>, MAX = <exp>
EV IN ( <exp>, <exp>, ... )
EV ( <variable> ) IN ( <exp>, <exp>, ... )
EV = INIT
EV ==
```

where:

- <exp> can be any of the expressions of the [Initialization Parameters on page 826](#), plus the following expressions:
- <delta> is the acceptable tolerance of the expected value and can be expressed:
- <variable> is an Ada variable

Description

The **EV** expressions are used to specify a test criteria by comparison with the value of a variable. The test is considered Passed when the actual value matches the <expected value> expression.

The **EV** value is calculated during the preprocessing phase, not dynamically during test execution.

An acceptable tolerance <delta> can be expressed:

- As an absolute value, by a numerical expression in the form described above
- As a percentage of the expected value. Tolerance is then written in the form <exp> %.

Expected values can be expressed in the following ways:

- **EV = <exp>** specifies the expected value of the variable when it is known in advance. The value of variable is considered correct if it is equal to <exp>.
- **EV = <exp>, DELTA = <tolerance>** allows a tolerance for the expected value. The value of variable is considered correct if it lies between <exp> - <tolerance> and <exp> + <tolerance>.
- **MIN = <exp>** and **MAX = <exp>** specify an interval delimited by an upper and lower limit. The value of the variable is considered correct if it lies between the two expressions. Characters and character strings are treated in dictionary order.

EV IN (*<exp>*, *<exp>*, ...) specifies the values expected successively, in accordance with the initial values, for a variable that is declared in **INIT IN**. It is therefore essential that the two lists have an identical number of values.

EV (<variable>) IN is identical to **EV IN**, but the expected values are a function of another variable that has previously been declared in **INIT IN**. As for **EV IN**, the two lists must have an identical number of values.

EV == allows the value of *<variable>* not to be checked at the end of the test. Instead, this value is read and displayed. The value of *<variable>* is always considered correct.

Expressions

The initialization expressions *<exp>* can be among any of the following values:

- Numeric (integer or floating-point), character, or character string literal values. Strings can be delimited by single or double quotes
- Native constants, which can be numeric, characters, or character strings
- Variables belonging to the test program or the module to be tested
- Ada functions
- The keyword **NIL** to designate a null pointer
- The keyword **NONIL**, which tests if a pointer is non-null
- Pseudo-variables **I**, **I1**, **I2** ..., **J**, **J1**, **J2** ..., where **I** *n* is the current index of the *n*th dimension of the parameter and **J** *m* the current number of the subtest generated by the test scenario's *m*th **INIT IN**, **INIT FROM** or **LOOP**; the **I** and **I1** variables are therefore equivalent as are **J** and **J1**; the subtest numbers begin at **1** and are incremented by **1** at each iteration
- An Ada expression with one or more of the above elements combined using any operators and casting, with all required levels of parentheses, the + operator being allowed to concatenate character string variables
- For arrays and structures, any of the above-mentioned expressions between brackets (**[]**) for Ada, including when appropriate:
 - For an array element, part of an array or a structure field, its index, interval or name followed by **'=>'** and by the value of the array element, common to all elements of the array portion or structure field
 - For structures you can test some fields only, by using the following syntax:
 - **[<fieldname> => <value> , <fieldname> => <value>]**

- The keyword **others** (written in lower case) followed by '=' and the default value of any array elements or structure fields not yet mentioned
- The pseudo-variable **INIT**, which copies the initialization expression. You cannot use the pseudo-variable **INIT** inside an array or structure. The keyword **INIT** applies to the entire expression.



Note: The following syntaxes cannot be used in an **ARRAY** instruction:

```
EV IN ( <exp>, <exp>, ... )
EV ( <variable> ) IN ( <exp>, <exp>, ... )
```

Additional rules

EV with **DELTA** is only allowed for numeric variables. The **STR** statement does not support **DELTA**.

MIN = <exp> and **MAX = <exp>** are only allowed for alphanumeric variables that use lexicographical order for characters and character strings.

MIN = <exp> and **MAX = <exp>** are not allowed for pointers.

Only **EV =** and **EV ==** are allowed for structured variables.

In some cases, in order to avoid generated code compilation warnings, the word **CAST** must be inserted before the **NIL** or **NONIL** keywords.

Example

```
VAR x, ..., EV = pi/4-1
VAR y[4], ..., EV IN (0, 1, 2, 3 )
VAR y[5], ..., EV(y[4]) IN ) (10, 11, 12, 13 )
VAR z.field, ..., MIN = 0, MAX = 100
VAR p->value, ..., EV ==
ARRAY y[0..100], ..., EV = cos(I)
ARRAY y, ..., EV = (50=>10,others=>0)
STR z, ..., EV = (0, "", NIL)
STR *p, ..., EV = (value=>4.9, valid=>1)
```

Related Topics

[VAR, ARRAY and STR on page 824](#) | [VAR, ARRAY and STR <expected> Parameter on page 830](#) | [VAR, ARRAY and STR <variable> Parameter on page 825](#)

Requirement

Purpose

The **Requirement** instruction allows the testers to link a test or a set of tests to one or a set of requirements. **Requirement** is optional.

Syntax

```
REQUIREMENT <requirement_name> {, [<attribute_name> =|:] <attribute_value>}
```

Argument

<requirement_name> is a mandatory identifier indicating the name of the requirement.

<attribute_name> is the name of one attribute of the requirement. It is an identifier.

<attribute_value> is the value of the attribute. The syntax may be: \$<identifier>. In this case, the attribute value is substituted with the content of an environment variable whose name is \$<identifier>.

Description

The **REQUIREMENT** instruction appears within **TEST** blocks, where it defines the requirements for this test or within **SERVICE** blocks where it defines the requirements for the tests including in this service or before the first **SERVICE** block where it defines the requirements for the all the tests in the file.

Requirements are cumulative between test and service.

rod2req is a binary that generates an XML file analyzing the rod files and describing the tracability matrix between tests and requirements with pass/failed status.

Example

```
TEST1
FAMILY nominal
REQUIREMENT req1, req2
COMMENT histogram computation on a black image
ELEMENT
```

C System Testing

System Testing driver script (.pts)

This section describes each System Testing driver script instruction, including:

- Syntax
- Functionality and rules governing its usage
- Examples of use

Notation Conventions

Throughout this guide, command notation and argument parameters use the following standard convention:

Notation	Example	Meaning
----------	---------	---------

BOLD	<code>ADD_ID</code>	Language keyword
<i><italic></i>	<code><filename></code>	Symbolic variables
[]	[<i><option></i>]	Optional items
{ }	{ <i><filenames></i> }	Series of values
[{ }]	[{ <i><file- names></i> }]	Optional series of vari- ables
	on off	OR operator

System test script keywords are case sensitive. All keywords must be entered in upper case.

For conventional purposes however, this document uses upper-case notation for the supervisor script keywords in order to differentiate from native source code.

Split statements

Statements may be split over several lines in a **.spv** supervisor script. Continued lines must start with the ampersand (&) symbol to be recognized as a continuation of the previous line. No tabs or spaces should precede the ampersand.

Statements must be shorter than 2048 characters, although this limit may be lower on some platforms.

Identifiers

A supervisor script identifier is a text string used as a label, such as the name of a message type.

Identifiers are made of an unlimited sequence of the following characters:

- a-z
- A-Z
- 0-9
- _ (underscore)

Spaces are not valid identifier characters.

System Testing keywords and identifiers are case sensitive. This means that **LABEL**, **label**, and **Label** are three different identifiers.

Related Topics

[System Testing driver script structure on page 835](#) | [System Testing driver script keywords on page 836](#) | [System Testing for C overview on page 586](#)

System Test Script keywords (PTS)

System Testing for C helps you solve complex testing issues related to system interaction, concurrency, and time and fault tolerance by addressing the functional, robustness, load, performance and regression testing phases from small, single threads or tasks up to very large, distributed systems. Test script file names must contain only plain alphanumerical characters. System Testing for C

Basic structure

A typical System Testing **.pts** test driver script is presented as follows:

```
HEADER "Registering", "1.0", "1.0"
```

```
SCENARIO basic_registration
```

```
FAMILY nominal
```

```
-- The body of my basic_registration test
```

```
END SCENARIO
```

```
SCENARIO extended_registration
```

```
FAMILY robustness
```

```
SCENARIO reg_priv_area
```

```
-- The body of my reg_priv_area test
```

```
END SCENARIO -- reg_priv_area
```

```
SCENARIO reg_pub_area LOOP 10
```

```
-- The body of my reg_pub_area test
```

```
END SCENARIO -- reg_priv_area
```

```
END SCENARIO
```

The overall structure of a C system test script must follow these rules:

- A test script always starts with the **HEADER** keyword.
- A test script is composed of one or several scenarios.
- All statements begin with a keyword.
- Statements are not case sensitive (except when C expressions are used).

- Statements start at the beginning of a line and end at the end of a line. You can, however, write an instruction over several lines using the ampersand (&) continuation character at the beginning of additional lines. In this case, the ampersand must be the very first character on that line; no spaces or tabs should precede it.
- Statements must be shorter than 2048 characters, although this limit may be lower on some platforms.

Structuring statements

The basic structuring statements are:

- **HEADER:** Specifies the name of the test script, the version of the tested system, and the version of the test script. This information will be included in the test report.
- **SCENARIO:** Indicates the beginning of a **SCENARIO** block. A **SCENARIO** block ends with an **END SCENARIO** statement. A **SCENARIO** block can be iterated multiple times using the **LOOP** keyword.
- **FAMILY:** Qualifies the scenario and all its sub-scenarios. The **FAMILY** attribute is optional. A list of qualifiers can be given such as: **FAMILY nominal, structural**.

Each scenario can be split into sub-scenarios.

Related information

[System Testing driver script \(.pts\) on page 833](#)

[System Testing driver script keywords on page 836](#)

[HEADER on page 855](#)

[SCENARIO ... LOOP ... END SCENARIO on page 876](#)

[FAMILY on page 852](#)

System Testing driver script keywords

Flow control instructions

- [CALL on page 839](#)
- [INCLUDE on page 857](#)
- [CASE on page 842](#)
- [IF on page 855](#)
- [PROC on page 872](#)
- [WHILE on page 890](#)

Adaptation layer instructions

- [ADD_ID](#) on page 838
- [COMMTYPE](#) on page 845
- [CHANNEL](#) on page 843
- [MESSAGE](#) on page 864
- [PROCSEND](#) on page 873
- [VAR](#) on page 883
- [SEND](#) on page 877
- [CALLBACK](#) on page 840
- [DEF_MESSAGE](#) on page 847
- [WAITTIL](#) on page 889
- [VIRTUAL CALLBACK](#) on page 885
- [VIRTUAL PROCSEND](#) on page 887
- [INTERSEND](#) on page 860
- [INTERRECV](#) on page 861

Instance instructions

- [DECLARE_INSTANCE](#) on page 846
- [INSTANCE](#) on page 858
- [RENDEZVOUS](#) on page 875

Environment instructions

- [ERROR](#) on page 849
- [EXIT](#) on page 851
- [EXCEPTION](#) on page 850
- [INITIALIZATION](#) on page 858
- [TERMINATION](#) on page 879

Time management instructions

- [TIME on page 880](#)
- [TIMER on page 880](#)
- [WTIME on page 891](#)
- [RESET on page 875](#)
- [PRINT on page 871](#)
- [PAUSE on page 870](#)

ADD_ID

System Testing Test Script Language.

Syntax

ADD_ID (<channel_identifier>, <connection_identifier>)

Description

The **ADD_ID** instruction dynamically adds the value of a connection identifier to a communication channel identifier.

A communication channel is a logical medium that integrates (multiplexes) the same type of connection between the virtual tester and remote applications under test.

When opening a connection with your communication API, you must dynamically link the connection identifier with a channel identifier.

You must declare a channel identifier with the **CHANNEL** instruction.

C connection identifiers must be compatible with C communication channels.

Examples

...

```
COMMTYPE ux_inet IS integer_t
```

```
CHANNEL ux_inet: ch
```

...

```
SCENARIO First
```

...

```
#integer_t id;
```

```
CALL socket(AF_UNIX, SOCK_STREAM, 0) @@ id
```

```
ADD_ID(ch, id)
```

```
....
```

Related Topics

[CHANNEL on page 843](#) | [CLEAR_ID on page 844](#) | [WAITTIL on page 889](#)

CALL

System Testing Test Script Language.

Syntax

```
CALL <identifier> ( <arguments> ) [ @ [ <expected_expr> ] @ [ <return_var> ] ]
```

Description

The **CALL** instruction lets you call a specific interface routine. This routine may be a function or a procedure.

You can check a function's return values for interface routine calls.

The @ character is a separator.

<expected_expr> gives the expected return value of the function.

<return_var> gives the variable in which the return value of the function is stored.

If <return_var> is specified, the return value is stored in <return_var>.

The **CALL** instruction can be used in a **PROC**, **SCENARIO**, **INITIALIZATION**, **TERMINATION**, or **EXCEPTION** blocks.

Example

```
#int return_val;
```

```
#int V_in;
```

```
#int V1_out, V2_out;
```

```
SCENARIO TEST_1
```

```
FAMILY nominal
```

```
...
```

```
CALL APL_function(V_in, REF(@0@V1_out),&1@0@V2_out)@1@return_val
```

...

Related Topics

[Function calls on page 599](#)

CALLBACK ... END CALLBACK

System Testing Test Script Language.

Purpose

The **CALLBACK** instruction dynamically recalls message reception and links a connection identifier value to a communication channel identifier.

Syntax

```
CALLBACK <message_type> : <msg> ON <commtype>: <id> [ <n> ]
```

```
END CALLBACK
```

<message_type> is a message type previously declared in a **MESSAGE** statement.

<msg> is the output parameter of <message_type> that must be initialized in the callback if a message is received.

<commtype> is the type of communication used for reading messages previously declared in a **COMMTYPE** statement.

<id> is the input connection parameter on which a message must be read.

Description

Callbacks are declared in the first part of the test script, before the first scenario.

<commtype> must be declared with the **COMMTYPE** instruction.

<message_type> must be declared with the **MESSAGE** instruction.

You can declare only one callback per combination of message and communication type.

Message reception in the **CALLBACK** statement must never be blocked. If no message is received, you must exit the block using the **NO_MESSAGE** instruction.

Use of both a **NO_MESSAGE** and **MESSAGE_DATE** statement is mandatory within the callback or a procedure called from a callback.

If the C <message_type> contains *unions*, you can define for each union the display and comparison field. The system implicitly defines a structured variable, named as **ATL_** followed by the name of the <message_type>. You can specify which field to use by specifying *select* attribute for the union.

Freeze Mode

Freeze mode is a blocking mode in which the **CALLBACK** waits for a message to be received. To use *freeze mode*, you must use only one **CALLBACK** block throughout the entire **WAITTIL** statement, messages can be read in *freeze mode*. In this mode, the **ATL_TIMEOUT** macro specifies the maximum wait delay for a message. The value of **ATL_TIMEOUT** is calculated from a **WTIME** expression used in the **WAITTIL** statement. Only **WTIME** must be specified in the **WAITTIL** statement. The **ATL_TIMEOUT** macro is an integer and uses the time unit defined in the Target Deployment Port. By default, the time unit is a hundredth of second.

Example

```
typedef enum { e_name, e_id, e_balance } client_kind_t;

typedef struct {

client_kind_t kind;

union {

char name[50];

int id;

float balance;

} my_union;

} client_info_t;

COMMTYPE socket IS socket_id_t

CHANNEL socket: ch

MESSAGE client_info_t: msg

CALLBACK client_info_t: info ON socket: id

CALL read(id, &info, sizeof(client_info_t))@@ret

IF (ret == 0) THEN

NO_MESSAGE

END IF

MESSAGE_DATE

VAR ATL_client_info_t.my_union.select, INIT=info.kind

END CALLBACK
```

Related Topics

[COMMTYPE on page 845](#), [MESSAGE on page 864](#), [WAITTIL on page 889](#), [MESSAGE_DATE on page 865](#), [NO MESSAGE on page 869](#), [VIRTUAL CALLBACK on page 885](#)

CASE ... IS ... WHEN OTHERS... END CASE

System Testing Test Script Language.

Syntax

```
CASE <expression> IS
```

```
WHEN <constant1> => <instructions>
```

```
WHEN <constant2> => <instructions>
```

```
WHEN <constant3> => <instructions>
```

```
WHEN OTHERS => <instructions>
```

```
END CASE
```

Description

The **CASE** instruction allows you to choose one of several sets of instructions according to the value of an expression.

The **CASE** instruction may appear in a **PROC**, **SCENARIO**, **INITIALIZATION**, **TERMINATION** or **EXCEPTION** block.

The list of options for the expression begins after **IS** and ends in **END CASE**.

WHEN identifies the different constant expressions that cause a specific process to be carried out. This process is defined by the instructions following the => symbol.

OTHERS processes all the values of expression that have not been explicitly processed in the **CASE**. This instruction set is optional.

<expression> must take an integer value.

Examples

```
##define ACK 0
```

```
##define NACK 1
```

```
#int choice;
```

```
SCENARIO TEST_1
```

```

FAMILY nominal

CALL ApiGetChoice(choice)

CASE (choice) IS

WHEN ACK => CALL ApiAcknowledge()

WHEN NACK => CALL ApiReset()

...

WHEN OTHERS => CALL Api_DefaultMsg()

END CASE

```

...

Related Topics

[Multiple Conditions on page 603](#) | [Conditions on page 601](#)

CHANNEL

System Testing Test Script Language.

Syntax

```
CHANNEL <communication_type> : <channel> {[, <channel> ]}
```

Description

The **CHANNEL** instruction allows you to declare a set of communication channels.

You must declare the <communication_type> with the **COMMTYPE** instruction.

Each <channel> variable identifies a new type of communication channel. A communication channel is a logical medium that integrates (multiplexes) the same type of connection among virtual testers and remote applications under test.

Use the **CHANNEL** instruction at the beginning of the test script, before the first scenario.

Examples

```

#ifdef int inet_id_t;

COMMTYPE ux_inet IS inet_id_t WITH MULTIPLEXING

CHANNEL ux_inet: ch_1, ch_2, ch_3

CHANNEL ux_inet: ch_out

```

Related Topics

[COMMTYPE on page 845](#) | [WAITTIL on page 889](#)

CLEAR_ID

System Testing Test Script Language.

Syntax

CLEAR_ID (<channel_identifier>)

Description

The **CLEAR_ID** instruction clears a communication channel.

The communication channel has no more links with remote applications under test.

You must declare a communication channel with the **CHANNEL** instruction.

Example

...

```
COMMTYPE ux_inet IS integer_t
```

```
CHANNEL ux_inet: ch
```

...

```
SCENARIO First
```

...

```
#integer_t id;
```

```
CALL socket(AF_UNIX, SOCK_STREAM, 0) @@ id
```

```
ADD_ID(id,ch)
```

...

```
CLEAR_ID(ch)
```

....

Related Topics

[ADD_ID on page 838](#) | [CHANNEL on page 843](#) | [WAITTIL on page 889](#)

COMMENT

COMMENT

System Testing Test Script Language.

Syntax

COMMENT

Description

The **COMMENT** instruction allows you to add comments to the results file by inserting text.

Its use in test scenarios is optional.

The position of the **COMMENT** instruction in the test program defines the position in which the comment appears in the test report.

The **COMMENT** instruction may appear in a **PROC**, **SCENARIO**, **INITIALIZATION**, **TERMINATION** or **EXCEPTION** block.

In the command line interface, you can deactivate the processing of comments by adding the **-NOCOMMENT** option to the [C Test Script Compiler on page 1112](#) command line.

Example

```
SCENARIO TEST_1
```

```
FAMILY nominal
```

```
COMMENT calling connection confirmation
```

```
CALL api_trspmt_connectionCF()
```

```
...
```

COMMTYPE

System Testing Test Script Language.

Syntax

```
COMMTYPE <identifier> IS <connection_id_type> [WITH MULTIPLEXING]
```

Description

The **COMMTYPE** instruction defines a type of communication. The C connection identifies the communication type.

The C <connection_id_type> must be a *typedef*, as defined in the interface file, an included file, or in the test script.

You can define the communication type as being able to multiplex connections for the read operation, using the multiplexing option.

You must use the **COMMTYPE** instruction at the beginning of the test script, before the first scenario.

Example

...

```
#typedef int inet_id_t;
```

```
COMMTYPE ux_inet IS inet_id_t WITH MULTIPLEXING
```

```
#typedef struct { int key; int id; } msgqueue_id_t;
```

```
COMMTYPE ux_msgqueue IS msgqueue_id_t
```

....

Related Topics

[CALLBACK on page 840](#)

DECLARE_INSTANCE

System Testing Test Script Language.

Syntax

```
DECLARE_INSTANCE <instance> {[,<instance>]}
```

Description

The **DECLARE_INSTANCE** instruction allows you to define the set of the instances described in the test script.

A **DECLARE_INSTANCE** instruction takes effect after you have declared it.

<instance> may be any identifier. The **DECLARE_INSTANCE** must have at least one instance name passed by parameter.

Example

```
HEADER "DEMO SOCKET", "1.0", "2.4"
```

```
DECLARE_INSTANCE client, server
```

```
SCENARIO Main
```

...

END SCENARIO

Related Topics

INSTANCE

DEF_MESSAGE

System Testing Test Script Language.

Syntax

DEF_MESSAGE <message>, **EV=** <cmp_expression>

Description

The **DEF_MESSAGE** instruction allows you to define a reference <message> variable. In order to do this, you must define the reference values with <cmp_expression>.

The message variable is the reference event variable initialized by the **DEF_MESSAGE** instruction. It has to be declared by the **MESSAGE** instruction.

Associated Rules

The **DEF_MESSAGE** instruction can appear in a **PROC**, **SCENARIO**, **INITIALIZATION**, **TERMINATION**, or **EXCEPTION** block.

You may partially define a reference message. The undefined <cmp_expression> fields are not used to compare incoming messages.

Interface File

```
typedef struct {
    int type;
    struct {
        char app_name[8];
        unsigned char class_name;
    } data;
    char userdata[100];
} message_t;
```

Example

MESSAGE message_t: msg

SCENARIO first

DEF_MESSAGE msg, EV= { code=>ConnectCF,

& data=>{ app_name=>"ATCMKD" }}

Related Topics

[MESSAGE on page 864](#) | [WAITTIL on page 889](#) | [VAR on page 883](#)

END

System Testing Test Script Language.

Syntax

END *<block>*

Description

The **END** instruction delimits an instruction block.

You use it to end the following:

- A callback: **END CALLBACK**
- A procedure: **END PROC**
- A message sending procedure: **END PROCSEND**
- An initialization block: **END INITIALIZATION**
- A termination block: **END TERMINATION**
- An exception block: **END EXCEPTION**
- A scenario: **END SCENARIO**
- An instance block: **END INSTANCE**
- A **CASE** instruction: **END CASE**
- An **IF** instruction: **END IF**
- A **WHILE** instruction: **END WHILE**

Example

```
INSTANCE tester1, tester2:
```

```

PROC clean1
...
END PROC
...
END INSTANCE
INITIALIZATION
...
END INITIALIZATION
SCENARIO TEST1
...
END SCENARIO

```

Related Topics

[CALLBACK on page 840](#) | [PROC on page 872](#) | [PROCSSEND on page 873](#) | [INITIALIZATION on page 858](#) | [TERMINATION on page 879](#) | [EXCEPTION on page 850](#) | [SCENARIO on page 876](#) | [INSTANCE on page 858](#) | [CASE on page 842](#) | [IF...THEN...ELSE on page 855](#) | [WHILE on page 890](#)

ERROR

System Testing Test Script Language.

Syntax

```
ERROR
```

Description

When an unexpected output value for a function or a **WAITTIL** causes a problem, the current scenario halts as a result. You may terminate the scenario deliberately with the **ERROR** instruction.

After an **ERROR** instruction, the **EXCEPTION** block is executed on the next scenario at the same level, if there is one.

Example

```
#int sock;
```

```
...
```

```
SCENARIO Main
```

```
SCENARIO Test1
```

```
...
```

```
IF (sock==1) THEN
```

```
ERROR
```

```
END IF
```

```
...
```

```
END SCENARIO
```

```
SCENARIO Test2
```

```
...
```

```
CALL ...
```

```
...
```

```
END SCENARIO
```

```
END SCENARIO
```

In the above example, you can stop the **Test1** scenario with the **ERROR** instruction. The virtual tester then proceeds to **Test2** scenario.

Related Topics

[EXIT on page 851](#)

EXCEPTION ... END EXCEPTION

System Testing Test Script Language.

Syntax

```
EXCEPTION [ <proc> ( [ <arg> { [, <arg> } ] ] ) ]
```

```
END EXCEPTION
```

Description

The **EXCEPTION** instruction or block deletes a specific environment by executing the set of instructions or the procedure <proc>. **END EXCEPTION** marks the end of the **EXCEPTION** block.

Associated Rules

An **EXCEPTION** block or instruction applies to the set of scenarios at its level.

It does not apply to subscenarios of these scenarios.

The **EXCEPTION** instruction or block is optional.

A maximum of one **EXCEPTION** block may occur in a scenario.

The **EXCEPTION** instruction is only executed if a scenario terminates with an error.

It does not matter where the **EXCEPTION** instruction is placed among scenarios in a given level.

Example

```
#int sock;
```

```
EXCEPTION
```

```
CALL close (sock)
```

```
...
```

```
END EXCEPTION
```

```
...
```

```
SCENARIO Main
```

```
...
```

```
END SCENARIO
```

Related Topics

[INITIALIZATION](#) on page 858 | [TERMINATION](#) on page 879

EXIT

System Testing Test Script Language.

Syntax

```
EXIT
```

Description

This instruction lets you exit from the virtual tester. It causes all scenarios to terminate.

After an **EXIT**, the virtual tester terminates. For an **EXIT** instruction, the end of execution code of the virtual tester process is -1.

The scenario in which the **EXIT** instruction was executed is deemed incorrect.

Example

```
#int sock;  
  
...  
  
SCENARIO Main  
  
SCENARIO Test1  
  
...  
  
IF (sock==1) THEN  
  
COMMENT stop tester  
  
EXIT  
  
END IF  
  
...  
  
END SCENARIO  
  
SCENARIO Test2  
  
...  
  
CALL ...  
  
...  
  
END SCENARIO  
  
END SCENARIO
```

FAMILY

System Testing Test Script Language.

Syntax

FAMILY <family> {[, <family>]}

Description

The **FAMILY** instruction allows you to group tests by families or classes.

This instruction appears just once at the beginning of a **SCENARIO** block, where it defines the family or families to which the scenario belongs.

When starting tests, you can request to execute only tests of a given family.

The *<family>* parameter indicates the name of the test family. You can define the following families: nominal, structural, robustness.

A test can belong to several families: in this case, the **FAMILY** instruction contains a *<family>* list, separated by commas.

<family> can be any identifier. You must have at least one family name.

The **FAMILY** instruction is optional. If omitted, the test belongs to every family.

Example

```
SCENARIO Test_1
```

```
FAMILY nominal
```

```
COMMENT ...
```

```
...
```

```
END SCENARIO
```

FLUSH_TRACE

System Testing Test Script Language.

Syntax

```
FLUSH_TRACE
```

Description

The **FLUSH_TRACE** instruction dumps the execution traces stored in the circular buffer to the **.rio** file.

This instruction is taken into account only when the **-TRACE=CIRCULAR** Test Script Compiler option is set.

The **FLUSH_TRACE** instruction can be used in a **PROC**, **SCENARIO**, **INITIALIZATION**, **TERMINATION**, or **EXCEPTION** block. You may not use **FLUSH_TRACE** in a **CALLBACK** or **PROCSEND** block.

Example

```
SCENARIO one
```

```
(...)
```

```
FLUSH_TRACE
```

```
(...)
```

```
END SCENARIO
```

Related Topics

-TRACE=CIRCULAR, TRACE_ON, TRACE_OFF

FORMAT

System Testing Test Script Language.

Syntax

FORMAT <variable> = <format>

FORMAT <type> = <format>

FORMAT <field> = <format>

Description

This **FORMAT** instruction modifies the way a variable, type, or field of a structure is tested and printed. All formats of the same type is modified.

The new format is defined in C.

A format can also specify a print mode in binary or hexadecimal, using the options **#B** and **#H**.

The **FORMAT** instruction is optional. You may use it at the beginning of the test script or in a block of instructions, depending on the required scope. However, **FORMAT** statements that apply to data contained in a **CALLBACK** or **PROCSEND** block must be located before:

1. any **CALLBACK** or **PROCSEND** block
- any **PROC** statements that contain **DEF_MESSAGE** or **SEND** instructions

Example

SCENARIO first

```
#char buffer[100];
```

```
#typedef struct {
```

```
# int ax_register;
```

```
# int bx_register;
```

```
# int cx_register;
```

```
#} 8088_register_t;
```

```

FORMAT buffer = unsigned char[50]

FORMAT 8088_register_t.ax_register = #B

FORMAT 8088_register_t.bx_register = #H

END SCENARIO

```

HEADER

System Testing Test Script Language.

Syntax

```
HEADER <test_name>, <version>, <test_plan_version>
```

Description

This instruction allows you to define a standard header at the beginning of the test script. The information contained in this header enables you to identify a list of scenarios.

The headers can be strings or environment variables.

<test_name> is the name for the test script.

<version> is the version of the system tested.

<test_plan_version> is the test script version.

This instruction must appear before the first instruction block and strings must be enclosed in double-quotes (" ").

Example

```
HEADER "DEMO SOCKET", $VERSION, "2.4"
```

```
INITIALIZATION
```

```
...
```

```
END INITIALIZATION
```

```
SCENARIO Main
```

```
....
```

```
END SCENARIO
```

IF...THEN...ELSE

System Testing Test Script Language.

Syntax

```
IF <condition> THEN
```

```
ELSE
```

```
END IF
```

Description

This is a control statement. The simplest form of an **IF** instruction begins with the keyword **IF**, is followed by a Boolean expression, and then the keyword **THEN**. A set of instructions follows. These instructions are executed if the expression is true. The last **END IF** marks the end of the set of instructions.

Other actions can be executed depending on the value of the condition. Add an **ELSE** block, followed by the set of instructions to be executed if the condition is false.

IF may be placed anywhere in the test program.

THEN must be placed at the end of a line.

ELSE must be on its own line.

END IF must be on its own line.

Example

```
HEADER "DEMO SOCKET RPC";"1.0a", "2.5"
```

```
#int sock;
```

```
INITIALIZATION
```

```
...
```

```
IF (sock==1) THEN
```

```
ERROR
```

```
ELSE
```

```
CALL listen(sock,5)
```

```
...
```

```
END IF
```

```
...
```

```
END INITIALIZATION
```

SCENARIO Main

....

Related Topics

[Conditions on page 601](#)

INCLUDE

System Testing Test Script Language.

Syntax

INCLUDE <*string*>

Description

The **INCLUDE** instruction lets you include scenarios in the current test script.

Its use in test scenarios is optional.

The **INCLUDE** instruction may appear in any scenario as long as the scenario does not contain any primary instructions. <*string*> is the name of the file to be included. The system searches for files in the current directory and then searches the list of paths passed on to the Test Script Compiler.

Example

SCENARIO Test_1

FAMILY nominal

INCLUDE "../common/initialization"

INCLUDE "scenario_1_and_2"

SCENARIO scenario_3

COMMENT call connection

CALL api_trsprt_connexionCF()

CALL ...

END SCENARIO

END SCENARIO

INITIALIZATION ... END INITIALIZATION

System Testing Test Script Language.

Syntax

```
INITIALIZATION [ <proc> ( [ <arg> { , <arg> } ] ) ]
```

```
END INITIALIZATION
```

Description

The **INITIALIZATION** instruction initializes a specific environment by executing a set of instructions or the procedure <proc>. **END INITIALIZATION** marks the end of the **INITIALIZATION** block.

An **INITIALIZATION** block or instruction applies to the set of scenarios at its level. It does not apply to sub-scenarios.

The **INITIALIZATION** instruction or block is optional.

A maximum of one **INITIALIZATION** block or instruction may occur at a given scenario level.

This instruction is executed before every scenario at the same level.

The **INITIALIZATION** instruction may appear anywhere among scenarios at a given level.

Example

...

```
INITIALIZATION
```

```
CALL socket (AF_INET, SOCK_DGRAM, 0)@@ds
```

...

```
FD_ADD(ds,SOCKAPI)
```

```
END INITIALIZATION
```

...

Related Topics

[TERMINATION on page 879](#) | [EXCEPTION on page 850](#)

INSTANCE ... END INSTANCE

System Testing Test Script Language.

Syntax

```
INSTANCE <instance>[ [ , <instance> ] ]:
```

END INSTANCE

Description

An **INSTANCE ... END INSTANCE** block allows you to specify associated declarations or the instructions.

When the **INSTANCE ... END INSTANCE** block is located before the top-level scenarios, it gives global declarations to the test script for all the specified instances.

At the block or nested scenario level, it gives instructions or local declarations to the wrapping block or scenario.

You may not nest instance blocks.

You cannot mix declarations and instructions in the same instance block.

Instance blocks containing instructions follow instance blocks containing declarations.

Examples

```
HEADER "DEMO SOCKET", $VERSION, "2.4"
```

```
DECLARE_INSTANCE client, server
```

```
INSTANCE server:
```

```
#static int var_c_time ;
```

```
END INSTANCE
```

```
INITIALIZATION
```

```
INSTANCE server:
```

```
var_c_time = 0;
```

```
END INSTANCE
```

```
END INITIALIZATION
```

```
SCENARIO Principal
```

```
...
```

```
INSTANCE client:
```

```
#int connectStatus ;
```

```
END INSTANCE
```

```
...
```

```
INSTANCE server:
```

```
var_c_time = TIME(globalTime);
```

```
END INSTANCE
```

```
END SCENARIO
```

INTERSEND

System Testing Test Script Language.

Syntax

INTERSEND(*<integer>*, *<identifier>*)

INTERSEND(*<string>*, *<identifier>*)

<identifier> is the unique identifier of a virtual tester to which the message is to be sent.

<integer> is a 32-bit integer value.

<string> is a string-type value.

Description

The **INTERSEND** statement allows the virtual tester to send a simple message to another virtual tester. The other virtual tester receives the incoming message with the **INTERRECV** statement.

The message can be either an integer or a string.

<identifier> is *<instance_name>_<occid>* or *<test_script.rio>_<occid>*

The default value for *<occid>* is 0.

Example

```
INSTANCE JUPITER:
```

```
INTERSEND( "How many messages did you receive from SUT?", "SATURN_0" )
```

```
INTERRECV( &transmitted_int)
```

```
END INSTANCE
```

```
INSTANCE SATURN:
```

```
INTERRECV( buffer, 1024 )
```

```
INTERSEND( 2, "JUPITER_0" )
```

```
END INSTANCE
```

Related Topics

[INTERRECV\(\) on page 861](#) | [ATL_OCCID on page 891](#)

INTERRECV

System Testing Test Script Language.

Syntax

INTERRECV(*<integer_pointer>*)

INTERRECV(*<string_pointer>* , *<buffer size>*)

<integer_pointer> indicates the memory location of a 32-bit integer message.

<string_pointer> points to a static or allocated memory zone containing the incoming message.

<buffer size> is the size of the memory zone starting at *<string_pointer>*.

Description

The **INTERRECV** statement allows the virtual tester to receive a simple message sent by an **INTERSEND** statement from another virtual tester.

Received messages are stored in static or allocated memory zone indicated by *<integer_pointer>* or *<string_pointer>*.

The message can be either an integer or a string. However if the message type expected by the **INTERRECV** mismatches the actual message type sent by **INTERSEND**, System Testing for C attempts to convert the message.

Example

INSTANCE JUPITER:

```
INTERSEND( "How many messages did you receive from SUT?" , "SATURN_0" )
```

```
INTERRECV( &transmitted_int)
```

```
END INSTANCE
```

INSTANCE SATURN:

```
INTERRECV( buffer, 1024 )
```

```
INTERSEND( 2 , "JUPITER_0" )
```

```
END INSTANCE
```

Related Topics

[INTERSEND\(\) on page 860](#), [ATL_OCCID on page 891](#)

MATCHED

System Testing Test Script Language.

Syntax

```
MATCHED( <ref_msg> {[, <channel> ]})
```

Description

<ref_msg> is a reference message variable declared with the **MESSAGE** instruction and initialized with the **DEF_MESSAGE** instruction.

<channel> is a communication channel declared with the **CHANNEL** instruction and initialized by the **ADD_ID** instruction.

MATCHED is a function that returns a Boolean value. It returns true if one of the messages received during a **WAITTIL** matches the reference message <ref_msg>. If you specify a channel, it returns true only if the matching message was received on this channel.

It returns true if at least one received message has the same values as those defined for the reference message.

MATCHED is only meaningful when used in a **WAITTIL** instruction or in control statements following a **WAITTIL**, such as **IF**, **WHILE**, or **CASE**.

The **MATCHED** return value changes when you reuse it in a **WAITTIL** statement.

Examples

...

```
CHANNEL ux_socket: ch
```

```
SCENARIO Main
```

```
DEF_MESSAGE msg_1, EV={100,10}
```

```
DEF_MESSAGE msg_2, EV={200,20}
```

...

```
WAITTIL(MATCHED(msg_1) && MATCHED(msg_2,ch),WTIME==10)
```

...

```
IF (MATCHED(msg_1,ch)) THEN
```

...

Related Topics

[CHANNEL](#) on page 843 | [DEF_MESSAGE](#) on page 847 | [WAITTIL](#) on page 889 | [MATCHING\(\)](#) on page 863

MATCHING

System Testing Test Script Language.

Syntax

MATCHING(*<ref_msg>* {[, *<channel>*]})

Description

MATCHING is a function that returns a Boolean value. It returns true if the last message received during a **WAITTIL** matches the reference message *<ref_msg>*. If you specify a channel, it returns true only if the matching message was received on this channel.

<ref_msg> is a reference message variable declared with the **MESSAGE** instruction and initialized with the **DEF_MESSAGE** instruction.

<channel> is a communication channel declared with the **CHANNEL** instruction and initialized by the **ADD_ID** instruction.

It returns true if the last received message has the same values as those defined for the reference message.

Associated Rules

MATCHING is only meaningful when used in a **WAITTIL** instruction and in control statements following a **WAITTIL**, such as **IF**, **WHILE**, or **CASE**.

The **MATCHING** return value changes when you reuse it in a **WAITTIL**.

Examples

...

```
CHANNEL ux_socket: ch
```

```
SCENARIO Main
```

```
DEF_MESSAGE msg_1, EV={100,10}
```

```
DEF_MESSAGE msg_2, EV={200,20}
```

...

```
WAITTIL(MATCHING(msg_1) || MATCHING(msg_2,ch),WTIME==10)
```

...

```
IF (MATCHING(msg_1,ch)) THEN
```

...

Related Topics

[CHANNEL](#) on page 843 | [DEF MESSAGE](#) on page 847 | [WAITTIL](#) on page 889 | [MATCHED\(\)](#) on page 862

MESSAGE

System Testing Test Script Language.

Syntax

```
MESSAGE <message_type> : <ref_msg> {[, <ref_msg>]}
```

Description

The MESSAGE instruction allows you to declare a list of reference messages <ref_msg> of the <message_type> type.

<message_type> is in C and must be defined by a **typedef** in the interface file, an included file, or the test script.

You must use the **MESSAGE** instruction at the beginning of the test script, before the first scenario.

The reference messages are global variables. After a **WAITTIL** instruction, the reference messages used contains the value of the last received message.

Interface file

```
typedef struct {  
  
int code;  
  
int flight_number;  
  
struct {  
  
char flight_name[8];  
  
unsigned char class_name;  
  
} data;  
  
} aircraft_data_t;
```

Examples

```
MESSAGE aircraft_data_t: air_msg
```

```
SCENARIO first
```

```
DEF_MESSAGE air_msg, EV= {code => FlightReport }
```

```
WAITTIL(MATCHING(air_msg), WTIME == 100)
```

```
...
```

```
IF (air_msg.flight_number == 321) THEN
```

```
...
```

Related Topics

[DEF_MESSAGE on page 847](#) | [WAITTIL on page 889](#)

MESSAGE_DATE

System Testing Test Script Language.

Syntax

```
MESSAGE_DATE
```

Description

The **MESSAGE_DATE** instruction marks the date the user receives the message.

For instance, this date may be the moment a message is present in a reception queue or when a message has been read and decoded. This instruction must appear once in a callback or in a procedure called in a callback.

The **MESSAGE_DATE** instruction must be used in a callback.

Examples

```
COMMTYPE socket IS socket_id_t
```

```
CHANNEL socket: ch
```

```
MESSAGE client_info_t: msg
```

```
CALLBACK client_info_t: info ON socket: id
```

```
CALL read(id, &info, sizeof(client_info_t))@@ret
```

```
IF (ret == 0) THEN
```

```
NO_MESSAGE
```

```
END IF
```

```
MESSAGE_DATE
```

```
END CALLBACK
```

Related Topics

[CALLBACK on page 840](#)

NIL

System Testing Test Script Language.

Syntax

NIL

Description

NIL is a macro that represents the value of a null pointer and can be used in any C expression.

Example

...

```
SCENARIO Main
```

```
CALL free_object(@NIL@object)
```

...

```
END SCENARIO
```

Related Topics

[NONIL on page 866](#)

NONIL

System Testing Test Script Language.

Syntax

NONIL

Description

NONIL is a macro that represents the value of a non-null pointer and can be used in any C expression.

NONIL is useful in a **CALL** or a **VAR** instruction. In these two cases, it verifies that the pointer does not have a null value.

Example

...

```
SCENARIO Main
```

```
CALL alloc_object( ) @ NONIL @ object
```

```
VAR object, VA = NONIL
```

```
...
```

```
END SCENARIO
```

Related Topics

[CALL on page 839](#) | [VAR on page 883](#) | [NIL on page 866](#)

NOTMATCHED

System Testing Test Script Language.

Syntax

```
NOTMATCHED( <ref_msg> [, <channel> ] )
```

Description

NOTMATCHED is a function that returns a Boolean value. It returns true if one of the messages received during a **WAITTIL** does not match the reference message <ref_msg>. If you specify a channel, it returns true only if the non-matching message was received on this channel.

<ref_msg> is a reference message variable declared with the **MESSAGE** instruction and initialized with the **DEF_MESSAGE** instruction.

<channel> is a communication channel declared with the **CHANNEL** instruction and initialized by the **ADD_ID** instruction.

It returns true if at least one received message has a value different from those defined for the reference message.

NOTMATCHED is only meaningful when used in a **WAITTIL** instruction or in control statements following a **WAITTIL**, such as **IF**, **WHILE**, or **CASE**.

The **NOTMATCHED** return value changes when reused in a **WAITTIL**.

Example

```
...
```

```
CHANNEL ux_socket: ch
```

```
SCENARIO Main
```

```
DEF_MESSAGE msg_1, EV={100,10}
```

```
DEF_MESSAGE msg_2, EV={200,20}
```

...

```
WAITTIL(WTIME==10, NOTMATCHED(msg_1))
```

...

```
IF (NOTMATCHED(msg_1,ch)) THEN
```

...

Related Topics

[CHANNEL on page 843](#) | [DEF_MESSAGE on page 847](#) | [WAITTIL on page 889](#)

NOTMATCHING

System Testing Test Script Language.

Syntax

```
NOTMATCHING( <ref_msg> [, <channel> ] )
```

Description

NOTMATCHING is a function that returns a Boolean value. It returns true if the last message received during a **WAITTIL** does not match the reference message <ref_msg>. If you specify a channel, it returns true only if the non-matching message was received on this channel.

<ref_msg> is a reference message variable declared with the **MESSAGE** instruction and initialized with the **DEF_MESSAGE** instruction.

<channel> is a communication channel declared with the **CHANNEL** instruction and initialized by the **ADD_ID** instruction.

It returns true if the value of the last received message differs from the values specified for the reference message.

NOTMATCHING is only meaningful when used in a **WAITTIL** instruction or in control statements following a **WAITTIL**, such as **IF**, **WHILE**, or **CASE**.

The **NOTMATCHING** return value changes when reused in a **WAITTIL**.

Example

...

```
CHANNEL ux_socket: ch
```

SCENARIO Main

DEF_MESSAGE msg_1, EV={100,10}

DEF_MESSAGE msg_2, EV={200,20}

...

WAITTIL(WTIME==10, NOTMATCHING(msg_2,ch))

...

IF (NOTMATCHING(msg_2,ch)) THEN

...

Related Topics

[CHANNEL on page 843](#) | [DEF_MESSAGE on page 847](#) | [WAITTIL on page 889](#)

NO_MESSAGE

System Testing Test Script Language.

Syntax

NO_MESSAGE

Description

The **NO_MESSAGE** instruction is used to exit the callback if no message has been received.

This instruction has to appear once in a callback or in a procedure called in a callback.

The **MESSAGE_DATE** instruction must be used in a callback.

Example

COMMTYPE socket IS socket_id_t

CHANNEL socket: ch

MESSAGE client_info_t: msg

CALLBACK client_info_t: info ON socket: id

CALL read(id, &info, sizeof(client_info_t))@@ret

IF (ret == 0) THEN

NO_MESSAGE

END IF

MESSAGE_DATE

END CALLBACK

Related Topics

[CALLBACK on page 840](#)

PAUSE

System Testing Test Script Language.

Syntax

PAUSE [*<duration>*]

<duration> is an integer specifying the length of the delay in multiples of 10ms by default.

Description

PAUSE introduces a delay in the execution of the supervisor script. It does not delay any other processes that are running on the machine.

The **PAUSE** instruction does not appear in generated reports.

<duration> is the duration of the delay in multiples of the time unit. By default the time unit is 10ms and can be customized in the TDP.

Example

In the following example, the first PAUSE statement introduces a delay of 200ms before resuming the execution of the script. The second PAUSE statement pauses the script for 1840ms.

```
#int hp = 3;
```

```
#int ds = 5;
```

```
PROC init (int sock_type)
```

```
...
```

```
PAUSE 20
```

```
...
```

```
END PROC
```

```
SCENARIO Main
```

```
...
CALL init( AF_UNIX )
PAUSE (hp+ds)*23
...
```

```
END SCENARIO
```

Related Topics

[WTIME on page 891](#)

PRINT

System Testing Test Script Language.

Syntax

```
PRINT <identifier>, <expression>
```

Description

The **PRINT** instruction prints the value of <expression> in the generated reports. The identifier names the value.

<expression> must be a C integer expression.

The same identifier can be used in different **PRINT** instructions.

Example

```
#int hp = 3;
```

```
#int ds = 5;
```

```
TIMER time
```

```
PROC init (int sock_type)
```

```
...
```

```
PRINT SocketValue, sockType
```

```
...
```

```
END PROC
```

```
SCENARIO Main
```

```
...
```

```
CALL init( AF_UNIX )  
PRINT HpDs, (hp+ds)*10  
PRINT elapsedTime, TIME (time)  
...  
END SCENARIO
```

Related Topics

[TIME on page 880](#) | [TIMER on page 880](#) | [VAR on page 883](#)

PROC ... END PROC

System Testing Test Script Language.

Syntax

```
PROC <arg> {[, <arg> ]}
```

```
END PROC
```

Description

The **PROC** instruction lets you define a local procedure inside a scenario. A procedure can take parameters defined as data types.

Any previously defined global variables declared in the test script are visible in the **PROC** block. Variables declared locally to a procedure block are only visible within that procedure.

Procedure parameters take basic data: *int*, *char*, and *float* as well as any data types defined by the a **typedef** statement.

Procedures must be located at the beginning of the test script file, before the highest-level scenarios.

Procedures can be called from any scenario.

Procedures do not return any parameters.

Example

```
#int hp,ds;  
PROC init (int sock_type)  
...  
CALL gethostbyname (serv_name)@@hp
```

```
CALL socket (sock_type, SOCK_DGRAM, 0)@@ds
```

```
...
```

```
END PROC
```

```
SCENARIO Main
```

```
...
```

```
CALL init( AF_UNIX )
```

```
...
```

```
END SCENARIO
```

Related Topics

[CALL on page 839](#)

PROCSEND

System Testing Test Script Language.

Syntax

```
PROCSEND <message_type> : <msg> ON <commtype> : <id>
```

```
END PROCSEND
```

Description

The **PROCSEND** instruction allows you to define a message-sending procedure. The **SEND** statement uses this instruction.

<message_type> is declared with the **MESSAGE** instruction.

<msg> is the input parameter of <message_type> that describes the message to be sent.

<commtype> is the communication method for sending messages.

Use the <id> formal input parameter to specify the connection on which a message has to be sent.

You must declare the message-sending procedure in the first part of the test script, before the first scenario.

Declare <commtype> with the instruction **COMMTYPE**.

Declare <message_type> with the instruction **MESSAGE**.

You only need to declare one message-sending procedure a message and communication type pair.

If the structured C `<message_type>` contains unions, you should declare the field of the union that you want to use. For this purpose, a structured variable is implicitly defined. Its name adds **ATL_** before the name of the `<message_type>`. An attribute selected for each union lets you define the field.

Example

```
typedef enum { e_name, e_id, e_balance } client_kind_t ;
```

```
typedef struct {
```

```
client_kind_t kind ;
```

```
union {
```

```
char name[50];
```

```
int id ;
```

```
float balance ;
```

```
} my_union
```

```
} client_info_t;
```

```
COMMTYPE socket IS socket_id_t
```

```
CHANNEL socket: ch
```

```
MESSAGE client_info_t: msg
```

```
#socket_id_t id;
```

```
PROCSEND message_t: msg ON appl_comm: id
```

```
...
```

```
CALL socket (sock_type, SOCK_DGRAM, 0) @ 0
```

```
...
```

```
END PROCSEND
```

```
SCENARIO Principal
```

```
...
```

```
ADD_ID(ch,id)
```

```
...
```

```
SEND (msg,ch)
```

...

END SCENARIO

Related Topics

[COMMTYPE on page 845](#) | [MESSAGE on page 864](#) | [SEND on page 877](#) | [VIRTUAL PROCSEND on page 887](#)

RENDEZVOUS

System Testing Test Script Language.

Syntax

RENDEZVOUS *<identifier>*

Description

The **RENDEZVOUS** instruction allows you to synchronize several virtual testers. A rendezvous name is the *<identifier>* following the keyword.

When the scenario is executed, the **RENDEZVOUS** instruction stops the execution until all virtual testers have reached the rendezvous point, thereby validating the rendezvous.

When the rendezvous is valid, the scenario resumes the execution.

A **RENDEZVOUS** identifier does not appear more than one time in a scenario.

Example

SCENARIO Connection

RENDEZVOUS begin

...

RESET

System Testing Test Script Language.

Syntax

RESET *<identifier>*

Description

The **RESET** instruction lets you reset the *<identifier>* timer.

Declare the timer identifier with the **TIMER** instruction.

You may use a timer identifier only once in the same block. The timer immediately restarts after being reset.

Example

TIMER time

SCENARIO Connexion

...

RESET time

...

END SCENARIO

Related Topics

[TIMER](#) on page 880, [TIME](#) on page 880

SCENARIO ... LOOP ... END SCENARIO

System Testing Test Script Language.

Syntax

```
SCENARIO <scenario> [ LOOP <iteration_factor> ]
```

```
END SCENARIO
```

Description

This instruction allows you to define a scenario block. This is the highest level of instruction.

<scenario> is the name of the scenario.

The optional **LOOP** keyword lets you state the identifier's scenario <iteration_factor>.

Associated Rules

Scenarios at the same level must have different names.

A scenario that contains other scenarios can only include **FAMILY** and **SCENARIO** statements.

<scenario> must begin with an upper or lower case letter and may contain letters, numbers, underscores, and dollar signs.

<iteration_factor> must be a positive integer.

Example

The J_n variable (n is the nesting level of the scenario that starts at 1) gives the current scenario iteration number.

```

SCENARIO principal LOOP 10
FAMILY nominal, robustness
...
SCENARIO number_one
...
SCENARIO number_one_two LOOP 10
CALL ...
PRINT iteration_number_one_two, J3
END SCENARIO
...
END SCENARIO
SCENARIO number_two LOOP 5
...
CALL ...
PRINT iteration_number_two, J2
PRINT global_iteration, J1
...
END SCENARIO
END SCENARIO

```

SEND

System Testing Test Script Language.

Syntax

SEND (<message>, <channel>)

Description

The **SEND** instruction allows you to send a <message> on a specific <channel>. It calls the message-sending procedure associated with the message and communication types.

The **SEND** instruction may be located in a **PROC**, **SCENARIO**, **INITIALIZATION**, **TERMINATION**, or **EXCEPTION** block.

Example

```
CHANNEL appl_comm: appl_ch
```

```
#message_t msg;
```

```
SCENARIO TEST_1
```

```
FAMILY nominal
```

```
...
```

```
SEND( msg, appl_ch )
```

Related Topics

[PROCSEND on page 873](#), [VIRTUAL PROCSEND on page 887](#)

SHARE

System Testing Test Script Language.

Syntax

```
SHARE <identifier>
```

Description

The **SHARE** instruction allows you to specify global static variables declared in a test script.

This allows all instances of the same test script, to share these variables in multi-thread environments.

Associated Rules

The **SHARE** instruction must be at the beginning of a test script, before the first block.

The identifier is the name of the global static variable declared at the beginning of the test script.

Example

```
#static int id_Connection;
```

```
#static int Synchro;
```

```
#static int buffer;
```

```
SHARE Synchro
```

```
SCENARIO Test1
```

FAMILY nominal

...

TERMINATION ... END TERMINATION

System Testing Test Script Language.

Syntax

TERMINATION [<proc>([*type identifier*]{ , type identifier })]

END TERMINATION

Description

The **TERMINATION** instruction deletes a specific environment by executing a set of instructions or the procedure <proc>. **END TERMINATION** marks the end of the **TERMINATION** block.

A **TERMINATION** block or instruction applies to the set of scenarios on its level. It does not apply to sub-scenarios.

The **TERMINATION** instruction or block is optional. A maximum of one **TERMINATION** block or instruction may occur at a given scenario level. The **TERMINATION** instruction is only executed when a scenario terminates without errors.

You may place a **TERMINATION** instruction anywhere among scenarios at the same level.

Example

```
#int sock;
```

```
TERMINATION
```

```
...
```

```
CALL close (sock)
```

```
...
```

```
END TERMINATION
```

```
...
```

```
SCENARIO Main
```

```
...
```

```
END SCENARIO
```

Related Topics

[INITIALIZATION on page 858](#) | [EXCEPTION on page 850](#)

TIME

System Testing Test Script Language.

Syntax

TIME (*<identifier>*)

Description

The **TIME** instruction gives the value of the identifier timer.

The timer *<identifier>* must be declared by a **TIMER** instruction.

The **TIME** instruction can only appear in a C expression (analyzed or not).

Example

```
#static int id_connexion;  
  
#static int Synchro;  
  
#static int buffer;  
  
TIMER globalTime  
  
SCENARIO TEST_1  
  
FAMILY nominal  
  
#unsigned long C_var_Time = TIME (globalTime);  
  
...  
  
PRINT time, TIME (globalTime)  
  
END SCENARIO
```

Related Topics

[TIMER on page 880](#) | [RESET on page 875](#)

TIMER

System Testing Test Script Language.

Syntax

TIMER *<identifier>*

Description

The **TIMER** instruction lets you define a timer (which automatically starts after being defined).

A timer *<identifier>* can be declared once in the same block. The scope of an identifier is its definition block. For example, an identifier declared in an exception block can only be used in this block. However, you may use an identifier declared in the global block in all the other blocks.

Example

```
#static int id_connexion;

#static int Synchro;

#static int buffer;

TIMER globalTime

PROC dummy

TIMER procTime

END PROC

SCENARIO TEST_1

FAMILY nominal

#unsigned long C_var_Time = TIME (globalTime);

...

PRINT time, TIME (globalTime)

END SCENARIO
```

Related Topics

[TIME on page 880](#) | [RESET on page 875](#)

TRACE_ON

System Testing Test Script Language.

Syntax

```
TRACE_ON
```

Description

The **TRACE_ON** instruction stores execution traces in the circular buffer.

This instruction is taken into account only when the **-TRACE=CIRCULAR** option is set.

Associated Rules

The **TRACE_ON** instruction can be used in **PROC**, **SCENARIO**, **INITIALIZATION**, **TERMINATION**, or **EXCEPTION** blocks, but not in **CALLBACK** or **PROSEND** blocks.

Example

```
SCENARIO one
```

```
...
```

```
TRACE_ON
```

```
...
```

```
END SCENARIO
```

Related Topics

[TRACE_OFF on page 882](#) | [FLUSH_TRACE on page 853](#)

TRACE_OFF

System Testing Test Script Language.

Syntax

```
TRACE_OFF
```

Description

The **TRACE_OFF** instruction turns off storage of execution traces in the circular buffer.

This instruction is taken into account only when the **-TRACE=CIRCULAR** option is set.

Associated Rules

The **TRACE_OFF** instruction can be used in **PROC**, **SCENARIO**, **INITIALIZATION**, **TERMINATION**, or **EXCEPTION** blocks, but not in **CALLBACK** or **PROSEND** blocks.

Example

```
SCENARIO one
```

```
...
```

```
TRACE_OFF
```

```
...
```

```
END SCENARIO
```

Related Topics

[FLUSH_TRACE](#) on page 853 | [TRACE_ON](#) on page 881

VAR

System Testing Test Script Language.

Syntax

VAR <variable> , **INIT**= <expression> | **EV**= <expression>

Description

This instruction allows you to initialize or check a variable. The first statement performs the initialization. The second statement compares the contents of the variable with the expression.

<variable> is a message or a variable that has previously been declared in native language. It may be any basic or structure type expression.

<expression> is in C and takes the following form:

cmp_expression ::= C_CPP_lang_exp

{cmp_init {,cmp_initialization}}

[atto_init {,atto_init}]

cmp_init ::= Constant => C_CPP_lang_exp |

Constant1 .. Constant2 => C_CPP_lang_exp |

C_CPP_lang_exp

field_name => C_language_expression

When controlling a numeric value (VAR ... EV=), you can check a range of values with one of following syntaxes:

VAR <variable>, EV= [<expr_min> ..]

VAR <variable>, EV= [.. <expr_max>]

VAR <variable>, EV= [<expr_min> .. <expr_max>]

This indicates that the value should be greater than <expr_min>, less than <expr_max>, or between the two expressions.

The **VAR** instruction may appear in a **PROC**, **SCENARIO**, **INITIALIZATION**, **TERMINATION** or **EXCEPTION** block.

The keyword **OTHERS** in a *<expression>* that represents ranges in an array or fields in a structure that have not been previously specified.

The identifiers **I1**, **I2**, ... **I20** are reserved to access different dimensions of an array. For a three-dimensional matrix, **I1** represents the index for the first dimension, **I2** the index for the second dimension, and **I3** the index for the third dimension.

Example

```
SCENARIO Main
```

```
#int matrix[3][3];
```

```
#struct {
```

```
# char name[30];
```

```
# char color[20];
```

```
# double size;
```

```
# } object;
```

```
# long x;
```

```
CALL compute(matrix)
```

```
VAR matrix, EV=[ [1, 1, 1], [2, 2, 2], [1, 1, 1] ]
```

```
-- OR
```

```
VAR matrix, EV=[ 2 => [2, 2, 2], OTHERS =>[1,1,1] ]
```

```
-- multiplication table:
```

```
VAR matrix, INIT= I1 I2
```

```
VAR object, INIT=[ name => "car", color => "rouge",
```

```
& size => 2.50 ]
```

```
VAR object, INIT=[ size => 0.10, OTHERS => "orange" ]
```

```
VAR x, EV=[11..28]
```

```
END SCENARIO
```

Related Topics

[CALL on page 839](#) |

VIRTUAL CALLBACK

System Testing Test Script Language.

C++ only.

The **VIRTUAL** keyword modifies the **CALLBACK** statement, allowing it to handle messages using C++ inheritance.

Syntax

```
VIRTUAL CALLBACK <message_type> : <msg> ON <commtype> : <id> [<n>]
```

```
END CALLBACK
```

Description

The **CALLBACK** instruction dynamically recalls message reception and adds a connection identifier value to a communication channel identifier.

<message_type> is a message type, previously declared with a C++ *typedef* statement. Syntax using <message_type> * is not allowed.

<msg> is the output parameter of <message_type> that must be a *polymorphic* C++ class, which means that it must contain at least one virtual method.

<commtype> is the type of communication used for reading messages.

<id> is the input connection parameter on which a message must be read.

Because a single **VIRTUAL CALLBACK** can read several message types, the implicit choice of a **CALLBACK** may be ambiguous. The following rules apply:

1. If a **CALLBACK** exists for a given <message type>, System Testing chooses it.

If not, and if the message type is actually a virtual class, then System Testing chooses the **VIRTUAL CALLBACK** with the closest type in terms of path in the inheritance diagram of <message_type>.

If more than one **VIRTUAL CALLBACK** can be chosen by following the above rules, the **CALLBACK** is ambiguous and System Testing produces an error.

Example

```
class high_level_message
{
public:
char from[12];
```

```
char applname[12];

virtual int get_type(){return 0;}

};

class ack : public high_level_message
{
public:
int get_type(){return ACK;}

};

class negack : public high_level_message
{
public:
int get_type(){return NEG_ACK;}

};

class data : public high_level_message
{
public:
char userdata[MAX_USERDATA_LENGTH];

int length;

int get_type(){return DATA;}

};

#ifdef high_level_message * pt_high_level_message;

VIRTUAL CALLBACK pt_high_level_message: msg ON appl_comm: id

CALL get_message ( &id, &msg, 0 ) @@ errcode

MESSAGE_DATE

IF ( errcode == err_empty ) THEN

NO_MESSAGE
```

```

END IF

IF ( errcode != err_ok ) THEN

ERROR

END IF

END CALLBACK

```

This VIRTUAL CALLBACK allows you to read **high_level_message**, **ack**, **negack** and **data** message types, as shown on the following lines:

```

MESSAGE data : a_data

MESSAGE ack : my_ack

MESSAGE negack : my_neg_ack

MESSAGE high_level_message : hm

DEF_MESSAGE my_ack, EV={}

WAITTIL (MATCHING(my_ack), WTIME==300)

DEF_MESSAGE a_data, EV={}

WAITTIL (MATCHING(a_data), WTIME==300)

```

Related Topics

[CALLBACK ... END CALLBACK on page 840](#) | [PROCSSEND ... END PROCSSEND on page 873](#) | [VIRTUAL PROCSSEND on page 887](#) | [MESSAGE on page 864](#)

VIRTUAL PROCSSEND

System Testing Test Script Language.

For C++ only.

The **VIRTUAL** keyword modifies the **PROCSSEND** statement, allowing it to handle messages using C++ inheritance.

Syntax

```
VIRTUAL PROCSSEND <message_type> : <msg> ON <commttype> : <id>
```

```
END CALLBACK
```

Description

The **PROCSEND** instruction allows you to define a message-sending procedure using C++ classes.

<message_type> is a message type, previously declared with a C++ *typedef* statement. Syntax using <message_type> * is not allowed.

<msg> is the output parameter of <message_type> that must be a *polymorphic* C++ class, which means that it must contain at least one virtual method.

<commtype> is the type of communication used for reading messages.

<id> is the input connection parameter on which a message must be read.

Associated Rules

Because a single **VIRTUAL PROCSEND** can read several message types, the implicit choice of a **PROCSEND** may be ambiguous. The following rules apply:

1. If a **PROCSEND** exists for a given <message type>, System Testing chooses it.

If not, and if the message type is actually a virtual class, then System Testing chooses the **VIRTUAL PROCSEND** with the closest type in terms of path in the inheritance diagram of <message_type>.

If more than one **VIRTUAL PROCSEND** can be chosen by following the above rules, the **PROCSEND** is ambiguous and System Testing produces an error.

Example

```
VIRTUAL PROCSEND pt_high_level_message : msg ON appl_comm : id_stack
```

```
CALL send_message (msg) @ err_ok
```

```
END PROCSEND
```

This **VIRTUAL PROCSEND** example allows you to send **high_level_message**, **ack**, **negack** et **data** message types, as shown on the following lines:

```
MESSAGE data : a_data
```

```
MESSAGE ack : my_ack
```

```
MESSAGE negack : my_neg_ack
```

```
MESSAGE high_level_message : hm
```

```
VAR a_data, INIT={applname=>"SATURN",userdata=>"Hello Saturn!"}
```

```
SEND( a_data , appl_ch )
```

```
VAR my_ack, INIT={applname=>"SATURN"}
```

```
SEND(my_ack , appl_ch )
```

```
VAR my_neg_ack, INIT={applname=>"SATURN"}
```

```
SEND(my_neg_ack , appl_ch )
```

Related Topics

[CALLBACK ... END CALLBACK on page 840](#) | [PROCSEND ... END PROCSEND on page 873](#) | [VIRTUAL CALLBACK on page 885](#) | [SEND on page 877](#)

WAITTIL

System Testing Test Script Language.

Syntax

WAITTIL (<passed_expr>, <failed_expr>)

Description

This instruction waits for several events and/or a timer.

<passed_expr> is a parameter that contains a Boolean expression. If this expression is true, the waiting process is disabled and the test sequence continues.

<failed_expr> is a parameter that contains a Boolean expression. If this expression is true, the waiting process is disabled and it ends with an error.

The expressions <passed_expr> and <failed_expr> can only use global variables.

When <failed_expr> is true, the execution of the scenario containing the **WAITTIL** is interrupted. The next scenario at the same level is then executed.

To use this instruction, you need to take the following actions:

1. Declare a type of communication with the **COMMTYPE** instruction.

Declare a communication channel with the **CHANNEL** instruction.

Declare the reference messages with the **MESSAGE** instruction.

Write a callback for a non-blocking read of communication and message type.

Define the expected values for each reference message with the **DEF_MESSAGE** instruction.

Associate the identifier of a communication connection with the **ADD_ID** instruction.

Use the four comparison operators, **MATCHING**, **MATCHED**, **NOTMATCHING**, **NOTMATCHED**, and the timer **WTIME**. Also use the **&&** (logical *and*) and **||** (logical *or*) operators.

You must use a global variable to pass parameters to a **WAITTIL** statement, as in the following example. It does not handle **PROC** parameters.

Example

The following lines are from the **Basestation** sample application delivered with the product.

```
#int tt; /* global var */  
  
PROC con (int timeout)  
  
VAR tt, INIT=timeout;  
  
DEF_MESSAGE mResponse, EV={command=>cmd_connection_established}  
  
WAITTIL ( MATCHING(mResponse,BaseStation), WTIME>tt )  
  
END PROC
```

Related Topics

[ADD_ID on page 838](#) | [MATCHED\(\) on page 862](#) | [MATCHING\(\) on page 863](#) | [NOTMATCHED\(\) on page 867](#) | [NOTMATCHING\(\) on page 868](#) | [WTIME on page 891](#)

WHILE ... END WHILE

System Testing Test Script Language.

Syntax

```
WHILE ( condition )  
  
END WHILE
```

Description

The instruction **WHILE** is a control structure. All the instructions between **WHILE** and **END WHILE** is executed if the condition is true.

Example

```
#int i = 0;  
  
SCENARIO Main  
  
CALL api1_func...  
  
WHILE (i<100)  
  
CALL api_val(i)
```

```
VAR i, INIT=i+1
```

```
END WHILE
```

```
...
```

```
END SCENARIO
```

Related Topics

[Iterations on page 602](#)

WTIME

System Testing Test Script Language.

Syntax

WTIME

Description

WTIME is a macro that acts as a timer in a **WAITTIL** instruction.

The value of **WTIME** is reset to zero before every **WAITTIL**. The value is a multiple of the time unit. By default the time unit is 10ms and can be customized in the TDP.

You can assign parameters to the timer's unit of time in the Target Deployment Port.

Example

```
...
```

```
SCENARIO Acknowledge
```

```
...
```

```
WAITTIL (MATCHING (OK), WTIME > 1000 )
```

```
END SCENARIO
```

ATL_OCCID

System Testing Test Script Language.

Description

ATL_OCCID is a macro that returns the value of the occurrence identification number (**OCCID**) that uniquely identifies a virtual tester.

You can change the occurrence identification number of a virtual tester by adding the **-OCCID=** *<number>* parameter to the command line of the generated virtual tester.

By default, the value of **ATL_OCCID** within a test script is **0**.

Example

```
HEADER "Client", "1.0", "3.0"
```

```
SCENARIO Main
```

```
...
```

```
PRINT occnumber, ATL_OCCID
```

```
...
```

```
END SCENARIO
```

Related Topics

[INTERRECV on page 861](#) | [INTERSEND on page 860](#)

ATL_TIMEOUT

System Testing Test Script Language.

Description

The value of **ATL_TIMEOUT** is calculated from a **WTIME** expression used in the **WAITTIL** statement. The **ATL_TIMEOUT** macro is an integer and uses the time unit defined in the Target Deployment Port. By default, the time unit is a hundredth of second.

Related Topics

[CALLBACK on page 840](#)

ATL_NUMINSTANCE

System Testing Test Script Language.

Description

ATL_NUMINSTANCE is a macro that returns the index number of an executed instance, according to the order defined in the **DECLARE_INSTANCE** instruction.

Note The number returned by **ATL_NUMINSTANCE** is the index number +1. For example, the first instance returns 2, the fourth instance returns 5.

Example

```

HEADER "Client", "1.0", "3.0"

DECLARE_INSTANCE client, server

SCENARIO Main

...

PRINT instanceNum, ATL_NUMINSTANCE

...

END SCENARIO

```

Related Topics

[DECLARE_INSTANCE on page 846](#)

System Testing supervisor script reference (.spv)

When using the System Testing tool, the machine running HCL OneTest™ Embedded runs a supervisor process.

This section describes each supervisor script instruction, including:

- Syntax
- Functionality and rules governing its usage
- Examples of use

Notation Conventions

Throughout this guide, command notation and argument parameters use the following standard convention:

Notation	Example	Meaning
BOLD	ADD_ID	Language keyword
<i><italic></i>	<filename>	Symbolic variables
[]	[<option>]	Optional items
{ }	{<filenames>}	Series of values
[{ }]	[{<file- names>}]	Optional series of variables
	on off	OR operator

System test script keywords are case sensitive. All keywords must be entered in upper case.

For conventional purposes however, this document uses upper-case notation for the supervisor script keywords in order to differentiate from native source code.

Split statements

Statements may be split over several lines in a **.spv** supervisor script. Continued lines must start with the ampersand (&) symbol to be recognized as a continuation of the previous line. No tabs or spaces should precede the ampersand.

Statements must be shorter than 2048 characters, although this limit may be lower on some platforms.

Identifiers

A supervisor script identifier is a text string used as a label, such as the name of a message type.

Identifiers are made of an unlimited sequence of the following characters:

- a-z
- A-Z
- 0-9
- _ (underscore)

Spaces are not valid identifier characters.

System Testing keywords and identifiers are case sensitive. This means that **LABEL**, **label**, and **Label** are three different identifiers.

Related Topics

[Supervisor script structure on page 894](#) | [Supervisor script keywords on page 895](#) | [System Testing supervisor on page 654](#)

Supervisor script structure

System Testing Supervisor Script Language.

System Testing manages the simultaneous execution of Virtual Testers distributed over a network. The supervisor script language allows you to create a supervisor process to:

- Set up target hosts to run the test
- Launch the virtual testers, the system under test and any other tools.
- Synchronize virtual testers during execution
- Retrieve the execution traces after test execution

Note When using the HCL OneTest™ Embedded graphical user interface, the **.spv** supervisor scripts are generated automatically. Experienced users can edit these files manually. See [System Testing supervisor on page 654](#)

Test script file names must contain only plain alphanumerical characters.

Basic structure

A typical System Testing **.spv** supervisor script looks like this:

```
HOST machine_1 IS localhost
```

```
HOST machine_2 IS 193.256.6.2(10098)
```

```
HOST machine_3 IS $HOSTNAME
```

```
COPY local_file machine_2:remote_file
```

```
DO machine_1:program
```

All instructions in a test script have the following characteristics:

- All statements begin with a keyword.
- Statements are not case sensitive.
- Statements start at the beginning of a line and end at the end of a line. You can, however, write an instruction over several lines using the ampersand (&) continuation character at the beginning of additional lines. In this case, the ampersand must be the very first character on that line; no spaces or tabs should precede it.
- Statements must be shorter than 2048 characters, although this limit may be lower on some platforms.

Supervisor script instructions are sequential. There is no hierarchical structure in the script.

Related Topics

[System Testing supervisor script \(.spv\) on page 893](#) | [Supervisor script keywords on page 895](#) | [System Testing supervisor on page 654](#)

Supervisor script keywords

- [COPY on page 899](#)
- [CHDIR on page 900](#)
- [DELETE on page 901](#)
- [DO on page 902](#)
- [ENDOF on page 903](#)

- [ERROR on page 904](#)
- [EXECUTE on page 905](#)
- [EXIT on page 906](#)
- [HOST on page 907](#)
- [IF ... THEN ... ELSE ... END IF on page 908](#)
- [INCLUDE on page 909](#)
- [MEMBERS on page 910](#)
- [MKDIR on page 911](#)
- [PAUSE on page 912](#)
- [PRINT on page 912](#)
- [PRINTLN on page 913](#)
- [RMDIR on page 914](#)
- [UNSET on page 914](#)
- [STATUS on page 915](#)
- [SHELL on page 917](#)
- [SET on page 917](#)
- [STOP on page 918](#)
- [TRACE ... FROM on page 919](#)
- [WHILE on page 919](#)

Related Topics

[Supervisor script structure on page 894](#) | [System Testing supervisor script \(.spv\) on page 893](#) | [Expressions on page 897](#)

Environment variables

System Testing Supervisor Script Language.

System Testing supervisor scripts can read and write environment variables on the System Testing Supervisor machine and on target machines.

Precede an environment variable name with a dollar sign (\$) to substitute the environment variable by its value within a statement.

To force a variable to refer to the environment of the System Testing Supervisor machine, precede the environment variable with the 'at' sign (@) instead of the dollar sign.

Example

HOST machine IS \$HOSTNAME

-- show the contents of the target home directory

DO machine: Is \$HOME

-- show the contents of the local home directory

SHELL Is \$HOME

Related Topics

[Expressions on page 897](#) | [SET on page 917](#) | [UNSET on page 914](#)

Expressions

System Testing Supervisor Script Language.

Supervisor scripts may contain integer expressions only.

You may use expressions in variable assignments, **IF** instructions, and **WHILE** instructions.

Expressions may contain the following operators:

Opera- tor	Description
==	Equals
!=	Does not equal
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
+	Plus
-	Unary or binary minus
*	Multiply

/	Divide
%	Modulo
!	Negation
&&	Logical AND
	Logical OR
ENDOF	See ENDOF on page 903
STATUS	See STATUS on page 915

Expressions may be nested with parentheses. Operators obey the following ascending order of priority:

- &&, ||
- ==, !=
- >, >=, <, <=
- +, Unary or binary -
- *, /, %
- **!, ENDOF, STATUS**

Example

```
HOST machine IS 193.6.2.1
```

```
EXECUTE proc_1 IS machine:program
```

```
i = 1
```

```
-- declaration of i
```

```
j = ( i + 3 2 ) + ( i <= 2 )
```

```
-- declaration of j
```

```
PRINTLN j
```

Related Topics

[Variables on page 920](#) | [ENDOF on page 903](#) | [IF on page 908](#) | [STATUS on page 915](#) | [WHILE on page 919](#)

COPY

System Testing Supervisor Script Language.

Purpose

The COPY instruction transfers a binary or ASCII file from the System Testing Supervisor machine to a target machine, or the opposite.

Syntax

COPY [*<hostname>* :]*<source>* [*<hostname>* :]*<destination>* [/ASCII]

where:

- *<source>* is the absolute or relative filename of the file to be copied.
- *<destination>* is the absolute or relative path to which *<source>* is to be copied.
- *<hostname>* is the optional name of the source or destination machine.

Description

When the *<hostname>* is not specified, the filename refers to a local file on the System Testing Supervisor machine. When a *<hostname>* is specified, the filename refers to a file on the corresponding remote host.

COPY instructions can only transfer files from the System Testing Supervisor machine to a remote machine, or from a remote machine to the System Testing Supervisor machine. Transfers from one remote machine to another must be performed using two COPY instructions.

By default, transfers are in binary mode. If you specify the keyword /ASCII, the transfer is performed in character mode, which insures that text files are correctly copied between different types of machines. In binary mode, the target file's access permissions are updated so that the file is executable.

A filename may contain environment variables that are local to the System Testing Supervisor machine or that are defined on the remote machine. For more information, refer to the section on [Environment variables on page 896](#).

If the file to be copied does not exist or is read-protected, you will receive an error message (see [ERROR on page 904](#)).

Path and filenames may contain long quoted pathnames, such as "C:\Program Files\HCL OneTest™ Embedded\HCL OneTest™ Embedded\".

Example

HOST target_1 IS antares

...

```
COPY localfile target_1:$HOME/file.bin
```

```
COPY target_1:remotefile localfile /ASCII
```

...

Related Topics

[DELETE on page 901](#)

CHDIR

System Testing Supervisor Script Language.

Purpose

The CHDIR instruction changes the current working directory of the System Testing Supervisor machine or of a target machine.

Syntax

```
CHDIR [ <hostname>: ] <directory>
```

where:

1. <hostname> is an optional logical name of a target machine (see [HOST on page 907](#))
2. <directory> is the relative or absolute path of a directory

Description

When supervisor execution starts, the working directory of the System Testing Supervisor machine is the current directory of the shell that runs the System Testing Supervisor.

When the script starts, the working directory of the target machine is the directory where the Agent has been started.

The <directory> path may contain local environment variables from the System Testing Supervisor machine, or remote environment variables defined on the target machine. For more information, refer to the section on [Environment variables on page 896](#).

If the operation fails, you will receive an error message (see [ERROR on page 904](#)).

The <directory> path may contain long quoted pathnames, such as "C:\Program Files\HCL OneTest™ Embedded\HCL OneTest™ Embedded\".

Example

```
HOST target IS workstation.domain.com
```

```
CHDIR localdir
```

```
CHDIR $ATS_DIR
```

```
CHDIR target:$HOME
```

```
CHDIR target:/tmp/project
```

```
SET DIR=C:\tmp
```

```
CHDIR $DIR
```

Related Topics

[MKDIR on page 911](#) | [RMDIR on page 914](#)

DELETE

System Testing Supervisor Script Language.

Purpose

The DELETE instruction deletes a local or remote file.

Syntax

```
DELETE <filename>
```

where:

- <filename> is a local or remote file to be deleted.

Description

<filename> may be specified with an absolute or relative path, or as <hostname> : <filename>, where <hostname> is a remote host running a System Testing Agent daemon.

The filename may contain environment variables that are local to the System Testing Supervisor machine or that are defined on the remote machine. For more information, refer to the section on Environment variables

If the file to be deleted does not exist or is write-protected, you will receive an error message. (See **ERROR**.)

Path and filenames may contain long quoted pathnames, such as "**C:\Program Files\HCL OneTest™ Embedded\HCL OneTest™ Embedded**".

Example

```
HOST target_2 IS 123.4.56.7(10098)
```

```
DELETE target_2:$DIR/../remote_file
```

```
DELETE local_file
```

Related Topics

[ERROR on page 904](#)

DO

System Testing Supervisor Script Language.

Purpose

The **DO** instruction executes a program on a remote machine and waits for the end of its execution.

Syntax

```
DO [ <process> IS ] <hostname> : <program> [ <parameters> ]
```

where:

- <process> optionally assigns a process name to the program
- <hostname> is the name of the remote machine as defined by a **HOST** instruction
- <program> is the name of the program to execute
- <parameters> is a set of optional parameters that can be sent to <program>

Description

DO is a blocking instruction that waits for the program to end.

The field <hostname> is mandatory and must specify a remote machine.

You can give a logical name to a program by including the clause <process> **IS**. You can then form expressions with the **ENDOF** and **STATUS** operators.

A process name may only appear once in a supervision script, otherwise you will receive an error when the scenario does not execute. If <process> **IS** is not present, the **ENDOF** and **STATUS** operators cannot be used.

While the program runs, all logs sent to the standard and error outputs are redirected to the supervisor, except if you have set **TRACE OFF**.

If the program does not start or does not have execution permission, an error message is produced. (See **ERROR**.)

Note If a logical process name is used in a **DO** instruction within a **WHILE** loop, the name refers not to a single process, but a group of processes. (See the **ENDOF** and **STATUS** operators.)

Example

```
HOST remote IS 192.3.2.1
```

```
DO process_1 IS remote: ls /tmp -l
```

```
i = 1
```

```
WHILE i < 10
```

```
DO group IS remote:program
```

```
i = i + 1
```

```
END WHILE
```

-- the variable group refers to a group of 9

-- executions of the process called program

Related Topics

[ENDOF on page 903](#) | [EXECUTE on page 905](#) | [STATUS on page 915](#) | [TRACE on page 919](#)

ENDOF

System Testing Supervisor Script Language.

Purpose

ENDOF is a Boolean function that tests whether *<process>* has ended or not. **ENDOF** is true if the execution of *<process>* has ended.

Syntax

ENDOF (*<process>*)

<process> is a logical process name, defined with an **EXECUTE** statement.

Description

You can use the **ENDOF** function in expressions analyzed by the supervisor.

ENDOF is a non-blocking operator.

If an unknown process identifier is specified, an error is generated during analysis of the supervision script before it is executed.

Note If an **EXECUTE** instruction is placed inside a **WHILE** loop, the process identifier denotes a group of processes. In this case, an **ENDOF** expression with this process identifier is true when all the processes associated with the identifier have ended.

Example

```
...  
i = 1  
WHILE i < 10  
EXECUTE proc_group IS machine:program  
i = i + 1  
END WHILE  
...  
IF ENDOF ( proc_group ) THEN  
PRINT "end of execution of all processes"  
END IF
```

Related Topics

[EXECUTE on page 905](#) | [IF on page 908](#) | [STATUS on page 915](#) | [WHILE on page 919](#)

ERROR

System Testing Supervisor Script Language.

Purpose

The **ERROR** instruction indicates to the supervisor whether or not execution of a scenario should be interrupted if an error occurs.

Syntax

ERROR [ON | OFF]

Description

Use **ERROR ON** to interrupt execution of the supervision script if an error is detected.

Use **ERROR OFF** to ignore errors and continue execution of the supervision script.

In both cases, you will still receive an error message through the standard output.

The use of **ERROR** in supervision scripts is optional. **ERROR ON** is the default setting.

You may use **ERROR ON** and **ERROR OFF** several times in the same supervisor script.

Example

```

...
COPY localfile_1 target:file_1

ERROR OFF

DELETE localfile_1

ERROR ON

...

ERROR OFF

EXECUTE target:file_1

ERROR ON

```

EXECUTE

System Testing Supervisor Script Language.

Purpose

The EXECUTE instruction executes the program *<program_name>* on the *<hostname>* defined by a previous **HOST** instruction.

Syntax

```
EXECUTE [ <process> IS ] <hostname>: <program> [ <parameters> ]
```

where:

- *<process>* optionally assigns a process name to the program
- <hostname>* is the name of the remote machine as defined by a **HOST** instruction
- <program>* is the name of the program to execute
- <parameters>* is a set of optional parameters that can be sent to *<program>*

Description

EXECUTE is a non-blocking instruction that asynchronously starts the *<program>* on *<hostname>*, and then returns.

The field *<hostname>* is mandatory and must specify a remote machine.

You can assign a logical name to the *<program>* by adding the optional *<process>* **IS** statement. You can use this logical name to form expressions with the **ENDOF** and **STATUS** operators.

Any logical process name must be unique to a supervision script, otherwise it will generate an error when the scenario execution fails.

If no logical process name is assigned to the program execution, the **ENDOF** and **STATUS** operators will generate an error during the analysis of the supervisor script.

While *<program>* is running, all logs normally sent to the standard and error outputs are redirected to the supervisor, except if you have used a **TRACE OFF** statement.

If the *<program>* file is missing or does not have execution permission, an error is generated.

Note If a logical process name is used in an **EXECUTE** instruction within a **WHILE** loop, the name refers not to a single process, but a group of processes. (See the **ENDOF** and **STATUS** operators).

Example

```
HOST remote IS 192.3.2.1
```

```
EXECUTE process_1 IS remote: ls /tmp -l
```

```
EXECUTE remote: myFoo
```

```
i = 1
```

```
WHILE i < 10
```

```
EXECUTE group IS remote:program
```

```
i = i + 1
```

```
END WHILE
```

-- the variable group refers to a group of 9

-- executions of the process called program

Related Topics

[DO on page 902](#) | [ENDOF on page 903](#) | [STATUS on page 915](#) | [TRACE on page 919](#)

EXIT

System Testing Supervisor Script Language.

Purpose

The **EXIT** instruction stops execution of the supervision script.

Syntax

EXIT [" <message> "]

<message> is a an optional character string delimited by double-quotes (").

Description

Stopping the supervisor causes all processes started by agents to stop as well.

The optional <message> is printed as an information message.

Note If you need to include a double-quote in the message, use \".

Example

```
HOST remote IS 192.6.2.1
```

```
...
```

```
IF ( i = 3 ) THEN
```

```
EXECUTE remote: ls /tmp -l
```

```
ELSE
```

```
EXIT "Exit on incorrect value of \"i\""
```

```
END IF
```

Related Topics

[ERROR on page 904](#)

HOST

System Testing Supervisor Script Language.

Purpose

The **HOST** instruction assigns a logical machine name to a target machine.

Syntax

```
HOST <logical_name> IS <address> [ ( < port_number> ) ]
```

<logical_name> is the identifier of the target machine.

<address> is the network address of the target machine

<port_number> is the network port to which the target machine's Agent is assigned.

Description

Executing a **HOST** instruction opens a connection with an agent on the target machine.

Logical machine names are used in **CHDIR**, **COPY**, **DO**, **DELETE**, **EXECUTE**, **MKDIR**, **RMDIR**, **SET**, **TRACE** and **UNSET** instructions to refer to target machines.

The host *<address>* may be:

1. a hostname (for example: **workstation.domain.com**),
an alias (for example: **workstation**),
or an IP address (for example: **155.22.9.3**).

The TCP/IP port number is optional. It helps specify the port used by the target machine's agent that listens for connection demands. By default, the port used by the supervisor is the one specified by the **ATS_PORT** environment variable, or 10000.

A logical machine name must be unique within the supervision script. If the System Testing Supervisor machine cannot connect to the agent, the supervisor produces an error message and terminates, regardless of any **ERROR** statement.

Example

```
HOST machine_1 IS localhost
```

```
HOST machine_2 IS 193.256.6.2(10098)
```

```
HOST machine_3 IS $HOSTNAME
```

```
COPY local_file machine_2:remote_file
```

```
DO machine_1:program
```

IF ... THEN ... ELSE ... END IF

System Testing Supervisor Script Language.

Purpose

The **IF ... END IF** statement allows you to define a conditional behavior based on the result of an expression.

Syntax

```
IF <expression> THEN
```

```
ELSE
```

```
END IF
```

<expression> is a Boolean expression. See [Expressions on page 897](#).

Description

IF defines the Boolean expression.

Instructions following the **THEN** keyword are executed if the expression is true.

Instructions following **ELSE** are executed if the expression is false.

END IF marks the end of the of the IF statement.

Example

```
HOST machine IS 193.6.2.1
```

```
DO prepro IS machine:preprocessing.exe
```

```
IF ( STATUS ( prepro ) == 0 ) THEN
```

```
PRINTLN "preprocessing OK"
```

```
ELSE
```

```
PRINTLN "preprocessing FAILED"
```

```
EXIT
```

```
END IF
```

Related Topics

[Expressions on page 897](#)

INCLUDE

System Testing Supervisor Script Language.

Purpose

The **INCLUDE** instruction allows you to nest supervision scripts.

Syntax

```
INCLUDE " <filename> "
```

<filename> is the absolute or relative file name of an included supervision script, delimited by double quotes (").

Description

There is no limit to the levels of nested **INCLUDE** commands.

If an infinite loop of included files is detected during analysis, you will receive an error message and the execution will fail.

INCLUDE instructions may appear anywhere in a supervision script, including inside a structured **IF** or **WHILE** instruction.

There is no default file extension. If the filename has an extension, you must state it in the **INCLUDE** instruction.

Example

```
HOST machine_1 IS 193.6.2.1
```

```
INCLUDE "included_file.spv"
```

```
...
```

```
DO test_1 IS machine_1:test_1
```

MEMBERS

System Testing Supervisor Script Language.

Purpose

The **MEMBERS** instruction lets you declare the number of members awaited at a given rendezvous.

Syntax

```
MEMBERS <rendezvous> <number>
```

where:

1. <rendezvous> is the rendezvous identifier

<number> is a positive integer representing the number of members to wait for

Description

MEMBERS lets you synchronize virtual testers with the **RENDEZVOUS** instructions or with other applications with the rendezvous Target Deployment Port.

A <rendezvous> identifier must be unique within the supervision script. If not, an error message is produced and the scenario execution fails.

Example

```
...
```

```
MEMBERS beginning 3
```

...

EXECUTE machine_1:test1

EXECUTE machine2:test2

...

RENDEZVOUS beginning

Related Topics

[RENDEZVOUS on page 922](#)

MKDIR

System Testing Supervisor Script Language.

Purpose

The **MKDIR** instruction creates a new directory on the System Testing Supervisor machine or on a target machine.

Syntax

MKDIR [*<hostname>*:] *<directory>*

where:

1. *<hostname>* is an optional logical name of a target machine (see [HOST on page 907](#))
<directory> is the relative or absolute path of a directory

Description

The directory path name may contains local environment variables of the System Testing Supervisor machine, or remote environment variables defined on the target machine.

If the operation fails, the script returns an error message.

Example

HOST target IS workstation.domain.com(10098)

MKDIR ../localdir

MKDIR target:\$HOME/tmp

Related Topics

[CHDIR on page 900](#) | [RMDIR on page 914](#)

PAUSE

System Testing Supervisor Script Language.

Purpose

You may use the **PAUSE** instruction to delay script execution.

Syntax

PAUSE <duration>

<duration> is an integer specifying the length of the delay in seconds.

Description

The **PAUSE** instruction introduces a delay in the execution of the supervisor script. **PAUSE** does not delay any other processes that are already running on the machines.

<duration> is expressed in seconds. It may be an integer constant or an integer expression.

Example

```
DELAY = 25
```

```
...
```

```
PAUSE 3
```

```
...
```

```
PAUSE DELAY
```

PRINT

System Testing Supervisor Script Language.

Purpose

The **PRINT** instruction prints <argument> to the supervision script execution log file without a carriage return or line feed.

Syntax

PRINT <argument>

where:

1. <argument> is a string or a variable that points to a string

Description

The **PRINT** instruction does not cause a carriage return or line feed after printing the value of *<argument>*.

<argument> can be a string constant, delimited by quote double-quotes, or a variable integer value used in the scenario.

If *<argument>* uses an unknown variable, the scenario execution exits with an error message.

Example

```
var_i = 25
```

```
PRINT "value of var_i "
```

```
PRINT var_i
```

Related Topics

[PRINTLN on page 913](#)

PRINTLN

System Testing Supervisor Script Language.

Purpose

The **PRINTLN** instruction prints *<argument>* to the supervision script execution log file with a carriage return or line feed.

Syntax

```
PRINTLN [ <argument> ]
```

<argument> is an optional string or identifier that is to be printed.

Description

The value of *<argument>* can be a string constant, delimited by double-quotes, or a variable integer value used in the scenario.

If you provide no argument, the instruction causes a carriage return or line feed.

If *<argument>* uses an unknown variable, the scenario execution exits with an error message.

Example

```
var_i = 25
```

```
PRINTLN "value of var_i "
```

PRINTLN var_i

Related Topics

[PRINT on page 912](#)

RMDIR

System Testing Supervisor Script Language.

Purpose

The **RMDIR** instruction deletes a directory from the System Testing Supervisor machine or from a target machine.

Syntax

RMDIR [*<hostname>*:] *<directory>*

where:

1. *<hostname>* is an optional logical name of a target machine (see [HOST on page 907](#))
2. *<directory>* is the relative or absolute path of a directory

Description

The directory path name may contain local environment variables of the System Testing Supervisor machine or remote environment variables defined on the target machine. For more information, refer to the section on [Environment variables on page 896](#).

If the operation fails, the script returns an error message.

The *<directory>* path may contain long quoted pathnames, such as "**C:\Program Files\HCL OneTest™ Embedded\HCL OneTest™ Embedded**".

Example

HOST target IS antares.tlse.fr(10098)

RMDIR ../localdir

RMDIR target:\$HOME/tmp

Related Topics

[CHDIR on page 900](#) | [MKDIR on page 911](#)

UNSET

System Testing Supervisor Script Language.

Syntax

UNSET [<hostname>:] <env_var>

where:

1. <hostname> is the logical name of the target machine (See HOST.)

<env_var> is the name of the environment variable

Purpose

The **UNSET** instruction deletes an environment variable from the System Testing Supervisor machine or from the target machine.

Description

Hostname is the logical name on a target machine as defined in the HOST instruction. If you do not specify a hostname, the **UNSET** instruction deletes a local variable.

When you execute the **UNSET** instruction, the environment variable deletes until the end of the execution, or until you reset it.

Example

HOST target IS workstation(10098)

...

SET LOCAL_TMP_DIR=/tmp

SET target:REMOTE_TMP_DIR=\$TMPDIR

...

UNSET LOCAL_TMP_DIR

UNSET target:REMOTE_TMP_DIR

...

Related Topics

[SET on page 917](#)

STATUS

System Testing Supervisor Script Language.

Purpose

STATUS is an integer operator that retrieves the code returned by a remote process when it terminates.

Syntax

STATUS (process)

where:

- <process> is a logical process identifier

Description

The execution of a **STATUS** expression does not block execution of the scenario.

Applying **STATUS** to an ongoing process always returns a zero value. We recommend you use the **STATUS** operator in conjunction with **ENDOF**.

Note If you place an **EXECUTE** or **DO** instruction inside a **WHILE** loop, the process identifier denotes a group of processes. In this case, a **STATUS** expression returns a binary result or code from all the processes in the group. For example, if ten processes terminate with a return code of 0 and one process terminates with the return code of 1, the **STATUS** operator returns the value 1.

Example

```
EXECUTE proc_1 IS machine:foo0098
```

```
WHILE !ENDOF(proc_1)
```

```
  PAUSE 1
```

```
END WHILE
```

```
j = STATUS ( proc_1 )
```

```
IF j != 0 THEN
```

```
  PRINT "incorrect termination of program -> "
```

```
  PRINTLN j
```

```
  EXIT
```

```
END IF
```

Related Topics

[DO on page 902](#) | [ENDOF on page 903](#) | [EXECUTE on page 905](#)

SHELL

System Testing Supervisor Script Language.

Syntax

SHELL command

Purpose

The **SHELL** instruction executes a command by the System Testing Supervisor machine.

Description

SHELL commands block execution of the supervision script until the command is complete.

The command's execution log is not recorded in the supervision script execution log.

Example

...

SHELL ls /tmp -l ...

Related Topics

[DO on page 902](#) | [EXECUTE on page 905](#)

SET

System Testing Supervisor Script Language.

Purpose

The **SET** instruction sets an environment variable on either the System Testing Supervisor machine or the target machine.

Syntax

SET [*<hostname>* :] *<env_var>* << *<expression>*

SET [*<hostname>* :] *<env_var>* = *<string>*

<hostname> is the logical name of the target machine,

<env_var> is the name of the environment variable,

<expression> is a numerical expression,

<string> is a text string.

Description

<hostname> must be previously declared with a **HOST** instruction. If you do not specify a hostname, the **SET** instruction sets a local environment variable on the supervisor machine.

The environment variable is set when the SET instruction executes. It keeps its value until the end of the execution, or until it resets.

The string from the equal sign (=) to the end of the line belongs to the expression.

To evaluate an expression and assign it to the variable, use the << symbol. The expression may contain variables.

Example

```
HOST target IS workstation(10098)
```

```
...
```

```
SET LOCAL_TMP_DIR=/tmp
```

```
SET target:REMOTE_TMP_DIR << $TMPDIR
```

```
SET target:NUMVALUE <<i+2
```

Related Topics

[UNSET on page 914](#)

STOP

System Testing Supervisor Script Language.

Syntax

```
STOP <process>
```

where:

1. <process> is the identifier of a process

Purpose

The **STOP** instruction stops a process began with the **EXECUTE** instruction.

Example

```
HOST target IS antares
```

```
EXECUTE server IS machine:server
```

...

STOP server

Related Topics

[ENDOF on page 903](#) | [EXECUTE on page 905](#)

TRACE ... FROM

System Testing Supervisor Script Language.

Syntax

TRACE ON | OFF [FROM <host_name>]

Purpose

The **TRACE** instruction enables or disables execution traces from the machine specified by **host_name**, where this name was defined by a **HOST** instruction.

The traces are consolidated into the supervisor log file.

The keyword **ON** enables traces.

The keyword **OFF** disables traces.

Description

If the clause **FROM host_name** is not present, all traces from all machines are enabled or disabled.

If the clause **FROM host_name** is present, traces from machine **host_name** are enabled or disabled.

If you specify an unknown host name, you will receive an error when scenario execution fails.

By default, traces follow the **HOST** instruction.

Example

```
HOST machine_1 IS 193.5.4.3
```

```
HOST machine_2 IS remote
```

```
TRACE OFF FROM machine_1
```

WHILE

System Testing Supervisor Script Language.

Syntax

WHILE expression

instructions

END WHILE

Purpose

The **WHILE** instruction creates an execution loop.

Example

HOST machine IS 193.6.2.1

EXECUTE proc_1 IS machine:program

i = 1

WHILE !ENDOF (proc_1)

PAUSE 1

i = i + 1

END WHILE

j = STATUS (proc_1)

PRINT "execution time: "

PRINTLN i

PRINT "return code: "

PRINTLN j

Related Topics

[Expressions on page 897](#)

Variables

System Testing Supervisor Script Language.

A supervision script may contain integer variables only.

The system implicitly declares variables the first time they appear. The variable must first appear in an assignment instruction.

A variable must have a different name from any logical hostname defined in a **HOST** instruction, from any logical process name defined in an **EXECUTE** instruction, and from any **RENDEZVOUS** name. Otherwise, you will receive an error when scenario execution fails.

Variable names must begin with an upper or lowercase letter or with an underscore (`_`), followed, if necessary, by a series of letters, digits, or underscore characters.

Variable names are case sensitive. For example, the variable **Aa5** is different from the variable **aA5**.

Example

```
HOST machine IS 193.6.2.1
```

```
EXECUTE proc_1 IS machine:program
```

```
i = 1
```

```
-- declaration of i
```

```
WHILE !ENDOF ( proc_1 )
```

```
PAUSE 1
```

```
i = i + 1
```

```
END WHILE
```

```
j = STATUS ( proc_1 )
```

```
-- declaration of j
```

```
PRINT "execution time "
```

```
PRINTLN i
```

```
PRINT "return code "
```

```
PRINTLN j
```

Related Topics

[Expressions on page 897](#) | [PRINT on page 912](#) | [PRINTLN on page 913](#)

TIMEOUT

System Testing Supervisor Script Language.

Syntax

```
TIMEOUT <integer>
```

Purpose

The **TIMEOUT** instruction lets you define the time to wait for a rendezvous.

The value is measured in seconds.

Description

You may use only one **TIMEOUT** instruction in a test script.

The default value is 300 seconds (5 minutes).

Example

```
HOST machine_1 IS 193.5.4.3
```

```
HOST machine_2 IS remote.domain.fr
```

```
TIMEOUT 40
```

```
RENDEZVOUS phase_1
```

Related Topics

[RENDEZVOUS on page 922](#)

RENDEZVOUS

System Testing Supervisor Script Language.

Purpose

The **RENDEZVOUS** instruction synchronizes virtual testers and other processes.

Syntax

```
RENDEZVOUS <rendezvous>
```

<rendezvous> is a rendezvous identifier, previously declared by a **MEMBERS** statement.

Description

When the scenario reaches a **RENDEZVOUS** statement, the script is halted until all declared members arrive at the rendezvous. When the rendezvous is met by all members, the supervisor orders all processes to resume.

RENDEZVOUS identifiers must be unique in the script, including from logical process names or variable names, otherwise you will receive an error when execution fails.

If the rendezvous does not occur before the end of the timeout delay, you will receive an error. The default delay is five minutes. You can modify the delay with the **TIMEOUT** instruction.

Example

...

MEMBERS test1_test2 3

EXECUTE machine_1:test1

EXECUTE machine_2:test2

RENDEZVOUS test1_test2

Related Topics

[MEMBERS on page 910](#) | [TIMEOUT on page 921](#) | [ERROR on page 904](#)

Chapter 6. Test Manager Guide

This guide applies only to HCL OneTest™ Embedded for Eclipse IDE.

Generating test reports

You can generate tests reports from the results of a test harness run. Custom reports can be generated from XSL transforms.

To generate a test report:

1. In the test navigator, right-click the test results and select **Generate Report**.
2. On the **Report Scope** page, select the scope of the report, and click **Next**.
Choose from:
 - Select **Full report** to generate a complete report with all variables and values. This report includes code coverage information.
 - Select **Filtered report** to specify the level of information that you want to include in the report.
3. On the **Report Format Selection** page, select the XSL transformation file to generate the report, and click **Finish**.

Results

The test report is generated in the **Reports** folder of the project.

Generating 2D and 3D chart data

You can use test data (initial values, expected values and obtained values) to generate various types of 2D and 3D graphs and charts in the chart viewer. This representation of test data is particularly valuable associated with a data pool to generate test patterns

Before you begin

To generate a chart, you must activate the chart feature and select the variables that you want to display before running the test. You can display the chart in the chart viewer after the run.

Charts are more relevant when variables are associated with series of values that produce a pattern. Therefore, they work best when you use data pools and initial expressions with series or multiple expressions.

To configure a test to generate a graph or chart:

1. Open a test case in the test case editor and select a Check block.
2. Click the **Activate Chart**  button.
This activates chart data generation for the current Check block.
3. Under Chart Configuration, click the **Edit** link to select the variables that you want to include in the chart.

Result

The **Chart Configuration** window opens.

4. In Chart Type, select a type of chart:

Choose from:

- Line chart: Use this type of chart to display test data as a series of curves created from single data points relative to each other.
- XY line chart: Use this type of chart to display test data on a 2 dimensional plane. This requires at least 2 data patterns for the X and Y axis.
- 3D chart: Use this type of chart to display test data in a 3 dimensional space. This requires at least 3 data patterns for the X, Y, and Z axis.

5. Click **Add Curve** () to generate the data for one of the checked variables from the test case, select a variable in the list, and click **OK**.

6. Edit the variable data with the following parameters.

- a. In the Name column, type a name that will be displayed in the chart on the corresponding axis.
- b. In the Axis column, specify the X, Y or Z axis on which the variable will be drawn.
Ensure that the axis values match the type of chart. If you select an axis that is not available in an Line or XY chart, then the variable data will be displayed on an available axis.
- c. In the Source column, select whether the curve uses initial values, expected values or obtained values (with a min and max option) to create the curve.



Note: These settings define how the variable data is recorded during the test run. You can modify the way the data is displayed in the chart viewer after the run.

7. Repeat steps 5 and 6 for all the variables that you want to use to generate the chart data.

8. Click **Close**.

What to do next

After running the test, you can open the chart in the chart viewer.

Related information

[Viewing 2D and 3D charts on page 938](#)

[Creating data pools on page 202](#)

Publishing HTML reports to the Server

You can publish HTML reports generated from HCL OneTest™ Embedded to HCL OneTest™ Server.

Before you begin

You must have installed to HCL OneTest™ Server and started it. For more information about HCL OneTest™ Server, see the [Information center](#).

About this task

With the current version of HCL OneTest™ Embedded, the following reports can be published:

- Coverage
- Code Review
- Data Coupling
- Control Coupling
- Performance profiling
- Stack Size profiling

This feature applies to Eclipse IDE.

You must first setup the server URL in the preferences:

1. Open the **Preferences** menu.
2. Click **OneTest Server**.
3. Enter the Server URL..
4. Click **Test Connection** to test the connection to the Server.
5. If you connect to Server for the first time, specify the offline user token that you created on the server. To create a token on HCL OneTest Server, see [Generating an offline token](#).



Note: You can publish HTML reports for the following tests: Applications, Test Harness, and Test Suites.

6. Click **Apply and close**.
7. Right-click on the test you want to publish, select **Publish on OneTest Server**.
8. In the dialog that opens, select the project and the list of reports to be published.
9. If you have already published these reports, click **Overwrite result** to overwrite those results.
10. Click **Publish** .

Results

To view the published results, log in to the Server, select the project, and then click **Results**.

Opening runtime analysis reports

After running an instrumented application or a test harness, runtime analysis results can be displayed in a series of specialized viewers or in HTML format reports.

Before you begin

Runtime analysis reports are available after having successfully run a test harness or an instrumented application. Only the reports for the runtime analysis tools that were selected during the run are available.

To open a runtime analysis report:

1. Right-click the results that are available in the **Project Explorer**, in your project under **Test > Application Results** after the instrumented application has run.



Note: The test results are listed with a timestamp.

2. Click **Open With** to select the viewer for the runtime analysis results that you want to see, or click **Open With > HTML reports** to select the appropriate HTML report.

Choose from:

- Coverage
- Memory Profiling
- Performance Profiling
- Application Profiling
- Stack Size Profiling
- Control Coupling
- Data Coupling
- Static Metrics
- Runtime Tracing
- Code review

About test reports

Test reports are displayed in the test report viewer.

The Report Explorer displays each element of a test report with a *Passed* or *Failed* symbol.

- Elements marked as *Failed* are either a failed test, or an element that contains at least one failed test.
- Elements marked as *Passed* are either passed tests or elements that contain only passed tests.

Test results are displayed for each instance, following the structure of the test harness.

Each test report contains a report header containing the following elements:

- The version of the product used to generate the test and the timestamp of the test report.
- The path and name of the project files used to generate the test.
- The total number of test cases *Passed* and *Failed*.

These statistics are calculated from the actual number of test elements listed in the report.

The graphical symbols in front of the node indicate if the test harness, test case, or variable check is Passed or Failed. A test is failed if it contains at least one failed variable check. Otherwise, the test is considered passed. Click the Information button to obtain the following information:

- Number of tests run
- Number of tests passed
- Number of tests failed

A variable check is failed if the expected expression and the obtained value are not identical, or if the obtained value is not within the expected range.

If a variable belongs to an environment, an environment header is edited. In the report, variables are edited according to the value of the **Display variables** setting for the test harness. The following table summarizes the editing rules:

Table 5.

Result	Display variables = All variables	Display variables = Incorrect variables	Display variables = Failed tests only
Passed	Variable edited automatically	Variable not edited	Variable not edited
Failed	Variable edited automatically	Variable edited automatically	Variable edited if incorrect

About coverage reports

The coverage report view displays code coverage information generated by the Code Coverage feature.

The coverage report contains the following elements:

- The **Source** page shows the source code under analysis, highlighted with the actual coverage information.
- The **Rates** page provides detailed coverage rates for each activated coverage type.
- The **Outline** view displays the source code components and with an coverage rate bar.

You can use the **Outline** view to navigate through the report. Click **Root** to display a global coverage graph, or click a source code component in the **Outline** to go to the corresponding line on the **Source** page. Jump directly to the next or previous uncovered portion of source code by using the **Next Uncovered Line** or **Previous Uncovered Line** buttons in the toolbar.

Source page

By default, the **Source** page of the coverage report displays covered and uncovered lines of code in the following colors:

- Green for covered lines of code,
- Red for uncovered lines of code,
- Orange for partially covered lines of code,
- Blue for justified lines of code,
- Blue with the + icon for justified but covered lines of code, which means that they should not be justified.
- Red with - icon for unreachable code.

You can change these colors in the code coverage report preferences. In the main toolbar, click **Window > Preferences > Viewers > Coverage viewer**, you can modify the text color for the covered lines, covered lines with justify, justified lines, partially covered lines, and uncovered lines.

```

int count(int x) {
    if (x >= 0) {
        return x - 1;
    } else {
#pragma attol cov_justify (0,block,, "block not reachable")
        return 0;
    }
}

int main(void) {
    puts("!\n");
#pragma attol cov_justify (0,cond, "cond not reachable")
#pragma attol cov_justify (0,cond, "cond not reachable")
    if (a >= 2) {
#pragma attol cov_justify (block, "block not reachable")
        int b = a + 1;
#pragma attol cov_justify (return, "return not reachable")
        return 1;
    }
#pragma attol cov_justify (2,cond, "a!=2:true", "cond not reachable")
#pragma attol cov_justify (0,mcde, "TF;TT;FX", "mcde not reachable")
    int b = a != 2 && (a >= 2) == 1;
    return count(b);
    int c = a;
}

```

For non-covered line of codes that are justified, click on the blue attributes value to see more details about the justification text.

```

int count(int x) {
    if (x_≥_0) {
        return x -1;
    } else {
#pragma attol cov_justify (0,block,, "block not reachable")
        return 0;
    }
}

int main(void) {
    puts("!!!Hello World!!!");
#pragma attol cov_justify (2,cond,:true, "cond not reachable")
#pragma attol cov_justify (implicit, "else not reachable")
    if (a_≥_0) {
#pragma attol cov_justify (block, "block not reachable")
        int b = a+1;
#pragma attol cov_just: Justification: return not reachable not reachable")
        return 1;
    }
#pragma attol cov_justify (2,cond, "a!=2:true", "cond not reachable")
#pragma attol cov_justify (0,mcdc, "TF;TT;FX", "mcdc not reachable")
    int b = a_!=_2 && (a_≠_2)_==_1;
    return count(b);
    int c = a;
}

```



Note: In C source files, the last bracket '}' in a function after a return statement is always displayed as uncovered in the coverage report, even if the function reports 100% coverage.

The **Source** page provides hypertext navigation throughout the source code:

- Click a plain underlined function call to jump to the definition of the function.
- Click a dashed underlined text to view additional coverage information in a pop-up window.
- Right-click any line of code and select **Edit Source** to open the source file in the source code editor.
- Some macro calls are preceded with a magnifying glass icon. Click the magnifying glass icon to expand the macro in a pop-up window with the usual coverage color codes.

A *test-by-test* analysis mode allows you to refine the coverage analysis. In *test-by-test* mode, an **Available tests** section in the **Outline** view allows you to select and combine coverage results for different runs. To enable this mode, select **Test-by-Test** in the toolbar.

The hit count tool displays the number of times that a selected branch was covered. Hit count is only available when *test-by-test* analysis is disabled and when the hit count option has been enabled for the selected configuration. To enable the hit count tool, right-click the source page and select **Hit Count**.

The cross reference tool displays the name of tests that executed a selected branch. Cross reference is only available in *test-by-test* mode. To enable the hit count tool, right-click the source page and select **Cross Reference**.

Rates page

The Rates page displays a table with the coverage information for each function.

To view the coverage rate and type for a particular component, select the component in the **Outline** view. Select the **Root** node to view coverage rates for all current files.

To toggle the displayed format between absolute values, percentages, or both, click on the **Display** line located just above the table. To sort the table by one of the values, click the column title. Coverage rates are updated dynamically as you navigate through the **Outline** view and as you select various coverage types.

About memory profiling reports

After execution of an instrumented application, the Memory Profiling report provides a summary diagram and a detailed report for both byte and memory block usage.

A memory block is a number of bytes allocated with a single malloc instruction. The number of bytes contained in each block is the actual amount of memory allocated by the corresponding allocation instruction.

Summary diagrams

The summary bar graph diagrams provide a quick overview of memory usage in blocks and bytes, where:

- **Allocated** is the total memory allocated during the execution of the application.
- **Unfreed** is the memory that remains allocated after the application was terminated.
- **Maximum** is the highest memory usage encountered during execution.

Detailed report

The detailed section of the report lists memory errors and warnings described in the following paragraphs.

You can use the Filter Errors and Warnings button to select the level of information that you want to display.

Detected memory errors

Error messages indicate invalid program behavior. These are serious issues you should address before you check in code.

Freeing Freed Memory (FFM)

An FFM message indicates that the program is trying to free memory that has previously been freed.

This message can occur when one function frees the memory, but a data structure retains a pointer to that memory and later a different function tries to free the same memory. This message can also occur if the heap is corrupted.

Memory Profiling maintains a free queue, whose role is to actually delay memory free calls in order to compare with upcoming free calls. The length of the delay depends on the Free queue length and Free queue threshold Memory Profiling Settings. A large deferred free queue length and threshold increases

the chances of catching FFM errors long after the block has been freed. A smaller deferred free queue length and threshold limits the amount of memory on the deferred free queue, taking up less memory at run time but providing a lower level of error detection.

Freeing Unallocated Memory (FUM)

An FUM message indicates that the program is trying to free unallocated memory.

This message can occur when the memory is not yours to free. In addition, trying to free the following types of memory causes a FUM error:

- Memory on the stack.
- Program code and data sections.

Freeing Invalid Memory (FIM)

An FIM message indicates that the program is trying to free allocated memory with the wrong instruction.

This message can occur when the memory free instruction mismatches the memory allocation instruction. For example, a FIM occurs when memory is freed with a free instruction when it was allocated with a new instruction.

Late Detect Array Bounds Write (ABWL)

An ABWL message indicates that the program wrote a value before the beginning or after the end of an allocated block of memory.

Memory Profiling checks for ABWL errors whenever `free()` or `dump()` routines are called, or whenever the free queue is actually flushed.

This message can occur when you:

- Make an array too small. For example, you fail to account for the terminating NULL in a string.
- Forget to multiply by `sizeof(type)` when you allocate an array of objects.
- Use an array index that is too large or is negative.
- Fail to NULL terminate a string.
- Are off by one when you copy elements up or down an array.

Memory Profiling actually allocates a larger block by adding a Red Zone at the beginning and end of each allocated block of memory in the program. Memory Profiling monitors these Red Zones to detect ABWL errors.

Increasing the size of the Red Zone helps HCL® OneTest™ Embedded catch bounds errors before or beyond the block at the expense of increased memory usage. You can change the Red Zone size in the Memory Profiling Settings.

The ABWL error does not apply to local arrays allocated on the stack.



Note: The ABWL error in the HCL® OneTest™ Embedded Memory Profiling tool only applies to heap memory zones and not to global or local tables.

Late Detect Free Memory Write (FMWL)

An FMWL message indicates that the program wrote to memory that was freed.

This message can occur when you:

- Have a dangling pointer to a block of memory that has already been freed (caused by retaining the pointer too long or freeing the memory too soon).
- Index far off the end of a valid block.
- Use a completely random pointer which happens to fall within a freed block of memory.

Memory Profiling maintains a free queue, whose role is to actually delay memory free calls in order to compare with upcoming free calls. The length of the delay depends on the Free queue length and Free queue threshold Memory Profiling Settings. A large deferred free queue length and threshold increases the chances of catching FMWL errors. A smaller deferred free queue length and threshold limits the amount of memory on the deferred free queue, taking up less memory at run time but providing a lower level of error detection.

Memory Allocation Failure (MAF)

An MAF message indicates that a memory allocation call failed. This message typically indicates that the program ran out of paging file space for a heap to grow. This message can also occur when a non-spreadable heap is saturated. After Memory Profiling displays the MAF message, a memory allocation call returns NULL in the normal manner. Ideally, programs should handle allocation failures.

Freeing Freed Memory (FFM)

An MAF message indicates that a memory allocation call failed. This message typically indicates that the program ran out of paging file space for a heap to grow. This message can also occur when a non-spreadable heap is saturated.

After Memory Profiling displays the MAF message, a memory allocation call returns NULL in the normal manner. Ideally, programs should handle allocation failures.

Core Dump (COR)

A COR message indicates that the program generated a UNIX core dump. This message can only occur when the program is running on a UNIX target platform.

Detected memory warnings

Warning messages indicate a situation in which the program might not fail immediately, but might later fail sporadically, often without any apparent reason and with unexpected results. Warning messages often pinpoint serious issues you should investigate before you check in code.

Memory in Use (MIU)

An MIU message indicates heap allocations to which the program has a pointer.



Note: On exit, small amounts of memory in use in programs that run for a short time are not significant. However, you should fix large amounts of memory in use in long running programs to avoid out-of-memory problems.

Memory Profiling generates a list of memory blocks in use when you activate the MIU Memory In Use option in the Memory Profiling Settings.

Memory Leak (MLK)

An MLK message describes leaked heap memory. There are no pointers to this block, or to anywhere within this block. Memory Profiling generates a list of leaked memory blocks when you activate the MLK Memory Leak option in the Memory Profiling Settings.

This message can occur when you allocate memory locally in some function and exit the function without first freeing the memory. This message can also occur when the last pointer referencing a block of memory is cleared, changed, or goes out of scope. If the section of the program where the memory is allocated and leaked is executed repeatedly, you might eventually run out of swap space, causing slow downs and crashes. This is a serious problem for long-running, interactive programs.

To track memory leaks, examine the allocation location call stack where the memory was allocated and determine where it should have been freed.

You can ignore memory leaks that do not have a call stack, for memory allocations that occur before the application starts by changing the configuration.

Memory Potential Leak (MPK)

An MPK message describes heap memory that might have been leaked. There are no pointers to the start of the block, but there appear to be pointers pointing somewhere within the block. In order to free this memory, the program must subtract an offset from the pointer to the interior of the block. In general, you should consider a potential leak to be an actual leak until you can prove that it is not by identifying the code that performs this subtraction.

Memory in use can appear as an MPK if the pointer returned by some allocation function is offset. This message can also occur when you reference a substring within a large string.

Alternatively, leaked memory might appear as an MPK if some non-pointer integer within the program space, when interpreted as a pointer, points within an otherwise leaked block of memory. However, this condition is rare.

Inspection of the code should easily differentiate between different causes of MPK messages.

Memory Profiling generates a list of potentially leaked memory blocks when you activate the MPK Memory Potential Leak option in the Memory Profiling Settings.

File in Use (FIU)

An FIU message indicates a file that was opened, but never closed. An FIU message can indicate that the program has a resource leak.

Memory Profiling generates a list of files in use when you activate the FIU Files In Use option in the Memory Profiling Settings.

Signal Handled (SIG)

A SIG message indicates that a system signal has been received.

Memory Profiling generates a list of received signals when you activate the SIG Signal Handled option in the Memory Profiling Settings.

About performance profiling reports

The performance profiling report provides function profiling data for your program and its components so that you can see exactly where your program spends most of its time.

Top functions

This section of the report provides a percentage graph of the largest time consumers detected by performance profiling in the application.

Performance summary

This section of the report indicates, for each instrumented function, procedure or method (collectively referred to as functions), the following data:

- Calls: The number times the function was called
- Function (F) time: This value indicates the total time spent executing the function, exclusive of any calls to its descendants.
- Function+descendant (F+D) time: The total time spent executing the function and any of its descendants (any other functions called by this function).



Note: Because each of the descendants may have been called by other functions, it is not enough to simply add the descendants' F+D to the caller function's F. In fact, it is possible for the descendants' F+D to be larger than the calling function's F+D. The following example demonstrates three functions *a*, *b* and *c*, where both *a* and *b* each call *c* once:

**Table 6.**

Func- tion	F	F+D
a	5	15
b	5	15
c	20	20

The F+D value of a is less than the F+D value of c because the F+D of a (15) equals the F of a (5) plus one half the F+D of c ($20/2=10$).

- F Time (% of root) and F+D Time (% of root): Same as above, expressed in percentage of total execution time
- Average F Time: The average time spent each time the function was executed.
- Min F+D: The minimum time spent executing the function and any of its descendants.
- Max F+D: The maximum time spent executing the function and any of its descendants.



Note: The Min and Max values are optional because their calculation uses a large amount of memory. To calculate these values, you must activate the option in the Configuration Settings for the corresponding node.

Click on a function in the table to open the source file in the source code editor. To sort the table by one of the values, click the column title.

About metrics results

The metrics report provides static testability and complexity measurements of the source files of your project. Source code metrics are created each time a source file is added to the project.

The scope of the metrics report depends on the selection made in the **Outline** view. This can be a file, one or several classes or any other set of source code components.

The metrics window provides hyperlinks to the actual source code. Click the name of a source component to open the source code editor at the corresponding line.

Complexity metrics

The $V(g)$ or *cyclomatic number* is a measure of the complexity of a function, which is correlated to the difficulty of testing the function. The typical $V(g)$ for a function is between 1 and 10. A value of 1 means that the code has no branching. The cyclomatic complexity of a function should not exceed 10.

Halstead complexity measurement was developed to measure a program module's complexity directly from source code, with emphasis on computational complexity. The measures were developed by the late Maurice Halstead as a

means of determining a quantitative measure of complexity directly from the operators and operands in the module. Halstead provides various indicators of the module's complexity.

The Metrics Viewer presents $V(g)$ and Halstead values of a function in the metrics report when a function is selected in the **Outline** view. At the **Root** level, the same statistical treatment is provided for all functions in the source file.

File level metrics

Comment only lines

The number of comment lines that do not contain any source code.

Comments

The total number of comment lines.

Empty lines

The number of lines with no content.

Source only lines

The number of lines of code that do not contain any comments.

Comment only lines

The number of comment lines that do not contain any source code

Lines

The total number of lines in the source file.

Comment rate

The percentage of comment lines against the total number of lines.

Source lines

The total number of lines of source code

File, Class or Package, and Root Level Metrics

These numbers are the sum of metrics measured for all the components of a given file, class or package.

Total statements

total number of statement in child nodes

Maximum statements

The maximum number of statements in the selected scope.

Maximum level

The highest nesting level reached in the selected scope.

Maximum $V(g)$

The average cyclomatic number of the selected scope.

Standard deviation V(g)

Standard deviation V(g) of the selected scope

Sum of V(g)

Total V(g) for the selected scope.

Viewing 2D and 3D charts

Use the chart viewer to display test data (initial values, expected values and obtained values) that was recorded during a run.

Before you begin

To display a chart, you must have activated the chart feature in the Check block of the test case and selected the variables to display before running the test. To configure a test case to record chart data, see [Generating 2D and 3D chart data on page 924](#).

To display a 2D or 3D chart in the chart viewer:

1. After running the test, right-click the test run in the test navigator and selecting **Open With > Chart**.

Result

The chart opens in the chart viewer.

2. Click the toolbar buttons to select alternative chart types.

The available chart types depend on the type and number of variables that were recorded during the test.

3. If necessary, you can edit the way the data is displayed in the chart viewer.

Choose from:

- Click the **Data** tab to display the data values that were recorded during the test.
- Use the **Outline** view to hide or show data sets and change their display color.
- If the chart is a 3D chart, you can click and drag the chart to change the view angle.
- Click the **Curve Definition** button to redefine colors, axis settings and the chart type.



Note: These settings only affect the way the recorded data is displayed in the chart viewer. It does not change the variable name and axis settings that were configured in the test case editor.

- Click the **Configure Chart** button to access advanced graph display options.

Related information

[Generating 2D and 3D chart data on page 924](#)

[Creating data pools on page 202](#)

Chapter 7. Reference Guide

Use these additional topics to gain more knowledge about the product.

UI reference

HCL OneTest™ Embedded preferences

Use these preferences to change general settings and file locations for HCL OneTest™ Embedded for Eclipse IDE.

To access the **Preferences**, click **Window > Preferences > .**

Installation directory

Specifies the directory in the filesystem where the product is installed. The product uses this path to locate its own resource.

Verbose mode

Enables verbose output to the **Console** window. If you disable this option... (?)

Clear temporary intermediate files before running a component test

Enable this option to clear temporary intermediate files before a run. If you disable this option, new results will be merged with the existing results in the report. (?)

Related information

[Overview on page 9](#)

Call graph preferences

Use these preferences to change how the call graph is displayed in HCL OneTest™ Embedded for Eclipse IDE.

To access the **Call Graph Preferences**, click **Window > Preferences > OneTest Embedded > Call Graph**.

Call graph colors and styles

Use these preferences to specify the colors and styles used in the call graph. Click a color to display a color picker.

Editor preferences

Use these preferences to change the behavior of the test case, test harness, and stub editors in HCL OneTest™ Embedded for Eclipse IDE.

To access the **Editor Preferences**, click **Window > Preferences > > Editors**.

Editor colors and styles

Use these preferences to specify the colors and styles used in the call graph. Click a color to display a color picker.

Display variable under test smart tooltip on check blocks

Select this option to display a tooltip over check blocks.

Display chart configuration section

Select this option to display the chart configuration section in the test case, test harness, and stub editors.

Marker colors

Use these preferences to specify the colors used for error and warning markers in the test case, test harness, and stub editors.

Diagram

Use these preferences to specify the colors used in the diagrams in the test case, test harness, and stub editors.

Coverage Bars

Use these preferences to specify the colors used in the coverage bars that are displayed in the test case, test harness, and stub editors.

Run Results

Use these preferences to specify how run results are displayed in the test case, test harness, and stub editors.

C Syntax coloring preferences

Use these preferences to change the color schemes for C code in HCL OneTest™ Embedded for Eclipse IDE.

To access the **C Syntax Coloring Preferences**, click **Window > Preferences > > Call Graph**.

Coloring styles

Select a style and specify a foreground color, a background color and font styles.

Sample

This area provides an example of C++ source code with the selected coloring styles.

Errors and warnings preferences

Use these preferences to change how errors and warnings are displayed in HCL OneTest™ Embedded for Eclipse IDE.

To access the **Errors and Warnings Preferences**, click **Window > Preferences > > Errors and Warnings**.

Message types

For each type of message in the list, specify whether to consider it an **Error**, a **Warning** or to **Ignore** the message. Errors and warnings are logged in the **Console**.

Navigator preferences

Use these preferences to change the behavior of the project navigator in HCL OneTest™ Embedded for Eclipse IDE.

To access the **Navigator Preferences**, click **Window > Preferences > > Navigator**.

Sort result files by ascending date

Select this option to sort the result files by date. If this option is disabled, the result files are sorted by alphabetical order.

Report generation preferences

Use these preferences to change how reports are generated in HCL OneTest™ Embedded for Eclipse IDE.

To access the **Report Generation Preferences**, click **Window > Preferences > > Report Generation**.

XML Generation Options

Specify the XML version and XML encoding information for the XML header.

Open the report after generation

Select this option if you want the report to be automatically opened after it is generated.

Target deployment port preferences

Use these preferences to change how target deployment ports (TDP) are generated in HCL OneTest™ Embedded for Eclipse IDE.

To access the **Target Deployment Port Preferences**, click **Window > Preferences > > Target Deployment Port**.

Default Target Deployment Port

Specify the target deployment port (TDP) that is selected by default when you create a project.

Search path

Specify a list of directories where TDPs are located. TDPs are searched in the specified order. Select a directory and click **Up** or **Down** to change the search order.

Test generation preferences

Use these preferences to change how tests are generated in HCL OneTest™ Embedded for Eclipse IDE.

To access the **Test Generation Preferences**, click **Window > Preferences > > Test Generation**.

Initialize char array as a character string (ex: var="")

Select this option to initialize character arrays as character strings by default.

Default array size for empty array in function parameter

Specify the default empty array size.

When function parameter is pointer type:

Specify the whether the default test generation behavior when pointer types are encountered as parameters of a function is to leave the parameter as a pointer or to instantiate the variable as a pointed type.

Except for following type

Specify the types for which the previous setting does not apply.

HCL OneTest™ Embedded preferences in Eclipse

HCL OneTest™ Embedded

for Eclipse CDT

The HCL OneTest™ Embedded preferences in the Eclipse workbench allow you to configure settings for HCL OneTest™ Embedded in Eclipse.

HCL OneTest™ Embedded preferences

The HCL OneTest™ Embedded preferences allow you to change the following settings:

- **Binary Directory:** Specifies the directory where HCL OneTest™ Embedded binaries are located.
- **Default TDP:** Specifies the default TDP that will be used in the **Default.settings** configuration when you enable HCL OneTest™ Embedded in a C or C++ project.
- **Verbose Mode:** Enables detailed information of HCL OneTest™ Embedded components in the console during execution.
- **Delete intermediate files:** Select this option to automatically delete previous intermediate files each time you run a test.

Results Editor preferences

The **Results Editor** preferences allow you to change the appearance of your Test and Runtime Analysis reports in Eclipse.

These preferences are identical to the corresponding preferences in the HCL OneTest™ Embedded user interface.

- **Styles:** This list allows you to select one or several styles that you want to change. To change several styles at the same time, you can perform multiple selections in the style list.
- **Font:** This allows you to change the font type and size for the selected style.
- **Text Color:** This allows you to change the foreground and background colors for the selected style. This opens a standard color palette.
- **Text Attributes:** This allows you to set the selected style to Bold, Italic, Underlined or Dashed.

To access the HCL OneTest™ Embedded preferences in Eclipse:

1. In Eclipse, select **Window > Preferences**.
2. HCL OneTest™ Embedded

In the **Preferences** window, expand HCL OneTest™ Embedded.

Related Topics

[User interface preferences on page 1009](#) ||

Viewer preferences

Modify these preferences to change how report viewers are displayed in HCL OneTest™ Embedded for Eclipse IDE.

To access the **Viewer Preferences**, click **Window > Preferences > > Viewers**.

Common preferences for viewers

Use these preferences to specify the colors and styles that are used by all viewers. Click a color to display a color picker.

Chart viewer colors and styles

Use these preferences to specify the colors and styles used in charts. Click a color to display a color picker.

Code coverage viewer colors and styles

Use these preferences to specify the colors and styles used in code coverage reports. Click a color to display a color picker.

Code review colors and styles

Use these preferences to specify the colors and styles used in code review reports. Click a color to display a color picker.

Memory profiling colors and styles

Use these preferences to specify the colors and styles used in memory profiling reports. Click a color to display a color picker.

Static metrics colors and styles

Use these preferences to specify the colors and styles used in static metrics reports. Click a color to display a color picker.

Performance profiling colors and styles

Use these preferences to specify the colors and styles used in performance profiling reports. Click a color to display a color picker.

Runtime Tracing colors and styles

Use these preferences to specify the colors and styles used in UML sequence diagrams displayed in the runtime tracing viewer. Click a color to display a color picker.

Test report colors and styles

Use these preferences to specify the colors and styles used in test reports. Click a color to display a color picker.

TDP configuration settings

Use the target deployment port (TDP) build settings to adapt the test harness to the target platform.

To access to the target deployment port build settings, right-click the project and select **Properties > C Build > Settings > Build TDP**.

Target deployment port

Target deployment port

This setting allows you to override the target deployment port (TDP) of the entire configuration for the selected element. Use this for example if you are mixing different languages or compilers within a single project. Any child elements will use the default configuration settings from this target deployment port, such as compilation flags. When you change the TDP within a configuration, the settings are overwritten using the default settings of the new TDP.

Directory

Specifies the TDP directory name or relative path. By default, HCL OneTest™ Embedded searches for the TDP directory in the directories that are declared in the project preferences.

Path

Indicates the location of the selected target deployment port.

Initial definition file

Points to the default `.ini` file in the TDP directory.

Source file language

Specifies the language of the TDP.

Object file extension

Specifies the default extension for object files produced with the current TDP.

Static library file extension

Specifies the file extension used for static library files.

Dynamic library file extension

Specifies the file extension used for dynamic library files.

Binary file extension

Specifies the default extension for executable binaries produced with the current TDP (for example: `.exe`).

Source file extension

Specifies the default extension for source files used with the current TDP.

Compiler options**Preprocessor flags**

Specify any compilation flags that are to be sent to the compiler.

Compiler flags

Specify any additional command line options to be sent to the compiler.

Preprocessor macro definitions

Specify any macro definition that are to be sent to both the compiler preprocessor (if used) and the Test Script Compilers. Several generation conditions must be separated by a comma ',' with no space.

You can use a comma inside a condition, preceded a backslash character. For example: `semTake(x`

```
\,y)=TestRTsemTake(x\,y),WIN32,_DEBUG
```

Include directories

Use this setting to specify include directories that are specific to the current TDP. Click the ... button to create or modify a list of directories for included files when the include statement is encountered in source code and test scripts. In the directory selection box, use the **Up** and **Down** buttons to indicate the order in which the directories are searched.

Linker options**Linker flags**

Specify any particular flags that are to be sent to the linker.

Libraries

Specify a list of object libraries to be linked to the generated executable. Type the command line option as required by your linker. See the documentation provided with your development tool for the correct syntax.

Library paths

Click the ... button to create or modify a list of directories for library link files. In the directory selection box, use the **Up** and **Down** buttons to indicate the order in which the directories are searched.

Advanced**Output buffer size**

Sets the size of the output buffer. A smaller output buffer can save memory when resources are limited. A larger buffer improves performance. The default setting for the output buffer is 1024 bytes.

Time measurement

Selects whether to use a real-time **Operating system clock** or a **Process or task clock** for time measurement, if both options are available in the current target deployment port. Otherwise, this setting is ignored.

Multi-threaded application

When selected, specifies whether to protect target deployment port global variables against concurrent access when you are working in a multithreaded environment such as Posix, Solaris or Windows™. This can cause an increase in size of the TDP and an impact on performance; therefore, only select this option when necessary.

Multi-process application

When selected, specifies whether to produce a different output file for each process in forked applications.

Maximum number of threads

When the multithread option is enabled, this setting sets the maximum number threads that can be run at the same time by the application.

Override compiler flags

By default, the TDP is compiled with the build compiler flags. Use this setting to override the build compiler flags with specific flags for compiling the TDP.

Use source compiler flags

Select **Yes** to use the source build compiler flags to compile the test driver. Select **No** to use the default TDP settings.

Link flags for library format (for library files only)

Link flags for generating the TDP as a shared library or DLL.

TDP output format

This setting specifies how the TDP is linked to the application. **None**: No TDP is generated. Use this setting if the TDP is already included in another section of the application. **Object file (.obj, .o)**: Default setting. Use this setting if your application does not use shared libraries. **Static library (.lib, .a)**: Use this setting to link the TDP as a static library. **Dynamic library (.dll, .so)**: Use this setting to link the TDP as a dynamic library for most cases when shared libraries are involved.

Use of unloadable libraries

Use the setting if your application uses shared libraries that can be unloaded dynamically from memory. See Unloadable libraries for details. **None**: The application does not dynamically unload libraries during execution. This is an unloaded library: Select this if the selected node is a library node that can be dynamically unloaded during execution. **Uses unloaded libraries**: Select this if the selected node is an application or test node that can use unloadable libraries.

Related reference

[Build configuration settings on page 947](#)

Related information

[Target deployment port overview on page 12](#)

Build configuration settings

Use the build configuration settings to change how the project is built in HCL OneTest™ Embedded for Eclipse IDE.

To access the build settings, right-click the project and select **Properties > C Build > Settings > Build Settings**.

General

Selective instrumentation

Build options

Specifies the runtime analysis options for the selected resource. This is also where you enable the runtime analysis tools. See [Enable runtime analysis tools on page 50](#).

Instrument inline methods

Extends instrumentation to inline methods.

Instrument included methods or functions

Extends instrumentation to included methods or functions.

Excluded files

Specifies a list of source files that are parsed during the run, but are not instrumented. Click the ... button and use the **Add** and **Remove** buttons to select the files to be excluded.

Excluded directories

Specifies a list of directories containing source files that are parsed during the run, but are not instrumented. Click the ... button and use the **Add** and **Remove** buttons to select the directories to be excluded.

Instrumentor hides all warnings

Set this option to **Yes** to hide instrumentation warnings.

Snapshot

In some case, such as with applications that never terminate or when working with timing or memory-sensitive targets, you might need to dump traces at specific points in your code. See [Generating SCI Dumps](#) for more information.

On function entry

Allows you to specify a list of function names, from your source code, that will dump traces at the beginning of the function. Click ... and use **Add** and **Remove** to create a list of function names.

On function return

Allows you to specify a list of function names, from your source code, that will dump traces at the end of the function. Click ... and use **Add** and **Remove** to create a list of function names.

On function call

Allows you to specify a list of function names, from your source code, that will dump traces before the function is called. Click ... and use **Add** and **Remove** to create a list of function names.

Static file storage

Depending on the runtime analysis feature, the product generates `.tsf` or `.fdc` temporary static data files during source code instrumentation of the application under analysis.

Code coverage static file storage (.fdc)

These settings apply to code coverage `.fdc` static trace files:

- **Build Directory:** Select this option to use the current directory for all generated files.
- **Other Directory:** Select this option to define a specific directory.
- **Source Directory:** Select this option to use the same directory as the source under analysis.
- **Use Single Temporary File (.fdc):** By default, code coverage produces one `.fdc` file for each instrumented source file. Select this option to use a single `.fdc` file for all instrumented source files, and specify its location.

FDC directory or name

If the **Use single temporary file (.fdc)** option is selected in the previous setting, specify a location for the `.fdc` file.

Memory profiling, performance profiling, runtime tracing static file storage (.tsf)

This setting applies to memory profiling, performance profiling and runtime tracing `.tsf` static trace files.

- **Build directory:** Select this option to use the current directory for all generated files.
- **Other directory:** Select this option to define a specific directory.
- **Source directory:** Select this option to use the same directory as the source under analysis.

- Use single temporary file (`.tsf`): By default, memory profiling, performance profiling and runtime tracing produce one `.tsf` file for each instrumented source file. Select this option to use a single `.tsf` file for all instrumented source files, and specify its location.

TSF directory or name

If the **Use single temporary file (.tsf)** option is selected in the previous setting, specify a location for the `.tsf` file.

Advanced options

Identification header

Select this option to add an identification header to files generated by the instrumentation. The header includes the command line used to generate the file, the version of the product, the date and operating system information.

Application includes system files

By default, unused methods within a template are ignored by the instrumentation. Set this option to **Yes** to analyze and instrument all template methods, even if they are not used.

Internal data allocation

Set this option to **Yes** if the application includes system files such as `windows.h` in Windows™ or `pthread.h` in UNIX™.

Full template instrumentation

Select either Static declaration or Dynamic allocation as the memory allocation method for storing intermediate variables in the instrumented source code.

Check internal data before use

This setting allows you to add command line options for the instrumentation. Normally, this line should be left blank.

Use alternate checksum

Select **Yes** to calculate a more unambiguous checksum for `.fdc` and `.tsf` files. Select No to remain compatible with existing `.fdc` and `.tsf` files. Before using this option, you must delete existing `.fdc` and `.tsf` files, which will be recreated with the new checksum. File keys are not changed by this option.

Additional instrumentor options

Set this option to **Yes** if you are experiencing crashes of the application when runtime analysis features are engaged. This option improves compatibility but increases memory usage.

Generate TPM files

Set this option to Yes if you want to generate metrics for the test process monitor.

Code Coverage

Instrumentation control

You can use the coverage type settings to declare various types of coverage. See [Coverage levels on page 55](#) for more information about these settings.

Coverage level functions

Select between function **Entries**, **With exits**, or **None**.

Coverage level calls

Select **Yes** or **No** to toggle call code coverage.

Coverage level blocks

Select the desired block code coverage type. You can combine, enable, or disable any of these coverage types before running the application node. All coverage types selected for instrumentation can be filtered out in the coverage viewer.

Exclude for loops

Select **Yes** to exclude for loops from instrumentation. Only *while* and *do* loops are instrumented.

Coverage level conditions

Selects the condition level of code coverage to be included in the report:

- **None**: The coverage report ignores conditions.
- **Basic**: Only basic conditions are included in the coverage report.
- **Modified (MC/DC)**: Only modified conditions are included in the coverage section of the test report.
- **Modified and Multiple**: Both modified and multiple conditions are included in the coverage report.
- **Forced Modified (MC/DC)**: The report includes modified conditions where all operators are replaced with bitwise operators.
- **Forced Modified and Multiple**: The report includes modified and multiple conditions where all operators are replaced with bitwise operators.

Condition in expressions

Select **Yes** to consider relational operators in an expression (for example: $y = (a > 0)$) as conditions.

Bitwise as logical

Select **Yes** to instrument bitwise operators as logical when both operands are booleans.

Ternary coverage

When this option is selected, code coverage reports ternary expressions as statement blocks.

Information mode

This setting specifies the information modes to be used by code coverage.

- **Default (Optimized for Code Size and Speed)**: This setting uses one byte per branch to indicate branch coverage.
- **Compact (Optimized for Memory)**: This setting uses one bit per branch. This method saves target memory but uses more CPU time.
- **Report Hit Count**: This adds information about the number of times each branch was executed. This method uses one integer per branch.

Excluded function calls

Specifies a list of functions to be excluded from the call coverage instrumentation type, such as *printf* or *fopen*. Use the **Add**, **Remove** buttons specify the functions to be excluded.

Not returning functions

Type the identifiers (not signatures) of the functions that do not return (functions that execute a *longjmp* or *exit*).

Advanced options**Trace file name (.tio)**

this allows you to specify a path and filename for the `.tio` dynamic coverage trace file.

Key ignore source file path

Identifies source files based only on the filename instead of the complete path. Use this option to consolidate test results when a same file can be located in different paths. This can be useful in some multi-user environments that use source control. If you use this option, make sure that the source file names used by your application are unique.

User comment

This adds a comment to the code coverage report. This can be useful for identifying reports produced under different configurations. To view the comment, click the a magnifying glass symbol that is displayed at the top of your source code in the coverage viewer.

Report summary

Select **Yes** to add the coverage summary to the summary text file of the selected node.

On-the-fly frequency dump

Specify the function call number after which the coverage results are updated dynamically during execution. 0 means no update during execution.

Memory Profiling

Instrumentation control

You can specify the type of memory errors and warnings that you want to detect. See [Memory profiling errors on page 958](#) and [Memory profiling warnings on page 960](#) for more information about these settings.

Detect File in Use (FIU)

When the application exits, this option reports any files left open.

Detect Memory in use (MIU)

When the application exits, this option reports allocated memory that is still referenced.

Free Invalid Memory (FIM)

This option activates the detection of invalid free memory instructions.

Detect Signal (SIG)

This option indicates the signal number received by the application forcing it to exit.

Detect Freeing Freed Memory (FFM) and Detect Free Memory Write (FMWL)

Select **Yes** to activate detection of these errors.

Free queue length (blocks)

Specifies the number of memory blocks that are kept free.

Free queue size (bytes)

Specifies the total buffer size for free queue blocks.

Largest free queue block size (bytes)

Specifies the size of the largest block to be kept in the free queue.

Detect Array Bounds Write (ABWL)

Select **Yes** to activate detection of ABWL errors.

Red zone length (bytes)

Specifies the number of bytes added by Memory Profiling around the memory range for bounds detection.

Number of functions in call stack

Specifies the maximum number of functions reported from the end of the CPU call stack. The default value is 6.

Only show memory leaks with call stack

Select this option to only record memory leaks that are associated with a call stack. Memory allocations that occurred before the application started do not have a call stack and are not included in the Memory Profiling report.

Line number link

Select **Statement** to link the line number in the report to the corresponding allocation or free statement in the function. Select **Function** to link only to the function entry and to improve performance.

Only show new memory leaks in each dump

In multi-dump report, Memory leaks (MLK) and potential leaks (MPK) are only reported once.

Advanced options**Trace File Name (.tpf)**

This setting allows you to specify a filename for the generated `.tpf` trace file.

Exclude block tracking before init

Disables memory profiling for any memory blocks allocated before the first execution of instrumented code. Use this option to prevent crashes when the system uses memory allocations that cannot be tracked.

Excluded global variables

Specifies a list of global variables that are not to be inspected for memory leaks. This option can be useful to save time and instrumentation overhead on trusted code. Use the **Add** and **Remove** buttons to add and remove global variables.

Exclude variables from directories

Specifies a list of directories from which any variables found in files are not to be inspected for memory leaks.

Break on error

Use this option to break the execution when an error is encountered. The break point must be set to `priv_check_failed` in debug mode.

ABWL and FMWL check frequency

Use this to check for ABWL and FMWL errors:

- Each time the memory is dumped (by default).
- Each time a manual check macro is encountered in the code.
- Each function return.

These checks can be performed either on all memory blocks or only a selection of memory blocks. See [Checking for ABWL and FMWL errors on page 70](#) for more information.

Preserve block content

Set this setting to **Yes** to preserve the content of memory blocks freed by the application. Use this setting to avoid application crashes when memory profiling is engaged. When this setting is enable, reads to freed blocks of memory are no longer detected.

Application Profiling

Stack size

You can configure the parameters to calculate the worst stack size.

Trace file name (.tzf)

This allows you to specify a path and filename for the .tzf dynamic stack trace file.

Measure Max Stack Used

This allows you to enable the Worst Stack Size feature. The default value is yes.

Report template

You can set your own report template. The default template id **ccreport.template**.

Debug Reports

You can specify the type errors and warnings that you want to detect.

Display path using biggest stack

Displays the called stack when the biggest stack size is detected during the execution. The selected number is the displayed called stack in the report.

Detect File In Use (FIU)

When the application exits, this option reports any open files.

Detect Signal (SIG)

This option indicates the signal number received by the application that caused to exit.

Performance Profiling

Trace file name (.tqf)

This box allows you to specify a filename for the generated .tqf trace file for performance profiling.

Compute F max and F+D max time

Indicate whether you want the maximum execution time for each function and descendants to be calculated, or not, or if it must be calculated with the Worst Case Execution Time.

Coupling

Control Coupling

You can specify parameter dedicated to the Control Coupling coverage.

Trace File name (.tgf)

Set the name of the trace file dedicated to the Control Coupling. It is the default name of the test with the extension .tgf.

Exclude libraries

Include or exclude the Control Couplings ending with a function call that is not part of the application (this option set the option -noccect of the report generator if it is set to yes).

Report template

You can change the template of the report generator. By default the template is ccreport.template.

Runtime Tracing**Instrumentation control****Runtime Tracing file name (.tdf)**

This allows you to force a filename and path for the dynamic .tdf file. By default, the .tdf carries the name of the application node.

Show data classes

When this option is disabled, structures or classes that do not contain methods are excluded from instrumentation. Disable this option to reduce instrumentation overhead.

Trace control**Split trace file**

When you use several runtime analysis tools together, the executable produces a multiplexed trace file, containing the output data for each tool. Use this option to split the generated atlout.spt output trace file into multiple files.

Maximum size (Kbytes)

This specifies the maximum size for a split .tdf file. When this size is reached, a new split .tdf file is created.

File name prefix:

By default, split files are named as att_<number>.tdf, where <number> is a 4-digit sequence number. This setting allows you to replace the att_ prefix with the prefix of your choice.

Automatic loop detection

Loop detection simplifies UML sequence diagrams by summarizing repeating traces into a loop symbol. Loops are an extension to the UML sequence diagram standard and are not supported by UML.

Additional options

This setting allows you to add command line options. Normally, this line should be left blank.

Display maximum call level

When selected, the target deployment port records the highest level attained by the call stack during the trace. This information is displayed at the end of the UML sequence diagram in the runtime tracing viewer as **Maximum calling level reached**.

Runtime options**Disable on-the-fly mode**

When selected, this setting stops on-the-fly updating of the dynamic `.tdf` file. This option is primarily for target deployment ports that use *printf* output.

Runtime tracing buffer and Partial Runtime Tracing flush

See [Advanced runtime tracing on page 84](#) for more information about these settings.

Maximum buffer size (events)

The maximum number of events recorded in the buffer before it is flushed.

User signal action

Specify an action to be performed when a user signal is detected:

- **No action:** nothing.
- **Flush call stack:** the call stack is flushed to the trace file.
- **Runtime tracing on/off:** toggles the runtime tracing feature on or off.

Record and display time stamp

This setting adds timestamp information to each element in the UML sequence diagram generated by runtime tracing.

Record and display heap size

This setting enables the heap size bar in the UML sequence diagram generated by runtime tracing.

Record and display thread info

This setting enables the Thread Bar in the UML sequence diagram generated by runtime tracing.

Static Metrics**One level metrics**

By default, `.met` static metric files are produced for source files as well as all dependency files that are found when parsing the source code. Set to **Yes** to restrict the calculation of static metrics only to the source files displayed in the navigator.

Analyzed directories

This setting allows you to restrict the generation of `.met` metric files only to files which are located in the specified directories.

Generate metrics in source directories

By default, all `.met` files are generated in the project directory, and use the same name as the source file. Select **Yes** on this setting to compute metrics for source files that have the same name but are located in different directories. In this case, each `.met` is generated in the source directory of each file.

Additional options

Use this setting to specify extra command line options. In most cases, this should be empty.

Code Review**Rule configuration**

This setting specifies the file containing the rules for the code review tool. Click **Browse ...** to select another rule configuration file. Click the **Edit** button to edit the rule configuration or to create a new rule configuration. See [Configuring code review rules on page 91](#) for more information.

Additional included system directories

This setting specifies system include directories that are to be ignored during the code review.

Review included system files

Select **Yes** to extend code review to system files that are *#included* in the source files.

Naming script file

This setting allows you to specify a perl script that can check your own naming rules.

Include files

Specify a list of files to preinclude. This is similar to the `-include=<files>` option in gcc.

Display errors/warnings

Specify the maximum number of errors and warnings that you want to display in the report. By default, All errors and warnings are displayed.

Related reference

[TDP configuration settings on page 944](#)

Related information

[Target deployment port overview on page 12](#)

Data pool editor reference

The data pool editor enables you to link a data pool to a CSV file located in the workspace or in the file system.

Select

Click this button to select the CSV file that is referred to by the current data pool.

Import

Character set

Specify the character set used to generate the CSV file.

Language

Specify the locale used to generate the CSV file. This defines the characters used to note decimals or thousands.

First row as column names

Select this option to use the first row as titles for the columns. Test data starts on the second row.

First row

Specify the first row to use for test data.

Activate line range

Select this option to limit the number of rows used for test data. When selected, use **Range row count** to specify the number of rows to use.

Text delimiter

Specify whether to use a quote or a double-quote for text.

Separator options

Specify one or several characters to use as a column separator. If you select **Other**, type a character to use as a separator.

UML sequence diagram reference

The runtime tracing viewer produces Unified Modeling Language (UML) sequence diagrams of the execution of your source code.

A sequence diagram is a UML diagram that provides a view of the chronological sequence of messages between instances (objects or classifier roles) that work together in an interaction or interaction instance. A sequence diagram consists of a group of instances (represented by lifelines) and the messages that they exchange during the interaction. You line up instances participating in the interaction in any order from left to right, and then you position the messages that they exchange in sequential order from top to bottom. Activations sometimes appear on the lifelines.

A sequence diagram belongs to an interaction in a collaboration or an interaction instance in a collaboration instance.

Memory profiling errors

Error messages indicate invalid program behavior. These are serious issues that you should address before you check in code.

Freeing Freed Memory (FFM)

An FFM message indicates that the program is trying to free memory that has previously been freed.

This message can occur when one function frees the memory, but a data structure retains a pointer to that memory and later a different function tries to free the same memory. This message can also occur if the heap is corrupted.

Memory profiling maintains a free queue, whose role is to actually delay memory free calls in order to compare with upcoming free calls. The length of the delay depends on the Free queue length and Free queue threshold, which are specified in the memory profiling configuration settings. A large deferred free queue length and threshold increases the chances of catching FFM errors long after the block has been freed. A smaller deferred free queue length and threshold limits the amount of memory on the deferred free queue, taking up less memory at run time but providing a lower level of error detection.

Freeing Unallocated Memory (FUM)

An FUM message indicates that the program is trying to free unallocated memory. This message can occur when the memory is not yours to free. In addition, trying to free the following types of memory causes a FUM error:

- Memory on the stack
- Program code and data sections

Freeing Invalid Memory (FIM)

An FIM message indicates that the program is trying to free allocated memory with the wrong instruction.

This message can occur when the memory free instruction mismatches the memory allocation instruction. For example, a FIM occurs when memory is freed with a free instruction when it was allocated with a new instruction.

Late Detect Array Bounds Write (ABWL)

An ABWL message indicates that the program wrote a value before the beginning or after the end of an allocated block of memory. HCL OneTest™ Embedded checks for ABWL errors whenever *free()* or *dump()* routines are called, or whenever the free queue is actually flushed. This message can occur in the following situations:

- An array is too small. For example, you fail to account for the terminating NULL in a string.
- You forgot to multiply by *sizeof(type)* when you allocate an array of objects.
- Code uses an array index that is too large or is negative.
- Fail to NULL terminate a string.
- Code is off by one when copying elements up or down an array.

Memory profiling actually allocates a larger block by adding a *red zone* at the beginning and end of each allocated block of memory in the program. These red zones are monitored to detect ABWL errors. Increasing the size of the red zone helps catch bound errors before or beyond the block, at the expense of increased memory usage. You can change the red zone size in the memory profiling configuration settings. The ABWL error does not apply to local arrays allocated on the stack.



Note: Unlike UNICOM PurifyPlus™, the ABWL error in the HCL OneTest™ Embedded tool only applies to heap memory zones and not to global or local tables.

Late Detect Free Memory Write (FMWL)

An FMWL message indicates that the program wrote to memory that was freed. This message can occur when in the following situations:

- There is a dangling pointer to a block of memory that has already been freed (caused by retaining the pointer too long or freeing the memory too soon).
- Index is far off the end of a valid block.
- A completely random pointer happens to fall within a freed block of memory.

Memory Profiling maintains a free queue, whose role is to actually delay memory free calls in order to compare with upcoming free calls. The length of the delay depends on the Free queue length and Free queue threshold Memory Profiling Settings. A large deferred free queue length and threshold increases the chances of catching FMWL errors. A smaller deferred free queue length and threshold limits the amount of memory on the deferred free queue, taking up less memory at run time but providing a lower level of error detection.

Memory Allocation Failure (MAF)

An MAF message indicates that a memory allocation call failed. This message typically indicates that the program ran out of paging file space for a heap to grow. This message can also occur when a non-spreadable heap is saturated. After displaying the MAF message, a memory allocation call returns NULL in the normal manner. Ideally, programs should handle allocation failures.

Core Dump (COR)

A COR message indicates that the program generated a UNIX™ core dump. This message can only occur when the program is running on a UNIX™ target platform.

Related reference

[Memory profiling warnings on page 960](#)

[Build configuration settings on page 947](#)

Related information

[Memory profiling overview on page 69](#)

Memory profiling warnings

Warning messages indicate a situation in which the program might not fail immediately, but might later fail sporadically, often without any apparent reason and with unexpected results. Warning messages often pinpoint serious issues you should investigate before you check in code.

Memory in Use (MIU)

An MIU message indicates heap allocations to which the program has a pointer.



Note: On exit, small amounts of memory in use in programs that run for a short time are not significant. However, you should fix large amounts of memory in use in long running programs to avoid out-of-memory problems.

Memory profiling generates a list of memory blocks in use when you activate the **MIU Memory In Use** option in the memory profiling configuration settings.

Late Detect Array Bounds Write (ABWL)

An MLK warning describes leaked heap memory. There are no pointers to this block or to anywhere within this block.

Memory Profiling generates a list of leaked memory blocks when you activate the MLK Memory Leak option in the Memory Profiling Settings.

This message can occur when you allocate memory locally in some function and exit the function without first freeing the memory. This message can also occur when the last pointer referencing a block of memory is cleared, changed, or goes out of scope. If the section of the program where the memory is allocated and leaked is executed repeatedly, you might eventually run out of swap space, causing slow downs and crashes. This is a serious problem for long-running, interactive programs. To track memory leaks, examine the allocation location call stack where the memory was allocated and determine where it should have been freed.

You can ignore memory leaks that do not have a call stack, for memory allocations that occur before the application starts by changing the following configuration setting: **Runtime Analysis > Memory Profiling > Instrumentation control > Only show memory leaks with call stack**.

Memory Potential Leak (MPK)

An MPK warning describes heap memory that might leak. There are no pointers to the start of the block, but there appear to be pointers pointing towards somewhere within the block. In order to free this memory, the program must subtract an offset from the pointer to the interior of the block. In general, you should consider a potential leak to be an actual leak until you can prove that it is not by identifying the code that performs this subtraction.

Memory in use can appear as an MPK if the pointer returned by some allocation function is offset. This message can also occur when you reference a substring within a large string. In rare cases, leaked memory might cause an MPK warning if some non-pointer integer within the program space, when interpreted as a pointer, points to an otherwise leaked block of memory. Inspection of the code should easily differentiate between different causes of MPK messages.

Memory profiling generates a list of potentially leaked memory blocks when you activate the MPK Memory Potential Leak option in the memory profiling configuration settings.

File in Use (FIU)

An FIU message indicates a file that was opened, but never closed. An FIU message can indicate that the program has a resource leak.

Memory profiling generates a list of files in use when you activate the FIU Files In Use option in the memory profiling configuration settings.

Signal Handled (SIG)

A SIG message indicates that a system signal has been received.

Memory profiling generates a list of received signals when you activate the SIG Signal Handled option in the memory profiling configuration settings.

Related reference

[Memory profiling errors on page 958](#)

[Build configuration settings on page 947](#)

Related information

[Memory profiling overview on page 69](#)

Command Line Interface

HCL OneTest™ Embedded was designed ground-up to provide seamless integration with your development process. To achieve this versatility, the entire set of features are available as command line tools.

In most cases when a CLI is necessary, the easiest method is to develop, set up and configure your project in the graphical user interface and to use the **studio** command line to launch the GUI and run the corresponding project node.

When not using the GUI to execute a node, you must create source files that can execute HCL OneTest™ Embedded tests or acquire runtime analysis data without conflicting with the your native compiler and linker. In both cases – that is, regardless of whether you are attempting to execute a Test or Application node – the native compiler and linker do the true work.

For Test nodes, the following commands convert HCL OneTest™ Embedded test scripts into source files supported by your native compiler and linker:

- **attolpreproC** for the C language
- **atoprepro** for the C++ language
- **attolpreproADA** for the Ada language

For Runtime Analysis, the primary choice is whether or not you wish to perform source code insertion (SCI) as an independent activity or as part of the compilation and linkage process. Of course, if no runtime analysis is required, source code insertion is unnecessary and should not be performed. To simply perform source code insertion, use the binaries:

- **attolcc1** for the C language
- **attolccp** or **attolcc4** for the C++ language
- **attolada** for the Ada language

However, if the user would like compilation and linkage to immediately follow source code insertion, use the binaries:

- **attolcc** for the C and C++ language
- inclusion of the **javic.jar** library, and calls to **javic.jar** classes, as part of an ant-facilitated build process

The following sections provide details about the most common use cases.

To learn about	See
Launching the GUI with the studio command and running a node from the command line	Running a Node from the Command Line on page 963
Preparing your environment for command line usage	Setting Environment Variables on page 702
Code coverage, runtime tracing, memory and performance profiling from the command line	Command Line Runtime Analysis for C and C++ on page 965
Testing C, C++ and Ada source code components from the command line	Command Line Component Testing for C, Ada and C++ on page 966
Testing message-based systems from the command line	Command Line System Testing for C on page 967
Using the Command Line Interface through a set of examples	Command Line Examples on page 968

Related Topics

[Using the Graphical User Interface on page 657](#) | [Automated Testing](#) | [Runtime Analysis on page 305](#)

Running a Node from the Command Line

Command Line Interface

Although the product contains a full series of command line tools, it is usually much easier to create and configure your runtime analysis specifications inside the graphical user interface (GUI). The CLI would then be used to simply launch the GUI with a project or project node as a parameter.

By doing this, you combine the ease and simplicity of the GUI with the ability to execute project nodes from a CLI.

Note This functionality can be used to execute any node in a project, including group nodes, application nodes, test nodes or the entire project.

An HTML output option produces a set of HTML reports in a specified directory. The output is the same as exporting to HTML from the original reports. With this option, it is not necessary to open the GUI to view the reports.

To run a specific node from a command line:

1. Set up and configure your project in the GUI.
2. Save your project and close the GUI.
3. Type the following command:

```
studio -r <node>.{[.<node>]} <project_file> [-html <directory>]
```

where:

1.
 - <node> is the node to be executed.
 - <project> is the **.rtp** project file.
 - <directory> is the output directory for the optional HTML output.

The <node> hierarchy must be specified from the highest node in the project (excluding the actual project node) to the target node to be executed, with periods ('.') separating each item:

```
<node>{[. <node>]}
```

Example

The following command opens the **project.rtp** project in the GUI, and runs the *app2* application node, located in *group1* of the sub-project *subproject1*:

```
studio -r subproject1.group1.app2 project.rtp -html project/output
```

Related Topics

[Graphical User Interface - studio on page 1053](#)

Command Line Runtime Analysis for C and C++

Command Line Interface

The runtime analysis tools for C and C++ include:

- Memory Profiling
- Performance Profiling
- Code Coverage
- Runtime Tracing

These features use Source Code Insertion (SCI) technology. When analyzing C and C++ code, the easiest way to implement SCI features from the command line is to use the C and C++ Instrumentation Launcher.

The Instrumentation Launcher is designed to fit directly into your compilation sequence; simply add the **attolcc** command in front of your usual compilation or link command line.

Note The **attolcc** binary is located in the **/cmd** directory of the applicable Target Deployment Port.

To perform runtime analysis on C or C++ source code:

1. First, set up the necessary environment variables. See [Setting Environment Variables on page 702](#).
1. Edit your usual makefile with the following command line:

```
attolcc [-options] [--settings] -- <compiler command line>
```

Where *<compiler command line>* is the command that you usually invoke to build your application.

For example:

```
attolcc -- cc -I../include -o appli appli.c bibli.c -lm
```

```
attolcc -TRACE -- cc -I../include -o appli appli.c bibli.c -lm
```

1. After execution of your application, in order to process SCI dump information (i.e. the runtime analysis results), you need to separate the single output file into separate, feature-specific, result files. See [Splitting the SCI Dump File on page 978](#).
1. Finally, launch the **Graphical User Interface** to view the test reports. See [Opening Reports from the Command Line on page 978](#).

Related Topics

[Runtime Analysis on page 305](#) | [Source code instrumentation overview on page 11](#) | [C and C++ Instrumentation Launcher - attolcc on page 1065](#)

Command Line Component Testing for C, Ada and C++

Command Line Interface

Use Component Testing for C and Ada and Component Testing for C++ to test individual components of your C, C++ and Ada source code.

To perform component testing on C, C++ or Ada source code:

1. First, set up the necessary environment variables. See [Setting Environment Variables on page 702](#).
2. Generate a set of test script templates based on your source files by using the Source Code Parser. See corresponding **Source Code Parser** command line section in the Reference section.
3. Use the generated **.ptu**, **.otc** or **.otd** templates to write a test script. See the Reference section for test script syntax.
4. If you are using an **.otc** Contract Check script, set up an **options.h** header file. See [Preparing an Options Header File on page 971](#).
5. Compile the generated test harness source file. See [Compiling the Test Harness on page 974](#)
6. If you are using any of the runtime analysis tools, instrument and compile the source code. See [Instrumenting and Compiling the Source Code on page 972](#).

If not, simply compile your source code with your usual compiler.

1. Set up the TDP configuration file, called **product.h**. See [Preparing a Products Header File on page 972](#).
2. Compile the TDP Library. See [Compiling the TDP Library on page 973](#).
3. Link the compiled files together to create an executable test binary. See [Linking the Application on page 975](#).
4. Execute the test binary. See [Running the Test Harness or Application on page 976](#).
5. After execution, to obtain the final test results, as well as any SCI dump information, you need to separate the output file into separate result files. See [Splitting the SCI Dump File on page 978](#).
6. Run the **Report Generator** to produce a test report. See the Reference section.
7. Finally, launch the **Graphical User Interface** to view the test reports. See [Opening Reports from the Command Line on page 978](#).

Related Topics

[Command Line Interface on page 962](#)

Command Line System Testing for C

Command Line Interface

Use System Testing to test message-based systems and subsystems written in C.

To perform message based testing on a system:

1. First, set up the necessary environment variables. See [Setting Environment Variables on page 702](#).
2. Write a System Testing **.pts** test script. See [System Testing language reference on page 893](#).
3. Write a System Testing **.spv** supervisor script. See [System Testing language reference on page 893](#).

Note Manually created supervisor scripts may be overwritten by the HCL OneTest™ Embedded graphical user interface.

1. Compile the generated test harness source file. See [Compiling the Test Harness on page 974](#).
2. If you are using any of the runtime analysis tools, instrument and compile the source code. See [Instrumenting and Compiling the Source Code on page 972](#).

If not, simply compile your source code with your usual compiler.

1. Set up the TDP configuration file, called **product.h**. See [Preparing a Products Header File on page 972](#).
2. Compile the TDP Library. See [Compiling the TDP Library on page 973](#).
3. Link the compiled files together to create an executable test binary. See [Linking the Application on page 975](#).
4. Ensure that the System Testing agents are running on all remote target hosts. See [Installing System Testing Agents on page 587](#).
5. Run the supervisor script on the supervisor machine (the machine running HCL OneTest™ Embedded) with the following command:

```
atsspv <supervisor.spv>
```

where supervisor is the name of the **.spv** supervisor script.

1. Run the **System Testing Report Generator** to produce a test report. See [System Testing Report Generator - atsmmerge on page 1109](#).
2. Finally, launch the **Graphical User Interface** to view the test reports. See [Opening Reports from the Command Line on page 978](#).

Related Topics

[Command Line Interface on page 962](#) | [About System Testing for C on page 586](#) | [System Testing Supervisor - atsspv on page 1106](#) | [System Testing Report Generator - atsmmerge on page 1109](#)

Command line examples

Command Line Interface

This section describes an example of using HCL OneTest™ Embedded Runtime Analysis tools through the Command Line Interface:

Note This example is for UNIX platforms only.

This example demonstrates using Runtime Analysis tools through the **attolcc** Instrumentation Launcher. The example application is the Apache Web Server, which is widely available for most platforms.

Additionally, the Apache Web Server is a multi-process, multi-tasking application written in C where particular attention must be paid to tracking memory leaks.

To prepare for the example:

1. Download the **apache_1.3.37.tar.gz** archive of the Apache web server source code from <http://www.apache.org/dist/httpd/>
2. Copy the archive file to the directory where you will perform the tests (for example, **/projects/Apache_Test**) and untar the archive:

```
cp /projects/download/apache_1.3.37.tar.gz .
tar zxvf apache_1.3.37.tar.gz
cd apache_1.3.37
```

3. To set up the HCL OneTest™ Embedded environment, enter the following command:

```
./<install_dir>/TestRealTime/testrtinit.sh
```

where *<install_dir>* is the installation directory of the product.

See the **Installation overview section** for information about setting up and starting the product.

Compiling the application with the Runtime Analysis features:

1. To configure the Apache release, enter the following command:

```
./configure --prefix=`pwd`
```

2. To compile the Apache server with instrumentation, enter the following commands:

```
REP=`pwd`
make -C src/main gen_test_char
make CC="attolcc -mempro -perfpro -trace -proc=ret -block=l -keep --atl_multi_process=1
--atl_traces_file=$REP/atlout.spt -- gcc"
```

The application is compiled with the following options:

- Memory Profiling instrumentation enabled
- Performance Profiling instrumentation enabled

- Runtime Tracing instrumentation enabled
- Instrumentation of procedure inputs, outputs, and terminal instructions
- Instrumentation of simple, implicit and logical blocks (loops)
- Keep instrumented files
- Multi-process support

To install the Apache server, type the following command:

```
make install
```

A message indicates that you have successfully built and installed the Apache 1.3 HTTP server.

To run the application and view runtime analysis results:

1. Optionally, edit the configuration file **apache_1.3.37/conf/httpd.conf**.
2. Enter the following command to start the Apache server:

```
/projects/Apache_Test/apache_1.3.37/bin/apachectl start
```

3. Start a web browser on port 8080 (see the **httpd.conf** file), type the following command:

```
firefox <IP Address>:8080
```

where <IP Address> is the IP address of the machine hosting the Apache server.

To stop the Apache server, type the following command:

```
/projects/Apache_Test/apache_1.3.37/bin/apachectl stop
```

To split the results, type the following command:

```
atlsplit *.spt
```

To start the HCL OneTest™ Embedded GUI to view the results, type the following command:

```
studio `find . -name "*.fdc"` `find . -name "*.tsf"` *.tio *.tpf *.tqf *.tdf
```

Related Topics

[Command Line Runtime Analysis for C and C++ on page 965](#) | [Command Line Interface on page 962](#)

Setting Environment Variables for CLI tools

The command line interface (CLI) tools require several environment variables to be set.

These variables determine, for example, the Target Deployment Port (TDP) that you are going to use. The available TDPs are located in the product installation directory, under **targets**. Each TDP is contained in its own sub-directory.

Prior to running any of the CLI tools, the following environment variables must be set:

- **TESTRTDIR** indicates the installation directory of the product
- **ATLTGT** and **ATUTGT** specify the location of the current TDP: **\$TESTRTDIR/targets/ <tdp>**, where *<tdp>* is the name of the TDP.
- **PATH** must include an entry to **\$TESTRTDIR/bin/ <platform> / <os>**, where *<platform>* is the hardware platform and *<os>* is the current operating system.

You must also add the product installation **bin** directory to your **PATH**.

Note Some command-line tools may require additional environment variables. See the pages dedicated to each command in **Reference** section of the help.

Most of these environment variables are set during installation of the product. Under Linux, use the `testrtinit.sh` script to set these variables. See the [Reference on page 981](#) section for more information about these scripts.

Automated Testing

If you are using Component Testing or System Testing features, the following additional environment variables must be set:

- **ATUDIR** for Component Testing, points to **\$TESTRTDIR/lib**
- **ATS_DIR**, for System Testing, points to **\$TESTRTDIR/bin/ <platform> / <os>**, where *<platform>* is the hardware platform and *<os>* is the current operating system.

Library Paths

UNIX platforms require the following additional environment variable:

- On Solaris and Linux platforms: **LD_LIBRARY_PATH** points to **\$TESTRTDIR/lib/ <platform> / <os>**
- On HP-UX platforms: **SHLIB_PATH** points to **\$TESTRTDIR/lib/ <platform> / <os>**
- **On AIX platforms: LIB_PATH** points to **\$TESTRTDIR/lib/ <platform> / <os>**

where *<platform>* is the hardware platform and *<os>* is the current operating system.

Example

The following example shows how to set these variables for HCL OneTest™ Embedded with a **sh** shell on a Suse Linux system. The selected Target Deployment Port is **clinuxgnu** .

```
TESTRTDIR=/opt/HCL OneTest™ Embedded/TestRealTime.v2002R2
```

```
ATCDIR=$TESTRTDIR/bin/intel/linux_suse
```

```
ATUDIR=$TESTRTDIR/lib
```

```

ATS_DIR=$TESTRTDIR/bin/intel/linux_suse
ATLTGT=$TESTRTDIR/targets/clinugnu
ATUTGT=$TESTRTDIR/targets/clinugnu
LD_LIBRARY_PATH=$TESTRTDIR/lib/intel/linux_suse
PATH=$TESTRTDIR/bin/intel/linux_suse:$PATH

export TESTRTDIR

export ATCDIR

export ATUDIR

export ATS_DIR

export ATLTGT

export ATUTGT

export LD_LIBRARY_PATH

export PATH

```

Creating an Options Header File for the Testing component for C++ or for System Testing

Command Line Interface

This step is necessary if you are using:

- System Testing for C
- or
- Testing component for C++

Use **attolcc** instrumentor launcher to create an **options.h** file and compile the generated source files.

To create the options.h file:

From the **attolcc** launcher, type the following command:

```
attolcc -otc=<mycontract.otc> <options> <settings> -- $CC $CFLAGS $<
```

<options> and <settings> are used to generate the content of the options.h file.

Related Topics

[Preparing a Products Header File on page 972](#) | [Command Line Component Testing for C, Ada and C++ on page 966](#) | [Command Line System Testing for C on page 967](#)

Preparing a Products Header File

Command Line Interface

Before you can compile the TDP library source files, you must set up a header file named **products.h** for C and C++. This file is generated by the Instrumentation Launcher (**attolcc**) and contains the options that describe how the TDP library files are to be compiled.

To set up a products header file, type the following command line:

```
attolcc -env_ATL_NO_TDP_COMPILE=1 -force_tdp_cc <instrumOptions>
```

where *<instrumOptions>* are the options used for the instrumentation.

Note Type **attolcc --help** to list a comprehensive list of options, including those of the C and C++ Instrumentor (attolccp or attolcc4 and attolcc1), for use with the instrumentation launcher.

Related Topics

[Preparing an Options Header File on page 971](#) | [C and C++ Instrumentation Launcher - attolcc or attolcc4 on page 1065](#) | [C and C++ Instrumentor - attolcc1 and attolccp on page 1075](#)

Instrumenting and Compiling the Source Code

Command Line Interface

The runtime analysis tools (Memory Profiling, Performance Profiling, Code Coverage and Runtime Tracing) as well as Component Testing for C++ Contract Check all use SCI instrumentation technology to insert analysis and SCI dump routines into your source code.

Requirements

Before compiling an SCI-instrumented source file, you must make sure that:

- A working C, Ada or C++ compiler is installed on your system
- If you use Component Testing for C++, you have [prepared a valid options.h file on page 971](#)
- If you compile on a target different from the host where the generated file has been produced, the instrumented file must have been produced using option **-NOPATH**, and the sub-directory lib of the selected Target Deployment Port directory must be copied onto the target.

There are two alternatives to instrument and compile your source code:

- Using the Instrumentation Launcher in your standard makefile
- Using the Instrumentor and Compiler separately.

Instrumentation Launcher

The Instrumentation Launcher replaces your actual compiler command in your makefiles. This launcher transparently takes care of source code preprocessing, instrumentation and compiling.

See the reference section:

- [C and C++ Instrumentation Launcher - attolcc on page 1065](#)

Instrumentation and Compilation

Alternatively, you can use the actual Instrumentor command line tools to instrument the source files.

See the reference section:

- [C and C++ Instrumentor - attolcc1, attolccp or attolcc4 on page 1075](#)
- [Ada Instrumentor - attolada on page 1088](#)

If you are compiling on a different target, you must copy the TDP /lib directory over to that target.

Add to the include search path the **/lib** sub-directory that you have copied onto the target. In C and C++, use the **-I** compiler option.

After this, simply compile the instrumented source file with your compiler.

Compiling the TDP Library

Command Line Interface

Before you can link your test harness or your instrumented application, you must compile the Target Deployment Port library. This section describes how to do this.

Requirements

To compile the Target Deployment Port library, make sure that:

- A working C or C++ Test Script Compiler is installed on your system
- You have [prepared a valid Products file on page 972](#)

Compilation

Depending on the language of your source file:

- For C: compile the **TP.c** file
- For C++: compile the **TP.cpp** file
- For Ada: compile the contents of the **/lib** directory

Do not forget to add to the *include* search path the directory where the **products.h** file is located (usually with option **-I** or **/I**, depending on the compiler).

Configuration Settings

A wide variety of compilation flags can be used by the command line tools, allowing you to select sub-components of the application under test. These flags are equivalent to the Test Configuration Settings dialog box of the graphical user interface and are covered in the [Reference on page 981](#) section.

Default settings are contained in the following Perl script. You can use this file to define your own customized configuration settings.

```
<InstallDir>/lib/scripts/BatchCCDefaults.pl
```

To run this script, type the following command:

```
$TESTRTDIR/bin/<cpu>/<os>/perl -I$TESTRTDIR/lib/perl
```

```
$TESTRTDIR/lib/scripts/TDPBatchCC.pl <my_env.pl>
```

where *<cpu>* is the architecture platform of the machine, *<os>* is the operating system, and *<my_env.pl>* is your customized copy of the **BatchCCDefaults.pl** file

The **TESTRTDIR** and **ATLTGT** environment variables must have been previously set.

Related Topics

[Preparing a Products Header File on page 972](#) | [Setting Environment Variables on page 702](#)

Compiling the Test Harness

Command Line Interface

Each of the Test Script Compilers converts a test script into test harness source code. This section explains how to compile the test harness source file.

Requirements

In order to compile a generated source file, you must check that:

- A working C, C++ or Ada compiler is installed on your system
- If you are using System Testing, you have [prepared a valid `options.h` file on page 971](#)
- If you are compiling on a target different from the host where the file was generated, the generated file must have been produced using the **-NOPATH** option (available with every Test Script Compiler), and the **/lib** sub-directory of the Target Deployment Port directory must be copied onto the target.

Compilation

If you are using Component Testing, System Testing or Component Testing for C++ alone without any of the runtime analysis tools, then simply compile the generated test harness source file with your C or C++ compiler.

If you are compiling on a remote target, do not forget to add to the *include* search path the **/lib** sub-directory that you have copied onto the target.

If you are using SCI instrumentation features (Memory Profiling, Performance Profiling, Code Coverage, Runtime Tracing and C++ **.otc** contract check), use the specific command line options for the Instrumentor in the [Reference on page 981](#) section:

- [C and C++ Instrumentor - `attolcc1` and `attolccp` on page 1075](#)
- [Ada Instrumentor - `attolada` on page 1088](#)
- [Java Instrumentor - `javi` on page 1094](#)

Related Topics

[Instrumenting and Compiling the Source Code on page 972](#) | [Command Line Interface on page 962](#)

Linking the Application

Command Line Interface

Once you have compiled all your source files, you need to link them to build an executable. This section describes linkage specifics when using a test or runtime analysis feature.

Requirements

In order to compile an instrumented source file, you must check that:

- A working C, C++ or Ada linker is installed on your system
- You have compiled every source file, including any instrumented source files, of your application under test
- If using a Component Testing for C, Ada or C++, or System Testing, you have compiled the test harness.
- You have compiled the Target Deployment Port library.

Linking

If you are using only runtime analysis feature (Runtime Tracing, Code Coverage, Memory Profiling, Performance Profiling, C++ Contract Check), you just have to add the Target Deployment Port library object to the object files linked together. If you are using a test feature, you also have to add the tester object to the linked files.

You just have to add the Target Deployment Port library object to the object files linked together.

Related Topics

[Compiling the TDP Library on page 973](#) | [Instrumenting and Compiling the Source Code on page 972](#)

Running the Test Harness or Application

Command Line Interface

Once you have produced a binary tester or instrumented application, you want to run it in order to obtain test or SCI analysis information.

By default, the generated SCI dump file is named **atlout.spt**.

To run the test application binary:

1. Check that the current directory is correct, relatively to the previously specified trace file, if the trace files was specified with a relative path.
2. Run the binary. When the application terminates, the trace file should be available.

Related Topics

[Splitting the SCI Dump File on page 978](#)

Troubleshooting Command Line Usage

Command Line Interface

The following information might help if you encounter any problems when using the command line tools.

Failure	Response
Compilation fails	Ensure that the selected Target Deployment Port matches your compiler; there may be several Target Deployment Ports for one OS, each of which targets a different compiler. If you are unsure, you can check the full name of a Target Deployment Port by opening any of the .ini files located in the Target Deployment Port directory.
Compiler reports that	Ensure that you have correctly prepared the options.h file on page 971 , and that this file is located in a directory that is searched by your compiler (this is usually specified with -I or /I option on the compiler command line).

options.h is missing	
Compiler reports that TP.h file is missing	<p>If you are compiling on a target different from the host where the generated file has been produced, double-check the above specific requirements to compilation on a different target.</p> <p>If the compiler and C/C++ Test Script Compiler are executed on the same machine, ensure you have not used the -NOPATH option on the Test Script Compiler command line, and that the ATLTGT environment variable was correctly set while the Test Script Compiler was executed.</p>
Compilation fails	Ensure that the selected Target Deployment Port matches your compiler; there may be several Target Deployment Ports for one OS, each of which targets a different compiler. If you are unsure, you can check the full name of a Target Deployment Port by opening any of the .ini files located in the Target Deployment Port directory.
TDP compilation fails	<p>When using the -I linker option, the TDP fails to compile. This is because the following line is added to the instrumented file:</p> <pre>#include "<path to target directory>/TP.h"</pre> <p>where TP.h includes other files using the #include syntax, such as:</p> <pre>#include "atclock.h"</pre> <p>where atclock.h is in the same directory as "TP.h". If you use the -I flag, the compiler no longer searches the same directory as the current file (TP.h) and therefore cannot find atclock.h. If you cannot remove the -I flag, you must add a -I flag for the compiler to find the include files required by the TDP.</p>
Linkage fails because of undefined references	<p>Ensure you have successfully compiled the Target Deployment Port library object, and have included it in your linked files</p> <p>Ensure you have correctly configured the products.h options file on page 972.</p> <p>If you are using a test feature, ensure that you are linking both source under test and additional files. You may also want to add some stubs in your .ptu or .otd test script.</p> <p>Ensure the options set in options.h on page 971 (if required) are coherent with the options set in products.h.</p>
Errors are reported through #error directives	You may have selected a combination of options in products.h which is incompatible. The error messages help you to locate the inconsistencies.

Splitting the trace dump file

Command Line Interface

When you use several features together, the executable produces a multiplexed trace file, containing several outputs targeting different features from HCL OneTest™ Embedded.

You must split the **atlout.spt** SCI dump file.

Splitting an SCI dump file

In most cases, you must split the **atlout.spt** trace file into several files for use with each particular Report Generator or the product GUI.

To do this, you must have a working *perl* interpreter. You can use the *perl* interpreter provided with the product in the **/bin** directory.

To split the trace file:

1. Use the **atlsplit** tool supplied in the **/bin** directory of HCL OneTest™ Embedded:

```
atlsplit atlout.spt
```

After the split, depending on the selected runtime analysis tools, the following file types are generated:

- **.rio test result files:** process with a Report Generator
- **.tio Code Coverage report files:** view with Code Coverage Viewer
- **.tdf dynamic trace files:** view with UML/SD Viewer
- **.tpf Memory Profiling report files:** view with Memory Profiling Viewer
- **.tqf Performance Profiling report files:** view with Performance Profiling Viewer

Related Topics

[TDF Splitter - attsplit on page 1100](#)

Opening Reports from the Command Line

Command Line Interface

Once the test harness or application has been successfully run, you will want to view the test result files in the HCL OneTest™ Embedded. To do this, simply invoke the **studio** binary with the corresponding result files. Some reports require at least two files to be opened simultaneously.

- Code Coverage: **.fdc** and **.tio** files
- Memory Profiling for C and C++: **.tsf** and **.tpf** files
- Performance Profiling: **.tsf** and **.tqf** files
- Runtime Tracing: **.tsf** and **.tdf** files
- Test Reports: **.xrd** files

Alternatively, you can launch the GUI (studio) and use the Browse Reports feature to open the report files. See [Opening a Report on page 683](#).

Report Viewers

The GUI opens the report viewer adapted to the type of report:

- The UML/SD Viewer displays UML sequence diagram reports.
- The Report Viewer displays test reports
- The [Code Coverage Viewer on page 349](#) displays code coverage reports.
- The [Memory Profiling Viewer on page 377](#) and [Performance Profiling Viewer on page 391](#) display Memory Profiling for Cand C++ and Performance Profiling results.

Examples

To open the Runtime Tracing UML sequence diagram:

```
studio MyApp.tsf MyApp.tdf
```

To open a Test Report file

```
studio MyTest.xrd
```

Related Topics

[Opening a Report on page 683](#) | [Splitting the Trace Dump File on page 978](#)

Code Coverage HTML Report Generator

The HTML Report Generator creates Code Coverage reports in an HTML format from the coverage data that are collated during the execution of the application under analysis.

Syntax

On Windows:

```
java -Djava.library.path=<installation folder>/bin/intel/linux_64 -jar <installation
folder>/bin/intel/linux_64/CoverageHTMLReport.jar -files { <file>} -report <report> -template
<template> -version
```

On Linux:

```
java -Djava.library.path=<installation folder>/bin/intel/linux_64 -jar <installation
folder>/bin/intel/linux_64/CoverageHTMLReport.jar -files { <file>} -report <report> -template
<template> -version
```

Where:

- *file* are the input files. It could be .fdc, .tio, .xtp. At least, one .fdc or .xtp is required.
- *report* is the output HTML file.
- *template* is the template file used for the report.

The *-version* option is used for the version of the Coverage HTML Report Generator.

RTistFdcConv command line

RTist is used to create models of the software system based on the Unified Modeling Language (UML) constructs, to generate the implementation code, compile, then run and debug the application. It is used from a command line.

Overview

- RTist is a UML modeling tool for embedded applications. HCL OneTest™ Embedded supports two new coverage levels related to the UML State Diagram. A state diagram is used to represent the condition of the system or part of the system at finite time instances. State diagrams are also referred to as State machines or State-chart Diagrams:
 - State coverage: Verifies that all the states of a diagram has been reached at least at once.
 - Transition coverage: Verifies that all the transitions between states have been executed at least at once.
- HCL OneTest™ Embedded integrates a new tool to translate .fdc files (static files resulting from instrumentation process) into new .fdc files. These files filter the generated code and add state and transition coverage information.
- The coverage result viewer has been updated to display this new coverage level. In this view, the report contains the list of states and transitions with the user code.

Command line to launch RTist

To launch RTist, run the following command:

```
RTistFdcConv <fdcFilesFolders> <convertedFdcFilesFolder> <RtistModelFiles>
```

<fdcFilesFolders> is the folder(s) containing *fdc* file generated by *attolcc*. If there are multiple folders, use a comma between each folder name: For example: *folder1, folder2, ...*

<convertedFdcFilesFolder> is the folder used to generate the *converted fdc*.

<RtistModelFiles> is a .emx file or list of .emx files containing the RTist model. If there are multiple files, use a comma between each file name: For example: model1.emx,model2.emx,...

Studio Reference

This section contains reference material for using classic HCL OneTest Embedded Studio command line tools and test script languages with your existing projects. This reference material does not apply to HCL OneTest Embedded for Eclipse IDE.

To learn about	See
General command line tools, configuration settings and HCL OneTest Embedded GUI components	User interface reference on page 981
Test script languages, component test and system test command line tools	Testing tools reference on page 715
Runtime analysis command line tools, trace probes and pragma macros	Runtime Analysis reference on page 1040

Reference Topics

[Target deployment technology overview on page 12](#)

User interface reference

This section contains advanced reference material for general command line tools, configuration settings and HCL OneTest™ Embedded GUI components.

To learn about	See
Configuration settings	Configuration settings reference on page 982
HCL OneTest™ Embedded GUI components	GUI elements on page 1019
Macros for configuring the Tools menu	GUI macro variables on page 1029
UML sequence diagram elements	UML sequence diagrams on page 397
Files used by HCL OneTest™ Embedded	File types on page 1032
Environment variables	Environment variables on page 1035

Configuration settings reference

Each configuration has its own set of configuration settings which are applied on each node of the test project. You can change these settings in the Configuration Settings dialog box.

The Configuration Settings provides access to the following settings families:

- General
- Build
- Runtime Analysis
- Testing

The actual settings available for each node depend on the type of node and the language of the selected Configuration.

General Settings

To learn about	See
Configuring the compiler and linker options	Build Settings on page 983
General project settings	General Settings on page 987
Adding a user-specified command line to the project	External Command Settings on page 1007
Controlling System Testing Probes	Probe Control Settings on page 1006
Checking compliance to coding guidelines	Code review settings on page 1008

Runtime Analysis

The Runtime Analysis setting family covers Configuration Settings for Memory Profiling, Performance Profiling, Code Coverage and Runtime Tracing.

To learn about	See
Setting up instrumentation and file storage locations	General Runtime Analysis Settings on page 989

Configuring Memory Profiling error and warning detection	Memory Profiling Settings on page 995
Specifying a trace file name for Performance Profiling	Performance Profiling Settings on page 997
Setting coverage levels and instrumentation options for Code Coverage	Code Coverage Settings on page 992
Configuring sequence diagram output	Runtime Tracing Control Settings on page 997

Automated Testing Settings

This setting family covers Configuration Settings for Component Testing and System Testing features.

To learn about	See
Setting up C and Ada test execution	Component Testing for C and Ada Settings on page 999
Setting up C++ test execution	Component Testing for C++ Settings on page 1001
Setting up system test execution	System Testing Settings on page 1003

Related Topics

[Modifying Configurations on page 662](#) | [Selecting Configurations on page 210](#) | [Project Explorer on page 1021](#)

Build Settings

The **Build** settings are part of the Configuration Settings dialog box, which allows you to configure settings for each node.

By default, the settings of each node are inherited from those of the parent node. When you override the settings of a parent node, changes are propagated to all child nodes within the same Configuration. Overridden fields are displayed in bold.

Build options settings

- **Target Deployment Port:** This setting allows you to override the TDP of the entire configuration for a specific node. Use this for example if you are mixing different languages or compilers within a single project. Any child nodes will use the default Configuration Settings from this Target Deployment Port, such as compilation flags.

When you change the TDP within a Configuration, the settings are overwritten using the default settings of the new TDP.

- **Build options:** Build options allow you to specify how the test is built and executed. This is also where you enable the Runtime Analysis tools. See [Selecting Build Options for a Node on page 699](#).
- **Environment variables:** This section allows you to specify any environment variables that can be used by the application under test. Click the "..." button to edit environment variables. String values must be entered with quotes ("").

You can enter GUI macro variables as values for environment variables. These will be interpreted by the GUI and replaced with the actual values for the current node. See [GUI macro variables on page 1029](#).

- **Ignored files (for Eclipse CDT only):** Specifies a list of files that are ignored by the instrumentor. Click the ... button and use the **Add** and **Remove** buttons to select the files to be excluded.
- **Instrumented files (for Eclipse CDT only):** Specifies a list of files that are to be explicitly instrumented. Any other files are ignored. Click the ... button and use the **Add** and **Remove** buttons to select the files to be excluded.
- **Clean result history:** Select **Yes** to clear results before the application starts. This ensures that displayed results are for the last execution only.
- **List tested files versions:** Select the configuration management tool to be used to handle file versioning in the test report.

Compiler settings

- **Assembler flags:** Specify any additional command line options to be sent to the assembler for assembler source files.
- **Preprocessing-only flags:** Specific compilation flags used only for preprocessing. If no preprocessing is performed, these flags are used as compiler flags.
- **Preprocessor flags:** Specific compilation flags to be sent to the compiler.
- **Compiler flags:** Specify any additional command line options to be sent to the compiler.
- **Preprocessor macro definitions:** Specify any macro definition that are to be sent to both the compiler preprocessor (if used) and the Test Script Compilers. Several generation conditions must be separated by a comma ',' with no space. You can use a comma inside a condition, preceded a backslash character. For example:

```
semTake(x\y)=TestRTsemTake(x\y),WIN32,_DEBUG
```

- **Default include directories:** Use this setting to specify include directories that are specific to the current TDP (if you change the TDP, these directories will be lost). Click the ... button to create or modify a list of directories for included files when the include statement is encountered in source code and test scripts. In the directory selection box, use the **Up** and **Down** buttons to indicate the order in which the directories are searched.
- **User include directories:** Use this setting to specify include directories that are independent of the current TDP (if you change the TDP, these directories will be retained). Click the ... button to create or modify a list of directories for included files when the include statement is encountered in source code and test scripts. In the directory selection box, use the **Up** and **Down** buttons to indicate the order in which the directories are searched.
- **User link file (for Ada only):** When using the Ada Instrumentor, you must provide a link file. See [Ada Link Files on page 322](#) for more information.

Linker settings

This area contains parameters to be sent to the linker during the build of the current node.

- **Additional objects or libraries:** A list of object libraries to be linked to the generated executable. Enter the command line option as required by your linker. Please refer to the documentation provided with your development tool for the correct syntax.
- **Library path:** Click the ... button to create or modify a list of directories for library link files. In the directory selection box, use the Up and Down buttons to indicate the order in which the directories are searched.
- **Link flags:** Flags to be sent to the linker.
- **Test driver filename :** The name of the generated test driver binary. By default, HCL OneTest™ Embedded uses the name of the test or application node.

Execution settings

These settings apply to Component Testing and System Testing nodes only.

- **Command line arguments:** Specifies any command line arguments that are to be sent to the application under test upon execution.
- **Main procedure (for Ada only):** Ada requires an entry point in the source code.

Target Deployment Port build settings

This area relates to the parameters of the Target Deployment Port on which is based the Configuration:

- **Output buffer size:** Sets the size of the output buffer. A smaller output buffer can save memory when resources are limited. A larger buffer improves performance.

The default setting for the output buffer is **1024** bytes.

- **Time measurement:** Selects between a real-time **Operating system clock** or a **Process or task clock** for time measurement, if both options are available in the current Target Deployment Port. Otherwise, this setting is ignored.
- **Multi-threaded application:** This box, when selected, protects Target Deployment Port global variables against concurrent access when you are working in a multi-threaded environment such as Windows. This can cause an increase in size of the Target Port as well as an impact on performance, therefore select this option only when necessary.
- **Multi-processed application:** When selected, this option produces a different output file for each process in forked applications.
- **Maximum number of threads:** When the multi-thread option is enabled, this setting sets the maximum number threads that can be run at the same time by the application.
- **Override compiler flags:** By default, the TDP is compiled with the build compiler flags. Use this setting to override the Build compiler flags with specific flags for compiling the TDP.
- **TDP output format:** This setting specifies how the TDP is linked to the application.
 - **None:** No TDP is generated. Use this setting if the TDP is already included in another section of the application.
 - **Object file (.obj, .o):** Default setting. Use this setting if your application does not use shared libraries.
 - **Static library (.lib, .a):** Use this setting to link the TDP as a static library.
 - **Dynamic library (.dll, .so):** Use this setting to link the TDP as a dynamic library for most cases when shared libraries are involved.
- **Link flags for library format (for library nodes only):** Link flags for generating the TDP as a shared library or DLL.
- **Use source compiler flags (for Eclipse only):** Select **Yes** to use the source build compiler flags to compile the test driver. Select **No** to use the default TDP settings.
- **Use of unloadable libraries:** Use the setting if your application uses shared libraries that can be unloaded dynamically from memory. See [Unloadable libraries on page 685](#) for details.

- ◦ **None:**The application does not dynamically unload libraries during execution.
- ◦ **This is an unloaded library:**Select this if the selected node is a library node that can be dynamically unloaded during execution.
- ◦ **Uses unloaded libraries:**Select this if the selected node is an application or test node that can use unloadable libraries.

To edit the Build settings for a node:

1. In the **Project Explorer**, click the **Settings**  button.
2. Select a node in the **Project Explorer** pane.
3. In the **Configuration Settings** list, expand **Build**.
4. Select **Compiler**, **Linker**, **Execution** or **Target Deployment Port**.
5. When you have finished, click **OK** to validate the changes.

Related Topics

[Configuration Settings on page 658](#)

General Settings

The General settings are part Configuration Settings dialog box, which allows you to configure settings for each node.

By default, the settings of each node are inherited from those of the parent node. When you override the settings of a parent node, changes are propagated to all child nodes within the same Configuration. Overridden fields are displayed in bold.

Host configuration settings

The Host Configuration area lets you specify any information about the machine on which the Target Deployment Port is to be compiled.

- **Hostname:** The *hostname* of the machine. By default this is the local host.
- **Address:** The IP address of the local machine. By default this is **127.0.0.1**. Leave this field blank to dynamically retrieve the actual IP address of the machine each time this setting is used.
- **System Testing agent TCP/IP port:** The port number used by System Testing Agents. The default is 10000.
- **Socket upload port:** The default value is 7777.

Directories settings

- **Temporary:** Enter the location for any temporary files created during the Build process
- **Report:** Specify the directory where test and analysis results are created.

Target Deployment Port data

The Target Deployment Port (TDP) Settings allow you to override the TDP used for a particular node in the current Configuration. By default, the TDP used is that of the current Configuration.

- **Name:** Displays the name of the TDP.
- **Directory:** Specifies the TDP directory name or relative path. By default, HCL OneTest™ Embedded searches for the TDP directory in the directories that are declared in the project preferences.
- **Use Directory as relative:** This option enables you to specify the TDP directory as a relative path from the project file (for example: `../TDP/clinuxgnu`). To use this option, select **True** and then, in the **Directory** setting, click ... to browse to the relative TDP directory. When this option is selected, the TDP directories declared in the project preferences are no longer searched.
- **Initial definition file:** Points to the default `.ini` file in the TDP directory.
- **Source file language:** Specifies the language of the TDP.
- **Object File Extension:** Specifies the default extension for object files produced with the current TDP.
- **Dynamic library file extension:** Specifies the file extension used for dynamic library files.
- **Static library file extension:** Specifies the file extension used for static library files.
- **Binary file extension:** Specifies the default extension for executable binaries produced with the current TDP (for example: `.exe`).
- **Source File Extension:** Specifies the default extension for source files used with the current TDP.

Source file information settings

The Source File Information settings are only available on the project node as they apply to how HCL OneTest™ Embedded extracts source file information and dependency files to be displayed in the **Asset Browser** view of the **Project Explorer**. These setting apply to the entire project and cannot be overridden at the node level.

- **Directories for include files:** Specifies a list of include directories for the file tagging facility.
- **Get struct definition like a class:** Extracts `struct` definitions and display them as classes in the **Asset Browser**.

CSV format settings

The CSV format settings allow you to specify options for importing a test data table from a **.csv** table file. The default values are inherited from the local settings of your operating system. See [Importing a Data Table \(.csv File\) on page 692](#) for more information.

- **CSV separator:** Specifies the character used to separate table columns.
- **CSV decimal separator:** Specifies the character used as a decimal separator.
- **CSV thousand separator:** Specifies the character used to mark thousands.

To edit the General settings for a node:

1. In the **Project Explorer**, click the **Settings**  button.
2. Select a node in the **Project Explorer** pane.
3. In the Configuration Settings list, expand **General**.
4. Select **Host Configuration**, **Directories** or **Target Deployment Port**.
5. When you have finished, click **OK** to validate the changes.

Related Topics

[About Configuration Settings on page 658](#)

Runtime Analysis settings

General runtime Analysis Settings

The General Runtime Analysis settings are part of the Configuration Settings dialog box, which allows you to configure settings for each node.

By default, the settings of each node are inherited from those of the parent node. When you override the settings of a parent node, changes are propagated to all child nodes within the same Configuration. Overridden fields are displayed in bold.

Selective instrumentation

By default, runtime analysis tools instrument all components of source code under analysis.

The Selective Instrumentation settings allow you to more finely define which units (classes or functions) you want to instrument and trace.

- **Instrument inline methods:** Extends instrumentation to inline methods.
- **Instrument included methods or functions:** Extends instrumentation to included methods or functions.

- **Selective unit instrumentation:** Click the...button to access a list of units (classes and functions) that can be explicitly selected for instrumentation. Click a unit to select or clear a unit. Use the **Select File** and **Clear File** buttons to select and clear all units from a source file.
- **Excluded files:** Specifies a list of files that are parsed by the instrumentor, but are not instrumented. Click the ... button and use the **Add** and **Remove** buttons to select the files to be excluded.
- **Excluded directories:** Click the ... button and use the **Add** and **Remove** buttons to select the files to be excluded.

Advanced options

- **Identification header:** Select this option to add an identification header to files generated by the Instrumentor, including the command line used to generate the file, the version of the product, date and operating system information.
- **Full template instrumentation:** By default unused methods within a template are ignored by the Instrumentor. Set this option to **Yes** to analyze and instrument all template methods, even if they are not used.
- **Application includes system files:** Set this option to **Yes** if the application includes system files such as **windows.h** in Windows or **pthread.h** in UNIX.
- **Internal data allocation:** Select either **Static declaration** or **Dynamic allocation** as the memory allocation method for storing intermediate variables in the instrumented source code.
- **Additional instrumentor options:** This setting allows you to add command line options for the Instrumentor. Normally, this line should be left blank.
- **Use alternate checksum:** Select **Yes** to calculate a more unambiguous checksum for .fdc and .tsf files. Select **No** to remain compatible with existing .fdc and .tsf files. Before using this option, you must delete existing fdc and tsf files, which will be re-created with the new checksum. File keys are not changed by this option.
- **Check internal data before use:** Set this option to **Yes** if you are experiencing crashes of the application when Runtime Analysis is engaged. This option improves compatibility but increases memory usage.
- **Generate TPM files:** Set this option to **Yes** if you want to generate metrics for the test process monitor.

Snapshot settings

In some case, such as with applications that never terminate or when working with timing or memory-sensitive targets, you might need to dump traces at specific points in your code.

- **On function entry:** Allows you to specify a list of function names, from your source code, that will dump traces at the beginning of the function.
- **On function return:** Allows you to specify a list of function names, from your source code, that will dump traces at the end of the function.
- **On function call:** Allows you to specify a list of function names, from your source code, that will dump traces before the function is called.

For each tab, click the ... button to open the function name selection box. Use the **Add** and **Remove** buttons to create a list of function names.

See [Generating SCI Dumps on page 1051](#) for more information.

Static file storage

Depending on the runtime analysis feature, the product generates **.tsf** or **.fdc** temporary static data files during source code instrumentation of the application under analysis.

- **Code Coverage static file storage (.fdc):** These settings apply to Code Coverage **.fdc** static trace files:
 - **Build Directory:** Select this option to use the current directory for all generated files.
 - **Other Directory:** Select this option to define a specific directory.
 - **Source Directory:** Select this option to use the same directory as the source under analysis.
 - **Use Single Temporary File (.fdc):** By default, Code Coverage produces one **.fdc** file for each instrumented source file. Select this option to use a single **.fdc** file for all instrumented source files, and specify its location.
- **FDC Directory or Name:** When using the **Use single temporary file (.fdc)** option in the previous setting, specify a location for the **.fdc** file.
- **Memory Profiling, Performance Profiling, and Runtime Tracing storage:** This setting applies to Memory Profiling, Performance Profiling and Runtime Tracing **.tsf** static trace files.
 - **Build directory:** Select this option to use the current directory for all generated files.
 - **Other directory:** Select this option to define a specific directory.

- **Source directory:** Select this option to use the same directory as the source under analysis.
- **Use single temporary file (.tsf):** By default, Memory Profiling, Performance Profiling and Runtime Tracing produces one **.tsf** file for each instrumented source file. Select this option to use a single **.tsf** file for all instrumented source files, and specify its location.
- **TSF directory or name:** When using the **Use single temporary file (.tsf)** option in the previous setting, specify a location for the **.tsf** file.

To edit the General Runtime Analysis settings for a node:

1. In the **Project Explorer**, click the **Settings**  button.
2. Select a node in the **Project Explorer** pane.
3. In the Configuration Settings list, expand **Runtime Analysis** and select **General**.
4. Select **Snapshot, Selective Instrumentation, Static File Storage** or **Miscellaneous**.
5. When you have finished, click **OK** to validate the changes.

Related Topics

[Using Runtime Analysis Features on page 306](#) | [Configuration Settings on page 658](#)

Code Coverage Settings

The Code Coverage settings are part of the **Runtime Analysis** node of the Configuration Settings dialog box, which allows you to configure settings for each node.

By default, the settings of each node are inherited from those of the parent node. When you override the settings of a parent node, changes are propagated to all child nodes within the same Configuration. Overridden fields are displayed in bold.

Instrumentation control settings

You can use the Coverage Type settings to declare various types of coverage.

- **Coverage level functions** : select between function **Entries, With exits**, or **None**.
- **Coverage Level Blocks:** select the desired block code coverage type. Please see [Selecting Coverage Types on page 310](#) for details on using each coverage type with each language.

You can combine, enable, or disable any of these coverage types before running the application node. All coverage types selected for instrumentation can be filtered out in the [Code Coverage Viewer on page 349](#).

- **Coverage level calls:** select **Yes** or **No** to toggle call code coverage. For Ada and C only.
- **Coverage level conditions:** select the condition level of code coverage to be included in the report. For Ada and C only:
 - **None:** The coverage report ignores conditions.
 - **Basic:** Only basic conditions are included in the coverage report.
 - **Modified (MC/DC):** Only modified conditions are included in the coverage section of the test report.
 - **Modified and Multiple:** Both modified and multiple conditions are included in the coverage report.
 - **Forced Modified (MC/DC):** The report includes modified conditions where all operators are replaced with bitwise operators.
 - **Forced Modified and Multiple:** The report includes modified and multiple conditions where all operators are replaced with bitwise operators.

See [Condition coverage \(C\) on page 330](#), [Condition coverage \(Ada\) on page 316](#), and [Bitwise MC/DC coverage on page 355](#) for more information about coverage levels.

- **Condition in expression:** Select **Yes** to consider relational operators in an expression (for example: $y = (a > 0)$) as conditions.
- **Ternary coverage (for C and C++ only):** For C and C++, when this option is selected, Code Coverage is extended to ternary expressions as statement blocks.
- **Information Mode:** This setting specifies the [Instrumentation Modes on page 309](#) to be used by Code Coverage.
 - **Default (Optimized for Code Size and Speed):** This setting uses one byte per branch to indicate branch coverage.
 - **Compact (Optimized for Memory):** This setting uses one bit per branch. This method saves target memory but uses more CPU time.
 - **Report Hit Count:** This adds information about the number of times each branch was executed. This method uses one integer per branch.
- **Ada specification (For Ada only):** Selecting this option extends instrumentation to Ada package specifications. Specifications can contain calls and conditions. In this case, the specification file must be included in the application node.
- **Excluded function calls:** Specifies a list of functions to be excluded from the call coverage instrumentation type, such as **printf** or **fopen**. Use the **Add**, **Remove** buttons to tell Code Coverage the functions to be excluded.

- **Exclude for loops (for C and C++ only):** Select **Yes** to exclude *for* loops from instrumentation. Only *while* and *do* loops are instrumented.
- **Bitwise as logical (for C and C++ only):** Select **Yes** to instrument bitwise operators as logical when both operands are booleans. See [Bitwise MC/DC coverage on page 355](#).
- **Not returning Functions (for C and C++ only):** Type the identifiers (not signatures) of the functions that do not return (functions that execute a `longjmp` or `exit`).
- **Generated package prefix (for Ada only):** Add a new prefix to Ada packages if the default Code Coverage prefix (**atc_**) generates conflicts.
- **Generated package suffix (for Ada only):** Specifies how Code Coverage names the instrumented Ada packages:
 - Select **Standard** to use the your package name as a suffix
 - Select **Short** to reduce the size of the generated package name for compilers that have a package name length limit.

Advanced Options

- **Trace file name (.tio):** this allows you to specify a path and filename for the **.tio** dynamic coverage trace file.
- **Key ignores source file path:** Identifies source files based only on the filename instead of the complete path. Use this option to consolidate test results when a same file can be located in different paths. This can be useful in some multi-user environments that use source control. If you use this option, make sure that the source file names used by your application are unique.
- **Compute deprecated metrics:** This setting is for compatibility with third party tools designed for earlier versions of the product (before v2002.05). Set this to **No** in most cases.
- **User comment:** This adds a comment to the Code Coverage Report. This can be useful for identifying reports produced under different Configurations. To view the comment, click the a magnifying glass symbol that is displayed at the top of your source code in the [Code Coverage Viewer on page 349](#).
- **Report summary:** Select **Yes** to add the coverage summary to the summary text file of the selected node.
- **On the fly frequency dump (for C and C++ only):** Specify the function call number after which the coverage results are updated dynamically during execution. **0** means no update during execution.

To change the Code Coverage Instrumentation Control setting for an application or test node.

1. In the **Project Explorer**, click the **Settings**  button.
2. Select a node in the **Project Explorer** pane.
3. In the Configuration Settings list, expand **Runtime Analysis** and select **Coverage**.

4. Select **Instrumentation Control**.
5. When you have finished, click **OK** to validate the changes.

Related Topics

[Using Runtime Analysis Features on page 306](#) | [Configuration Settings on page 658](#) | [Selecting Coverage Types on page 310](#)

Memory Profiling settings

The **Memory Profiling** settings are part of the **Runtime Analysis** node of the Configuration Settings dialog box, which allows you to configure settings for each node.

By default, the settings of each node are inherited from those of the parent node. When you override the settings of a parent node, changes are propagated to all child nodes within the same Configuration. Overridden fields are displayed in bold.

Instrumentation control

- **Detect File in Use (FIU)**: When the application exits, this option reports any files left open. See [File in Use \(FIU\) on page 371](#).
- **Detect Memory in use (MIU)**: When the application exits, this option reports allocated memory that is still referenced. See [Memory in Use \(MIU\) on page 370](#).
- **Free Invalid Memory (FIM)**: This option activates the detection of invalid free memory instructions. See [Freeing Invalid Memory \(FIM\) on page 367](#).
- **Detect Signal (SIG)**: This option indicates the signal number received by the application forcing it to exit. See [Signal Handled \(SIG\) on page 372](#).
- **Detect Freeing Freed Memory (FFM) and Detect Free Memory Write (FMWL)**: Select **Yes** to activate detection of these errors. See [Freeing Freed Memory \(FFM\) on page 366](#) and [Late Detect Free Memory Write \(FMWL\) on page 368](#).
- **Free queue length (blocks)**: specifies the number of memory blocks that are kept free.
- **Free queue size (bytes)**: specifies the total buffer size for *free queue* blocks. See [Freeing Freed Memory \(FFM\) on page 366](#) and [Late Detect Free Memory Write \(FMWL\) on page 368](#).
- **Largest free queue block size (bytes)**: Specifies the size of the largest block to be kept in the free queue.
- **Detect Array Bounds Write (ABWL)**: Select **Yes** to activate detection of this error. See [Late Detect Array Bounds Write \(ABWL\) on page 367](#).
- **Red zone length (bytes)** specifies the number of bytes added by Memory Profiling around the memory range for bounds detection.

- **Number of functions in call stack:** specifies the maximum number of functions reported from the end of the CPU call stack. The default value is **6**.
- **Only show memory leaks with call stack:** select this option to only record memory leaks that are associated with a call stack. Memory allocations that occurred before the application started do not have a call stack and are not included in the Memory Profiling report.
- **Line number link:** Select **Statement** to link the line number in the report to the corresponding allocation or free statement in the function. Select **Function** to link only to the function entry and to improve performance.
- **Only show new memory leaks in each dump:** In multi-dump report, Memory leaks (MLK) and potential leaks (MPK) are only reported once.

Advanced options

- **Trace File Name (.tpf):** This setting allows you to specify a filename for the generated **.tpf** trace file.
- **Exclude block tracking before init:** This setting disables memory profiling for any memory blocks allocated before the first execution of instrumented code. Use this option to prevent crashes when the system uses memory allocations that cannot be tracked.
- **Excluded global variables:** Specifies a list of global variables that are not to be inspected for memory leaks. This option can be useful to save time and instrumentation overhead on trusted code. Use the **Add** and **Remove** buttons to add and remove global variables.
- **Exclude variables from directories:** Specifies a list of directories from which any variables found in files are not to be inspected for memory leaks.
- **Break on error:** Use this option to break the execution when an error is encountered. The break point must be set to **priv_check_failed** in debug mode.
- **ABWL and FMWL check frequency:** Use this to check for ABWL and FMWL errors:
 - Each time the memory is dumped (by default).
 - Each time a manual check macro is encountered in the code.
 - Each function return.

These checks can be performed either on all memory blocks or only a selection of memory blocks. See [Checking for ABWL and FMWL errors on page 378](#) for more information.

- **Preserve block content:** Set this setting to **Yes** to preserve the content of memory blocks freed by the application. Use this setting to avoid application crashes with Memory Profiling is engaged. However, reads to freed blocks of memory are no longer detected.

Related Topics

[About Memory Profiling on page 363](#) | [JVMPI Technology on page 383](#) | [About Configuration Settings on page 658](#)

Performance Profiling settings

The **Performance Profiling** settings are part of the **Runtime Analysis** node of the Configuration Settings dialog box, which allows you to configure settings for each node.

By default, the settings of each node are inherited from those of the parent node. When you override the settings of a parent node, changes are propagated to all child nodes within the same Configuration. Overridden fields are displayed in bold.

- **Trace file name (.tqf)**: This box allows you to specify a filename for the generated **.tqf** trace file for Performance Profiling.
- **Compute min max times**: This setting specifies whether you want to record minimum and maximum function execution times. By default this setting is disabled because the option can use a large amount of memory, which may cause issues on embedded systems.

To edit the Performance Profiling settings for one or several nodes:

1. In the **Project Explorer**, click the **Settings**  button.
2. Select a node in the **Project Explorer** pane.
3. In the Configuration Settings list, expand **Runtime Analysis** and select **Performance Profiling**.
4. When you have finished, click **OK** to validate the changes.

Related Topics

[About Performance Profiling on page 384](#) | [Performance Profiling Results on page 78](#) | [About Configuration Settings on page 658](#)

Runtime Tracing settings

The Runtime Tracing Control settings are part of the **Runtime Analysis** node of the Configuration Settings dialog box, which allows you to configure settings for each node.

By default, the settings of each node are inherited from those of the parent node. When you override the settings of a parent node, changes are propagated to all child nodes within the same Configuration. Overridden fields are displayed in bold.

Instrumentation Control

- **Runtime Tracing file name (.tdf):** This allows you to force a filename and path for the dynamic **.tdf** file. By default, the **.tdf** carries the name of the application node.
- **Show unnamed classes:** For C++ only. When this option is disabled, unnamed class are not instrumented.
- **Show data classes:** When this option is disabled, structures or classes that do not contain methods are excluded from instrumentation. Disable this option to reduce instrumentation overhead.
- **Display one note for class templates:** For C++ only. With this option, the UML/SD Viewer will not display notes for each instance of template classes.
- **Display return functions as sequence:** For C only. With this option, the UML/SD Viewer displays calls located in return expressions as if they were executed sequentially and not in a nested manner.

Trace Control

- **Split trace file:** See Splitting trace files for more information on this setting.
- **Maximum size (Kbytes):** This specifies the maximum size for a split **.tdf** file. When this size is reached, a new split **.tdf** file is created.
- **File name prefix:** By default, split files are named as **att_<number>.tdf**, where **<number>** is a 4-digit sequence number. This setting allows you to replace the **att_** prefix with the prefix of your choice.
- **Automatic loop detection:** Loop detection simplifies UML sequence diagrams by summarizing repeating traces into a loop symbol. Loops are an extension to the UML sequence diagram standard and are not supported by UML.
- **Additional options:** This setting allows you to add command line options. Normally, this line should be left blank.
- **Display maximum call level:** When selected, the Target Deployment Port records the highest level attained by the call stack during the trace. This information is displayed at the end of the UML Sequence Diagram in the UML/SD Viewer as **Maximum Calling Level Reached**.

Runtime options

These settings allow you to set compilation flags that define how the Runtime Tracing feature interacts with the Target Deployment Port. These are general settings for the Target Deployment Port.

- **Disable on-the-fly mode:** When selected, this setting stops on-the-fly updating of the dynamic **.tdf** file. This option is primarily for Target Deployment Ports that use **printf** output.
- **Runtime tracing buffer and Partial Runtime Tracing flush:** Please see [Trace Item Buffer on page 413](#) and [Partial Trace Flush on page 411](#) for more information about these settings.

- **Maximum buffer size (events):** Maximum number of events recorded in the buffer before it is flushed.
- **User signal action:** Specify an action to be performed when a user signal is detected: No action, Flush call stack, Runtime Tracing on/off
- **Record and display time stamp:** This setting adds time stamp information to each element in the UML sequence diagram generated by Runtime Tracing.
- **Record and display heap size:** This setting enables the Heap Size Bar in the UML sequence diagram produced by Runtime Tracing.
- **Record and display thread info:** This setting enables the Thread Bar in the UML sequence diagram produced by Runtime Tracing.

To edit the Runtime Tracing settings for one or several nodes:

1. In the **Project Explorer**, click the **Settings**  button.
2. Select a node in the **Project Explorer** pane.
3. Select the **Runtime Analysis** node and the **Runtime Tracing** node.
4. In the Configuration Settings list, select **Instrumentation Control**, **Trace Control** or **Target Deployment Port Settings**.
5. When you have finished, click **OK** to validate the changes.

Related Topics

[About Configuration Settings on page 658](#) | [Multi-Thread Support on page 410](#) | [Partial Trace Flush on page 411](#) | [Trace Item Buffer on page 413](#) | [Splitting Trace Files](#)

Studio automated Testing settings

Component Testing Settings for C and Ada

The Component Testing settings are part of the **Component Testing for C and Ada** node of the Configuration Settings dialog box, which allows you to configure settings for each node.

By default, the settings of each node are inherited from those of the parent node. When you override the settings of a parent node, changes are propagated to all child nodes within the same Configuration. Overridden fields are displayed in bold.

Test Script Compiler

- **Intermediate test result file name (.rio):** Specifies the filename of the files produced during test execution. Changing this setting will modify the name of the intermediate .rio file, the tester (.c in C or .adb in Ada), and the generated main procedure/package name (in Ada).
- **Continue test build despite warnings:** Select this option to ignore warning during the test compilation phase.
- **Break on error:** Select this option to call a breakpoint function whenever a test failure occurs in a .ptu Test Driver script. To use this feature, you must set a breakpoint on the function `priv_check_failed ()`, located in the `<target_deployment_port> /lib/priv.c`, file. You can use this option for debugging purposes.
- **Authorize stubbing:** This setting determines the conditional generation of code in the test program when using `SIMUL` blocks in the .ptu test script.
- **Allow stubbing of module functions:** Set this option to Yes to allow stubbing of functions that are in the same source file as the functions under test.
- **Enable additional options:** Set this option to **Yes** if you want to specify additional command line options. In most cases this should be set to **No**.
- **Additional options:** Use this setting to specify extra command line options. In most cases this should be empty.
- **Test point name (for Ada only):** Entry point for the test program. The default entry point is `ATTOL_TEST`.
- **Test point packages (for Ada only):** List of packages containing the test program entry point.

Report generator

- **Display variables:** lets you select the level of detail of the Component Testing output report:
 - **Collapse tests:** collapses 'test loop' and 'init in' tests into one block.
 - **Incorrect:** shows only incorrect variables.
 - **Only for failed tests:** shows all variables for failed tests."
- **Displays initial and expected values:** the way in which the values assigned to each variable are displayed in the report. See [Initial and Expected Values on page 459](#).
- **Display arrays and structures:** indicates the way in which Component Testing processes variable array and structure statements. See [Array and Structure Display on page 510](#) for more information.
- **Generate report without coverage:** This setting hides coverage information in the Component Testing Report.
- **Generate graphics report:** This setting generates a graphics report when value arrays are produced by `loop`, `for` and `init in` statements.

- **Max tests per report:** When large reports are generated, this option allows you to split the results into multiple report files that contain the specified number of tests.
- **Compare two test runs:** This setting activates the comparison option. See [Comparing C Test Reports on page 510](#) or [Comparing Ada Test Reports on page 584](#).
- **Enable additional options:** Set this option to **Yes** if you want to specify additional command line options. In most cases this should be set to **No**.
- **Additional options:** Use this setting to specify extra command line options. In most cases this should be empty.

Related Topics

[About Configuration Settings on page 658](#) | [Stub Simulation Overview on page 475](#)

Component Testing for C++ settings

The Component Testing settings for C++ are part of the Configuration Settings dialog box, which allows you to configure settings for each node in your workspace.

The Component Testing for C++ node settings lets you to customize the parameters for the Component Testing for C++ feature of HCL OneTest™ Embedded.

By default, the settings of each node are inherited from those of the parent node. When you override the settings of a parent node, changes are propagated to all child nodes within the same Configuration. Overridden fields are displayed in bold.

Contract Check Options

These options are used by the [C++ Contract-Check Script on page 513](#).

- **Break on assertion failure:** Select this option to call a breakpoint function whenever an assertion fails in an **otcContract Check** script. To use this feature, you must set a breakpoint on the function **priv_check_failure()**, located in the `<target_deployment_port>/lib/priv.c`, file. You can use this option, for example, to debug your application when an assertion fails.
- **Report only failed assertions:** Select this option to hide passed assertions in the UML Sequence Diagram generated by Component Testing for C++. Only failed assertions are displayed. This option also reduces the size of the intermediate trace file.
- **Trace unchanged states:** Select this option to report states in UML Sequence Diagram generated by Component Testing for C++ each time states are evaluated. If the option is disabled, states are reported in UML Sequence Diagram only when they change. This affects both trace size and UML Sequence Diagram display size, but has no impact on execution time.

- **Check 'const' methods:** Usually C++ *const* methods are not checked for state changes because they cannot modify a field of the *this* object. Instead, *const* methods are only evaluated once for invariants. In some cases, however, the *this* object may change even if the method is qualified with *const* (by assembler code, or by calling another method that casts the *this* parameter to a non-const type). There may also be pointer fields to objects which logically belong to the object, but the C++ Test Script Compiler will not enforce that these pointed sub-objects are not modified. Select this option only if your code contains such code implementations.
- **Reentrant objects:** Select this option if your application is multi-threaded and objects are shared by several threads. This ensures granularity for state evaluation. This option has no effect if multi-thread support is not activated in the Target Deployment Compilation Settings.
- **Enforce 'const' assertions:** When this option is selected, the compiler requires that invariant and state expressions are constant. Disable this option if you do not use the *const* qualifier on methods that are actually constant.

Testing Options

These options are used for the [C++ Test Driver Script on page 514](#).

- **Add #line directive into instrumented source file:** This option allows use of *#line* statements in the source code generated by Component Testing for C++. Disable this option in environments where the generated source code cannot use the *#line* mechanism. By default *#line* statements are generated.
- **Application includes system files:** Set this option to **Yes** if the application includes system files such as **windows.h** in Windows or **pthread.h** in UNIX.
- **Break on CHECK failure:** Select this option to call a breakpoint function whenever a check failure occurs in an *.otd* Test Driver script. To use this feature, you must set a breakpoint on the function `priv_check_failed()`, located in the `<target_deployment_port>/lib/priv.c`, file. You can use this option for debugging purposes.
- **Test class friend of class under test:** Set this setting to **Yes** if you want the test to access any private or protected members (friend classes) of the components under test. The class must be mentioned in an OTC file to be recognized as a friend of the test class.
- **Instances stack size:** This value defines the maximum level for C++ Test Driver Script calls that you expect to reach when running an *.otd* Test Driver script. The C++ Test Driver Script calling stack includes **RUN**, **CALL** and **STUBs**. The default value is 256 and should be large enough for most cases. When using recursive stubs, you may need to increase this value.
- **Display only failed CHECKs:** Select this option to hide notes related to passed CHECK statements in the UML Sequence Diagram generated by Component Testing for C++. Failed checks are still displayed. The component testing report is not affected by this option. This option also reduces the size of the intermediate trace file.

- **Display all PRINT arguments in a single note:**Select this option to display only one UML note for all arguments of a**PRINT**statement in the Component Testing for C++ UML Sequence Diagram. This option requires use of a**PRINT**buffer, which uses memory on the target machine. Disable this option if memory on the target is an issue. In this case, Component Testing for C++ generates one UML note for each argument of each**PRINT**statement.
- **PRINT buffer size (bytes):**With the previous option selected, this option defines the size, in bytes, of the buffer devoted to the**PRINT**instructions during the execution. This buffer should be large enough to handle the complete result of a**PRINT**instruction. You may have to increase this value if your**PRINT**statements contain many arguments, or if arguments are long strings.

Advanced options

This area specifies the path and filenames for the intermediate files generated by the Component Testing for C++ feature during the test execution.

- **Test driver file name:**contains the location and name of a **.cpp**source file generated from the C++ Test scripts by Component Testing for C++
- **Contract check file name:**contains the path and file name of a temporary **.otif**file created during source code instrumentation by Component Testing for C++
- **Test report file name (.xrd):**contains the location and name of the **.xrd**report file generated by Component Testing for C++
- **Maximum test compilation errors displayed:**Specifies the maximum number of error messages that can be displayed by the C++ Test Script Compiler. The default value is 30.

Related Topics

[About Configuration Settings on page 658](#) | [C++ Test Driver Script on page 514](#) | [C++ Contract-Check Script on page 513](#)

System Testing for C Settings

The Test Script Compiler settings are part of the **System Testing** node of the Configuration Settings dialog box, which allows you to configure settings for each node.

By default, the settings of each node are inherited from those of the parent node. When you override the settings of a parent node, changes are propagated to all child nodes within the same Configuration. Overridden fields are displayed in bold.

Test Script Compiler

- **Message definition files:** Use the Add and Remove buttons to create a list of message definition files. These are source files that define the structure declarations required by Virtual Testers.
- **Generate virtual testers as threads:** By default, virtual testers are generated as processes. Use this setting with multi-threaded applications.
- **Multi-thread entry point:** Specifies the name of a main function to act as an entry point in multi-threaded applications.
- **Virtual Tester Memory Allocation Method:** Allows you to allocate memory to the Virtual Tester for internal data storage.
 - **Static** - use global static variables for internal data storage. This allows the Virtual Tester to run on systems that do not support dynamic memory allocation or that have limited execution stacks.
 - **Stack** - store internal data in a local variable of the *main()* function. Necessary memory is then allocated on the execution stack.
 - **Heap** - allocate memory through a Target Deployment Port dynamic allocation function, which is configurable.
- **Duplicate user-defined static global variables:** When using multiple Virtual Tester threads, this setting allows you to duplicate static global variables for each thread. This allows multiple instances of a virtual tester to all run in the same process with their own variables.
- **Use TDP thread launcher:** Specify **Yes** if the current TDP supports launching virtual tester threads. If not, then you must write a specific program to perform this task, and set this option to **No**. See [Launching virtual tester threads on page 597](#) for more information.
- **Additional options:** Use this setting to specify extra command line options. In most cases this should be empty.
- **Continue on WAITTIL error:** Select **Yes** to force the scenario to resume after encountering an error in a **WAITTIL** statement. You can use this setting to debug virtual testers.
- **Trace Buffer Optimization:** See [Optimizing Execution Traces on page 594](#).
 - Select **Time stamp only** to generate a normal trace file.
 - Select **Block start/end only** to generate traces for each scenario beginning and end, all events, and for error cases.
 - Select **Errors only** to generate traces only if an error is detected during execution of the application.

- **Circular buffer:** Select this option to activate the [Circular Trace Buffer on page 655](#).
- **Trace buffer size (Kbytes):** This box specifies the size - in kilobytes - of the circular trace buffer. The default setting is 10Kb.

Report generator settings

- **Display initial and expected values:** the way in which the values assigned to each variable are displayed in the report. See [Initial and Expected Values on page 459](#).
- **Report generated form:** This option specifies the form of the report generated.
 - **Full:** provides a full report of all variables for each test.
 - **All failed test variables:** All the variables that are in a failed test are displayed. If all tests are passed, then the report is empty.
 - **Only failed variables:** Only failed variables are displayed. If all tests are passed, then the report is empty.
- **Sort by time stamp:** By default, the report is sorted by test script structure blocks. Select this option to force the report to follow a fully chronological order.

Advanced for System Testing Settings

- **Kill VT when RENDEZVOUS fails:** Enable this setting to force the supervisor to kill any remaining virtual testers each time a **RENDEZVOUS** fails. This prevents uncontrollable processes from running on the computer.
- **RENDEZVOUS timeout (seconds):** This specifies the timeout associated to **RENDEZVOUS** statements.
- **INTERRECV timeout (seconds):** This specifies the timeout associated to [inter-tester communications on page 621](#).
- **Agent target directory:** Specifies the directory where system testing agent is located.
- **Run without deployment:** This allows you to launch the test execution without going through the deployment phase.
- **Compress trace data:** This option performs internal compression of trace data. Select this for hard real-time constraints. If you select NO, no compression of trace data is performed.
- **Trace data buffer size (bytes):** Specifies the size of the trace data buffer.
- **On-the-fly tracing:** This option enables [on-the-fly tracing on page 656](#) at Target Deployment Port level.
- **On-the-fly tracing buffer size (bytes):** This specifies the size of the trace buffer for on-the-fly tracing. By default the buffer size is 4096 bytes.

Related Topics

[About Configuration Settings on page 658](#) | [About System Testing for C on page 586](#)

Probe Control Settings

The Probe Control settings are part of the Configuration Settings dialog box, which allows you to configure settings for each node.

By default, the settings of each node are inherited from those of the parent node. When you override the settings of a parent node, changes are propagated to all child nodes within the same Configuration. Overridden fields are displayed in bold.

Probe Control Settings

- **Probe enable:** This setting enables or disables the Trace Probe feature as implemented with System Testing for C. See [Trace Probes on page 414](#).
- **Probe settings:** These settings allow you to select the Trace Probe output mode. See [Trace Probe output modes on page 416](#).
- **USER custom files directory:** Specifies the location of the user-defined **probecst.c** and **probecst.h** files when the USER output mode is selected. See [Customizing the USER output mode on page 418](#).
- **Message definition files:** Use the **Add** and **Remove** buttons to create a list of message definition files. These are source files that define the structure declarations required by Virtual Testers.
- **Script generation flags:** Use this setting to send command line arguments to the Probe Processor for generating a **.pts** test script for use with System Testing for C. See [Traces Probes and System Testing for C on page 417](#).
- Compress trace data:
- Trace data buffer size (bytes):
- On-the-fly tracing:
- On-the-fly tracing buffer size (bytes):

To edit the Probe Control settings for a node:

1. In the **Project Explorer**, click the **Settings**  button.
2. Select a node in the **Project Explorer** pane.
3. In the Configuration Settings list, select **Probe Control**.
4. When you have finished, click **OK** to validate the changes.

Related Topics

[About Configuration Settings on page 658](#) | [Trace Probes on page 414](#)

External command settings

The External Command settings are part of the Configuration Settings dialog box, which allows you to configure settings for each node.

Use the External Command setting to set a command line for External Command nodes. An External Command is a command line that can be included at any point in your workspace. External Commands can contain HCL OneTest™ Embedded Graphical User Interface macro variables, making them context-sensitive. See the **GUI Macro Variables** chapter in the **Reference Manual**.

By default, the settings of each node are inherited from those of the parent node. When you override the settings of a parent node, changes are propagated to all child nodes within the same Configuration. Overridden fields are displayed in bold.

To edit the External Command settings for one or several nodes:

1. In the **Project Explorer**, click the **Settings**  button.
2. Select a node in the **Project Explorer** pane.
3. In the Configuration Settings list, select **External Command** and enter a **Command line**.
4. When you have finished, click **OK** to validate the changes.

Related Topics

[About Configuration Settings on page 658](#) | [Creating an External Command Node on page 681](#)

Static Metric Settings

The Static Metric settings are part of the Configuration Settings dialog box, which allows you to configure settings for each node.

Use the Static Metric settings to change any project settings related to the calculation of static metrics.

By default, the settings of each node are inherited from those of the parent node. When you override the settings of a parent node, changes are propagated to all child nodes within the same Configuration. Overridden fields are displayed in bold.

- **One level metrics (for C and C++ only)** : By default, **.met** static metric files are produced for source files as well as all dependency files that are found by the Source Code Parser. Set **One level metrics** to **Yes** to restrict the calculation of static metrics only to the source files displayed in the Project Browser. In Ada, this setting is ignored.
- **Analyzed directories**: This setting allows you to restrict the generation of **.met** metric files only to files which are located in the specified directories.
- **Generate metrics in source directories**: By default, all **.met** files are generated in the project directory, and use the same name as the source file. Select **Yes** on this setting to compute metrics for source files that have the same name but are located in different directories. In this case, each **.met** is generated in the source directory of each file.
- **Additional options**: Use this setting to specify extra command line options. In most cases this should be empty.

To edit the Static Metrics settings for one or several nodes:

1. In the **Project Explorer**, click the **Settings**  button.
2. Select a node in the **Project Explorer** pane.
3. In the Configuration Settings list, select **Static Metrics**.
4. When you have finished, click **OK** to validate the changes.

Related Topics

[About Configuration Settings on page 658](#) | [About Static Metrics on page 225](#)

Code Review settings

The Code Review settings are part of the Configuration Settings dialog box, which allows you to configure settings for each node.

Use the Code Review settings to change any project settings related to the code review tool.

By default, the settings of each node of a project are inherited from those of the parent node. When you override the settings of a parent node, changes are propagated to all child nodes within the same Configuration. Overridden fields are displayed in bold.

- **Rule configuration**: This setting specifies the file containing the rules for the code review tool. Click Browse ... to select another rule configuration file. Click the **Edit**  button to edit the rule configuration or to create a new rule configuration. See [Configuring code review rules on page 294](#) for more information.
- **Additional included system directories**: This setting specifies system include directories that are to be ignored during the code review.

- Review included system files: Select Yes to extend code review to system files that are #included in the source files.
- Naming script file: This setting allows you to specify a perl script that can check your own naming rules. See [How to customize a naming script file. on page 94](#) for more information.
- **Include Files:** Specify a list of files to preinclude, like gcc would do with the **-include= <files>** option .
- Display Errors/Warnings: Specify the maximum number of errors and warnings that you want to display in the report. By default, All errors and warnings are displayed.

To edit the Code Review settings for one or several nodes:

1. In the **Project Explorer**, click the **Settings**  button.
2. Select a node in the **Project Explorer** pane.
3. In the Configuration Settings list, select **Code Review**.
4. When you have finished, click **OK** to validate the changes.

Related Topics

[About Configuration Settings on page 658](#) | [About Static Metrics on page 225](#) | [Configuring code review rules on page 294](#)

User interface preferences

HCL OneTest™ Embedded has many **Preference** settings that allow you to configure various components of the graphical user interface.

To learn about	See
General GUI-related preferences	Project preferences on page 1015
Changing data table defaults.	Data table preferences on page 1011
Changing preferences for source control software.	CMS preferences on page 1011
Changing the behavior of the text editor.	Text Editor preferences on page 1012 and Text Editor Syntax Coloring on page 697
ClearQuest Preferences	ClearQuest preferences on page 1012
Changing the appearance of the Memory Profiling Viewer	Memory Profiling Viewer preferences on page 1019
Changing the appearance of the Report Viewer.	Report Viewer preferences on page 1018
Changing the appearance of the Metrics Viewer.	Metrics Viewer preferences on page 1017

Changing the appearance of the UML/SD Viewer.	UML/SD Viewer preferences on page 1016
Changing the appearance of the Code Coverage Viewer.	Code Coverage Viewer preferences on page 1014
Changing the appearance of the Performance Profiling Viewer.	Performance Profiling Viewer preferences on page 1013
HCL OneTest™ Embedded preferences in the Eclipse workbench.	Eclipse CDT HCL OneTest™ Embedded preferences on page 942
Changing the appearance of the Code Review viewer.	Code Review viewer preferences on page 1014

To edit product preferences:

1. From the **Edit** menu, select **Preferences**.
2. In the tree-view, select the component that you want to configure.
3. Make any changes to the preferences.
4. Click **OK**.

Related Topics

[Using the Graphical User Interface on page 657](#)

General

Connection Preferences

The **Preferences** dialog box allows you to customize HCL OneTest™ Embedded.

The **Connections** node of the **Preferences** dialog box lets you set the network parameters for the graphical user interface.

- **Allow remote connections:** This allows external commands and tools to send messages to the GUI over a network. For example, this enables the Runtime Tracing on-the-fly capability on remote hosts.
- For information only, the Current TCP/IP port is automatically selected by GUI.

Related Topics

[Editing Preferences on page 1009](#)

Output window preferences

The general colors and font preferences panel allows you to specify the colors and fonts used in the output window.

This panel opens from menu **Edit > Preferences**. You can choose **Output Window** style or **Output Window Error** style.

Output window/Output window errors

In this panel, you can change the color and the font style used to display the build output messages or the standard error messages in the Output window. This windows opens from the menu **View > Other windowsOutput Window**.

Source control (CMS) preferences

The **Preferences** dialog box allows you to change the settings related to the integration of the product with Rational® ClearCase® or other configuration management software (CMS).

- **Repository directory:** Use this box to specify the location of the vault directory for the CMS tool.
- **Selected Configuration Management System:** Use this box to select Rational® ClearCase® or a different CMS tool. Before setting this option, make sure that the CMS system has been configured in Tools menu.

Related Topics

[Editing Preferences on page 1009](#) | [Working with Rational ClearCase on page 33](#) | Customizing Configuration Management

Data table preferences

The **Data table preferences** dialog box lets you specify how HCL OneTest™ Embedded handles the import of .csv data tables into the project by default.

These options define the default behavior, which can be overridden at the project or node level by changing the data table settings in the [General Settings on page 987](#).

The Project preferences contain a main page and two additional pages:

- **CSV Decimal Separator:** Specifies the character used as a decimal separator.
- **CSV Separator:** Specifies the character used to separate table columns.
- **CSV Thousand Separator:** Specifies the character used to mark thousands.

See [Importing a Data Table \(.csv File\) on page 692](#) for more information.

Related Topics

[Importing a Data Table \(.csv File\) on page 692](#) | [General Settings on page 987](#)

Internationalization preferences

The **Internationalization preferences** allow you to specify the codec that HCL OneTest™ Embedded uses to handle international character sets.

- **Codec for the locale:** Specifies the character set to be used. Change this setting if some international character sets are not displayed properly in HCL OneTest™ Embedded. In most cases, select **Auto Detect**.

Related Topics

[User interface preferences on page 1009](#)

Environment preferences

The **Environment preferences** allow you to add, remove and modify environment variables that can be required for HCL OneTest™ Embedded.

To add an environment variable, type the name of the variable and the value, and click Add.

Related Topics

[User interface preferences on page 1009](#)

ClearQuest preferences

The **Preferences** dialog box allows you to specify the location of the Rational® ClearQuest® database.

Please refer to the documentation provided with Rational® ClearQuest® for more information.

- **Schema Repository:** Use this box to select the schema repository you want to use.
- **Database:** Use this box to enter the location of the ClearQuest database.
- **User Name and Password:** Enter the user information provided by your ClearQuest administrator.

Related Topics

[Editing Preferences on page 1009](#) | [Working with Rational ClearQuest on page 34](#)

Editor preferences

The **Preferences** dialog box allows you to change the appearance of the source code and scripts in the text editor.

Editor

- **Font:** This allows you to change the general font type and size for Editor.
- **Global Colors:** This is where you select background colors for text categorized as **Normal**, **Information** or **Error** as well as the general background color. Click a color to open a standard color palette.

- **Autodetect parenthesis and bracket mismatch** - When this option is selected, the **Error** color is used when the Editor detects a missing bracket "[" or parenthesis ")".
- **Tabulation length**: This specifies the tabulation length, which is equivalent to a number of inserted spaces.

Syntax Colors

- **Elements**: This list allows you to select one or several styles that you want to change. To change several styles at the same time, you can perform multiple selections in the style list.
- **Text Color**: This allows you to change the foreground and background colors for the selected style. This opens a standard color palette.
- **Text Attributes**: This allows you to set the selected style to Bold, Italic, Underlined or Dashed.

Related Topics

[Editing Preferences on page 1009](#) | [About the Text Editor on page 693](#)

Performance Profiling viewer preferences

The **Preferences** dialog box allows you to change the appearance of your Performance Profiling reports.

To choose Performance Profiling report colors and attributes:

Performance Profiling Viewer

- **Background color**: This allows you to choose a background color for the **Performance Profiling Viewer** window.
- **Automatic raise viewer on tree selection change**: Specifies that the viewer is give focus when an code review element is selected.

Styles

- **Styles**: This list allows you to select one or several styles that you want to change. To change several styles at the same time, you can perform multiple selections in the style list.
- **Font**: This allows you to change the font type and size for the selected style.
- **Text Color**: This allows you to change the foreground and background colors for the selected style. This opens a standard color palette.
- **Text Attributes**: This allows you to set the selected style to Bold, Italic, Underlined or Dashed.

Related Topics

[Editing Preferences on page 1009](#) | [Using the Performance Profiling Viewer on page 391](#)

Performance Profiling viewer preferences

Performance Profiling for C and C++

The **Preferences** dialog box allows you to change the appearance of your code review reports.

Code Review Viewer

- **Background color:** This allows you to choose a background color for the Code Review viewer window.
- **Automatic raise viewer on tree selection change:** Specifies that the viewer is give focus when an code review element is selected.
- **Report Contents Depth Open:** Specifies the depth of the report tree.

Styles

- **Styles:** This list allows you to select one or several styles that you want to change. To change several styles at the same time, you can perform multiple selections in the style list.
- **Font:** This allows you to change the font type and size for the selected style.
- **Text Color:** This allows you to change the foreground and background colors for the selected style. This opens a standard color palette.
- **Text Attributes:** This allows you to set the selected style to Bold, Italic, Underlined or Dashed.

Related Topics

[Editing Preferences on page 1009](#) | [Using the code review viewer on page 298](#)

Code Coverage viewer preferences

The **Preferences** dialog box allows you to change the appearance of your Code Coverage reports.

Code Coverage Viewer

- **Background color:** This allows you to choose a background color for the **Code Coverage Viewer** window.
- **Choose the display value type for rates:** This allows you to choose the format of the rates in your Code Coverage reports. You can choose between absolute, percentage or both. The setting is taken into account when the reports are generated.

Styles

- **Styles:** This list allows you to select one or several styles that you want to change. To change several styles at the same time, you can perform multiple selections in the style list.
- **Font:** This allows you to change the font type and size for the selected style.
- **Text Color:** This allows you to change the foreground and background colors for the selected style. This opens a standard color palette.
- **Text Attributes:** This allows you to set the selected style to Bold, Italic, Underlined or Dashed.

Related Topics

[Editing Preferences on page 1009](#) | [About the Code Coverage Viewer on page 349](#)

Project preferences

The **Project Preferences** dialog box lets you set parameters for the HCL OneTest™ Embedded project.

The Project preferences contain a main page and two additional pages:

- Source File Types
- TDP Directories

In the **Preferences** dialog box, select Project to change the project preferences.

- **Automatic file tagging:** Select this option to activate the Project Explorer's automatic parsing mode, in which all source code and script components are automatically listed. If disabled, you will have to manually [refresh the File View on page 679](#) each time you modify the structure of a file.

Note If the structure of a source files has changed since the last file refresh, metrics calculation cannot be performed. This impacts the Component Testing Wizard, where the Unit Selection view will be disabled.

- **Compute static metrics for Component Testing Wizard:** Select this option so that when you open the Component Testing Wizard, the static metrics are recalculated whenever a new activity is created (file added, modified or refreshed). To open the Component Testing Wizard, click in the main toolbar **File > New > New activity > Component testing**.
- **Verbose output:** Select this option to prompt the HCL OneTest™ Embedded GUI to report detailed information to the Output Window during execution. Use this option to debug any compilation issues.
- **Show result nodes in Project Explorer:** Select this option to display test and runtime analysis reports in the Project Browser once they have been successfully generated. Result nodes appear inside their test or application nodes. If the option is not selected, the result nodes do not display.

- **Use Automatic Relative Project's Path computing in the settings file selectors:** Select this option to calculate the relative path to the project in configuration settings. If the option is not selected, this is the absolute path that is calculated.
- **Keep the execution node results between two sessions:** Select this option to save the node results for the current project so that it is available in another session. When a test is executed in a project's node, a green check is displayed on the left of the node if the result is successful, or a red cross if it failed. If this option is enabled, when you exit HCL OneTest™ Embedded Studio, the node status is saved in the project file and when you open the project in another session, you can see that the status on the last build executed is kept. If the preference is not enabled, the status is not saved.

The Project Preferences contains two additional pages:

- Source File Types
- TDP Directories

Source File Types

Use this page to specify any new file types that you want to use in HCL OneTest™ Embedded projects.

Click the New button to add a new line. In the extension column, enter the file extension in wildcard format, for example: *.asm. In the Description column, enter a description for the file type, for example: **Assembler source files**.

TDP Directories

If you have used the TDP Editor to generate Target Deployment Ports (TDPs) in locations other than the default target directory, use this page to specify a list of directories where HCL OneTest™ Embedded will search for TDPs. Directories are searched in the order defined in this list.

Select to Use default Target Deployment Port directory to use the default **targets** directory only, which is located in the HCL OneTest™ Embedded installation directory.

Related Topics

[Editing Preferences on page 1009](#)

UML/SD viewer preferences

The **Preferences** dialog box allows you to change the appearance of the UML Sequence Diagram reports.

UML/SD Viewer

- **Background:** This allows you to choose a background color for the UML sequence diagram.
- **Panel:** This allows you to choose a background color for panels in the UML sequence diagram.
- **Panel Background:** This allows you to choose a background color for selected panels.

- **Coverage Bar:** This allows you to choose a background color for the coverage bar.
- **Memory Usage:** This allows you to choose a background color for the memory usage bar.
- **Print Page header:** Select this option to print a page header.
- **Print Page footer:** Select this option to print a page footer.
- **Display Page Breaks:** When this option is selected, the UML/SD Viewer displays horizontal and vertical dash lines representing the page size for printing.
- **Show tooltip in UML/SD Viewer:** Use this option to hide or show the information tooltip in the UML/SD Viewer.
- **Time Stamp Format:** Use the editable box to select the format in which time stamps are displayed in the UML/SD Viewer. See [Time Stamping on page 403](#).

Styles or Styles System Test:

- **Styles:** This list allows you to select one or several styles that you want to change. To change several styles at the same time, you can perform multiple selections in the style list.
- **Font:** This allows you to change the font type and size for the selected style.
- **Text Color:** This allows you to change the foreground and background colors for the selected style. This opens a standard color palette.
- **Text Attributes:** This allows you to set the selected style to Bold, Italic, Underlined or Dashed.

Related Topics

Editing Preferences | About the UML/SD Viewer | [Time Stamping on page 403](#)

Metrics viewer preferences

Static Metrics for C, C++ and Ada

The **Preferences** dialog box allows you to change the appearance of the Static Metrics reports.

Metrics Viewer

- **Background color:** This allows you to choose a background color for the **Metrics Viewer** window.
- **Stroud number:** This parameter modifies the results of [Halstead Metrics on page 231](#).

Styles

- **Styles:** This list allows you to select one or several styles that you want to change. To change several styles at the same time, you can perform multiple selections in the style list.
- **Font:** This allows you to change the font type and size for the selected style.
- **Text Color:** This allows you to change the foreground and background colors for the selected style. This opens a standard color palette.
- **Text Attributes:** This allows you to set the selected style to Bold, Italic, Underlined or Dashed.

Related Topics

[Editing Preferences on page 1009](#) | [Viewing Static Metrics on page 226](#) | [Halstead Metrics on page 231](#)

Graphics viewer preferences

The **Preferences** dialog box allows you to change the appearance of graphs produced by HCL OneTest™ Embedded.

Graphics Viewer

- **Background color:** This allows you to choose a background color for the **Graphics Viewer** window.
- **Color and Background Color:** This allows you to choose the color and background color for the **Graphics Viewer** panel.

Styles

- **Styles:** This list allows you to select one or several styles that you want to change. To change several styles at the same time, you can perform multiple selections in the style list.
- **Font:** This allows you to change the font type and size for the selected style.
- **Text Color:** This allows you to change the foreground and background colors for the selected style. This opens a standard color palette.
- **Text Attributes:** This allows you to set the selected style to Bold, Italic, Underlined or Dashed.

Report viewer preferences

The **Preferences** dialog box allows you to change the appearance of your Test and Runtime Analysis reports.

Report Viewer

- **Background color:** This allows you to choose a background color for the **Report Viewer** window.

Syntax Color

- **Styles:** This list allows you to select one or several styles that you want to change. To change several styles at the same time, you can perform multiple selections in the style list.
- **Font:** This allows you to change the font type and size for the selected style.
- **Text Color:** This allows you to change the foreground and background colors for the selected style. This opens a standard color palette.
- **Text Attributes:** This allows you to set the selected style to Bold, Italic, Underlined or Dashed.

Related Topics

[Editing Preferences on page 1009](#) | [Using the Report Viewer on page 705](#)

Memory Profiling viewer preferences

Memory Profiling for C and C++

The **Preferences** dialog box allows you to change the appearance of your Memory Profiling reports for C and C++.

Memory Profiling Viewer

- **Background color:** This allows you to choose a background color for the **Memory Profiling Viewer** window.

Styles

- **Styles:** This list allows you to select one or several styles that you want to change. To change several styles at the same time, you can perform multiple selections in the style list.
- **Font:** This allows you to change the font type and size for the selected style.
- **Text Color:** This allows you to change the foreground and background colors for the selected style. This opens a standard color palette.
- **Text Attributes:** This allows you to set the selected style to Bold, Italic, Underlined or Dashed.

Related Topics

[Editing Preferences on page 1009](#) | [Using the Memory Profiling Viewer on page 377](#)

Studio GUI elements

When you launch the HCL OneTest™ Embedded Graphical User Interface (GUI), you are first greeted with the Start Page and a series of windows. Click the elements below to learn how to use them:

- The [Start Page on page 1020](#) is your main starting point when you launch the GUI
- The [Project Explorer on page 1021](#) is where you create, develop and execute your project nodes
- The [Properties Window on page 1023](#) provides information about node properties
- The [Output Window on page 1021](#) displays the output of command line tools and compilers
- The [Standard Toolbars on page 1025](#) provide quick and convenient access to the most commonly used features
- The [Report Explorer on page 1024](#) allows you to navigate through analysis reports

Related Topics

[Using the Graphical User Interface on page 657](#) | [Activity Wizards on page 663](#) | [GUI Components and Tools on page 658](#)

Start Page

When you launch the graphical user interface, the first element that appears is the HCL OneTest™ Embedded Start Page.

The Start Page is the central location of the application. From here, you can create a new project, start a new activity and navigate through existing project reports.

The Start Page contains the following sections:

- **Welcome:** General information for first-time users of the product.
- **Get Started:** This section lists your recent projects as well as a series of sample projects provided with HCL OneTest™ Embedded.
- **Activities:** This section displays a series of new activities. Click a new activity to launch the corresponding activity wizard. A project must be open before you can select a new activity.
- **Examples:** A set of sample projects for tutorial or demonstration purposes. You can use these projects to get familiar with the product.
- **Support:** Links to Customer Support and online documentation.

To reset the recent files list, select the Start page and click the **Reset**  toolbar button, and then click the **Reload**  toolbar button to reload the Start page.

Related Topics

[Using the Graphical User Interface on page 657](#) | [Activity Wizards on page 663](#)

Output Window

The Output Window displays messages issued by product components or custom features.

The first tab, labelled **Build**, is the standard output for messages and errors. Other tabs are specific to the built-in features of the product or any user defined tool that you may have added.

To switch from one console window to another, click the corresponding tab. When any of the Output Window tabs receives a message, that tab is automatically activated.

When a console message contains a filename, double-click the line to open the file in the Text Editor. Similarly when a test report appears in the Output Window, double-click the line to view the report.

Output Window Actions

Right-click the Output Window to bring up a pop-up menu with the following options:

- **Edit Selected File:** Opens the editor with the currently selected filename.
- **Copy:** Copies the selection to the clipboard.
- **Clear Window:** Clears the contents of the Output Window.

To hide or show the Output Window, from the **View** menu, select **Other Windows** and **Output Window**.

Related Topics

[Project Explorer on page 1021](#) | [Using the Tools Menu on page 712](#)

Project Explorer

The Project Explorer allows you to navigate, construct and execute the components of your project. The Project Explorer organizes your workspace from two viewpoints:

- **Project Browser:** This tab displays your project as a tree view, as it is to be executed.
- **Asset Browser:** Source code and test script components are displayed on an object or elementary level.

To change views, select the corresponding tab in the lower section of the **Project Explorer** window.

Project Browser

The **Project Browser** displays the following hierarchy of nodes:

- **Projects:** the Project Explorer's root node. Each project can contain one or more sub-projects.
- **Results:** after execution, this node can be expanded to display the resulting report sub-nodes and files, allowing you to control those files through a CMS system such as Rational ClearCase.

- **Test groups:** provide a way to group and organize test or application nodes into one or more test campaigns
- **Test nodes:** these contain test scripts and source files:
- **Test Scripts:** for Component Testing or System Testing
- **Source files:** for code-under-test as well as additional source files
- Any other test related files
- **Application nodes:** represent your application, to which you can apply SCI instrumentation for Memory Profiling, Performance Profiling, Code Coverage and Runtime Tracing. Application nodes can also contain Contract Check scripts for C++.
- **Library nodes:** allow you to specify library files that can be used by any test or application node.
- **External Command nodes:** these allow you to add shell command lines at any point in the Test Campaign.

After execution of a test or application node, double-click the node to open all associated available reports.

When you run a **Build** command in the **Project Browser**, the product parses and executes each node from the inside-out and from top to bottom. This means that the contents of a parent node are executed in sequence before the actual parent node.

Asset Browser

The Asset Browser displays all the files contained in your project. The product parses the files and displays individual components of your source files and test scripts, such as classes, methods, procedures, functions, units and packages.

Use the Asset Browser to easily navigate through your source files and test scripts.

In Asset Browser, you can select the type of Asset Browser in the **Sort Method** box at the top of the **Project Explorer** window. Each view type can be more or less relevant depending on the programming language used:

- **By File:** This view displays a classic source file and dependency structure
- **By Object:** Primarily for C++, this view type presents objects and methods independently from the file structure
- **By Directory:** Displays packages and components

Use the **Sort ** button to activate or disable the alphabetical sort.

Double-click a node in the Asset Browser to open the source file or test script in the text editor at the corresponding line.

To switch Project Explorer views, click the **Project Browser** or **Asset Browser** tab.

To hide or show the Project Explorer, right-click an empty area within the toolbar, and then select or clear the Project Window menu item; or from the **View** menu, select **Other Windows** and **Project Window**.

Related Topics

[Report Explorer on page 1024](#) | [Discovering the GUI on page 1019](#)

Properties Window

The **Properties Window** box contains information about the node selected in the Project Explorer. It also allows you to modify this information. The information available in the Properties Window depends on the view selected in the Project Explorer:

- Project Browser
- Asset Browser

When relevant, the properties can use environment variables.

Project Browser

Depending on the node selected, any of the following relevant information may be displayed:

- **Name:** Is the name carried by the node in the Project Explorer.
- **Exclude from Build:** Excludes the node from the Build process. When this option is selected a cross is displayed next to the node in the **Project Explorer**.
- **Execute in background:** Enables the build and execution of more than one test or application node at the same time.
- **Relative path:** Indicates the relative path of the file.
- **Full path:** Indicates the entire path of the file.
- **Instrumented:** Indicates whether the source file is instrumented or not. You can select either **Yes** or **No**.

Note The **Instrumented** property is ignored for Component Testing for C++ if the **.otd** test script contains a **CHECK METHOD** statement or if an **.otc** contract check script is used. In these cases, the source files are always instrumented.

Asset Browser

Select the type of Object View in the **Sort Method** box at the top of the **Project Explorer** window: **By Object**, **By Files**, or **By Packages**. Depending on the sort method selected, and the type of object or file, any of the following relevant information may be displayed:

- **Name:** is the name carried of the file, object or package.
- **Filters (for folders):** is the file extension filter for files in that folder. See [Creating a Source File Folder on page 684](#).
- **Name:** is the name carried of the file or package.
- **Relative path:** indicates the relative path of the file.
- **Full path:** indicates the entire path of the file.

To open the Properties window, in the **Project Explorer**, right-click a node, and then select **Properties...** in the pop-up menu.

To hide or show the Properties window, right-click an empty area within the toolbar, and then select or clear the *<object>* **Property** menu item; or from the **View** menu, select **Other Windows** and *<object>* **Property**.

Related Topics

[Report Explorer on page 1024](#) | [Project Explorer on page 1021](#) | [Excluding a Node from a Build on page 700](#)

Report Explorer

The **Report Explorer** allows you to navigate through all text and graphical reports, including:

- Test reports generated by Component or System Testing
- Memory Profiling, Performance Profiling and Code Coverage reports
- UML Sequence Diagram reports from the Runtime Tracing feature
- Metrics produced by the Metrics Viewer

The actual appearance of the Report Explorer contents depends on the nature of the report that is currently displayed, but generally the Report Explorer offers a dynamic hierarchical view of the items encountered in the report.

Click an item in the Report Explorer to locate and select it in the **Report Viewer** or **UML/SD Viewer** window.

To hide or show the Report Explorer, right-click an empty area within the toolbar, and then select or clear the **Report Explorer** menu item.

Related Topics

[Using the Report Viewer on page 705](#) | [About the Code Coverage Viewer on page 349](#) | [Using the Memory Profiling Viewer on page 377](#) | [Using the Performance Profiling Viewer on page 391](#) | [Viewing Static Metrics on page 226](#)

Toolbars

The toolbars provide shortcut buttons for the most common tasks.

To hide or show a toolbar, right-click an empty area within the toolbar, select and clear those toolbars you want to display or hide, and then click **OK**; or from the **View** menu, select **Toolbars** and the toolbars that you want to display or hide.

Main Toolbar

The main toolbar is available at all times:

- The **New File**  button creates a new blank text file in the [Text Editor on page 693](#).
- The **Open**  button allows you to load any project, source file, test script or report file supported by the product.
- The **Save File**  button saves the contents of the current window.
- The **Save All**  button saves the current workspace as well as all open files.
- The **Cut** , **Copy**  and **Paste**  buttons provide the standard clipboard functionality.
- The **Undo**  and **Redo**  buttons allow you to undo or redo the last command.
- The **Find**  button allows you to locate a text string in the active Text Editor or report window.

View Toolbar

The View toolbar provides shortcut buttons for the Text Editor and report viewers.

- The **Choose zoom Level** box and the **Zoom In**  and **Zoom Out**  buttons are classic Zoom controls.
- The **Reload**  button refreshes the current report in the report viewer. This is useful when a new report has been generated.
- The **Reset Observation Traces**  button clears cumulative reports such as those from Code Coverage, Memory Profiling or Performance Profiling.

Build Toolbar

The build toolbar provides shortcut buttons to build and run the application or test.

- The **Configuration** box allows you to select the target [configuration on page 210](#) on which the test will be based.
- The **Build**  button launches the build and executes the node selected in the Project Explorer. You can configure the Build Options for the workspace by selecting the **Options**  button.

- The **Stop** button stops the build or execution.
- The **Clean Parent Node**  button removes files created by previous tests.
- The **Execute Node**  button executes the node selected in the Project Explorer.

Status Bar

The Status bar is located at the bottom of the main GUI window. It includes a **Build Clock** which displays execution time, and the **Green LED** which flashes when work is in progress.

Text Editor Toolbar

The text editor toolbar provides shortcut buttons for editing source files and test scripts. Some buttons may only be available when editing certain file types.

- The **Comment** (-- or //) button allows you to add the comment prefix for the corresponding language to the selected lines.
- The **Comment** (-- or //) button removes the comment prefix for the corresponding language.
- The **Add Test** (T) button adds a **TEST ... END TEST** statement block to a **.ptu** test script.
- The **Add Note**  button inserts the **_ATT_USER_NOTE** instrumentation pragma into your source code to produce notes in the UML sequence diagram of the execution.
- the **Insert Dump**  button inserts the **_ATCPQ_DUMP** instrumentation pragma into your source code to introduce a manual trace dump when required for runtime analysis tools.

Report Viewer Toolbar

The Report toolbar eases report navigation with the **Report Viewer**.

Report Viewer commands are available when a **Report Viewer** window is open:

- The **Previous Failed Test** and **Next Failed Test** buttons allow you to quickly navigate through the Failed items.
- The **Failed Tests Only** or **All Tests** button toggles between the two display modes.

Code Coverage Toolbar

The Code Coverage toolbar is useful for navigating through code coverage reports generated by the Code Coverage tool.

These buttons are available when the Code Coverage viewer is active.

- The **Previous Link** and **Next Link** buttons allow you to quickly navigate through the Failed items.
- The **Previous Uncovered Line** and **Next Uncovered Line** buttons allow you to quickly navigate through the Failed items.
- The **Failed Tests Only** or **All Tests** button toggles between the two display modes.
- The **F** button allows you to hide or show functions
- The **E** button allows you to hide or show function exits
- The **B** button allows you to hide or show statement blocks
- The **I** button allows you to hide or show implicit blocks
- The **L** button allows you to hide or show loops.

UML/SD Viewer Toolbar

The UML/SD Viewer toolbar provides shortcut buttons to commands related to viewing graphical test reports and UML sequence diagrams.

UML/SD Viewer commands are only available when a UML sequence diagram is open.

- The **Filter** button allows you to define a sequence diagram filter.
- The **Trigger** button sets sequence diagram triggers.

The following buttons are only available when using the Step-by-Step mode.

- The **Step** button moves the UML/SD Viewer to the next selected event.
- The **Select** button allows you to select the type of event to trace.
- The **Continue** button draws everything to the end of the trace diagram.
- The **Restart** button restarts Step-by-Step mode.
- The **Pause** button pauses the On-the-Fly display mode. The application continues to run.

The TDF file selector is only available when using the Split TDF File feature.

- Click the button to select a **.tdf** dynamic trace file from the list.
- Click the **Previous** and **Next** buttons to select the previous or next file in the list.

Test process monitor toolbar

The test process monitor (TPM) toolbar is useful for navigating through TPM charts.

These buttons are available when a TPM window is open:

- The **Clear** button removes all curves from the chart.
- The **Hide Event** button hides the displayed event markers.
- The **Floating Schedule** button toggles the automatic location of new curves.

Related Topics

[Report Explorer on page 1024](#) | [Start Page on page 1020](#) | [GUI elements on page 1019](#)

UML/SD Viewer Toolbar

The UML/SD Viewer toolbar provides shortcut buttons to commands related to viewing graphical test reports and UML sequence diagrams.

UML/SD Viewer commands are only available when a UML sequence diagram is open.

- The **Filter** button allows you to define a sequence diagram filter.
- The **Trigger** button sets sequence diagram triggers.

The following buttons are only available when using the Step-by-Step mode.

- The **Step** button moves the UML/SD Viewer to the next selected event.
- The **Select** button allows you to select the type of event to trace.
- The **Continue** button draws everything to the end of the trace diagram.
- The **Restart** button restarts Step-by Step mode.
- The **Pause** button pauses the On-the-Fly display mode. The application continues to run.

The TDF file selector is only available when using the Split TDF File feature.

- Click the button to select a **.tdf** dynamic trace file from the list.
- Click the and buttons to select the previous or next file in the list.

Related Topics

[Standard Toolbars on page 1025](#) | [About the UML/SD Viewer on page 402](#)

GUI macro variables

Some parts of the graphical user interface (GUI) allow you to specify command lines, such as in the Tools menu or in User Command nodes.

To enhance the usability of this feature, the product includes a macro language, allowing you to pass system and application variables to the command line.

Usage

Macro variables are preceded by **\$\$** (for example: **\$\$WSPNAME**).

Macro functions are preceded by **@@** (for example: **@@PROMPT**).

Environment variables are also accessible, and start with **\$** (for example: **\$DISPLAY**).

When specifying a command line, variables and functions are replaced with their value.

In Windows, when long filenames are involved, it is necessary to add double quotes (" ") around filename variables. For example:

```
"C:\Program Files\Internet Explorer\IEXPLORE.EXE" "$NODEPATH"
```

Node variables are context-sensitive: the variable returned relates to the node selected in the File or Test Browser.

Multiple selections are supported. If a node variable is invoked when there is no selection, no value is returned by the variables.

Macro variables and functions are case-insensitive. For clarity, they are represented in this document in upper case characters.

Language Reference

- Global variables: not node-related, include Workspace and application parameters.
- Node attribute variables: general attributes of a node.
- Functions: return a value to the command line after an action has been performed.

Global Variables

Global variables always return the same value throughout the Workspace.

Environment Variable	Description
----------------------	-------------

\$\$PRJ- NAME	Returns the name of the current .rtp Project file
\$\$PRJDIR	Returns the directory name of the current .rtp Project file
\$\$PRJ- PATH	Returns the absolute path of the current .rtp Project file
\$\$VCSDIR	Returns the local repository for files retrieved from Rational ClearCase, as specified in the ClearCase Preferences dialog box
\$\$CPPIN- CLUDES	Returns the directory of C and C++ include files, as specified in the Directories Preferences dialog box
\$\$PERL	Returns the full command-line to run the PERL interpreter included with the product. \$\$PERL allows to execute any perl script when its absolute or relative (from project) path is specified. If no path is specified, then the script will be searched in the following locations: <ol style="list-style-type: none">1. Current project path (\$\$PRJPATH)2. \$TESTRTDIR/lib/scripts Examples: To remove foo.txt and bar.obj from project directory by using the script perlrm.pl: \$\$PERL perlrm foo.txt bar.obj To copy file to target directory by using the script perlcp.pl: \$\$PERL perlcp file.txt
\$\$CLIP- BOARD	Returns the text content of the clipboard
\$ \$VCSITEMS	Returns a list of files managed by the CMS tool. If a single source file is selected, \$\$VCSITEMS returns the absolute path of the file. If a group or test node is selected, \$\$VCSITEMS returns a list of all files that must be registered in the configuration database (test script and reports).
\$\$OUTDIR	Returns the INTDIR perl value, as specified in the temporary directory, where the temporary files are created during the Build process are located.
\$\$REPORT- DIR	Returns the OUTDIR perl value, as specified in the report directory, where test and analysis results are created.

Node Attribute Variables

These variables represent the attributes of a selected node. If no node is selected, these variables return an empty string.

Environment Variable	Description
\$\$NODENAME	Returns the name of the node. In the case of files, this is the node's short filename
\$\$NODEPATH	Returns the absolute path and filename of the selected node
\$\$CFLAGS	Returns the compilation flags
\$\$LDLIBS	Returns the filenames of link definition libraries
\$\$LDFLAGS	Returns the flags used for link definition
\$\$ARGS	Returns all arguments sent to the command line
\$\$OUTDIR	Returns the name of the product features output directory
\$\$REPORTDIR	Returns name of the text report output directory
\$\$TARGETDIR	Returns the absolute path to the current Target Deployment Port
\$\$BINDIR	Returns the binary directory where the product is installed
\$\$OBJECTS	Returns a list of .o or .obj object files generated by the compiler
\$\$TIO	Returns the name of the current .tio trace file generated by Code Coverage
\$\$TSF	Returns the name of the current UML/SD .tsf static file generated by Runtime Tracing
\$\$TDF	Returns the name of the current UML/SD .tdf dynamic file generated by Runtime Tracing
\$\$TDC	Returns the name of the current Code Coverage .tdc correspondence file
\$\$ROD	Returns the name of the current .rod report file
\$\$FDC	Returns the name of the current .fdc correspondence files for Code Coverage

Functions

Functions process an input value and return a result. Input values are typically a global or node variable.

Environment Variable	Description
@@PROMPT(' <message>')	Opens a prompt dialog box, allowing the user to enter a line of text.

The optional <message> parameter allows you to define a prompt message, surrounded by single quotes (').

@@EDITOR(<filename>) Opens the product Text Editor.

@@OPEN(<filename>) Opens <filename>. <filename> must be a file type recognized by the product. This is the equivalent of selecting **Open** from the **File** menu.

File types

This table summarizes all the file types generated and used by HCL OneTest™ Embedded.

File Type	De- fault Ex- ten- sion	Generated By	Used By
Component Testing for C++ Declaration Files	.dcl	C++ Source code Parser on page 1131*	C++ Test Script Compiler on page 1130
Component Testing for Ada Intermediate File	.ddt	Ada Test Script Compiler on page 1140	Ada Test Report Generator on page 1145
Code Coverage Correspondence File	.fdc	Instrumented application (Code Coverage)	Code Coverage Report Generator on page 1101
System Testing Component Testing for Ada Intermediate File	.hts .mdt	Ada Test Script Compiler on page 1140	Ada Test Report Generator on page 1145
Static Metrics File	.met	C++ Source code Parser on page 1131 C Source Code Parser on page 1118 Ada Source Code Parser on page 1137	GUI Metrics Viewer

Component Testing for C++ Contract Check Script	.otc	C++ Source code Parser on page 1131*	C++ Test Script Compiler on page 1130
Component Testing for C++ Test Driver Script	.otd	C++ Source code Parser on page 1131*	C++ Test Script Compiler on page 1130
Component Testing for C++ Instrumentation File	.oti	C++ Test Script Compiler on page 1130	C and C++ Instrumentor on page 1075
Component Testing for C++ Intermediate File	.ots	C++ Test Script Compiler on page 1130	C++ Test Report Generator on page 1129
System Testing for C Test Script	.pts	User	System Testing Script Compiler on page 1112
Component Testing for C and Ada Test Script	.ptu	C Source Code Parser on page 1118*	C Test Script Compiler on page 1121
System Testing for C Result File	.rio	Test Driver	System Testing Report Generator on page 1109
Component Testing for C and Ada Result File		(System Testing for C) Test Driver (Component Testing for C and Ada)	C Test Report Generator on page 1126 Ada Test Report Generator on page 1145
Project File	.rtp	GUI	GUI
Project Model File	.rtpl	GUI	GUI
Workspace File	.rtw	GUI	GUI
Graphic Report	.rtx	C Test Report Generator on page 1126 Ada Test Report Generator on page 1145	GUI Report Viewer
System Testing for C Supervision Script	.spv	User (via CLI) or	System Testing for C Supervisor

Virtual Tester Deployment Wizard			
Target Output File	.spt	Target Deployment Port	GUI
Component Testing for C++ Stub Files	.stb	C++ Source Code Parser on page 1131*	C++ Test Script Compiler on page 1130
System Testing for C Intermediate File	.tdc	System Testing Script Compiler on page 1112 C Test Script Compiler on page 1121	System Testing Report Generator on page 1109
Component Testing for C and Ada Intermediate File		Ada Test Script Compiler on page 1140	C Test Report Generator on page 1126 Ada Test Report Generator on page 1145
UML/SD Dynamic Trace File	.tdf	Instrumented application (Runtime Tracing and Component Testing for C++)	GUI UML/SD Viewer
Code Coverage Intermediate File	.tio	Instrumented application (Code Coverage)	Code Coverage Report Generator on page 1101
Memory Profiling for C and C++ Dynamic Trace File	.tpf	Instrumented application (Memory Profiling)	GUI Memory Profiling Viewer
Performance Profiling Dynamic Trace File	.tqf	Instrumented application (Performance Profiling)	GUI Performance Profiling Viewer
Static Trace File	.tsf	C++ Test Script Compiler on page 1130 C and C++ Instrumentor on page 1075	GUI UML/SD Viewer
Target Deployment Port Customization File	.xdp	TDP Editor	TDP Editor
XML Report File	.xrd	C Test Report Generator on page 1126 Ada Test Report Generator on page 1145 C++ Test Report Generator on page 1129	GUI Report Viewer

[System Testing Report Generator on page 1109](#)

Report Viewer Summary File	.xtp	TestRTcc (eclipse)	TestRT eclipse viewer
			GUI Report Viewer

* Indicates files that are generated test script templates. Use these files to write your own test scripts.

Environment variables

Mandatory environment variables

The following environment variables **MUST** be set to run the product:

- **TESTRTDIR** for the graphical user interface
- **ATUDIR** for Component Testing for C and Ada
- **ATS_DIR** for System Testing for C
- **ATLTGT** in the command line interface

Environment variable list

Environment Variable	Description
TESTRTDIR	A mandatory environment variable that points to the installation directory of the product.
ATTOLSTUDIO_VERBOSE	Setting this variable to 1 forces the product GUI to display verbose messages, including file paths, in the Build Message Window.

Runtime Analysis features

The Runtime Analysis Features use the following environment variables:

Environment Variable	Description
----------------------	-------------

ATLTGT A mandatory environment variable that points to the Target Deployment Port directory when you are using the product in the command line interface.

When you are using the Instrumentation Launcher or the product GUI, you do not need to set **ATLTGT** manually, as it is calculated automatically.

ATL_TMP_DIR Indicates the location for temporary files. By default, they are placed in **/tmp** for UNIX or the current directory for Windows.

ATL_EXT_SRC This variable allows you to instrument additional files with filename extensions other than the defaults (**.c** and **.i**). The **.c** extension is reserved for C source files that require preprocessing, while **.i** is for already pre-processed files. All other extensions supported by this variable are assumed to be of source files that need to be preprocessed.

ATL_EXT_OBJ Lets you specify an alternative extension to **.o**(UNIX) or **.obj**(DOS) for object files.

ATL_EXT_ASM Lets you specify more than **.s** extension for assembler source files when the compiler offers an option to generate an assembler listing without compiling it to the object file.

ATL_EXT_TMP_CMD Windows only. Lets you specify an alternative extension to the Windows temporary options file. Defaults to **._@@**.

ATL_EXT_SRCCP The variable lets you add C++ source file extensions (defaults are **.C**, **.cpp**, **.c++**, **.cxx**, **.cc**, and **.i**) to specify the C++ source files to be instrumented. Extensions **.C** to **.cc** in the list are reserved for source files under analysis. The **.i** extension is reserved for those to be processed, if the **ATL_FORCE_CPLUSPLUS** variable is set to **ON**. Any other extension implies that pre-processing is to be performed.

ATL_FORCE_CPLUSPLUS If set to **ON**, this variable allows you to force C++ instrumentation whether the file extension is **.c**, **.i**, or any added extension.

Component Testing for C and Ada

Component Testing for C and Ada uses the following environment variables:

Environment Variable	Description
ATUDIR	Points to the /lib directory in the product installation directory.
ATUTGT	Points to the Target Deployment Port directory for Component Testing for C and Ada.

You can change default extensions for Component Testing for C and Ada through the use of environment variables when the Test Script Compiler or Test Report Generator is started.

The following table summarizes these environment variables and the extensions they modify.

Environment Variable	File	Default extension
ATTOLPTU	Test script	.ptu
ATTOLTDC	Table of correspondence file	.tdc
ATTOLLIS	List of errors	.lis
ATTOLRIO	Trace file	.rio
ATTOLRO	Test report	.ro
ATTOLROD	Unformatted test report	.rod
ATTOLDEF	Standard definitions file	.def
ATTOLSMB	Symbol table file	.smb

The rule whereby a "2" is added to the extension of the **.rio** trace file when the **-compare** option is used still applies if the default extension is changed in the **ATTOLRIO** environment variable.

System Testing for C

System Testing for C uses the following environment variable:

Environment Variable	Description
ATS_DIR	Points to the directory containing the System Testing binaries for C.

Test Process Monitor

The Test Process Monitor uses the following environment variables.

Environment Variable	Description
AT-TOL_TPM_ROOT	This variable indicates the directory where Test Process Monitor databases are located for a project. AT-TOL_TPM_ROOT is a mandatory variable and must be set when a project is created. It should be a shared directory accessible by all users who work on a project.
AT-TOL_TPM_USER	This optional variable specifies the name of the user. If this variable is not set, the Test Process Monitor uses the current user, if possible.

C and C++ Instrumentation Launcher

The Instrumentation Launcher uses the following additional variables:

Environment Variable	Description
ATTOL_BIN	If set, this variable must contain the path to the Instrumentor binaries. If not, this path is determined automatically from the PATH variable. This variable can be useful if the Target Deployment Port has been moved to a non-standard location.
AT-TOLOBJ	If set, this variable points to a valid directory where the products.h file is generated and the Target Deployment Port (TP.oorTDP.obj) is compiled. By default, these files are generated in the current directory.
ATL_OVER_SET	This variable must indicate the path to a copy of the BatchCCDefaults.pl file if you want to change any Target Deployment Port compilation flags contained in that file.
ATL_EX_T_LIB	Lets you specify additional alternative extensions for library files. By default .a or .lib are used.
ATL_FORCE_C_TDP	If set to ON , the tp.ini file is used instead of the tpcpp.ini file (used for C++ language). If the Target Deployment Port supports only C language, the tp.ini file is always used.

ATL_ - As an alternative to using the --settings of the Instrumentation Launcher, you can copy and modify the
 OVER_ - <InstallDir> /lib/scripts/**BatchCCDefaults.pl** file. In this case, set **ATL_OVER_SET** to the directory and file-
 SET name of the new copy of this file.

Ada tools

The Ada Link File Generator and Ada Unit Maker use the following additional variables:

Environment Variable	Description
----------------------	-------------

ATTOLCHOP	Selects the default naming convention. The following values can be used:
-----------	--

ATTOLCHOP="APEX" : for Apex naming

ATTOLCHOP="GNAT" : for Gnat naming

All other values end with a fatal error. By default, Gnat naming is used.

ATTOLALK_ - EXT	Specifies allowed extensions separated by the semicolon (;) character on UNIX systems and (;) on Windows.
--------------------	---

By default, the allowed extension list is ".ada;.ads;.adb"

ATTOLALK_ - NOEXT	Specifies forbidden extensions separated by the ':' character on UNIX systems and ';' on Windows.
----------------------	---

By default, the forbidden extension list is empty.

LD_ - LIBRARY_ - PATH	Specifies the location of libraries required by the Ada Link File Generator on page 1147 . By default, these libraries are located in the /lib directory of the installation directory.
-----------------------------	---

Related Topics

[Setting Environment Variables on page 702](#)

Target Deployment Port window

From HCL OneTest™ Embedded Studio, you can see the list of Target Deployment Ports installed on your platform and reload the list if any TDP had been updated.

To open the **Target Deployment Port** window in HCL OneTest™ Embedded Studio, click **Project > Target Deployment Port** in the main menu toolbar.

Installed TDP list

This panel displays the list of all Target Deployment Ports installed. To have more information on a TDP, select one of them in the list and click **Details**. The name of the TDP, its path, and other details show up in another pane. If any TDP has been modified, click **Reload** to refresh the list.

Runtime and static analysis reference

The command line interface allows you to integrate HCL OneTest™ Embedded runtime analysis tools into your build process.

To learn about	See
Using the command line tools for Runtime Analysis	Command line interface reference on page 1065
Inserting trace probes in your code	Trace probe macros on page 1040
Inserting instrumentation macro commands in your code	Instrumentation pragmas on page 1046

Related Topics

[Using the command line interface on page 962](#) | [Command line Runtime Analysis for C and C++ on page 965](#)

Trace probe macros

Trace probe macros

Trace Probes for C

Trace Probes macros allow you to manually instrument your source code under test to add message tracing capability to your test binary. This feature is only available for C.

Upon execution of the instrumented binary, the probes write trace information on the exchange of specified messages to the **.rio** output file.

Please refer to the section about Trace Probes for C in the User Guide for more information about using this feature.

Using Probe Macros

Before adding Trace Probe macros to your source code, add the following **#include** statement to each source file that will contain a probe:

```
#include "atlprobe.h"
```

The [atl_start_trace\(\)](#) on page 1041 and [atl_end_trace\(\)](#) on page 1045 macros must be called respectively when the application under test starts and terminates.

Other macros must be placed in your source code in locations that are relevant for the messages that you want to trace.

Trace Probe macros

- [atl_dump_trace\(\)](#) on page 1044
- [atl_end_trace\(\)](#) on page 1045
- [atl_recv_trace\(\)](#) on page 1042
- [atl_select_trace\(\)](#) on page 1043
- [atl_send_trace\(\)](#) on page 1043
- [atl_start_trace\(\)](#) on page 1041
- [atl_format_trace\(\)](#) on page 1046

atl_start_trace()

Trace Probes for C

Purpose

Initializes the environment of an instance trace. This macro must be executed before any other probe macro. Ideally, it can be placed at the start of the application.

Syntax

```
atl_start_trace( <handle> , "<path>", <instance>, <size> )
```

where:

- *<handle>* is the handle of the media storage and the handle of the result file (.rio). It relates to the instance name. This handle is used by other macros for all messages sent or received by this instance. This parameter must be in a valid variable name format and a non existing variable.
- *<path>* is the path to the .rio file to which the traces are to be written (with quotes ""). It can be used to open the intermediate binary file.
- *<instance>* is the logical name of the life line showing messages sent or received by the application instance. it could be the process/thread name or the layer name.
- *<size>* specifies the memory size used in bytes in FIFO or USER mode.

Example

```
int main(int argc, char** argv)
{
...
atl_start_trace(atl_client, "../res/", client, 0);
atl_start_trace(atl_serv, "../res/", serv, 0);
...
atl_end_trace(atl_client);
atl_end_trace(atl_serv);
...
}
```

Related Topics

[Probe macros on page 1040](#) | [atl_end_trace\(\) on page 1045](#)

atl_recv_trace()

Trace Probes for C

Purpose

Traces the reception of message.

Syntax

```
atl_recv_trace( <handle>, <dist>, <msg>, <type>, <msgname> )
```

where:

- *<handle>* is the handle linked to an instance.
- *<dist>* is the identifier of the emitter of a message.
- *<msg>* is the message address to trace.
- *<type>* is the message type as defined in the header files.
- *<msgname>* is the logical name of the message traced in the report.

Example

```
atl_rcv_trace(atl_client,f1,serv,t_cost,cost);
```

```
atl_send_trace
```

Related Topics

[Probe macros on page 1040](#) | [atl_send_trace\(\) on page 1043](#)

atl_select_trace()

Trace Probes for C

Purpose

Specifies for a given union type, the field to use for a message instance.

Syntax

```
atl_select_trace( <handle>, <idx>, <rank> )
```

where:

- *<handle>* is the handle linked to an instance.
- *<idx>* is a union type name.
- *<rank>* is the rank of the field used in the union type, starting at 0 for the first rank.

Example

```
atl_rcv_trace(atl_client,f1,serv,t_cost,cost);
```

```
atl_send_trace
```

Related Topics

[Probe macros on page 1040](#) | [atl_send_trace\(\) on page 1043](#)

atl_send_trace()

Trace Probes for C

Purpose

Traces a message sent.

Syntax

```
atl_send_trace( < ctx>, <dist>, <msg>, <type>, <msgname> )
```

where:

- *<handle>* is the handle linked to an instance.
- *<dist>* is the identifier of the receiver of a message.
- *<msg>* is the message identifier.
- *<type>* is the message type as defined in the **msg_type.h** file.
- *<msgname>* is the name of the message traced in the report.

Example

```
atl_send_trace(atl_client,f1,serv,t_cost,cost);
```

Related Topics

[Probe macros on page 1040](#) | [atl_start_trace\(\) on page 1041](#) | [atl_rcv_trace\(\) on page 1042](#)

atl_dump_trace()

Trace Probes for C

Purpose

Writes traces from the custom location to the .rio result file, when **FIFO**, **FILE** or **USER** buffer mode is selected in the Probe settings.

This macro is ignored in **DEFAULT** mode.

Syntax

```
atl_dump_trace()
```

Example

```
int main(int argc, char** argv)
{
...
atl_start_trace(atl_client, "../res/", client, 0);
atl_start_trace(atl_serv, "../res/", serv, 0);
...
atl_end_trace(atl_client);atl_end_trace(atl_serv);
```

```
atl_dump_trace();
```

```
...
```

```
}
```

Related Topics

[Probe macros on page 1040](#) | [atl_end_trace\(\) on page 1045](#)

atl_end_trace()

```
atl_end_trace()
```

Trace Probes for C

Purpose

Closes the trace environment of an instance. This macro must be executed before the application terminates.

Syntax

```
atl_end_trace(< ctx>)
```

where:

- *<handle>* is the handle linked to an instance.

Example

```
int main(int argc, char** argv)
```

```
{
```

```
...
```

```
atl_start_trace(atl_client,"client.rio",client,1000);
```

```
atl_start_trace(atl_serv, "serv.rio", serv, 2000);
```

```
...
```

```
atl_end_trace(atl_client);
```

```
atl_end_trace(atl_serv);
```

```
...
```

```
}
```

Related Topics

[Probe macros on page 1040](#) | [atl_start_trace\(\) on page 1041](#)

atl_format_trace()

Trace Probes for C

Description

This macro allows you to include a format file for the trace output.

Syntax

```
atl_format_trace(<file>)
```

where:

- *<file>* is the name of a format file, containing System Testing FORMAT instructions for C.

Example

```
int main(int argc, char** argv)
{
  ...
  atl_start_trace(atl_client, "client.rio", client, 1000);
  atl_start_trace(atl_serv, "serv.rio", serv, 2000);
  ...
  atl_format_trace("atl_format.hts");
  ...
  atl_end_trace(atl_client);
  atl_end_trace(atl_serv);
  ...
}
```

Related Topics

[Probe macros on page 1040](#) | [atl_start_trace\(\) on page 1041](#) | [FORMAT on page 854](#)

Instrumentation pragmas

The Runtime Tracing feature allows the user to add special directives to the source code under test, known as instrumentation pragma directives. When the source code is instrumented, the Instrumentor replaces instrumentation pragma directives with dedicated code.

Usage

```
#pragma attol <pragma name> <directive>
```

Example

```
int f ( int a )
{
```

```
#pragma attol att_insert if ( a == 0 ) _ATT_DUMP_STACK

return a;

}
```

This code will be replaced, after instrumentation, with the following line:

```
/*#pragma attol att_insert*/ if ( a == 0 ) _ATT_DUMP_STACK
```

Note Pragma directives are implemented only if the routine in which it is used is instrumented.

Instrumentation Pragma Names

```
#pragma attol insert <directive>
```

This code must be replaced with the following instrumentation if any of Code Coverage, Runtime Tracing, Memory Profiling or Performance Profiling is/are selected:

```
/*#pragma attol insert*/ <directive>
```

```
#pragma attol atc_insert <directive>
```

This code must be replaced with the following instrumentation if Code Coverage is selected:

```
/*#pragma attol atc_insert*/ <directive>
```

```
#pragma attol att_insert <directive>
```

This code must be replaced with the following instrumentation if Runtime Tracing is selected:

```
/*#pragma attol att_insert*/ <directive>
```

```
#pragma attol atp_insert <directive>
```

This code must be replaced with the following instrumentation if Memory Profiling is selected.

```
/*#pragma attol atp_insert*/ <directive>
```

```
#pragma attol atq_insert <directive>
```

This code must be replaced with the following instrumentation if Performance Profiling is selected.

```
/*#pragma attol atq_insert*/ <directive>
```

```
#pragma attol type_boolean= <myType>
```

For Code Coverage, this code declares the variable type <myType> as a Boolean for MC/DC coverage of bit-wise operations.

```
#pragma attol type_modifier= <keyword>
```

This pragma indicates a specific type modifier to the parser. For example:

```
#pragma attol type_modifier = __far
```

```
#pragma attol type_modifier = __pascal
```

will analyze silently :

```
int __pascal func ( int ) { /* ... */ }
```

```
char __far *pointer;
```

```
#pragma attol stop_analyze #pragma attol start_analyze
```

```
#pragma attol stop_instr #pragma attol start_instr
```

These pragmas can be used to start and stop analysis or instrumentation.

Stopping analysis also stops instrumentation. Starting instrumentation also starts analysis.

```
#pragma attol rename_local_var = <FuncName>:<localVarName>
```

This pragma allows the user to change the declaration name of the local variable named <localVarName> into the method <FuncName> in the source code. This instrumentation pragma directive can be used in a test to get access to the <localVarName> variable.

In the following example, the prama is used to change the name of the local declaration var 'tata' into the translate method. The following code that uses this variable will search for an external variable method instead of the local one.

```
#pragma attol rename_local_var = translate:tata
```

```
int Point::translate(void)
```

```
{
```

```
static int tata=4;
```

```
return tata++; //
```

tata is a local variable that cannot be modified or read outside the method

becomes

```
int Point::translate(void)
```

```
{
```

```
static int _atu_stub_tata=4;
```

```
return tata++; //
```

This 'tata' variable is now a global variable that can be created from the test driver script.

This functionality is available only if using attolcc4 in 8.0.1 release

```
#pragma attol cov_justify <directive>
```

This pragma allows the user to add a justification statement for non-coverage of a branch of code.

For more details on the directives, see [Justification of non-covered lines of code on page 58](#).

As this branch of code is covered, the code coverage percentage. The coverage report will highlight the branch in blue. The justification is displayed in the coverage viewer when you hover over this branch with your mouse.

If this branch is covered, an error will be reported in the coverage report.

Code Review Directives

In some cases, it can be useful to temporarily ignore a rule non-conformance on a short portion of source code, while providing a justification of why you are allowing the non-conformance.

```
#pragma attol crc_justify (<rule>[,<lines>], "<text>")
```

This macro justifies a deviation on the first non-conformance to a rule that follows the pragma, where:

<rule> is the name of the code review rule (for example: "Rule M8.5").

<lines> is the optional number of lines (including blank lines), after the current pragma line, that are covered by the deviation. The default value is 1 meaning that the deviation only applies to the next line. Specify **all** to apply the deviation to all lines until the end of the file.

<text> is the justification of why the deviation applies here.

```
#pragma attol crc_justify_all (<rule>,<lines>,"<text>")
```

This macro justifies all non-conformance instances to a rule that follows the pragma statement, where:

<rule> is the name of the code review rule (for example: "Rule M8.5").

<lines> is the number of lines (mandatory)

<text> is the justification of why the rule is ignored here.

```
#pragma attol crc_justify_everywhere (<rule>[,<lines>], "<text>")
```

This macro justifies all non-conformance instances to a rule for all source files in the current project, including traps located before the pragma statement, where:

<rule> is the name of the code review rule (for example: "Rule M8.5").

<lines> is an ignored parameter.

<text> is the justification of why the rule is ignored here.

The recommended usage for **crc_justify_everywhere** is to create a specific source file containing only the list of pragma statements and to add this file to the project.

Code Coverage, Memory Profiling and Performance Profiling Directives

The following macros must be used only with Memory Profiling and Performance Profiling.

_ATCPQ_DUMP(<reset>)

where <reset> can be one or more of the following values:

- **_ATCPQ_COV** to dump coverage results.
- **_ATCPQ_RESET_COV** to reset the coverage status after the dump.
- **_ATCPQ_QTF** to dump performance results.
- **_ATCPQ_RESET_QTF** to reset the performance status after the dump.
- **_ATCPQ_FREE_FRQ** to free all old memory blocks after the dump.
- **_ATCPQ_PRF** to dump memory profiling results.
- **_ATCPQ_CHK_WL** to dump ABWL and FMWL results.
- **_ATCPQ_ALL** to dump everything.

By default, **_ATCPQ_RESET** is set to **_ATCPQ_ALL** but can be redefined with a compilation command. Values of **0** or **1** are equivalent to **_ATCPQ_ALL**.

_ATCPQ_CLOSE closes the result file at the end of the requested dump so that an open action can be done on the next result dump using the runtime **priv_open** method.

_ATCPQ_DUMP can be automatically inserted by clicking the Insert Dump  button in the Text Editor toolbar.

_ATP_CHECK(@REFLINE)

This macro indicates a manual dump point in the source code for checking for ADWL or FMWL errors when using Memory Profiling. The corresponding setting must be selected.

The @REFLINE parameter is mandatory.

_ATP_TRACK(<pointer>)

This macro adds <pointer> to a list of selected memory blocks to check for ABWL or FMWL errors. A manual dump point in the source code for checking for ADWL or FMWL errors when using Memory Profiling.

Runtime Tracing Directives

When using this mode, the Target Deployment Package only sends messages related to an instance creation and destruction, or user notes. All other events are ignored. See Partial message dump for more information about this feature.

`_ATT_START_DUMP`

`_ATT_STOP_DUMP`

These directives enable and disable the partial message dump mode.

`_ATT_TOGGLE_DUMP`

This directive toggles the dump mode on and off. `_ATT_TOGGLE_DUMP` can be used instead of `_ATT_START_DUMP` and `_ATT_STOP_DUMP`.

`_ATT_DUMP_STACK`

When invoked, this directive dumps the contents of the call stack at that moment.

`_ATT_FLUSH_ITEMS`

When entered in Target Deployment Package buffer mode, this directive flushes the buffer. All buffered trace information is dumped. Flushing the buffer is useful before entering a time-critical phase of the trace.

`_ATT_USER_NOTE(<text>)`

This directive associates a text note to the function or method instance. `<text>` is a user-specified alphanumeric string containing the note text of type `char*`. The length of `<text>` must be within the maximum note length specified in the Runtime Tracing Settings dialog box. This pragma statement can be automatically inserted by clicking the **Add Note**  button in the Text Editor toolbar.

Generating SCI Dumps

By default, the system call `atexit()` or `on_exit()` invokes the Target Deployment Port (TDP) function that dumps the trace data. You can therefore instrument either all or a portion of the application as required.

When instrumenting embedded or specialized applications that never terminate, it is sometimes impractical to generate a dump on the `atexit()` or `on_exit()` functions. If you exit such applications unexpectedly, traces may not be generated.

In this case, you must either:

- Specify one or several explicit dump points in your source code, or
- Use an external signal to call a dump routine, or
- Produce an snapshot when a specific function is encountered.

Explicit Dump

Code Coverage, Memory Profiling and Performance Profiling allow you to explicitly invoke the TDP dump function by inserting a call to the `_ATCPQ_DUMP(1)` instrumentation pragma (the parameter 1 is ignored).

Explicit dumps should not be placed in the main loop of the application. The best location for an explicit dump call is in a secondary function, for example called by the user when sending a specific event to the application.

The explicit dump method is sometimes incompatible with *watchdog* constraints. If such incompatibilities occur, you must:

- Deactivate any hardware or software watchdog interruptions
- Acknowledge the watchdog during the dump process, by adding a specific call to the Data Retrieval customization point of the TDP.

You can automatically add an explicit dump your C and C++ source code by clicking the **Add Dump** button  in the text editor. This inserts the `_ATCPQ_DUMP` instrumentation pragma into your source code.

Dump on Signal

Code Coverage allows you to dump the traces at any point in the source code by using the `_ATC_SIGNAL_DUMP` environment variable.

When the signal specified by `_ATC_SIGNAL_DUMP` is received, the Target Deployment Port function dumps the trace data and resets the signal so that the same signal can be used to perform several trace dumps.

Before starting your tests, set `_ATC_SIGNAL_DUMP` to the number of the signal that is to trigger the trace dump.

The signal must be redirectable signal, such as `SIGUSR1` or `SIGINT` for example.

Instrumentor Snapshot

The Instrumentor snapshot option enables you to specify the functions of your application that will dump the trace information on entry, return or call.

In snapshot mode, the Runtime Tracing feature starts dumping messages only if the **Partial Message Dump** setting is activated. Code Coverage, Memory Profiling and Performance Profiling features all dump their internal trace data.

Frequency Dump

when all functions listed in `DUMPRETURNING`, `DUMPENTERING` and `DUMPCALLING` are executed too often by the application, a call divider number can be used to get a result dump less frequent than the functions call frequency, after multiple dump requests.

`_ATL_OBSTOOLS_DUMP_FREQ`: perl variable used in `envNode.pl`

- 0 is used to specified that the additional code is disabled.
- 1 is used to specified that the dump is made on each call of the methods listed in snapshot method lists.
- 10 is used to specified that the dump is made every 10 calls of the methods listed in snapshot method lists.

Related Topics

[General Runtime Analysis Settings on page 989](#) | [Instrumentation pragmas on page 1046](#)

Command line interface

This section contains advanced reference material for the general command tools, the runtime analysis command line interface, the C system testing command line interface and the component testing command line interface.

General command line tools

Graphical User Interface - studio

The HCL OneTest™ Embedded Graphical User Interface (GUI) is an integrated environment that provides access to all of the capabilities packaged with the product Studio.

Syntax

```
studio [<options>] [<filename>{ <filename>}]
```

```
studio<.jpt file><.txf file><.tpf file>
```

where:

- *<filename>* can be an **.rtp** project or **.rtw** workspace file, as well as source files (.c, .cpp, .h, .ada) or any report files that can be opened by the GUI, such as **.tdf**, **.tsf**, **.tpf**, **.tqf**, **.xrd** files.

Options

```
-r<node>
```

where *<node>* is a project node to be executed.

The **-r** option launches the GUI and automatically executes the specified node. Use the following syntax to indicate the path in the Project Explorer to the specified node:

```
<workspace_node>{[.<child_node>]}
```

Nodes in the path are separated by period ('.') symbols. If no node is specified, the GUI executes the entire project.

When using the **-r** option, an **.rtp** project file must be specified.

```
-html <directory>
```

where *<directory>* is an output directory for HTML reports.

When used with the **-r** option, the **-html** option directly outputs all reports in HTML format to the specified directory.

-config*<configuration>*

where *<configuration>* is a valid Configuration name.

The **-config** option allows you to specify a particular Configuration which is used when the studio GUI is opened.

When combined with the **-r** option, you can build and execute any particular node with any particular Configuration.

Example

The following command opens the **project.rtp** project file in the GUI, and runs the **app_2** node, located in **app_group_1** of **user_workspace**:

```
studio -r user_workspace.app_group_1.app_2 project.rtp
```

The following example opens a UML sequence diagram created by Runtime Tracing.

```
studio my_app.tsf my_app.tdf
```

Return Codes

After execution, the program exits with the following return codes

Code	Description
0	Execution of the project failed
1	Execution succeeded
5	Execution did not complete

These codes help you decide on a course of action after studio has finished running the test. You can obtain these return codes after execution with the following methods:

Windows:

```
studio -r -config test.rtp
```

```
echo "Build Result : " %ERRORLEVEL%
```

UNIX:

```
studio -r -config test.rtp
```

```
echo "Build Result : " $?
```

All messages are sent to the standard error output device.

Trace Receiver - trtpd

Purpose

The Trace Receiver is a graphical client that receives and splits trace dump data through a socket.

Syntax

trtpd [*<options>*] [*<file>* [, *<file>*]]

where:

- *<file>* is one or several dynamic trace output files
- *<options>* is a set of optional parameters

Description

If a set of user-defined I/O functions uses sockets in a customized Target Deployment Port (TDP), the Trace Receiver can be used to receive the dump data and to split the trace files on-the-fly. Use the Target Deployment Port Editor to customize the TDP.

The Trace Receiver uses its own graphical user interface to display reception and split progression.

To use the Trace Receiver, you must:

- Customize the TDP to produce trace buffer output through a socket by setting the `SOCKET_UPLOAD` setting of the TDP to *True*
- Specify a delimiter character in the `SOCKET_UPLOAD_DELIMITER` setting of the TDP

The Runtime Trace Receptor uses the delimiter to find useful trace data and directs the output to the trace file formats. If no filenames are provided, the following files are produced:

- **testing.rio** for Component Testing output to be processed by a Report Generator
- **purifylt.tpf** for Memory Profiling data
- **quantifylt.tqf** for Performance Profiling data
- **attolcov.tio** for Code Coverage data
- **tracer.tdf** for Runtime Tracing data

Options

-p|--port *<number>*

Port number. Specifies the decimal number of the port. The default port number is 7777.

-d|--delimiter <delimiter-byte>

Delimiter byte. Specified the decimal number of the delimiter byte. The default number is 94 ("^" in ASCII).

-o|--oneshot

Oneshot. Exits the Trace Receiver when the first client closes.

Example

The following trace dump is sent to the socket, using the "^" character as a delimiter:

...

^TU "ms"

SF 1 1dch 9527b66bh

TI 1 1 5

TM 726h

HS 95fch

ME 3 1

NI 6 1

SF 2 10edh 72cbacbh

TM b68h

HS bea4h

^ ...

Use the following command line to receive and split the trace dump into the correct output file formats.

```
trtpd -port 7778 --delimiter 95 -o c:\\temp\\coverage.tio
```

```
c:\\temp\\trace.tdf c:\\temp\\profiling.tqf
```

You can also launch the Trace Receiver with no parameters. In this case, default parameters are assumed:

```
trtpd
```

Related Topics

[Dump File Splitter on page 1057](#)

Dump File Splitter - atlsplit

Purpose

The dump file splitter (**atlsplit**) tool separates the unique multiplexed trace data file generated by the runtime analysis command line tools into specific trace files that can be processed by the runtime analysis and test feature Report Generators.

Syntax

atlsplit [*<options>*]*<trace_file>*

where:

1. *<trace_file>* is the name of the generated trace file (atlout.spt)

Description

The dump file splitter actually launches a *perl* script. You must therefore have a working perl interpreter such as the one provided with the product in the **/bin** directory.

Alternatively, you could use the following command line:

```
perl -I<install_dir>/lib/perl <install_dir>/lib/scripts/BatchSplit.pl atlout.spt
```

where *<install_dir>* is the installation directory of the product.

The script automatically detects which test or runtime analysis feature were used to generate the file and produces as many output files.

After the split, depending on the data contained in the trace file, the following files are generated:

- **.rio test result files:** process with [C Test Report Generator on page 1126](#), [Ada Test Report Generator on page 1126](#) or [System Testing Report Generator on page 1109](#)
- **.tio Code Coverage report files:** view with Code Coverage Viewer
- **.tdf Dynamic trace files:** view with UML/SD Viewer
- **.tpf Memory Profiling report files:** view with Memory Profiling Viewer
- **.tqf Performance Profiling report files:** view with Performance Profiling Viewer

Options

-verbose

Runs the program with verbose output.

-#

Runs the program with verbose output but does not split the trace data file.

-check

Verifies files before performing splitting the trace data file. Defective files are ignored.

-studio_log= <log-file>

This option is for internal usage only.

Uprint Localization Utility - uprint

Purpose

The Uprint is a utility that can help if you are experiencing localization issues with HCL OneTest™ Embedded.

Syntax

uprint

uprint <hex_min>..<hex_max>

uprint --mimename

uprint --utf8 <string>

where:

- <hex_min> and <hex_max> specify a range of 16-bit unicode characters expressed in hexadecimal notation.
- <string> is a character string encoded in the current locale.

Description

When used with no argument, **uprint** returns the following information about the current locale:

- Mib name
- mimeType
- Locale name

When used with a <hex_min>..<hex_max> argument, **uprint** also returns a list of locale-encoded characters from <hex_min> to <hex_max>.

When used with the **--utf8** option, **uprint** translates a specified locale-encoded *<string>* into a UTF-8 compliant backslashed hexadecimal string for use in C or C++ source code.

When used with the **--mimename** option, **uprint** returns the name of the Unicode Mime encoding.

Examples

The following command returns information about the current locale:

```
>uprint
```

```
Mib:111 mimeType:"ISO-8859-15" locale:"fr_FR@euro"
```

The following command translates the word "éric" into a UTF-8 compliant string:

```
>uprint --utf8 éric\xc3\xa9\xc3\xa9\xc3\xa9\xc3\xa9
```

Test Process Monitor - tpm_add

Purpose

Use the Test Process Monitor tool (**tpm_add**) to create and update Test Process Monitor databases from a command line.

Syntax

```
tpm_add -metric= <metric> [-file=<filename>] [-user=<user>] {<value_field>}
```

where:

- *<metric>* is the name of the metric.
- *<filename >* contains the name of the file under test to which the metric applies. This allows metrics for several files to be saved within the same database.
- *<user>* is the name of the product user who performed the measured value.
- *<value_field>* are the values attributed to each field

Description

The Test Process Monitor (TPM) provides an integrated monitoring feature that helps project managers and test engineers obtain a statistical analysis of the progress of their development effort.

Metrics generated by a test or runtime analysis feature are stored in their own database. Each database is actually a three-dimensional table containing:

- **Fields:** Each database contains a fixed number of fields. For example a typical Code Coverage database records.
- **Values:** Each field contains a series of values.
- **Filenames:** Values can be attributed to a filename, such as the name of the file under analysis. This way, the TPM Viewer can display result graphs for any single filename as well as for all files, allowing detailed statistical analysis.

Each field contains a set of values.

Note Although you specify a filename for the file under analysis, the TPM Viewer currently only displays a unique **FileID** number for each file.

The TPM database is made of two files that use the following naming convention:

`<metric> . <user> . <nb_fields>.idx`

`<metric> . <user> . <nb_fields>.tpm`

where `<nb_fields>` is the number of fields contained in the database.

In the GUI, the Test Process Monitor gathers the statistical data from these database file and generates a graphical chart based on each field.

There are 3 steps to using TPM:

- Creating a database for the metric
- Updating the database
- Viewing the results in the GUI

Creating a Database

Before opening the Test Process Monitor in the product, you must create a database.

Database files are created by using the **tpm_add** command line tool.

If you are using Code Coverage from the GUI, it automatically creates and updates a TPM code-coverage database.

If you are using the product in the command line interface you can invoke **tpm_add** from your own scripts.

To create a new metric database with `tpm_add`:

1. Type the following command:

```
tpm_add -metric=<name> -file=<filename> <value1>[ {<value2>... }]
```

where *<name>* is the name of the new metric and *<value>* represents the initial value of each field in the database. *<filename>* is the name of the source file to which these values are related.

Updating a Database

The Test Process Monitor adds a record to the database each time it encounters an existing database.

To add a new record to this database:

1. Type the **tpm_add** command:

```
tpm_add -metric=<name> <value1>[ <value2>... ]
```

where *<name>* is the name of the new metric and *<value>* represents the initial value of each field in the database. The number of values must be consistent with the number of fields in the table.

Note It is important to remain consistent and supply the correct number of fields for your database. If you run the **tpm_add** command on an existing metric, but with a different number of fields, the feature creates a new database.

```
tpm_add -metric=stats 5 -6 5.4 3 0
```

Viewing TPM Reports

Use the Test Process Monitor menu in the product to display database. Please refer to the User Guide for further information.

Examples

The following command creates a user metric called *stats*, made up of five fields, containing initial values **1**, **0.03**, **0**, **3** and **-4.7**.

```
tpm_add -metric=stats -file=/project/src/myapp.c 1 0.03 0 3 -4.7
```

The new database is contained in the following files:

```
stats.user.5.idx
```

```
stats.user.5.tpm
```

The following line adds a new record to the *stats* database, pertaining to the **myapp.c** source file:

```
tpm_add -metric=stats -file=/project/src/myapp.c 5 -6 5.4 3 0
```

The following line adds a new set of values to the *stats* database, this time related to the **mylib.c** source file:

```
tpm_add -metric=stats -file=/project/src/mylib.c 5 -6 5.4 3 0
```

The metrics related to **myapp.c** and **mylib.c** are stored in the same database and can be viewed either jointly or separately in the product Test Process Monitor Viewer.

If the following command is issued:

```
tpm_add -metric=stats -file=myapp.c 5 -6 3 0
```

A new database is created with four fields:

stats.user.4.idx

stats.user.4.tpm

TDP Generator - tdpngen

Purpose

Use the TDP Generator tool (**tdpngen**) to generate target deployment ports (TDP) from an .xdp file from a command line.

Syntax

```
tdpngen <XDP filename> <target directory>
```

where:

- <XDP filename> is the name of the .xdp target deployment port file.
- <target directory > is the name of the location where the TDP will be created.

Description

The purpose of this tool is to generate a TDP from a command line without using the TDP Editor.

Examples

The following command generates a TDP for GNU C++ in the targets directory.

```
tdpngen "%TESTRTDIR%\targets\xml\cpcgnu.xdp" "%TESTRTDIR%\targets"
```

Studio Report - studioreport

Purpose

Creates a temporary project with the test result files passed as parameter. These results can then be exported to HTML.

Syntax

studioreport [-help] [-html <dir>] [-keep] [-clean[All]] [-verbose] [<report files>]

where:

- <dir> is the name of the output directory for the HTML reports.
- <report files> is a list of file names separated by space characters. Only test result files are accepted.

Description

Report files must have the following extensions

- **.spt** global result file generated by the execution
- **.xrd** Component testing report file
- **.rtx** Graphical report file
- **.crc** Rule checker report file
- **.met** Static Metric report file
- **.fdc, .tio** Code coverage report files
- **.tsf, .tpf** Memory profiling report files
- **.tsf, .tqf** Performance profiling report files
- **.tsf, .tdf** Runtime tracing report file
- **.log, .xtp** file providing the listing of files like **attolccReport.xtp**

Files must have absolute paths or be relative to the current directory.

When no parameters are specified, **studioreport** looks for **attolccfiles.log**, **TestRTccfiles.log**, **TestRTccfiles.xtp**, or **attolccReport.xtp** in the current directory.

Note **attolcc** generates **attolccReport.xtp** after the application linkage.

Typing the **studioreport** command with the **.spt** file path, if it has not been generated locally, starts the viewer for the instrumented application.

Options

-help

Displays the help message

-html <dir>

Creates the directory for HTML exports

-keep

Keeps the temporary project that was generated

-clean

Removes all dynamic results, -cleanAll removes all report files

-verbose

Shows all files listed as studio parameters

Binary Version Lister - binList

Purpose

The Binary Version Lister is a utility that lists the versions of all the binaries of HCL OneTest™ Embedded.

Syntax

binList

Description

When invoked, the Binary Version Lister (**binList.sh** in UNIX and **binList.bat** in Windows) lists the versions of all the command line tools that are part of HCL OneTest™ Embedded.

ROD Converter - rod2xrd

Purpose

This command line tool converts a .rod file produced by the C or Ada Test Report Generator (attolpostpro and attolpostproada) into an .xrd file that can be viewed in HCL OneTest Embedded Studio.

Syntax

rod2xrd <.rod_file>

where <rod file> is the name of the .rod file to convert.

Options

-o <log-file>

This option allows you to specify the name of the output file. By default, the generated .xrd has the same name as the .rod file.

-h <header file>

This option allows you to specify the customized header file for the report. See REPORTHEADER in the TDP Editor help for more information.

-g

If the .ptu test script contains loops, this option generates graph data with the test results.

-s <max_nb_of_tests>

When large reports are generated, this option allows you to split the results into multiple report files that contain the specified number of tests.

Runtime Analysis command line interface reference

The command line interface allows you to integrate HCL OneTest™ Embedded runtime analysis tools into your build process.

To learn about	See
Using TestRealTime instrumentation in your standard C and C++ build procedure.	C and C++ Instrumentation Launcher - attolcc on page 1065
Instrumenting your C and C++ source code from the command line for runtime analysis and testing.	C and C++ Instrumentor - attolcc1, attolccp or attolcc4 on page 1075
Instrumenting your Ada source code from the command line for runtime analysis and testing.	Ada Instrumentor - attolada on page 1088
Producing static metrics of your Ada source files.	Ada Metrics Generator - metada on page 1099
Producing reports from a .tdf trace dump file.	TDF Splitter - attsplit on page 1100
Producing reports for Code Coverage.	Code Coverage Report Generator - attolcov on page 1101
Parsing code for trace probes.	Probe Code Parser - parsecode.pl on page 1104

Related Topics

[Runtime Analysis reference on page 1040](#) | [Using the command line interface on page 962](#) | [Command line Runtime Analysis for C and C++ on page 965](#)

C and C++ Instrumentation Launcher - attolcc

The Instrumentation Launcher instruments and compiles C and C++ source files. The Instrumentation Launcher is used by Memory Profiling, Performance Profiling, Runtime Tracing and Code Coverage, as well as the Component Testing Contract Check feature for C++.

Syntax

```
attolcc [{<-options>}] [{<-settings>}] -- <compilation_command>
```

```
attolcc --help
```

where:

- *<compilation_command>* is the standard compiler command line that you would use to launch the compiler if you are not using the product.
- "--" is the command separator preceded and followed by spaces.
- *<options>* is a series of optional parameters settings is a series of optional instrumentation settings.

Description

The Instrumentation Launcher fits into your compilation sequence with minimal changes.

The Instrumentation Launcher is suitable for use with only one compiler and only one Target Deployment Port. To view information about the driver, run **attolcc** with no parameters.

The **attolcc** binary is located in the **/cmd** directory of the Target Deployment Port.



Note: Some Target Deployment Ports do not have an **attolcc** binary. In this case, you must manually run the instrumentor, compiler and linker.

General Options

The Instrumentation Launcher accepts all command line parameters for either the [C or C++ Instrumentor on page 1075](#), including runtime analysis feature options. This allows the Instrumentation Launcher to automatically compile the selected Target Deployment Port.

In addition to Instrumentor parameters and Code Coverage parameters, the following options are specific to the Instrumentation Launcher. Command line options can be abbreviated to their shortest unambiguous number of characters and are not case-sensitive.

```
--HELP
```

Type **attolcc --help** to list a comprehensive list of options, including those of the C and C++ Instrumentor (attolccp or attolcc4, and attolcc1), for use with the instrumentation launcher.

```
-VERBOSE | -#
```

The **-VERBOSE** option shows commands and runs them. The **-#** option shows commands but does not execute them.

```
-TRACE
```

```
-MEMPRO
```

```
-PERFPRO
```

These options activate specific instrumentation for respectively the Runtime Tracing, Memory Profiling and Performance Profiling runtime analysis feature.

-OTIFILE=<file>[{,<file>}]

When using the Contract Check capability of Component Testing for C++, the **-OTIFILE** option allows you to specify one or several Component Testing **.oti** instrumentation files for C++. These files are generated by the C++ Test Script Compiler and contain the Component Testing instrumentation rules for C++.

-AUTO_OTI

When using the Contract Check capability of Component Testing for C++, this option specifies that Component Testing instrumentation files (**.oti**) for C++ are to be searched and loaded from the directory specified with option **-OTIDIR**, or in current directory if this option is not used. **.oti** files are searched according to the source file names. For instance, if class A is found in file **myfile.h**, the **.oti** searched will be **myfile.oti**. An information message is issued for each **.oti** file loaded automatically.\$

-FORCE_TDP_CC

This option forces the Instrumentation Launcher to attempt to compile the Target Deployment Port even if the link phase has not yet been reached before the **TP.o** or **TP.obj** is built.

-NOSTOP

This option forces the initial command to resume when a failure occurs during preprocessing, instrumentation, compilation or link. This means that the build chain is not interrupted by errors, but the resulting binary may not be fully instrumented. Use this option when debugging instrumentation issues on large projects.

Each error is logged in an **attolcc.log** file located in the directory where the error occurred.

Code Coverage Options

The following parameters are specific to the Code Coverage runtime analysis feature. These options do not activate Code Coverage. To activate Code Coverage, use the Code Coverage Level options (**-PROC**, **-CALL**, **-COND** and **-BLOCK**).

-PASS | **-COUNT** | **-COMPACT**

Pass mode only indicates whether a branch has been hit. The default setting is pass mode.

Count mode keeps track of the number of times each branch is exercised. The results shown in the code coverage report include the number of hits as well as the pass mode information.

Compact mode is equivalent to pass mode, but each branch is stored in one bit, instead of one byte as in pass mode. This reduces the overhead on data size.

-COMMENT | **-NOCOMMENT**

The comment option lets the user associate a comment string with the source in the code coverage reports and in Code Coverage Viewer.

By default, the Instrumentation Launcher sends the preprocessing command as a comment. This allows you to distinguish the source file that was preprocessed and compiled more than once with distinct options.

Use **-NOCOMMENT** to disable the comment setting.

```
-IGNORE=<filename>[ { , <filename> } ]
```

-IGNORE explicitly specifies the files that are to be ignored both by preprocessing and instrumentation, where *<filename>* is a C or C++ source file. All other source files are instrumented. Files that are ignored are not analyzed. Use this option to avoid errors that may occur with a file using the **-EXFILE** option.

<filename> may contain a path (absolute or relative from the current working directory). If no path is provided, the current working directory is used.

```
-NO_SYS_INCLUDE
```

Use this option if the application includes system files, for example: **windows.h** or **pthread.h**.

Metrics Options

```
-metrics=<output directory>
```

Generates static metrics for the specified source files in the specified *<output directory>*. This option replaces the **metcc** command line tool, which is deprecated.

```
-one_level_metrics
```

By default, the calculation of static metrics is applied to the specified source files, and extended to any files included in those source files. Use the **-one_level_metrics** option to ignore included files when calculating static metrics.

```
-restrict_dir_metrics<directory>
```

Use the the **-restrict_dir_metrics** option to calculate static metrics of the specified source files, extended to any files included in those source files but limited to those files located in the specified *<directory>*.

-studio_log

This option is for internal use only.

Instrumentation Settings

The instrumentation settings apply to the compilation of the Target Deployment Port Library.

The **0** or **1** values for many conditional settings mean false for 0 and 1 for true.

Compiler Settings

```
--cflags=<compilation flags>
```

```
--cppflags=<preprocessing flags>
```

```
--include_paths=<comma separated list of include paths>
```

```
--defines=<comma separated list of defines>
```

Enclose the flags with quotes ("") if you specify more than one. These flags are used while compiling the Target Deployment Port Library.

By default, the corresponding **DEFAULT_CPPFLAGS**, **DEFAULT_CFLAGS**, **DEFAULT_INCLUDE_PATHS** and **DEFAULT_DEFINES** from the <ATLTGT>/tp.ini or <ATLTGT>/tpcpp.ini file are used.

General Settings

```
--atl_multi_threads=0|1
```

To be set to 1 if your application is multi-threads (default **0**).

```
--atl_threads_max=<number>
```

Maximum number of threads at the same time (default **64**).

```
--atl_multi_process=0|1
```

To be set to 1 if your application uses fork/exec to run itself or another instrumented application (default **0**). Traces files are named atlout.<pid>.spt.

```
--atl_buffer_size=<bytes>
```

Size of the Dump Buffer in bytes (default **16384**).

```
--atl_traces_file=<file-name>
```

Name of the file that is flushed by execution and to be split (default **atlout.spt**).

Memory Profiling Settings

```
--atp_call_stack_size=<number of frames>
```

Number of functions from the stack associated to any tracked memory block or to any error (default **6**).

```
--atp_reports_fiu=0|1
```

File In Use detection and reporting (default **1**)

```
--atp_reports_sig=0|1
```

POSIX Signal detection and reporting (default **1**)

```
--atp_reports_miu=0|1
```

Memory In Use detection and reporting, ie: not leaked memory blocks (default 0).

```
--atp_reports_ffm_fmwl=0|1
```

Freeing Freed Memory and Late Detect Free Memory Write detection and reporting (default 1).

```
--atp_max_freeeq_length=<number of tracked memory blocks>
```

Free queue length, ie: maximum number of tracked memory blocks whom actual free is delayed (default 100).

```
--atp_max_freeeq_size=<bytes number>
```

Sets the free queue size, ie: the maximum number of bytes actually unfreed (default 1048576 = 1Mb)

```
--atp_reports_abwl=0|1
```

Late Detect Array Bounds Write detection and reporting (default 1).

```
--atp_red_zone_size=<bytes number>
```

Size of each of the two Red Zones placed before and after the user space of the tracked memory blocks (default 16).

```
--atp_dump_unfreed_only_with_stack
```

Use this option to only record memory leaks that are associated with a call stack. Memory allocations that occurred before the application started do not have a call stack and are not included in the Memory Profiling report.

```
--linenumoptim
```

By default, memory profiling reports the exact line number where the memory allocation statement is located, which requires extensive instrumentation and can cause performance issues. Use this option to reduce instrumentation overhead by reporting only the function in which the memory allocation occurs.

Performance Profiling Settings

```
--atq_dump_driver=0|1
```

Enable the Performance Profiling Dump Driver API atqapi.h (default 0).

Code Coverage Settings

```
--atc_dump_driver=0|1
```

Enables the Coverage Dump Driver API apiatc.h (default 0).

Runtime Tracing Settings

```
--att_on_the_fly=0|1
```

If set to 1, implies that each tdf lines are flushed immediatly in order to be read on-the-fly by the UML/SD Viewer in Studio (default 0).

```
--att_item_buffer=0|1
```

Enable Trace Buffer (not Dump Buffer) if set to 1 (default 0).

```
--att_item_buffer_size=<bytes>
```

Maximum number of recorded Trace elements before Trace Buffer flush (default 100).

```
--att_partial_dump=0|1
```

Partial Message Dump is on if set to 1 (default 0).

```
--att_signal_action=0|1|2
```

- 0 means no action when handling a signal (default)
- 1 means toggling dump of messages
- 2 means only flushing the current call stack

```
--att_record_max_stack=0|1
```

Display largest call stack length in a note (default 1).

```
--att_timestamp=0|1
```

If enabled, record and display time stamp (default 0).

```
--att_thread_info=0|1
```

If 1 record and display thread information (default 1).

Component Testing for C++ Contract Check Settings

```
--atk_stop_on_error=0|1
```

Call breakpoint function on assertion failure (default 0).

```
--atk_dump_success=0|1
```

By default (0), only failed assertions are reported. If enabled, both failed and passed assertions are reported.

```
--atk_report_reflexive_states=0|1
```

Trace unchanged states (default 1).

Example

```
attolcc -- cc -I../include -o appli appli.c bibli.c -lm
```

```
attolcc -TRACE -- cc -I../include -o appli appli.c bibli.c -lm
```

Return codes

The return code from the Instrumentation Launcher is either the first non-zero code received from one of the commands it has executed, or 0 if all commands ran successfully. Due to this, the Instrumentation Launcher is fully compatible with the make mechanism.

If an error occurs while the Instrumentation Launcher - or one of the commands it handles - is running, the following message is generated:

```
ERROR : Error during C pre-processing
```

All messages are sent to the standard error output device.

Command line to launch code coverage for assembler files

You can launch code coverage for assembler files with a command.

COMMAND AND OPTIONS

The command used to launch code coverage for assembler files is the following one:

```
attolgas <lst> <instr> <asmopt> [ options ]
```

The only supported modes instructions are **BLOCK**, **PROC** and **CALL**. The values that are supported and enabled for the **options** parameter are the following ones:

- [-PROC[=<RET>]]
- [-BLOCK[=<IMPLICIT|DECISION|LOGICAL>]]



Note: The parameters added after the option -BLOCK are not active.

- [-NOPROC]
- [-CALL|-NOCALL]
- [-NOBLOCK]
- [-NOINLINE]
- [-NOINCLUDE]
- [-COMPACT]
- [-COUNT]
- [-COMMENT=<text>]
- [-NOPATH]

- [-NOFULLPATHKEY]
- [-ALTCHECKSUM]
- [-NOCVI]
- [-NOSOURCE]
- [-NOINFO]
- [-MAIN=<service>]
- [-EXCALL=<file>]
- [-RENAME=<name>[,name]]
- [-REMOVE=<name>[,name]]
- [-NOINSTRDIR=<directory>[,directory]]
- [-FDCNAME=<file>]
- [-FDCCDIR[=<directory>]]
- [-NOWARNINGS]
- [-SHOWINFO]
- [-STUDIO_LOG[=<log-file>]] (for Studio only)

Other options values can be entered but there are not active:

- [-NOTERNARY]
- [-CALLMAYTERMINATE=<service>[,service]]
- [-WHILEONLY]
- [-CONSTLOOP]
- [-COND[=<MODIFIED|COMPOUND|FORCEEVALUATION|FORCEBITWISE|BITWISE>]]
- [-NOCOND]
- [-CONDEXPRESSION]
- [-KEEP]
- [-METRICS|-NOMETRICS]
- [-DUMPCALLING=<service>[,service]]
- [-DUMPINCOMING=<service>[,service]]
- [-DUMPRETURNING=<service>[,service]]
- [-FILE=<file>[,file]]
- [-EXFILE=<file>[,file]]

- [-UNIT=<name>[,name]]
- [-EXUNIT=<name>[,name]]
- [-SERVICE=<name>[,name]]
- [-EXSERVICE=<name>[,name]]
- [-INSTANTIATIONMODE=<USED|ALL>]
- [-PCHMODE[=<CREATE|USE>]]
- [-OTIFILE=<file>[,file]]
- [-BODY=<MAP_FILE|NAME_CONV|INLINE>]
- [-AUTO_OTI]
- [-OTIDIR[=<directory>]]
- [-NO_OTC]
- [-NO_OTD]
- [-MAPFILE[=<file>]]
- [-INSTR_CONST]
- [-MTSUPPORT]
- [-FRIEND_TEST_CLASS[=<name>][,name]]
- [-TRACE|-NOTRACE]
- [-NO_DATA_TRACE]
- [-NO_UNNAMED_TRACE]
- [-NO_TEMPLATE_NOTE]
- [-MEMPRO]
- [-PURIFYLT]
- [-NOINSPECT[=<variable>][,variable]]
- [-NOINSPECTDIR=<directory>[,directory]]
- [-PERFPRO[=<OS|PROCESS>]]
- [-QUANTIFYLT[=<OS|PROCESS>]]
- [-TSFNAME=<file>]

- [-TSFDIR[=<directory>]]
- [-NODLINE]
- [-SET=<name>[,name]]
- [-APPSTACK]
- [-CCOUPLING]
- [-STACKSIZE]
- [-FEOPT=<name>[,name]]
- [-C99]
- [-BEFORE_RETURN_EXPR]

C and C++ Instrumentor

C/C++ Instrumentor - attolcc1, attolccp and attolcc4

Purpose

The Instrumentor for C and C++ inserts functions from a Target Deployment Port library into the C or C++ source code under test. The C/C++ Instrumentor is used for:

- Memory Profiling
- Performance Profiling
- Code Coverage
- Runtime Tracing



Note: The **attolccp** or **attolcc4** binary replaces the **attolcc1** binary, which is still provided for legacy projects and does not support C++ source files. The **attolccp** or **attolcc4** binary instruments both C and C++ source files and should be preferred for all new projects. If you want to use the legacy **attolcc1** binary on new projects, you must edit the Target Deployment Port Basic Settings and define the **USE_ATTOLCC1** setting.

Syntax

```
attolccp or attolcc4 <src> <instr> <def> <opp> [{ <-options> }]
attolccp or attolcc4 <src> <instr> <hpp> <opp> [{ <-options> }]
attolcc1 <src> <instr> <def> [{ <-options> }]
```

Where:

- `<src>` is the preprocessed source file (input)
- `<instr>` is the instrumented file (output)
- `<def>` is the standard definition file **atus_c.def** for C.
- `<hpp>` is the standard definition file **atl.hpp** for C++
- `<opp>` is the parser options file **atl.opp** for C and C++.

The usage of either the **atus_c.def** or the **atl.hpp** file defines whether the Instrumentor runs in C or C++ mode.

The `<src>` input file must be preprocessed beforehand with macro definitions expanded, include files included, **#if** and directives processed.

The instrumentor expects **atus_c.def**, **atl.hpp**, and **atl.opp** files to be located in the \$ATLTGT directory.

When using the Instrumentor in C language mode, all arguments are functions. When using the Instrumentor in C++ mode, arguments are qualified functions, methods, classes, and namespaces, for example: **void C::B::f(int)**.

Description

The C/C++ Instrumentor builds an output source file from an input source file, by adding special calls to the Target Deployment Port function definitions.

The Runtime Analysis tools are activated by selecting the command line options:

- **-MEMPRO** for Memory Profiling
- **-PERFPRO** for Performance Profiling
- **-TRACE** for Runtime Tracing
- **-PROC**, **-CALL**, **-COND** and **-BLOCK** for Code Coverage (code coverage levels). For more information about compliance with code coverage levels, see 'How do Code Coverage levels match DO-178B coverage levels' technote.



Note: there is no **-COVERAGE** option; the following rules apply for the Code Coverage feature:

- If no code coverage level is specified, nor Runtime Tracing, Memory Profiling, or Performance Profiling or C++ Component Testing Contract Check, then the default is to have code coverage analysis at the **-PROC** and **-BLOCK=DECISION** level.
- If no code coverage level is specified while one or more of the aforementioned features are selected, then code coverage analysis is not performed.

Detailed information about command line options for each feature are available in the sections below.

The C/C++ Instrumentor **attolccp** or **attolcc4** supports preprocessed C files ANSI 89, ANSI 99, or K&R C standard source code, and preprocessed C++ files compliant with the ISO/IEC 14882:1998 standard. Depending on the Target Deployment Port, **attolccp** or **attolcc4** can also accept the C ISO/IEC 9899:1990 standard, the ISO/IEC 9899:1999 (C99) standard, and other C or C++ dialects.

The legacy C Instrumentor (**attolcc1**) supports preprocessed C files ANSI 89, ANSI 99, or K&R C standard source code.

For C99 support, you must use the **-C99** option.

The C/C++ Instrumentor accepts either C or C++-style comments.

Attol pragmas start with the # character in the first column and end at the next line break.

The `<def>` and `<header>` parameters must not contain absolute or relative paths. The Code Coverage Instrumentor looks for these files in the directory specified by the **ATLTGT** environment variable, which must be set.

You can select one or more types of coverage at the instrumentation stage.

When you generate reports, results from some or all of the subset of selected coverage types are available.

General options

-FILE= `<filename>[{,<filename>}]` | **-EXFILE=** `<filename>[{,<filename>}]`

-FILE specifies the only files that are to be explicitly instrumented, where `<filename>` is a C or C++ source file. All other source files are ignored. Use this option with multiple C or C++ files that can be found in a preprocessed file (`#includes` of files containing the bodies of C or C++ functions, `lex` and `yacc` outputs, and so forth).

-EXFILE explicitly specifies the files that are to be excluded from the instrumentation, where `<filename>` is a C or C++ source file. All other source files are instrumented. You cannot use this option with the option **-FILE**. Files that are excluded from the instrumentation process are still analyzed. Any errors found in those files are still reported.

`<filename>` may contain a path (absolute or relative from the current working directory). If no path is provided, the current working directory is used.

-FILE and **-EXFILE** cannot be used together.

-UNIT= `<name>[{,<name>}]` | **-EXUNIT=** `<name>[{,<name>}]`

-UNIT specifies code units (functions, procedures, classes or methods) whose bodies are to be instrumented, where `<name>` is a unit which is to be explicitly instrumented. All other functions are ignored.

-EXUNIT specifies the units that are to be excluded from the instrumentation. All other units are instrumented. Functions, procedures, classes or methods that are excluded from the instrumentation process with the **-EXUNIT** option are still analyzed. Any errors found in those units are still reported.

If *<name>* contains commas (","), these must be preceded by a backslash character. For example: "\,"

-UNIT and **-EXUNIT** cannot be used together.

Note These options replace the **-SERVICE** and **-EXSERVICE** options from previous releases of the product.

In C++, if a method is defined in several files, you can specify *<name>* by preceding the method name with a filename, separated by a dot ("."). For example: **-EXUNIT=class.cpp.method**. If the filename does not contain an extension, then the option will apply to all files that use the base filename. For example: **-UNIT=class.method** instruments **method** from **class.cpp** and **class.h**. The *<name>* parameter cannot contain directory paths.

-MAIN= *<service>*

Specifies that the return of the main function, which is identified as *<service>*, will be instrumented to dump the complete results. This is useful in cases where the main entry is not called "main".

-RENAME= *<function>* [, *<function>*]

For C only. The **-RENAME** option allows you to change the name of C functions *<function>* defined in the file to be instrumented. Doing so, the *f* function will be changed to **_atu_stub_ f**. Only definitions are changed, not declarations (prototypes) or calls. Component Testing for C can then define stubs to some functions inside the source file under test.

If you used the **-RENAME** option of the C Test Script Compiler (attolpreproC), then you can pass the stub renaming information contained in the generated file with the syntax **attolccp @** or **attolcc4 <filename>** .

-REMOVE= *<name>* [, *<name>*]

This option removes the definition of the function (or method) *<name>* in the instrumented source code. This allows you to replace one or several functions (or methods) with specialized custom functions (or methods) from the TDP.

-NOFULLPATHKEY

Identifies source files based only on the filename instead of the complete path. Use this option to consolidate test results when a same file can be located in different paths. This can be useful in some multi-user environments that use source control. If you use this option, make sure that the source file names used by your application are unique.

-ALTCHECKSUM

Use this option to calculate a more unambiguous checksum for .fdc and .tsf files. Before using this option, you must delete existing fdc and tsf files, which will be re-created with the new checksum. File keys are not changed by this option.

-NOWARNING

This option deactivates the warning display for signature analysis. The Instrumentor's signature analyzer accepts any non-ambiguous signature, and more permissive than most compilers. Warnings indicates signatures that which are accepted by the instrumentor, but would be rejected by compilers.

-NO_DATA_TRACE

For C++ only. Excludes from instrumentation structures or classes that do not contain methods. This reduces instrumentation overhead.

-NOINSTRDIR= <directory>[, <directory>]

Specifies that any C or C++ function found in a file in any of the <directories> or a sub-directory are not instrumented.

Note You can also use the **attol incl_std** pragma with the same effect in the standard definitions file.

-NOINSPECTDIR= <directory>[, <directory>]

Specifies directories excluded from inspection of variables found in include files. Use this option to avoid the inspection of variables from 3rd party libraries.

-INSTANTIATIONMODE=ALL

For C++ only. When set to **ALL**, this option enables instantiation of unused methods in template classes. By default, these methods are not instantiated by the Instrumentor.

-DUMPCALLING= <name>[{,<name>}]

-DUMPINCOMING=[<class> ::]<name>[{,<class> ::]<name>}]

-DUMPRETURNING= <name>[{,<name>}]

In some cases, such as with applications that never terminate or when working with timing or memory-sensitive targets, you might need to dump traces at specific points in your code. These options allow you to explicitly define upon which incoming, returning or calling functions the trace dump must be performed.

- **-DUMPINCOMING:** Allows you to specify a list of function names, from your source code, that will dump traces at the beginning of the function.
- **-DUMPRETURNING** (for C and C++ only): Allows you to specify a list of function names, from your source code, that will dump traces at the end of the function. In C++, you can use the following syntax to specify a method within a class: `-dumpreturning=class::name`
- **-DUMPCALLING:** Allows you to specify a list of function names, from your source code, that will dump traces before the function is called.

See [Generating SCI Dumps on page 1051](#) for more information.

-NOPATH

Disables generation of the path to the Target Deployment Package directory in the `#include` directive. This lets you instrument and compile on different computers.

-NOINFO

Prohibits the Instrumentor from generating the identification header. This header is normally written at the beginning of the instrumented file, to strictly identify the instrument used.

-NODLINE

Prohibits the Instrumentor from generating `#line` statements which are not supported by all compilers. Use this option if you are using such a compiler.

-TSFDIR[= <directory>]

Not applicable to Code Coverage (see **FDCDIR**). Specifies the destination `<directory>` for the `.tsf` static trace file which is generated following instrumentation for each source code file. If `<directory>` is not specified, each `.fdc` file is generated in the corresponding source file's directory. If you do not use this option, the `.tsf` directory is the current working directory (the **attolcc1** or **attolccp** or **attolcc4** execution directory). You cannot use this option with the **-TSFNAME** option.

-TSFNAME= <name>

Not applicable to Code Coverage (see **FDCNAME**). Specifies the `.tsf` file name `<name>` to receive the static traces produced by the instrumentation. You cannot use this option with the **-TSFDIR** option.

-NOINCLUDE

This option excludes all included files from the instrumentation process. Use this option if there are too many excluded files to use the **-EXFILE** option.

-C99

This option enables support for the C99 specification (ISO/IEC 9899:1999).

Code coverage options

The following parameters are specific to the Code Coverage runtime analysis feature.

-PROC[=RET]

This option enables coverage of procedure inputs (C/C++ functions). This is the default setting.

The **-PROC=RET** option enables coverage of procedure inputs, outputs, and terminal instructions.

The **-NOPROC** option specifies that procedure coverage is disabled.

The **-BLOCK=IMPLICIT | DECISION | LOGICAL** option enables coverage of simple blocks only.

The **IMPLICIT** or **DECISION** option (these are equivalent) enables coverage of implicit blocks (unwritten else instructions), as well as simple blocks.

The **LOGICAL** option enables coverage of logical blocks (loops), as well as simple and implicit blocks.

By default, the Instrumentor instruments implicit blocks.

The **-NOBLOCK** option specifies that block coverage is disabled.

The **-CALL** option enables coverage of C or C++ function calls.

The **-EXCALL=<filename>** option applies to C language only. It excludes from coverage all calls to the C functions whose names are listed in <filename>. The names of functions (identifiers) must be separated by space characters, tab characters, or line breaks. No other types of separator can be used.

The **-NOCALL** option specifies that call coverage is disabled.

The **-COND[=MODIFIED | =COMPOUND | =FORCEEVALUATION]** option specifies the level of condition coverage. When **-COND** is used with no parameter, Code Coverage enables coverage of basic conditions.

The **MODIFIED** option enables coverage of modified conditions.

The **COMPOUND** option enables coverage of multiple (or compound) conditions.

The **FORCEEVALUATION** option enables coverage of forced conditions. This includes coverage of modified conditions.

The **-NOCOND** option specifies that condition coverage is disabled.

The **-CONDEXPRESSION** option causes relational operators in an expression (for example: $y = (a > 0)$) to be considered as conditions.

The **-COUNT** option specifies count mode.

The **-COMPACT** option specifies compact mode.

The **-FDCDIR= <directory>** option specifies the destination <directory> for the **.fdc** static correspondence file, which is generated for Code Coverage after the instrumentation for each source file. Correspondence files contain static information about each enumerated branch and are used as inputs to the Code Coverage Report Generator. If <directory> is not specified, each **.fdc** file is generated in the directory of the corresponding source file. If you do not use this option, the default **.fdc** files directory is the working directory (the **attolcc** execution directory). You cannot use this option with the **-FDCNAME** option.

With the **-FDCNAME= <name>** option, by default, the instrumentor generates one **.fdc** static correspondence file for each source file involved in the code to be instrumented. Use this option to specify a single static file for all source files in order to avoid file access conflicts, for example when a parallel build is involved. When this option is specified, the generated **.fdc** file contains one FDC section per source file. You cannot use this option with the **-FDCDIR** option.

-NOCVI: Disables generation of a Code Coverage report that can be displayed in the Code Coverage Viewer.

-METRICS: Provides static metric data for compatibility with old versions of the product. Use the static metrics features of the Test Script Compiler tools instead. By default no static metrics are produced by the Instrumentors.

-NOSOURCE: Replaces the generation of the colorized viewer source listing by a colorized viewer pre-annotated report containing line number references.

-COMMENT= *<comment>*: Associates the text from either the Instrumentation Launcher (preprocessing command line) or from the source file under analysis and stores it in the **.fdc** correspondence file to be mentioned in Code Coverage reports. In the Code Coverage Viewer, a magnifying glass appears next to the source file, allowing you to display the comments in a separate window. The comment text must not contain commas or non-alphanumeric characters.

-NOTERNARY: Specifies that ternary statements are not instrumented.

-CALLMAYTERMINATE= *<service>*[*<service>*]: This option specifies a list of functions that may not return.

-WHILEONLY: This option specifies that *for* loops are not instrumented as loops.

Memory Profiling Specific Options

The following parameters are specific to the Memory Profiling runtime analysis feature.

-MEMPRO: Enables the memory profiling feature.

-NOINSPECT= *<variable>*[*<variable>*]: Specifies global variables that are not to be inspected for memory leaks. This option can be useful to save time and instrumentation overhead on trusted code.

Performance Profiling Specific Options

The following parameters are specific to the Performance Profiling runtime analysis feature.

-PERFPRO[=*<os>***|***<process>***]**: Enables the performance profiling feature.

The optional *<os>* parameter allows you to specify a clock type. By default the standard operating system clock is used.

The *<process>* parameter specifies the total CPU time used by the process.

The *<os>* and *<process>* options depend on target availability.

Runtime Tracing Specific Options

The following parameters are specific to the Runtime Tracing analysis feature.

-TRACE: Enables the Runtime Tracing analysis feature.

-NO_UNNAMED_TRACE: For C++ only. With this option, unnamed *structs* and *unions* are not instrumented.

-NO_TEMPLATE_NOTE: For C++ only. With this option, the UML/SD Viewer will not display notes for template instances for each template class instance.

-BEFORE_RETURN_EXPR: For C only. With this option, the UML/SD Viewer displays calls located in return expressions as if they were executed sequentially and not in a nested manner.

Component Testing Options for C++

The following parameters are specific to Component Testing for C++.

-OTIFILE= *<filename>*{, *<filename>*}: Name of one or several Component Testing **.oti** instrumentation files for C++. These files contain rules for Component Testing instrumentation for C++ (they are generated by the C++ Test Script Compiler).

-AUTO_OTI: If this option is specified, Component Testing **.oti** instrumentation files for C++ will be searched and loaded in the directory specified with option **-OTIDIR**, or in current directory if this option is not used. **.oti** files are searched according to the source file names. For instance, if class **A** is found in file **myfile.h**, the **.oti** searched will be **myfile.oti**. An information message is issued for each **.oti** file loaded automatically.

-OTIDIR=*<directory>*: This option specifies, when option **-AUTO_OTI** is active, which directory is to be searched. If no directory is specified (i.e. **-OTIDIR=** is specified), **.oti** files will be searched in the same directory as the source file they are matching.

-FRIEND_TEST_CLASS: Use this option if you want the test to access any private or protected members (friend classes) of the components under test. The class must be mentioned in the OTC file to be recognized as a friend of the test class.

-BODY=MAP_FILE|NAME_CONV|INLINE: This option specifies where generated methods body should be generated.

Use **INLINE** to generate method bodies in each instrumented source file as inline routines. This is the default, since there is little chance that the generated code cannot be accepted by a compiler, except with template classes on some compilers.

Use **NAME_CONV** to generate routine bodies in the **.cpp**, **.cc** or **.C** file whose name matches the **.h** file that contains the class definition of the generated method.

Use **MAP_FILE** when you provide a map file with the option **-MAPFILE**. This generates method bodies according to the map file.

-MAPFILE= *<filename>* : When you add the **-BODY=MAP_FILE** option, this option must be provided. The **-MAPFILE** option specifies a user-created map file, describing where the methods of each class are to be generated.

This file must have the following format:

```
<source file>
  <class name>
  <class name>
  ...
<source file>
```

```
<class name>
...
...
```



Note: The character before a class name MUST be a tabulation.

Example

```
a.cpp
A
b.cpp
B
```

This specifies that class **A** methods bodies have to be generated in file **a.cpp**, and **B** methods bodies have to be generated in file **b.cpp**.

The options **-NO_OTC** and **-NO_OTD** specify that Component Testing instrumentation rules for C++ issued from, respectively, an **.otc** contract check test script, or an **.otd** test driver script should be ignored.

The option **-SHOWINFO** activates a diagnosis for each signature analysis. Usually, analysis of ill-formed signatures is silent. This option allows you to find ignored signatures



Note: A signature is a string describing a class, a method, or a function, and is used in **.otc** and **.otd** files.

-INSTR_CONST: Usually a C++ *const* method cannot modify any field of the *this* object. That's why the *const* methods are not checked for state changes, and are only evaluated once for invariants. But in some cases, the *this* object may change even if the method is qualified with *const* (by assembler code or by calling another method with casting the *this* parameter to a *non-const* type).

There may also be pointers fields to objects which logically belong to the object, but the C++ compiler does not guarantee that these pointed sub-objects are not modified. Use this option if the source code contains such pointers.

-MTSUPPORT: Use this option if your application is multi-threaded and objects are shared by several threads. This will ensure the specificity of each object for state evaluation.



Note: To use multi-thread support in the product, you must also compile the Target Deployment Port with multi-thread support.

-STUDIO_LOG: This option is for internal usage only.

Return codes

After the test execution, the program exits with the following return codes:

Code	Description
0	End of execution with no errors

- 7 End of execution because of fatal error
- 9 End of execution because of internal error

All messages are sent to the standard error output device.

C Code Review Compiler - crccc

Purpose

The C Code Review Compiler compiles C source files for code review static analysis. It produces an .xob output file, which must be linked by using the C Code Review Linker crclid.

Syntax

crccc <source> <xob> <def> <opp> [<options>]

where:

- <source> is the C source file under analysis
- <xob> is the name of the generated object file
- <def> is the standard definitions file. This file is searched in the \$ATLTGT directory.
- <opp> is the parser options file. This file is searched in the \$ATLTGT directory.

Description

Analyses the code and produces an .xob object file for use with the C Code Review Linker (crclid).

Options

Command line options can be abbreviated to their shortest unambiguous number of characters and are not case-sensitive.

[-RULE= <file>]

Specifies the code review rule configuration file. By default, a default internal configuration rule set is used

[-INCL= <directory>{[,<directory>}]

Specifies the locations of included files.

[-STD_INCL= <directory>[,<directory>]]

Specifies the location of additional system include directories.

[-CHECKSYSINCLUDE]

By default, files from system include directories are not analyzed by Code Review. Use this options to force the analysis of system include files.

```
[-DEFINE= <ident>[ = <value>] {[, <ident>[ = <value>]]}
```

This option specifies conditions to be applied when the Code Review Compiler starts. These conditions allow you to define C symbols that apply conditions to the generation of any **IF ... ELSE ... END IF** blocks in the test script.

If the option is used with one of the conditions specified in the **IF** instruction, the **IF ... ELSE** block (if **ELSE** is present) or the **ELSE ... END IF** block (if **ELSE** is not present) is analyzed and generated. The **ELSE ... END IF** block is eliminated.

If the option is not used or if none of the conditions specified in the **IF** instruction are satisfied, the **ELSE ... END IF** block is analyzed and generated.

All symbols defined by this option are equivalent to the following line in C

```
-define <ident> [<value>]
```

By default, the **ELSE ... END IF** blocks are analyzed and generated.

```
[-UNDEF= <identifier>[, <identifier>]]
```

Allows you to undefine symbols. This is equivalent to the -U option of the compiler.

Example

```
crcc mysource.c output.xob atus_c.def atl.opp
```

```
-rule=$TESTRT_DIR/plugins/Common/lib/confrule.xml
```

Related Topics

[Code review overview on page 233](#) | [Running a code review on page 297](#) | [C Code Review Linker - crclld on page 1086](#)

C Code Review Linker - crclld

The C Code Review Linker links and analyzes the files produced by the C Code Review Compiler (crcc) for code review static analysis. It produces a .crc code review file that can be displayed in HCL OneTest™ Embedded.

Syntax

```
crclld <xob>[ <xob>] -CRC=<output> [<options>]
```

where:

- `<xob>` is the name of the object file generated by the Code Review Compiler.
- `<output>` is the generated .crc code review report file.

Description

The C Code Review Compiler fits into your compilation sequence with minimal changes.

Options

Command line options can be abbreviated to their shortest unambiguous number of characters and are not case-sensitive.

[-RULE=<file>]

Specifies the code review rule configuration file. By default, a default internal configuration rule set is used

[-TEST]

Disables the verification of undefined symbols when using the code review link checker in test mode from HCL OneTest™ Embedded. So some of the MISRA rules are not verified. To verify these rules, they must be directly run from an application node in HCL OneTest™ Embedded.

List of rules that are not verified when the code review is run in test mode with the link checker:

FOR MISRA -C:2004

- - *Rule M8.7*: Global object should not be declared if they are used only from within a single function.
 - *Rule M8.9.2*: The global object or function `<name>` should have exactly one external definition. No definition found.
 - *Rule M8.10.1*: The global object `<name>` that is used only within the same file should be declared using the static storage-class specifier.
 - *Rule M8.10.2*: The global function `<name>` that is used only within the same file should be declared using the static storage-class specifier.
 - *Rule E8.51*: The object `<name>` is never referenced.
 - *Rule E16.50*: The function `<name>` is never referenced.

FOR MISRA C:2012

- *M8.9*: An object should be defined at block scope if its identifier only appears in a single function
- *E8.10*: The global object or function `<name>` should have exactly one external definition. No definition found.

- *M8.7.1* : Global object <name> that is only used within the same file should be declared using the static storage-class specifier.
- *M8.7.2*: Global function <name> that is only used within the same file should be declared using the static storage-class specifier.
- *E8.12*: The object <name> is never referenced.
- *M2.2.2*: The function <name> is never referenced.

Example

```
crcld object.xob main.xob -crc=main.crc
```

Related Topics

[Code review overview on page 233](#) | [Running a code review on page 297](#) | [C Code Review Compiler - crccc on page 1085](#)

Ada Instrumentor - attolada

Purpose

The source code insertion (SCI) Instrumentor for Ada inserts functions from a Target Deployment Port library into the Ada source code under test. The Ada Instrumentor is used for Code Coverage only.

Syntax

```
attolada <src> <instr> [<options>]
```

where:

- <src> is the source file (input)
- <instr> is the instrumented output file

Description

The Instrumentor builds an output source file from an input source file, by adding special calls to the Target Deployment Port function definitions.

The Ada Instrumentor (**attolada**) supports Ada83 and Ada95 standard source code without distinction.

You can select one or more types of coverage at the instrumentation stage (see the User Guide for more information).

When you generate reports, results from some or all of the subset of selected coverage types are available.

Options

-PROC [=RET]

-PROC alone instruments procedure, function, package, and task entries. This is the default setting.

The **-PROC=RET** option instruments both entries and exits.

-CALL

Instruments Ada functions or procedures.

-BLOCK [=IMPLICIT | DECISION | LOGICAL | ATC]

This option specifies how blocks are to be instrumented.

- The **-BLOCK** option alone instruments simple blocks only.

Use the **IMPLICIT** or **DECISION** option to instrument implicit blocks (unwritten else instructions), as well as simple blocks.

Use the **LOGICAL** parameter to instrument logical blocks (loops), as well as the simple and implicit blocks.

Use the **ATC** parameter to extend the instrumentation to asynchronous transfer control (**ATC**) blocks.

By default, the Instrumentor instruments implicit blocks.

-COND [=MODIFIED | COMPOUND | FORCEEVALUATION]

When **-COND** is used with no parameter, the Instrumentor instruments basic conditions.

- **MODIFIED** or **COMPOUND** are equivalent settings that allow measuring the modified and compound conditions.

FORCEEVALUATION modifies the code to force the execution of all conditions in the decision.

-NOPROC

Disables instrumentation of procedure inputs, outputs, or returns, etc.

-NOCALL

Disables instrumentation of calls.

-NOBLOCK

Disables instrumentation of simple, implicit, or logical blocks.

-NOCOND

Disables instrumentation of basic conditions.

-NOFULLPATHKEY

This option forces the product to ignore the full path of files. Use this option if you need to consolidate test results when a same file can be identified with various paths, for example in a multi-user development environment using source control.

-UNIT=<name>[,{<name>}] | -EXUNIT=<name>[,{<name>}]

-UNIT specifies Ada units (packages or functions or procedures in packages) whose bodies are to be instrumented, where *<name>* is an Ada unit which is to be explicitly instrumented. All other functions are ignored.

-EXUNIT specifies packages, or functions or procedures in packages that are to be excluded from the instrumentation. All other Ada units are instrumented. Units that are excluded from the instrumentation process with the **-EXUNIT** option are still analyzed. Any errors found in those files are still reported. For example:

-EXUNIT=MYPACKAGE or **-EXUNIT=MYPACKAGE.MYFUNCTION**

-UNIT and **-EXUNIT** cannot be used together.

-LINK= <filename>[,{<filename>}]

Provides a set of link files to the Instrumentor.

-INJECT= <unit>[,{<unit>}]

-NAMEINJECT[=<name>]

These options allow you to inject a procedure definition into the instrumented source code. **-INJECT** specifies the package(s) that contain the procedure definition. **-NAMEINJECT** specifies the name of the procedure that is injected. If *<name>* is not specified, then the procedure name **ATTOL_TEST** is assumed. These options must be used together.

-STDLINK= <filename>

Provides a standard link file to the Instrumentor.

-FDCDIR= <directory>

Specifies the destination *<directory>* for the **.fdc** correspondence file, which is generated for Code Coverage after the instrumentation for each source file. Correspondence files contain static information about each enumerated branch and are used as inputs to the Code Coverage Report Generator. If *<directory>* is not specified, each **.fdc** file is generated in the directory of the corresponding source file. If you do not use this option, the default **.fdc** files directory is the working directory (the **attolada** execution directory). You cannot use this option with the **-FDCNAME** option.

-FDCNAME= <name>

By default, the instrumentor generates one **.fdc** static correspondence file for each source file involved in the code to be instrumented. Use this option to specify a single static file for all source files in order to avoid file access conflicts, for example when a parallel build is involved. When this option is specified, the generated **.fdc** file contains one FDC section per source file. You cannot use this option with the **-FDCDIR** option.

-DUMPINCOMING= <name>[{{<name>}}]

-DUMPRETURNING= <name>[{{<name>}}]

These options allow you to explicitly define upon which incoming or returning function(s) the trace dump must be performed. Please refer to **General Runtime Analysis Settings** in the **User Guide** for further details.

-COMMENT= <comment>

Associates the text from either the Code Coverage Launcher (preprocessing command line) or from you with the source file and stores it in the FDC file to be mentioned in coverage reports. In Code Coverage Viewer, a magnifying glass is put in front of the source file. Clicking on this magnifying glass, shows this text in a separate window. The comment text must not contain commas or non-alphanumeric characters.

-NOMETRICS

Saves the metrics basic data calculation time.

-RESTRICTION =NOEXCEPTION|NOGENERIC|CSMART

Use this option to set a restriction.

- **NOEXCEPTION** deactivates instrumentation of exception block branches encountered in the source file. When this option is active, no coverage information is available on exception blocks or on instructions contained in exception blocks.

NOGENERIC deactivates the instrumentation using a generic Target Deployment Port call. When this option is active, the generated source code may contain uninstrumentable calls. If used with the **-CALL** option, this can generate compilation errors depending on your application if, for example, you use private packages as well as private sub-packages.

CSMART generates **CSMART** compliant code.

-NOSOURCE

Replaces the generation of the colored viewer source listing by a colored viewer pre-annotated report containing line number references.

-NOCVI

Disables generation of a Code Coverage report that can be displayed in the Code Coverage Viewer.

-METRICS

Provides static metric data for compatibility with old versions of the product. Use the static metrics features of the Test Script Compiler tools instead. By default no static metrics are produced by the Instrumentors.

-GENERATEDNAME = CHECKSUM | *<filename>*

-USERNAME = *<NAME>*

Use these options to add a package to the header of the generated file to store coverage traces. You can specify the name of the generated package using one of the following three options:

- **-GENERATEDNAME=CHECKSUM** uses a checksum calculated on the instrumented file to create a package name under the form **ATC_<checksum>**, where *<checksum>* has a maximum of four letters.
- **-GENERATEDNAME= *<filename>*** uses the name of the file to be instrumented, this name is transformed into an Ada identifier and prefixed by **ATC_**.
- **-USERNAME= *<username>***: A name you choose freely by the user and provide on the command line.

<File> is used without checking whether it is a valid Ada identifier.

By default, the **-GENERATEDNAME=<FILE>** option is used.

-ALTCHECKSUM

Use this option to calculate a more unambiguous checksum for .fdc and .tsf files. Before using this option, you must delete existing fdc and tsf files, which will be re-created with the new checksum. File keys are not changed by this option.

-PREFIX= *<prefix>*

You can prefix some instrumentations (name of the generated package, variables, etc.) if there are any semantic ambiguities. Thus, packages generated by **attolada** can be recognized by giving them a known prefix.

By default, no prefix is used.

Note The prefix you provide is used, without checking whether it is a valid Ada identifier.

-SPECIFICATION

Extends instrumentation of calls and conditions to source code inside package specifications.

-MAIN= <unit>[{, <unit>}]

Forces a trace dump at the end of the main unit of your application.

-EXCALL= <unit>[{, <unit>}]

Only applies when **-CALL** is used. Excludes all specified calls to the function or procedure units from the instrumentation. The <unit> names must be fully qualified names, for example: **package.procedure**

-ADA83 | **-ADA95**

Choose specifies the Ada language used by the Instrumentor. This language is applied to the analyzed and generated file.

-INSTRUMENTATION=[COUNT|INLINE]

Specifies the Instrumentation Mode:

- **COUNT**: Default Pass mode, each branch generates in 32 bits for profiling purposes. This offers the best compromise between code size and speed overhead.
- **INLINE**: Compact mode. functionally equivalent to *Pass* mode, except that each branch needs only one bit of storage instead of one byte. This implies a smaller requirement for data storage in memory, but produces a noticeable increase in code size (shift/bits masks) and execution time.

By default, count mode is used, which is a compromise between the flow mode (everything is a call to the Target Deployment Package) and the inline mode (when possible, the code is directly inserted into the generated file).

-NOINFO

Asks the Instrumentor not to generate the identification header. This header is normally written at the beginning of the instrumented file, to strictly identify the instrument used.

-STUDIO_LOG

This option is for internal usage only.

Return Codes

After execution, the program exits with the following return codes

Code	Description
0	End of execution with no errors
7	End of execution because of fatal error

- 9 End of execution because of internal error

All messages are sent to the standard error output device.

Java Instrumentor - javi

Purpose

The SCI Instrumentor for Java inserts methods from a Target Deployment Port library into the Java source code under test. The Java Instrumentor is used for:

- Performance Profiling
- Code Coverage
- Runtime Tracing

Memory Profiling for Java uses the JVMPI Agent instead of source code insertion (SCI) technology as for other languages.

Syntax

```
javi <src> {[,<src> ]} [<options>]
```

where:

- <src> is one or several Java source files (input)

Description

The SCI Instrumentor builds an output source file from each input source file by adding specific calls to the Target Deployment Port method definitions. These calls are used by the product's runtime analysis features when the Java application is built and executed.

The Runtime Analysis tools are activated by selecting the command line options:

- **-MEMPRO** for Memory Profiling
- **-PERFPRO** for Performance Profiling
- **-TRACE** for Runtime Tracing
- **-PROC** and **-BLOCK** for Code Coverage (code coverage levels).

Note that there is no **-COVERAGE** option; the following rules apply for the Code Coverage feature:

- If no code coverage level is specified, nor Runtime Tracing, Memory Profiling, or Performance Profiling, then the default is to have code coverage analysis at the **-PROC** and **-BLOCK=DECISION** level.
- If no code coverage level is specified while one or more of the aforementioned features are selected, then code coverage analysis is not performed.

Detailed information about command line options for each feature are available in the sections below.

The Java Instrumentor creates the output files in a **javi.jir** directory, which is located inside the current directory. By default, this directory is cleaned and rewritten each time the Instrumentor is executed.

Although the Java Instrumentor can take several input source files on the command line, you only need to provide the file containing a **main** method for the Instrumentor to locate and instrument all dependencies.

The **\$CLASSPATH** or **\$EDG_CLASSPATH** environment variable must point to the native classes required by the Instrumentor. Alternatively, you can specify one or several additional classpaths by using the **-CLASSPATH** option of the Java Instrumentor. The **-CLASSPATH** option overrides the **\$CLASSPATH** and **\$EDG_CLASSPATH** environment variables -in that order- during instrumentation.

When using the Code Coverage feature, you can select one or more types of coverage at the instrumentation stage (see the User Guide for more information). When you generate reports, results from some or all of the subset of selected coverage types are available.

Options

-FILE= *<filename>*{*<filename>*} | **-EXFILE=** *<filename>*{*<filename>*}

-FILE specifies the only files that are to be explicitly instrumented, where *<filename>* is a Java source file. All other source files are ignored.

-EXFILE explicitly specifies the files that are to be excluded from the instrumentation, where *<filename>* is a Java source file. All other source files are instrumented.

Files that are excluded from the instrumentation process with the **-EXFILE** option are still analyzed. Any errors found in those files are still reported.

<filename> may contain a path (absolute or relative from the current working directory). If no path is provided, the current working directory is used.

-FILE and **-EXFILE** cannot be used together.

-NOFULLPATHKEY

This option forces the product to ignore the full path of files. Use this option if you need to consolidate test results when a same file can be identified with various paths, for example in a multi-user development environment using source control.

-CLASSPATH=<classpath>

The **-CLASSPATH** option overrides the **\$CLASSPATH** and **\$EDG_CLASSPATH** environment variables -in that order- during instrumentation.

In <classpath>, each path is separated by a colon (":") on UNIX systems and a semicolon (";") in Windows.

-OPP=<filename>

The **-OPP** option allows you to specify an optional definition file. The <filename> parameter is a relative or absolute filename.

-DESTDIR= <directory>

The **-DESTDIR** option specifies the location where the **javi.jir** output directory containing the instrumented Java source files is to be created. By default, the output directory is created in the current directory.

-PROC [=RET]

The **-PROC** option alone causes instrumentation of all classes and method entries. This is the default setting.

The **-PROC=RET** option instruments procedure inputs, outputs, and terminal instructions.

-BLOCK=IMPLICIT | DECISION | LOGICAL

The **-BLOCK** option alone instruments simple blocks only.

Use the **IMPLICIT** or **DECISION** (these are equivalent) option to instrument implicit blocks (unwritten else instructions), as well as simple blocks.

Use the **LOGICAL** parameter to instrument logical blocks (loops), as well as the simple and implicit blocks.

By default, the Instrumentor instruments implicit blocks.

-NOTERNARY

This option allows you to abstract the measure from simple blocks. If you select simple block coverage, those found in ternary expressions are not considered as branches.

-NOPROC

Specifies no instrumentation of procedure inputs, outputs, or returns, and so forth.

-NOBLOCK

Specifies no instrumentation of simple, implicit, or logical blocks.

-COUNT

Specifies count mode. By default, the Instrumentor uses pass mode. See the User Guide.

-COMPACT

Specifies compact mode. By default, the Instrumentor uses pass mode. See the User Guide.

-UNIT=<name>[,{<name>}] | -EXUNIT=<name>[,{<name>}]

-UNIT specifies Java units whose bodies are to be instrumented, where *<name>* is an Java package, class or method which is to be explicitly instrumented. All other units are ignored.

-EXUNIT specifies the units that are to be excluded from the instrumentation. All other Java units are instrumented.

-UNIT and **-EXUNIT** cannot be used together.

-DUMPINCOMING= <service>[,{<service>}]

-DUMPRETURNING= <service>[,{<service>}]

-MAIN= <service>

These options allow you to precisely specify where the SCI dump must occur. **-MAIN** is equivalent to **-DUMPRETURNING**.

-COMMENT= <comment>

Associates the text from either the Code Coverage Launcher (preprocessing command line) or from you with the source file and stores it in the FDC file to be mentioned in coverage reports. In Code Coverage Viewer, a magnifying glass is put in front of the source file. Clicking this magnifying glass shows this text in a separate window.

-NOCVI

Disables generation of a Code Coverage report that can be displayed in the Code Coverage Viewer.

-JTEST | -NOJTEST

The **-JTEST** option provides UML sequence diagram output for Component Testing for Java with HCL OneTest Embedded. **-NOJTEST** disables this output.

-NOCLEAN

When this option is set, the Instrumentor does not clear the **javi.jir** directory before generating new files.

-FDCDIR= <directory>

Specifies the destination *<directory>* for the **.fdc** correspondence file, which is generated for Code Coverage after the instrumentation for each source file. Correspondence files contain static information about each enumerated branch and are used as inputs to the Code Coverage Report Generator. If *<directory>* is not specified, each **.fdc** file is generated in the directory of the corresponding source file. If you do not use this option, the default **.fdc** files directory is the current working directory. You cannot use this option with the **-FDCNAME** option.

-FDCNAME= *<name>*

By default, the instrumentor generates one **.fdc** static correspondence file for each source file involved in the code to be instrumented. Use this option to specify a single static file for all source files in order to avoid file access conflicts, for example when a parallel build is involved. When this option is specified, the generated **.fdc** file contains one FDC section per source file. You cannot use this option with the **-FDCDIR** option.

-NO_UNNAMED_TRACE

With this option, anonymous classes are not instrumented.

-PERFPRO

This option activates Performance Profiling instrumentation. This produces output for a Performance Profile report.

-TRACE

This option activates Runtime Tracing instrumentation. This produces output for a UML sequence diagram.

-TSFDIR=*<directory>*

Specifies the destination *<directory>* for the **.tsf** static trace file, which is generated for Code Coverage after the instrumentation of each source file. If *<directory>* is not specified, each **.tsf** static trace file is generated in the directory of the corresponding source file. If you do not use this option, the default **.tsf** static trace file directory is the current working directory. You cannot use this option with the **-TSFNAME** option.

-TSFNAME=*<filename>*

Specifies the *<name>* of the **.tsf** static trace file that is to be produced by the instrumentation. You cannot use this option with the **-TSFDIR** option.

-INSTRUMENTATION=[FLOW|COUNT|INLINE]

Choose specifies the instrumentation mode. By default, count mode is used, which is a compromise between the flow mode (everything is a call to the Target Deployment Package) and the inline mode (when possible, the code is directly inserted into the generated file).

Warning: Inline mode must be used only in pass mode. Do not use this option if you want to know how many times a branch is reached.

-NOINFO

Asks the Instrumentor not to generate the identification header. This header is normally written at the beginning of the instrumented file.

-STUDIO_LOG

This option is for internal usage only.

Return Codes

After execution, the program exits with the following return codes

Code	Description
0	End of execution with no errors
7	End of execution because of fatal error
9	End of execution because of internal error

All messages are sent to the standard error output device.

Ada Metrics Generator - metada

Purpose

The Ada Metrics Generator produces **.met** static metric files for the specified source files.

Syntax

metada <source_file> [-**output_dir**= <output_directory>]

metada @ <options_file>

where:

- <source_file> is the name of the source file to be analyzed.
- <output_directory> is the absolute path of the location where the .met static metric file is to be generated.
- <options_file> points to a plain text file containing a list of options.

Description

The Ada Metrics Calculator analyzes a specified Ada source file and produces a .met static metric file, which can be opened with the HCL OneTest Embedded GUI.

Note For other languages, the **.met** static metric files are produced by the C and C++ Source Code Parsers.

Return Codes

After execution, the program exits with the following return codes

Code	Description
0	End of execution with no errors
3	End of execution with one or more warning messages
5	End of execution with one or more errors
7	End of execution because of fatal error
9	End of execution because of internal error

All messages are sent to the standard error output device.

TDF Splitter - attsplit

Purpose

For use with Runtime Tracing. The **.tdf** splitter (**attsplit**) tool allows you to separate large **.tdf** dynamic trace files into smaller—more manageable—files.

Syntax

```
attsplit [<options>] <tcf file> <tsf_file> <tdf file>
```

where:

- *<tcf_file>* is always **\$TESTRTDIR/lib/tracer.tcf**
- *<tsf_file>* is the name of the generated **.tsf** static trace file
- *<tdf file>* is the name of the original **.tdf** dynamic trace file

Description

Trace **.tdf** files that contain loops cannot be split.

Options

-p *<prefix>*

Specifies the filename prefix for the split **.tdf** files. By default, split **.tdf** filenames start with **att**.

-s *<bytes>*

Sets the maximum file size for the split **.tdf** files. By default, the original **.tdf** dynamic trace file is split into 1000 byte split **.tdf** files

Specifies

-v | -vw

Activates verbose mode (**-v**) or verbose mode for written files only (**-vw**).

-nt

Disables the writing of time information. By default, time information is written to the split **.tdf** files.

-fopt *<filename>*

Uses a text file to pass options to the **attsplit** command line.

-studio_log

This option is for internal usage only.

Code Coverage Report Generator - attolcov

Purpose

The Report Generator creates Code Coverage reports from the coverage data gathered during the execution of the application under analysis.

Syntax

attolcov {*<fdc file>*} {*<traces>*} [*<options>*]

where:

- *<fdc files>* The list of correspondence files for the application under test, with one file generated for each source file during instrumentation

<traces> is a list of trace files. (default name **attolcov.tio**)

<options> represents a set of options described below.

Parameters can use wild-card characters ('*' and '?') to specify multiple files. They can also contain absolute or relative paths.

Description

Trace files are generated when an instrumented program is run. A trace file contains the list of branches exercised during the run.

You can select one or more coverage types at the instrumentation stage.

All or some of the selected coverage types are then available when reports are generated.

The Report Generator supports the following coverage type options:

-PROC[=RET]

The **-PROC** option, with no parameter, reports procedure inputs.

Use the **RET** parameter to reports procedure inputs, outputs, and terminal instructions.

-CALL

Reports call coverage.

-BLOCK[=IMPLICIT | DECISION | LOGICAL | ATC]

The **-BLOCK** option, with no parameter, reports statement blocks only.

- **IMPLICIT** or **DECISION** (equivalent) reports implicit blocks (unwritten else and default blocks), as well as statement blocks.

LOGICAL Reports logical blocks (loops, as well as statement and implicit blocks.

ATC Reports asynchronous transfer control (ATC) blocks, as well as statement blocks, implicit blocks, and logical blocks.

-COND[=MODIFIED|COMPOUND]

The **-COND** option, with no parameter, reports basic conditions only.

MODIFIED reports modified conditions as well as basic conditions.

COMPOUND reports compound conditions as well as basic and modified conditions.

Explicitly Excluded Options

Each coverage type can also be explicitly excluded.

-NOPROC

Procedure inputs, outputs, or returns are not reported.

-NOCALL

Calls are not reported.

-NOBLOCK

Simple, implicit, or logical blocks are not reported.

-NOCOND

Basic conditions are not reported.

Additional Options

The following options are also available:

-FILE= *<file>*{[, *<file>*]} | **-EXFILE=** *<file>*{[, *<file>*]}

Specifies which files are reported or not. Use **-FILE** to report only the files that are explicitly specified or **-EXFILE** to report all files except those that are explicitly specified. Both **-FILE** and **-EXFILE** cannot be used together.

-SERVICE=*<service>*{[, *<service>*]} | **-EXSERVICE=***<service>*{[, *<service>*]}

Specifies which functions, methods, and procedures are to be reported or not. Use **-SERVICE** to report only the functions, methods and procedures that are explicitly specified or **-EXSERVICE** to report all functions, methods, and procedures except those that are explicitly specified. Both **-SERVICE** and **-EXSERVICE** cannot be used together.

-TEST= *<test>*{[, *<test>*]} | **-EXTEST=** *<test>*{[, *<test>*]}

Specifies which tests are reported or not. Use **-TEST** to report only the tests that are explicitly specified or **-EXTEST** to report all tests except those that are explicitly specified. Both **-TEST** and **-EXTEST** cannot be used together.

-OUTPUT= *<file>*

Specifies the name of the report file (*<file>*) to be generated. You can specify any filename extension and can include an absolute or relative path.

-LISTING[=*<directory>*]

This option requires annotated listings to be generated from the source files. Annotated listings carry the same name as their corresponding source files, but with the extension **.lsc**. The optional parameter *<directory>* is the absolute or relative path to the directory where the listings are to be generated. By default, a listing file is generated in the directory where its corresponding source file is located.

-NOGLOBAL

Reports the results of each test found in the trace file, followed by a conclusion summarizing all the tests. If a test has no name, it is identified as **"#"** in the report. A test is an execution of an instrumented application, a **TEST** as defined for Component Testing for C and Ada, or a dump-on-signal. By default, the report is not structured in terms of tests.

-BRANCH=COV

Reports branches covered rather than branches not covered. It does not affect listings, where only branches not covered are indicated with the source code line where they appear.

-CLEAN=<file.tio>

Generates a new cleaned up **.tio** file that takes up less disk space. You can delete the original **.tio** file after using this option.

-MERGETESTS

When using the **-CLEAN** option, merges previous results in order to produce a more compact file.

-SUMMARY=CONCLUSION | FILE | SERVICE

This option sets the verbosity of the summary:

- **CONCLUSION** reports only the overall conclusion.
- FILE** reports only the conclusion for each source file, and the overall conclusion.
- SERVICE** reports only the levels of coverage for each source file, each C function, and overall. The list of branches covered or not covered is not included.

-STUDIO_LOG

This option is for internal usage only.

Return Codes

After execution, the program exits with the following return codes:

Code	Description
0	End of execution with no errors
7	End of execution because of fatal error
9	End of execution because of internal error

All messages are sent to the standard error output device.

Trace Probe Processor - parsecode.pl

When using the Probe runtime analysis feature of the product, the Probe Processor takes C source files and message definition files and generates a set of source files containing the message trace functions called by the probe macros.

Syntax

parsecode.pl [*<options>*] {*<msg_files>* [, *<msg_files>*]} {*<source>* [, *<source>*]}

where:

1. *<msg_files>* are **.h** message type definition files

<source> are probed C source files

<options> is a set of optional parameters among those described below.

Definition

The Probe Processor is only available for the C language.

The message traces are written to a **.rio** output file, one for each instance. The probed binary is produced by compiling the **atlprobe.c** file, which is generated by the Probe Processor, and linked to the application object files with the Target Deployment Port.

Optional Parameters

[-mode=DEFAULT|CUSTOM]

Default mode writes the **.rio** output file on-the-fly. In this case, the **atl_dump_trace** macro is not required.

Custom mode allows the probes to write traces to a temporary location, such as memory, a stack or a buffer file. In custom mode, the traces are flushed to the **.rio** file only when an **atl_dump_trace** macro is encountered.

The I/O functions for probe trace output to the temporary location are defined in the **probecst.c** source file delivered with the product. You can modify this file to adapt the probe mechanism to your application and platform.

In custom mode, the compilation and link phase generates write operations from the probed application and the **probecst.c** file, and read operations from the **atlprobe.c**, **probecst.c** files and the **TP.o** Target Deployment Port file.

[-preopts=-INCL= <include directories>]

The **-preopts** option allows you to send a list of include directories specified with a C Test Script Compiler **-INCL** option. See the [C Test Script Compiler on page 1112](#) **-INCL** option.

[-outdir= <output directory>]

This option allows you to specify the target directory for the **atlp**.

[-accuracy= <time>]

This option expressed the desired accuracy to be used if you are generating a **.pts** test script for use with System Testing for C, where *<time>* is expressed in milliseconds (ms).

[-polling= <time>]

This option expressed the desired polling interval to be used if you are generating a **.pts** test script for use with System Testing for C, where <time> is expressed in milliseconds (ms).

[-studio_log]

This option is for internal usage only.

Related Topics

[Probe Macros on page 1040](#) | [C Test Script Compiler on page 1112](#) | [System Testing Report Generator on page 1109](#)

C system testing command line interface

System Testing Supervisor - atsspv

The System Testing Supervisor executes **.spv** supervisor script files.

Syntax

```
atsspv spv_script options
```

Where

spv_script is the **.spv** supervisor script to execute.

options is a series of command line options. See the section Options.

Description

System Testing manages the simultaneous execution of Virtual Testers distributed over a network. When using System Testing, the job of the Supervisor is to:

- Set up target hosts to run the test.
- Launch the Virtual Testers, the system under test and any other tools.
- Synchronize Virtual Testers during execution.
- Retrieve the execution traces after test execution.

The System Testing Supervisor uses a **.spv** supervisor deployment script to control System Testing Agents installed on each distributed target host. Agents can launch either applications or Virtual Testers.

While the agent-spawned processes are running, their standard and error outputs are redirected to the supervisor.



Note: You must install and configure the agents on the target machines before execution.

The Supervisor generates traces during analysis and execution. These traces are displayed on the screen and written to a log file named as **spv_script.lis**.

Confirmation with telnet interface

You can check that the System Testing Agent is correctly configured by using the telnet interface. Launch a telnet session to the computer on which the Agent is running, on the System Testing port (by default 10000) and type **Jef** username after the welcome prompt. The exchange should be the same as follows:

```
> telnet <computer> 10000
210 hello, pleased to meet you.>
Jef <username>
```

The answer should provide the status of the user on the computer.

Options

- **-CHECK**

This option specifies that the scenario is to be analyzed but not executed. This allows you to check for errors in the .spv script.

- **-NOLOG**

Disables supervisor output of error messages and warnings to the screen. Traces are still written to the .lis log file.

- **STUDIO_MACH=localhost**

By default, the supervisor uses the IP address 127.0.0.1 to connect to the HCL OneTest™ EmbeddedGraphical User Interface. Use **-STUDIO_MACH= localhost** to resolve problems when the supervisor fails to connect.

- **-STUDIO_LOG**

This option is for internal usage only.

Return Codes

Code	Description
0	End of execution with no errors
3	End of execution with one or more warning messages
5	End of execution with one or more errors
7	End of execution because of fatal error

Code	Description
9	End of execution because of internal error

System Testing Load Report Generator - atslload

Purpose

The Load Report Generator produces a report describing messages and execution time.

Syntax

atsload **-SEPARATOR='** <sep_string> ' **-TITLE** <rdt file> {[, <rdt file>]}

where:

- <rdt_file> is an **.rdt** output file generated by the Report Generator.
- <sep_string> is the separator string.
- <options> is a set of optional parameters among those described below.

Description

The System Testing Load Report Generator tool processes **.rdt** file of a virtual tester from the Report Generator and produces the following output:

- **TITLE**: The optional header for each column of the report.
- **SCENARIO**: total execution time
- **SEND**: timestamp of a SEND message relative to the beginning of the SCENARIO
- **MESSAGE**: timestamp of a WAITTIL relative to the beginning of the SCENARIO
- **PRINT**: value of the numeric parameter

You can use the Load Report Generator to compare between several virtual testers. Data is presented in columns, separated by a separator string. Each column represents a particular virtual tester.

There must be one **.rdt** file for each virtual tester.

Optional Parameters

-TITLE

This option adds a **TITLE** line to the report, containing the name of the virtual tester for each column.

-STUDIO_LOG

This option is for internal usage only.

Example

```
atsload -SEPARATOR=':' vt1.rod vt2.rod vt2.rod
```

Return Codes

After execution, the program exits with the following return codes:

Code	Description
0	End of execution with no errors
7	End of execution due to a fatal error
9	End of execution due to an internal error

All messages are sent to the standard error output device.

System Testing Report Generator - atsmmerge

Syntax

```
atsmmerge <file> {[, <file> ]} [<options>]
```

where:

1. <file> lists the **.rio** intermediate result files generated during the virtual tester execution phase and the **.tdc** correspondence table files generated during compilation.

<options> is a list of options described below.

Description

The system generates a **.rod** result file for each **.rio** file, which is saved in the **rio** directory. The **.rod** filename uses the **.rio** filename with a **.rod** extension.

If one of the files cannot be found, the Report Generator produces a fatal error. The Report Generator does not support spaces in a filename.

The Report Generator produces a warning message each time it encounters any incorrect data.

If the report contains any synchronization errors between the **.tdc** and the **.rio** file, the Report Generator produces a fatal error.

Options

The options can be in any order. They may be upper or lower case and written in an incomplete form, provided the selected option is clear.

-TIME

This option enables you to merge reports that do not contain structure instructions. Structural instructions are beginning and ending block instructions (scenario, initialization, exception, termination).

If the **.rio** and **.tdc** files come from different test scripts, the **-TIME** option is enabled.

-RDD = <RDD report filename>

This option enables you to specify the output report filename.

By default, the report is named **atsrdd.rdd** and generated in the current directory.

-RA [=ERR | =TEST]

This option specifies the form of the report generated.

With **-RA = TEST**, only variables that are in a failed test are displayed.

With **-RA = ERR**, no variables are displayed.

In both cases, if the test is correct, only general information on this test is displayed.

The default option is **-RA** (with no parameters), which provides a full report of all variables for each test.

-VA =EVAL | NOEVAL | COMBINE

This option lets you specify the way in which initial and expected values of each variable is displayed in the test report.

1. With **-VA = EVAL**, the initial, expected value of each variable evaluated during execution is displayed in the report. This option is only visible for variables whose initialization or expected value is not reduced in the test script.

Note: For structures in which one of the fields is an array, this evaluation is not given for the initial values. For expected values, it is only given for incorrect elements.

1. With **-VA = NOEVAL**, for each variable, the report generator displays in the test report the initial and expected values described in the test script.

Use **-VA = COMBINE** to combine the previous two options, that is, for each variable, the report generator displays in the test report the initial and expected values described in the test script as well as the initial and expected values evaluated during execution.

By default **-VA = EVAL** is used.

-SUMMARY | NOSUMMARY

This option produces a summary of the test execution in the test report.

This option gives a quick overview of the execution of the set of test scenarios. It only summarizes the execution of the test scenarios.

The default option is **-NOSUMMARY**.

-COMMENT | -NOCOMMENT

In the System Testing Language, the **COMMENT** keyword displays a comment in the test report. You can use **-NOCOMMENT** to disable these comments, and **-COMMENT** to make them visible.

By default comments are displayed.

-STUDIO_LOG

This option is for internal usage only.

Log File

-LOG | -NOLOG

With the **-LOG** option, errors found during analysis of **.rio** and **.tdc** files are displayed on screen. Use the **-NOLOG** option to disable this behavior.

By default the **-LOG** option is used.

Example

```
atsmerge fic01.rio fic02.rio fic01.tdc fic02.tdc ...
```

Return Codes

After execution, the program exits with the following return codes:

Code	Description
0	End of execution with no errors
3	End of execution with one or more warning messages
5	End of execution with one or more errors
7	End of execution because of fatal error
9	End of execution because of internal error

All messages are sent to the standard error output device.

System Testing Script Compiler - atspreproC

The System Testing Script Compiler preprocesses the Test Script and converts it into a native source test harness.

Syntax

```
atspreproC <test script> <interface_file> {[,<interface_file>]} <file name> [<options>]
```

where:

- <test script> is the test script to be compiled.
- <interface file> lists interface files that contain event structure definitions, *includes* for interface prototypes, and their types. These files may have any extension.

Note If read access to these files is denied, System Testing for C produces a fatal error.

- <file name> is the name of the C code file generated from the test script. If you do not specify an extension, the system uses the **ATS_SRC** environment variable extension, or the default extension **.c**.
- <options> is a set of optional parameters among those described below.

Description

If you do not specify an extension, the system uses the **ATS_PTS** environment variable extension or the default **.pts** extension.

If an input file is absent or read access is denied, System Testing for C produces a fatal error.

After execution, the code is generated in the **code.c** file. If it is not possible to create the file, you will receive a fatal error.

If the Report Generator detects incorrect tests, System Testing for C produces a warning message.

If the report detects a synchronization error between the **.tdc** and the **.rio** file, System Testing for C produces a fatal error.

Optional Parameters

Options can be in any order. They may be upper or lowercase and can be abbreviated to their shortest unambiguous number of characters.

-ALLOCATION[=STACK | =DYNAMIC]

This option allows you to specify the method for allocating the work of the test program in the compiler.

If this option is present, the test program uses only allocated data on the execution stack (**=STACK**).

By default, the work context is global static data.

-BUFSIZE= *<size>*

This option sets the size of the trace buffer in kilobytes. The trace buffer is only used with the **-TRACE** option.

The default buffer size is 10KB.

-DEFINE= *<list of conditions>*

This option lets you specify the conditions to apply during test compilation. This option is equivalent to compiler option **-D**.

You can specify particular conditions or give them a value (**-define=condition=value**). Symbols defined with this option are equivalent to the following line in C:

```
#define <symbol> [ <value> ]
```

-FAMILY= *<family>* {[, *<family>*]} | **-EXFAMILY=** *<family>* {[, *<family>*]}

-FAMILY specifies the only test families that are to be explicitly executed. Any other test families are ignored.

-EXFAMILY explicitly specifies the families that are to be ignored. All other families are executed.

-FAMILY and **-EXFAMILY** cannot be used together. The Test Script Compiler generates a warning message if no scenarios are generated.

By default, all test families are executed.

-SCN= *<scenario>* {[, *<scenario>*]} | **-EXSCN=** *<scenario>* {[, *<scenario>*]}

-SCN specifies the only scenarios that are to be explicitly executed. Any other scenarios are ignored.

-EXSCN explicitly specifies the scenarios that are to be ignored. All other scenarios are executed.

-SCN and **-EXSCN** cannot be used together.

To specify a sub-scenario, name the set of scenarios in which it is included and separate with full stops. If you exclude a scenario that contains sub-scenarios, all its sub-scenarios are also excluded.

The Test Script Compiler generates a warning message if no scenarios are generated.

-FAST | -NOFAST

The **-FAST** option tells the Test Script Compiler to analyze only those scenarios that you want to generate. This option accelerates execution of the Test Script Compiler if you use a selection option. The option is useful when using **-SCN**, **-EXSCN**, **-FAMILY**, **-EXFAMILY**.

The **-NOFAST** option disables this behavior.

By default, the **-FAST** option is used.

-INCL= <directory> {[, <directory>]}

This option lists directories where included files are located. Using this option enables you to:

- Establish the list of include files in the tested source file

Execute the **INCLUDE** instructions

Execute the C #include instruction

The system first searches the current directory, next in the directories specified with the **-INCL** option, and finally the default C system files directory.

-LANG=C

This option allows you to select the language of the generated code. You can generate C virtual testers.

By default, virtual testers are generated in C.

-LOG | -NOLOG

With the **-LOG** option, the system displays and stores errors found during the analysis of interface files and test script. The name of the log file is the name of the test script with the **.lis** extension.

If you select **-NOLOG**, these errors are not displayed.

By default, the **-LOG** option is used.

-NOCOMMENT

Use this option to deactivate the processing of **COMMENT** statements in order to improve performance issues.

-NOTSHARED

This option allows you to disable sharing of global static data between instances. When using this option, you must apply different names to all global variables within a test script. No local variable, constant, or function parameter should have the same name as a global static variable in the test script.

This used only by the **-ALLOCATION** and **-THREAD** options.

By default global variables of the test script are shared by all instances.

-STD_DEFINE= <standard definitions file>

This option provides the C parser with a C source file describing the characteristics of the compiler used.

If the specified file cannot be found, the Test Script Compiler stops and you will receive a fatal error.

By default, no compiler characteristics are specified.

-THREAD [=<function name>]

This option allows you to create a test function with a name other than **main**.

If <function name> is omitted, the function name becomes the source file name appended with **_start**.

By default, the generated function is called **main**.

-TRACE=CIRCULAR | ERROR | SCN | TIME

The Test Script Compiler uses a buffer to store the result of the test script execution. This buffer is saved on disk each time selected events (**ERROR**, **SCN**, **TIME**) occur. This option reduces the size of the virtual tester execution file. It is most useful during an endurance test.

- **-TRACE=CIRCULAR** tells the virtual tester to use a circular buffer to store execution traces. The circular buffer stores the execution traces in memory. Traces are flushed into the **.rio** file only after virtual tester execution or if explicitly requested in the test script (see the [FLUSH_TRACE on page 853](#) keyword).

-TRACE=ERROR saves the buffer each time a test script error occurs.

-TRACE=SCN has the same functionality as the **ERROR** parameter, and additionally saves scenario begin and end marks.

-TRACE=TIME has the same functionality as the **SCN** parameter; and additionally saves timed events (**WAITTIL** and **PRINT**).

These options generate incomplete reports - some information is filtered - but the report always includes plan test errors.

If the buffer is too small, some traces are lost and the generated report is incomplete. You can change buffer size with the **-BUFSIZE** option.

-STUDIO_LOG

This option is for internal usage only.

-SPVGEN

This option is for internal usage only.

Examples

```
atsprepro gen.pts interface.h code -EXSCN=Main.send.test_1, Main.receive.test_1
```

Return Codes

After execution, the program exits with the following return codes

Code	Description
0	End of execution with no errors
3	End of execution with one or more warning messages
5	End of execution with one or more errors
7	End of execution because of fatal error
9	End of execution because of internal error

All messages are sent to the standard error output device.

Virtual Tester

Syntax

```
<virtual tester> [-INSTANCE= <instance>] [-OCCID= <id_number>] [-RIO= <trace_file>]
```

Description

Virtual testers are multiple contextual incarnations of a single **.pts** System Testing test script.

One virtual tester can be deployed simultaneously on one or several targets, with different test configurations. A same virtual tester can also have multiple clones on the same target host machine.

Deployment of virtual testers is controlled by either the GUI or a System Testing **.spv** supervisor script when running in the command line interface. Do not edit **.spv** scripts when using the GUI.

System Testing for C generates virtual testers from a test script according to the declared instances.

Note A System Testing Agent for C must be installed and running on each target host before deploying virtual testers to those targets.

Following the execution architecture and constraints needed to comply, the System Testing Script Compiler provides several ways to generate the virtual testers.

Options

Virtual testers can take the following command line options:

-INSTANCE=<instance>

If the **.pts** test script contains **DECLARE_INSTANCE** instructions, this option specifies which behavioral instance the virtual tester is to initiate. By default, the virtual tester generates all behaviors contained in the test script, but on execution, only one instance is adopted.

If no instances are selected even though instances do exist in the test script, the virtual tester stops with a fatal error message.

-RIO= <trace_file>

This syntax specifies the name of the execution trace file to be generated by the virtual tester.

If you do not define a trace filename, the name *<virtual tester>.rio* will be used.

-OCCID= <occurrence_id_number>

This allows you to specify the occurrence identification number to use in the virtual tester identifier when using communication between virtual testers. See the [INTERSEND on page 860](#) and [INTERRECV on page 861](#) statements for more information.

-STUDIO_LOG

This option is for internal usage only.

Component testing command line interface

Component testing for C

The Component Testing for C feature of HCL OneTest™ Embedded provides a unique, fully automated, and proven solution for applications written in C, dramatically increasing test productivity.

To learn about	See
C component test script keywords	C Component Testing script language on page 716
Using the Source Code Parser to generate test scripts	C Source Code Parser - attolstartC on page 1118
Using the Test Script Compiler	C Test Script Compiler - attolpreproC on page 1121
Using the Test Report Generator	C Test Report Generator - attolpostpro on page 1126

Using the C Instrumentor and Instrumentation Launcher [Runtime analysis tools reference on page 1040](#)

Related Topics

[Component Testing for C overview on page 447](#) | [Writing a test script on page 450](#) | [Command line component testing for C, Ada and C++ on page 966](#)

C Source Code Parser - attolstartC

When creating a new Component Testing test campaign for C, the C Source Code Parser creates a C test script template based on the analysis of the source code under test.

Syntax

```
attolstartC<source_file> <test_script> [{<-option>}]
```

```
attolstartC<source_file> -metrics [{<-option>}]
```

```
attolstartC@<option file >
```

where:

- <source under test> this required parameter is the name of the source file to be tested.
- <test script> is the name of the test script that is generated
- <options> is a list of options as defined below.
- <option file> is the name of a plain-text file containing a list of options.

Description

The C Source Code Parser analyzes the source file to be tested in order to extract global variables and testable functions.

Each global variable is automatically declared as external, if this has not already been done at the beginning of the test script. Then, an environment is created to contain all these variables with default tests. This environment has the name of the file (without the extension).

For each function under test, the generator creates a **SERVICE** which contains the C declaration of the variables to use as parameters of the function.

Parameters passed by reference are declared according to the following rule:

- `char* <param>` causes the generation of `char <param>[200]`
- `<type>* <param>` causes the generation of `<type> <param>` passing by reference

It is sometimes necessary to modify this declaration if it is unsuitable for the tested function, where `<type>*` `<param>` can entail the following declarations:

- `<type>*` `<param>` passing-by-value,
- `<type>` `<param>` passing-by-reference,
- `<type>` `<param>`[10] passing-by-reference.

File names can be related or absolute. Path names must not contain commas or non-alphanumeric characters.

If the generated file name does not have an extension, the C Source Code Parser automatically attaches `.ptu` or the extension specified by the **ATTOLPTU** environment variable. This name may be specified relatively, in relation to the current directory, or as an absolute path.

If the test script cannot be created, the C Source Code Parser issues a fatal error and stops.

If the test script already exists, the previous version is saved under the name `<generated test script>_bck` and a warning message is generated.

When the **-metrics** option is specified, the Source Code Parser produces static metrics for the specified source files. In this case, no other output is produced.

Options

Options can be in any order. They may be upper or lowercase and can be abbreviated to their shortest unambiguous number of characters.

If **attolstartC** is invoked without parameters, the binary returns a list of options. Some of those options are deprecated and listed only for legacy purposes.

Included Files

-insert

With this option the source file under test is included into the test script with an `#include` directive, ensuring that all the internal functions and variables (declared static) are visible to the test script. The C Source Code Parser adds the `#include` directive before the `BEGIN` instruction and after any `#includes` added by the `-use` option.

-use=*<file used>*{[,*<file used>*]}

This option gives the C Source Code Parser a list of header files to include in the test script before the `BEGIN` instruction. This avoids declaring variables or functions that have already been declared in a C header file of the application under test.

The C Source Code Parser adds the `#include` directive before the `BEGIN` instruction. Then, for each file, an environment is created, containing all variables with a default test. This environment has the name of the included file.

By default, no files are included in the test script.

Integrated Files

-integrate=<additional file>{[,<additional file>]}

This option provides a list of additional source files whose objects are *integrated* into the test program after linking.

The C Source Code Parser analyzes the additional files to extract any global variables that are visible from outside. For each global variable the Parser declares an external variable and creates a default test which is added to an environment named after the corresponding additional file.

By default, any symbols and types that could be exported from the source file under test are declared again in the test script.

Simulated Files

-simulate=<simulated file>{[,<simulated file>]}

This option gives the C Source Code Parser a list of source files to simulate upon execution of the test. List elements are separated by commas and may be specified relatively, in relation to the current directory, or as an absolute path.

The Parser analyzes the simulated files to extract the global variables and functions that are visible from outside. For each file, a DEFINE STUB block, which contains the simulation of the file's external global variables and functions, is generated.

Example:

```
attolstart add.c add.ptu -SIMULATE="source.c" -USE="source.c"
```

By default, if **attolstart** detects that a function is not used, the stub block is commented out in the code and no simulation instructions are generated. If this function is needed, uncomment the stub block in the code so that the stub can be generated.

Static Metrics

-metrics=<output directory>

This option generates static metrics for the specified source files. Resulting **.met** static metric files are produced in specified <output directory>. When the **-metrics** option is used, no other action is performed by the Source Code Parser.

-one_level_metrics: For use with the **-metrics** option only. When the **-metrics** option is used, by default, the calculation of static metrics is applied to the specified source files, and extended to any files included in those source files. Use the **-one_level_metrics** option to ignore included files when calculating static metrics.

-restrict_dir_metrics<directory>: For use with the **-metrics** option only. Use the the **-restrict_dir_metrics** option to calculate static metrics of the specified source files, extended to any files included in those source files but limited to those files located in the specified <directory>.

Other Option

-studio_log: This option is for internal usage only.

Return Codes

After execution, the program exits with the following return codes

Code	Description
0	End of execution with no errors
3	End of execution with one or more warning messages
5	End of execution with one or more errors
7	End of execution because of fatal error
9	End of execution because of internal error

All messages are sent to the standard error output device.

C Test Script Compiler - attolpreproC

Purpose

The C Test Script Compiler tool pre-processes a **.ptu** test script and converts it into a native source code test harness.

Syntax

```
attolpreproC <test_script> <generated_file> [ <target_directory> ] { [ <options> ] }
```

```
attolpreproC @<option_file>
```

where:

1. <test_script> is a required parameter that specifies the name of the test program to be generated.

<generated_file> is a required parameter that specifies the name of the test harness that is generated from the test script.

<target_directory> is an optional parameter. It specifies the location where Component Testing for C will generate the trace file. By default, the trace file is generated in the workspace directory.

<options> is a set of optional command line parameters as specified in the following section.

<option_file> is the name of a plain-text file containing a list of options.

Description

Options can be in any order. They may be upper or lowercase and can be abbreviated to their shortest unambiguous number of characters.

Source File Under Test

-source= *<source file>*

This option specifies the name of the source file being tested, allowing the Test Script Compiler to:

- Maintain the source file name in the table of correspondence files so that the Test Report Generator can display this name in the header of the results obtained file.
- Establish the list of include files in the tested source file.

The name of the tested source file may be specified with a relative or absolute directory in a syntax recognized by the operating system, or, in UNIX, by an environment variable.

By default, the list of include files in the tested source file and the source file name are not displayed in the Results Obtained file.

Condition Definition

-define= *<ident>[= <value>]* {[, *<ident>[= <value>]*}

This option specifies conditions to be applied when the Test Script Compiler starts. These conditions allow you to define C symbols that apply conditions to the generation of any **IF ... ELSE ... END IF** blocks in the test script.

If the option is used with one of the conditions specified in the **IF** instruction, the **IF ... ELSE** block (if **ELSE** is present) or the **ELSE ... END IF** block (if **ELSE** is not present) is analyzed and generated. The **ELSE ... END IF** block is eliminated.

If the option is not used or if none of the conditions specified in the **IF** instruction are satisfied, the **ELSE ... END IF** block is analyzed and generated.

All symbols defined by this option are equivalent to the following line in C

```
-define <ident> [<value>]
```

By default, the **ELSE ... END IF** blocks are analyzed and generated.

Specifying Tests, Families, and Services

-test= <test>{[, <test>]} | **-extest=** <test>{[, <test>]}

This option specifies a list of tests to be executed.

Use **-test** to only generate the source code related to the specified tests, and **-extest** to specify the tests for which you do not want to generate source code.

Both **-test** and **-extest** cannot be used together.

By default, all tests are selected.

-family=<family>{, <family>} | **-exfamily=**<family>{, <family>}

Use **-family** to only generate the source code related to the specified families, and **-exfamily** to specify the families for which you do not want to generate source code.

Both **-family** and **-exfamily** cannot be used together.

By default, all families are selected.

-service=<service>{[, <service>]} | **-exservice=** <service>{[, <service>]}

Use **-service** to only generate the source code related to the specified services, and **-exservice** family to specify the services for which you do not want to generate source code.

Both **-service** and **-exservice** cannot be used together.

By default, all services are selected.

-STD_DEFINE= <TDP definition file>

This option allows you to specify a TDP definition file that enables compatibility with the compiler. Typically, this is <targetPath>/ana/atus_c.def

-RENAME= <stub rename file>

Use this option to stub methods that are located within the source code. <stub rename file> specifies the name of a generated file that contains the stub renaming options for the C/C++ Instrumentor. You can pass this filename as a parameter for the C/C++ Instrumentor (attolccp or attolcc4/attolcc1) with the syntax **attolccp (or attolcc4)@ <stub rename file>** .

Test Script Parsing

-fast | **-nofast**

The **-fast** option tells the C Test Script Compiler to analyze only those tests that you want to generate. This setting considerably speeds up the Test Script Compiler when you use the **-service**, **-exservice**, **-family**, **-exfamily**, **-test**, or **-extest** options.

The **-fast** option is selected by default.

If you want a full test script analysis, this option can be de-selected using the **-nofast** option.

-noanalyse

This option disables the native language parser.

By default, native language lines are analyzed. This option enables you to disable this parsing.

-noedit

This option limits unit test code generation to the initialization of variables, making it possible to generate tighter code for special purposes such as debugging. If you specify the **-noedit** option, you cannot generate a test report.

By default, code is generated normally.

-nopath

Use this option if you do not want to generate long pathnames on the open and close execution trace file call, and on the Target Deployment Port header file include directive. This can be useful, for example, to preserve memory on embedded targets.

By default, full pathnames are generated.

-nosimulation

This option determines the conditional generation related to simulation in the source file generated by the Test Script Compiler. Blocks delimited by the keywords **SIMUL ... ELSE_SIMUL ... END SIMUL** can be included in the test scripts.

See [SIMUL on page 734](#) blocks in the [C Test Script Language Reference on page 716](#).

-restriction=ANSI | KR | NOEXCEPTION | NOIMAGE | NOPOS | SEPAR

This option lets you modify the behavior of test script parser.

1. **ANSI** enables C native code to be analyzed according to the ANSI standard (C only).

KR enables C native code to be analyzed according to the KERNIGHAN & RITCHIE version 2 standard (C only).

1. **noexception**: tells the Test Script Compiler to skip EXCEPTION blocks when generating a test harness. This allows the use of compilers that do not implement exception handling. By default, EXCEPTION blocks are generated in the test program.

noimage: initialization, expected, and obtained values display as integers instead of character strings. By default, reports are generated with IMAGE attributes.

npos: modifies the way enumerated variables are displayed in the test report by not generating any POS or IMAGE attributes. Initialization and expected values are displayed as they are written in the test script, whereas obtained values do not appear (although they are tested). Use this option to save memory on restricted target platforms. By default, reports are is generated with IMAGE attributes.

separ: modifies the format of the generated test program. In place of a main procedure including a sub-procedure for each service, the C Test Script Compiler generates one separate procedure for each service. With this restriction, the Test Script Compiler generates several compilation units and avoids overflow errors on compilation. By default, code is generated normally.

Several **-restriction** options can be used on the same command line. The ANSI and KR parameters, however, cannot be used together.

Other options

-STUDIO_LOG

This option is for internal usage only.

-TARGET

This option is deprecated.

Using an Option File

@<parameter file>

This syntax allows the compiler to pass options to the preprocessor through a file. The parameter file name can be written in absolute or relative format.

The format of the file must follow these rules:

1. One or more options can occur per line.

Each option must follow the same syntax as the command line version, with the character that usually introduces the option being '-' under UNIX and '/' under Windows.

You may not use both an option file and command line options.

By default, no file is taken into account.

If the option file is not found, a fatal error is generated and the preprocessor stops.

Examples

```
attolprepro C add.ptu Tadd.cpp -service=add -test=1,2,3 -family=nominal
```

```
attolprepro CPP @add.opt
```

In this case, the parameter file **add.opt** would contain:

```
add.ptu Tadd.cpp
```

-service=add

-test=1,2,3

-family=nominal

Return Codes

After execution, the program exits with the following return codes

Code	Description
0	End of execution with no errors
3	End of execution with one or more warning messages
5	End of execution with one or more errors
7	End of execution because of fatal error
9	End of execution because of internal error

These codes help you decide on a course of action once the Test Script Compiler has finished test execution. For example, if the return code in the command file shows there have been incorrect tests, you can save certain files in order to analyze them later.

All messages are sent to the standard error output device.

Related Topics

[C++ Test Script Compiler on page 1130](#) | [System Testing Script Compiler on page 1112](#)

C Test Report Generator - attolpostpro

Purpose

The C Test Report Generator processes a trace file produced during test execution, and generates a test report.

Syntax

```
attolpostpro <trace_filename> <report_filename> [<options>]
```

```
attolpostpro @ <option file>
```

where:

1. `<trace_filename>` specifies the root (filename without extension) of the trace file that is generated when the program runs.

`<report_filename>` specifies the name of the **.rod** compact report file produced by the Test Report Generator.

`<options>` can be any of the optional parameters specified below.

`<option_file>` is the name of a plain-text file containing a list of options.

Description

The Test Report Generator uses `<trace_filename>` to find the names of both the **.rio** trace file and the **.tdc** table of correspondence file that are generated by the Test Script Compiler.

If `<report_filename>` is provided without an extension, the Test Report Generator attaches **.rod**.

If either `<trace_filename>` or `<report_filename>` are omitted, the Test Report Generator produces an error message and terminates.

If the either `<trace_filename>` **.rio** or `<trace_filename>` **.tdc** do not exist, cannot be read, or contain synchronization errors, the Test Report Generator produces an error message and terminates.

If the Test Report Generator cannot create the **.rod** compact report file, generation of the report is terminated. If the file already exists, the newly generated file replaces the existing report.

The **.rod** compact report file is an intermediate low-footprint format that can be stored on remote targets. The **.rod** files must be converted to the **.xrd** report file format to be displayed by the HCL OneTest™ Embedded GUI with the **rod2xrd** command line tool.

Options

Options can be in any order. They may be upper or lowercase and can be abbreviated to their shortest unambiguous number of characters.

-cio= *<coverage result file>*

This option allows you to insert coverage results in the report file. This option must be used only in conjunction with the Code Coverage feature.

-compare[=strict]

This option lets you compare the results from two test runs. A trace file generated during the first run has a **.rio** extension, and the one generated during the second run has a **.ri2** extension.

When making a comparison, the Test Report Generator generates the test report from:

1. The **.tdc** table of correspondence file

The **.rio** trace file generated during the first run

The **.ri2** trace file generated during the second run

The same root name is used for the names of the three files.

When comparing values, a variable will only be deemed *correct* if the two obtained values are the same as the expected value, or within the specified validity interval for that variable. With the **compare=strict** option, the two results must have the same value.

-ra[=test | error]

This option specifies the form of the output report generated by the Test Report Generator.

Use **-ra** with no parameter, to display ALL test variables and mark any variables that are incorrect for a given test. This option is used by default.

Use the **-ra=test** option to display ALL test variables, with incorrect variables marked. This option provides a comprehensive display of variables for an incorrect test, which can prove useful in a complex test environment.

Use **-ra=error** to display only erroneous test variables.

For both **-ra=test** and **-ra=error**, if no errors are detected in the test, only general information about the test is produced.

-va=eval | noeval | combined

This option lets you specify the way in which initial and expected values of each variable are displayed in the test report.

Use **-va=eval** if you want the test report to show the initial and expected value of each variable evaluated during execution of the test. This is only relevant for variables whose initial or expected value expressions are not reducible in the test script.

Note: For arrays and structures in which one of the members is an array, the initial values are not evaluated. For the expected values, only incorrect elements are evaluated.

Use **-va=noeval** if you want the test report to show the initial and expected values described in the test script.

The **-va=combined** option combines both **eval** and **noeval** parameters. For each variable, the Report Generator includes the initial and expected values described in the test script, as well as the initial and expected values evaluated during execution, if these values differ.

By default, the **-va=eval** parameter is used.

-studio_log

This option is for internal usage only.

Return Codes

After execution, the program exits with the following return codes

Code	Description
0	End of execution with no errors
3	End of execution with one or more warning messages
5	End of execution with one or more errors
7	End of execution because of fatal error
9	End of execution because of internal error

All messages are sent to the standard error output device.

Component testing for C++**C++ Test Report Generator - atopospro**

Purpose

The C++ Test Report Generator processes a trace file produced during test execution, and generates a test report that can be viewed in the GUI.

Syntax

atopospro -ots {<ots files>} **-tdf** <tdf file> **-xrd** <xrd file>

where:

- <ots files> is a list of **.ots** intermediate files generated by the [C++ Test Script Compiler on page 1130](#).
- <tdf file> is the **.tdf** dynamic trace file generated during the execution of the application under test.
- <xrd file> is the **.xrd** report file to be generated by the Report Generator.

Example

```
atopospro -ots script.ots contract1.ots contract2.ots -tdf bar.tdf -xrd report.xrd
```

Return Codes

After execution, the program exits with the following return codes

Code	Description
0	End of execution with no errors
1	Abnormal termination

All messages are sent to the standard error output device.

C++ Test Script Compiler - atoprepro

Purpose

The C++ Test Script Compiler compiles the **.otd** C++ Test Driver Script and **.otc** C++ Contract Check Scripts into C++ source code.

Syntax

```
atoprepro [<OTD Script>][<OTC Scripts>] -G C++ -O <cpp file> -OTI <oti file> -TDF <tdf file>
```

where:

- <otd script> is an **.otd** C++ Test Driver Script file.
- <otc scripts> is a set of one or more **.otc C++** Contract Check Script files.
- <cpp file> is the name of the **.ccor.cpp** source file to be generated by Component Testing for C++ and linked to the application under test.
- <oti file> is the name of the **.oti** instrumentation file to be generated. This file is used by the [C++ Instrumentor on page 1075](#).
- <tdf file> is the **.tdf** dynamic trace file to be generated during the execution of the application under test.

Options

The C++ Source Code Parser supports the following options:

-E <number of errors>

Specifies the maximum number of error messages that can be displayed by the C++ Test Script Compiler. The default value is 30.

-NODLINE

Deactivates the generation of *#line* statements. This can be useful in environments where the generated source code cannot use the *#line* mechanism. By default *#line* statements are generated.

-NOPATH

This option tells the C++ Test Script Compiler not to use the full path to the TDP from the **\$ATLTGT** environment variable before the name of **TP.h** in the *#include* directive.

This option is useful for embedded targets when compilation of the generated source does not occur on the same host as the C++ test compilation.

-STUDIO_LOG

This option is for internal usage only.

Example

```
atoprepro script.otd contract1.otc contract2.otc -G C++ -O app.cc -OTI foo.oti -TDF bar.tdf -E 60
```

Return Codes

After execution, the program exits with the following return codes

Code	Description
0	End of execution with no errors
1	Abnormal termination

All messages are sent to the standard error output device.

C++ Source Code Parser - atostart

Purpose

The C++ Source Code Parser takes a set of C++ source files containing classes to generate template **.otd** C++ Test Driver Scripts and **.otc** C++ Contract Check Scripts to fully cover the application under test.

Syntax

```
atostart {[-i] <source file> <options>
```

```
atostart {<source file>} -metrics <options>
```

where:

- *<tested file>* is the list of files containing classes to be tested. If no class is specified with the option **-test_class**, the tested classes will be either the classes defined in a file under test, or the classes for which a method is defined in a tested file.
- *<options>* is a series of command line options. See the section **Options**.

If a tested file is specified with option **-i**, this file will be *included* by the generated **.otd** script. As a consequence everything defined in this tested file will be available in the script (especially types, classes, static variables, and functions). This option is ignored if you choose not to generate an **.otd** C++ Test Driver Script.

Description

The tested files and additional files (see option **-integrate**) are parsed by the integrated C++ analyzer. A candidate classes list is automatically deduced from the content of tested files (this list can be viewed in the header of the generated **.otd** and **.otc** scripts). If no **-test_class** or **-do_not_test_class** option is used, then all the candidate classes will have generated code to test them.

The C++ Source Code Parser generates only one **.otd** C++ Test Driver Script that contains all classes under test. It also generates two files associated to this test script: a **.dcl** declaration file (declaring and including every resource needed to compile the test script) and a **.stb** stub file (containing stub declarations deduced from used but not defined entities found in the parsed files).

The C++ Source Code Parser generates one **.otc** C++ Contract Check Script per encountered **.h** file defining a class.

When the **-metrics** option is specified, the Source Code Parser produces static metrics for the specified source files.

Options

The C++ Source Code Parser supports the following options:

{-integrate <additional file>}

Specifies additional files or directories to be analyzed. These files do not contain any classes under test, but they do contain code which is to be linked with the tested application. Basically, this option tells the C++ Source Code Parser which files not to stub.

Three types of additional files or directories are supported :

- **body files:** Only the entities defined within the file are considered defined.
- **header files:** Every declaration within the file is considered as having a matching definition in a non-provided body file or in a library. Use additional header files when linking to code for which the source is not available.
- **directories:** Every declaration found in a file belonging to an additional directory is considered as having a matching definition in a non-provided body file or in a library (an additional directory can be viewed as a collection of additional header files).

Note A header file is recognized as such from its content, and not from its extension. A header file does not contain any definition, other than inline functions, and template functions, or else it is considered as a body file.

This option is ignored when no **.otd** generation is required. This option can be used more than once to specify multiple files.

{-insert|-i <included file>}

Specifies included files. These are source files which, instead of being compiled separately during the test, are included and compiled with the **.otd** test driver script.

In most cases, you do not have to specify files to be included. Header files are automatically considered as included files, even if they are not specified as such.

Source files under test should be specified as included when:

- The file contains the class definition of a class you want to test
- A function or a variable definition depends upon a type which is defined in the file under test itself
- You need access in your test script to a static variable or function, defined in the file under test

This option is ignored when no **.otd** generation is required. This option can be used more than once to specify multiple files.

-o|-otd <test script>

Specifies the name of the generated **.otd** script. Two associated files are also generated with the same name, but with extension **.dcl** and **.stb**. If the filename extension of *<test script>* is not **.otd**, then a warning is issued.

This option is ignored when no **.otd** generation is required.

-otc <test script>

Specifies the name of the generated **.otc** script. If the filename extension of *<test script>* is not **.otc**, then a warning is issued.

This option is ignored when no **.otc** generation is required.

-otcdir <OTC directory>

Specifies the directory where **.otc** files are to be generated.

This option is ignored when no **.otc** generation is required.

-opp <compiler option file>

Specifies the name of the Target Deployment Port C++ parser option file. This file is searched for in **/ana** subdirectory of the current Target Deployment Port (see [ATLTGT environment variable on page 1035](#)), and should not include any path.

If this option is not provided, the default filename **atl.opp** will be searched for.

-hpp *<compiler configuration file>*

Specifies the name of the Target Deployment Port C++ parser configuration file. This file is searched for in **/ana** subdirectory of the current Target Deployment Port (see [ATLTGT environment variable on page 1035](#)), and should not include any path.

If this option is not provided, the default filename **atl.hpp** will be searched for.

{-test_class|-tc *<class under test>*

Specifies the classes to be explicitly tested. The classes must belong to the candidate classes. This option cannot be used simultaneously with the options **-do_not_test_class** (**-dtc**).

This option can be used more than once to specify multiple classes.

{-do_not_test_class|-dtc *<excluded class>*

Specifies the classes, among the candidate classes, which should not be tested. This option cannot be used simultaneously with the options **-test_class** (**-tc**).

This option can be used more than once to specify multiple classes.

-test_struct

Specifies whether structs and unions should be treated as classes, and therefore should be considered as potential tested classes. This option is not significant when **-test_class** option is used (you can specify structs or unions as classes to be tested).

-test_method|-tm *<method name>* *<line>*

Specifies the methods to be explicitly tested. *<method_name>* is the fully qualified name of the method (fully qualified class name with method name, without return values or parameters). *<line>* is the line number of the method. For example:

```
-test_method "class::method1" "50" "class::method2" "70"
```

This option can be used more than once to specify multiple methods.

-test_class_prefix *<prefix>*

Specifies the prefix used to name the generated test classes. By default, **atostart** uses 'Test'.

`-test_each_instance`

By default, a template class is tested as a generic template class. Use this option if you want to generate a specific test for each found instance of a template class.

`{-force_template <template instance>}`

This option forces the instantiation of the specified templates classes. Use it if no automatic template instantiation occurs while parsing the code. This option is useful only in conjunction with **-test_each_instance** option.

This option can be used more than once to specify multiple templates.

`-overwrite`

By default, the Test Template Generator creates a backup file of every file it overwrites. Use this option if you really intend to overwrite these files without backing up them.

`-ignore_line_directives`

Although the C++ test generator includes a preprocessor and can parse unpreprocessed source code, preprocessed code is also accepted. In the case of preprocessed source code, the test generator tries to detect included header files by looking at `#line` directives. In some cases, such as code generated by a code generator (for example `lex` or `yacc`), relying on `#line` directives does not produce effective test code. In this case, use **-ignore_line_directives** to have the generator ignore `#line` directives found in the source code.

Note In most cases, this option has no effect because unpreprocessed code does not usually contain `#line` directives.

This option is ignored when no **.otd** generation is required.

`{-I<include directory>}`

This option specifies directories where included files are to be searched for. You can use the option **-I-** to introduce the system *includes*: only directories specified after **-I-** will be looked up when the include directives use angular brackets (**#include <...>**).

This option can be used more than once to specify multiple directories.

`{-D <macro>[=<value>]}`

This option adds a predefinition for `<macro>` to `<value>`.

This option can be used more than once to specify multiple macros.

-E

This options generates preprocessing output to standard output. This option is mainly for debugging purpose.

-include={relative|absolute|copy}

This option specifies how *#include* directives should be generated in the test script. When relative is chosen, includes use relative path to the directory where the generated script is put. When absolute is chosen, absolute paths are generated. When copy is chosen, the way files are included in the test script is the same as they are included in the tested files, you should in this case ensure that the test script is generated in the same directory than the source files.

This option is ignored when no **.otd** generation is required.

-no_otc

This option deactivates **.otc** script generation. Use this option if you only want an **.otd** test driver script.

-no_otd

This option deactivates **.otd** script generation. Use this option if you only want an **.otc** Contract-Check script.

Note If no candidate class is found, nothing will be generated.

-studio_log

This option is for internal usage only.

Static Metrics Options

-metrics <output directory>

Generates static metrics for the specified source files. Resulting **.met** static metric files are produced in specified <output directory>. When the **-metrics** option is used, no other action is performed by the Source Code Parser.

-one_level_metrics

For use with the **-metrics** option only. When the **-metrics** option is used, by default, the calculation of static metrics is applied to the specified source files, and extended to any files included in those source files. Use the **-one_level_metrics** option to ignore included files when calculating static metrics.

-restrict_dir_metrics <directory>

For use with the **-metrics** option only. Use the the **-restrict_dir_metrics** option to calculate static metrics of the specified source files, extended to any files included in those source files but limited to those files located in the specified <directory>.

This option can be used more than once to specify multiple directories.

Return Codes

After execution, the program exits with the following return codes

Code	Description
0	End of execution with no errors
1	End of execution with error

All messages are sent to the standard error output device.

Component testing for Ada

Ada Source Code Parser - attolstartADA

Purpose

When creating a new Component Testing test campaign for Ada, the Ada Source Code Parser creates an Ada test script template based on the analysis of the source code under test.

When the **-metrics** option is specified, the Source Code Parser produces static metrics for the specified source files.

Syntax

attolstart ADA *<source_under_test >* *<test_script>* [*<-option>*]

attolstart ADA @ *<option file >*

where:

1. *<source under test>* this required parameter is the name of the source file to be tested.

<test script> is the name of the test script that is generated

<options> is a list of options as defined below.

<option file> is the name of a plain-text file containing a list of options.

Description

The Ada Source Code Parser analyzes the source file to be tested in order to extract global variables and testable functions.

Each global variable is automatically declared as external, if this has not already been done at the beginning of the test script. Then, an environment is created to contain all these variables with default tests. This environment has the name of the file (without the extension).

For each function under test, the generator creates a SERVICE which contains the Ada declaration of the variables to use as parameters of the function.

Parameters passed by reference are declared according to the following rule:

1. *char** *<param>* causes the generation of *char <param>[200]*
*<type>** *<param>* causes the generation of *<type> <param>* passing by reference

It is sometimes necessary to modify this declaration if it is unsuitable for the tested function, where *<type>** *<param>* can entail the following declarations:

1. *<type>** *<param>* passing-by-value,
<type> <param> passing-by-reference,
<type> <param>[10] passing-by-reference.

File names can be related or absolute.

If the generated file name does not have an extension, the Ada Source Code Parser automatically attaches **.ptu** or the extension specified by the ATTOLPTU environment variable. This name may be specified relatively, in relation to the current directory, or as an absolute path.

If the test script cannot be created, the Ada Source Code Parser issues a fatal error and stops.

If the test script already exists, the previous version is saved under the name *<generated test script>_bck* and a warning message is generated.

Options

Options can be in any order. They may be upper or lowercase and can be abbreviated to their shortest unambiguous number of characters.

Static Metrics

-metrics=<output directory>

Generates static metrics for the specified source files. Resulting **.met** static metric files are produced in specified *<output directory>*. When the **-metrics** option is used, no other action is performed by the Source Code Parser.

Included Files

-insert

With this option the source file under test is included into the test script with an **#include** directive, ensuring that all the internal functions and variables (declared static) are visible to the test script. The Ada Source Code Parser adds the **#include** directive before the BEGIN instruction and after any **#includes** added by the **-use** option.

Additional Files

-integrate= <additional file>{[, <additional file>]}

This option provides a list of additional source files whose objects are *integrated* into the test program after linking.

The Ada Source Code Parser analyzes the additional files to extract any global variables that are visible from outside. For each global variable the Parser declares an external variable and creates a default test which is added to an environment named after the corresponding additional file.

By default, any symbols and types that could be exported from the source file under test are declared again in the test script.

Simulated Files

-simulate= <simulated file>{[, <simulated file>]}

This option gives the Ada Source Code Parser a list of source files to simulate upon execution of the test. List elements are separated by commas and may be specified relatively, in relation to the current directory, or as an absolute path.

The Parser analyzes the simulated files to extract the global variables and functions that are visible from outside. For each file, a DEFINE STUB block, which contains the simulation of the file's external global variables and functions, is generated.

By default, no simulation instructions are generated.

Header Files

-use= <file used>{[, <file used>]}

This option gives the Ada Source Code Parser a list of header files to include in the test script before the BEGIN instruction. This avoids declaring variables or functions that have already been declared in an Ada header file of the application under test.

The Ada Source Code Parser adds the #include directive before the BEGIN instruction. Then, for each file, an environment is created, containing all variables with a default test. This environment has the name of the included file.

By default, no files are included in the test script.

Other options

-studio_log

This option is for internal usage only.

Return Codes

After execution, the program exits with the following return codes

Code	Description
0	End of execution with no errors
3	End of execution with one or more warning messages
5	End of execution with one or more errors
7	End of execution because of fatal error
9	End of execution because of internal error

All messages are sent to the standard error output device.

Ada Test Script Compiler - attolpreproADA

Purpose

The Ada Test Script Compiler tool pre-processes the Ada test script and converts it into a native source test harness.

Syntax

```
attolpreproADA <test_script> <generated_file> [ <target_directory> ] { { <options> } }
```

```
attolpreproADA @<option_file>
```

where:

- <test_script> is a required parameter that specifies the name of the test program to be generated.
- <generated_file> is a required parameter that specifies the name of the test harness that is generated from the test script.
- <target_directory> is an optional parameter. It specifies the location where Component Testing for Ada will generate the trace file. By default, the trace file is generated in the workspace directory.
- <options> is a set of optional command line parameters as specified in the following section.
- <option_file> is the name of a plain-text file containing a list of options.

Description

Options can be in any order. They may be upper or lowercase and can be abbreviated to their shortest unambiguous number of characters.

The Ada Test Script Compiler produces a series of **.tdc**, **.ddt** and **.mdt** files, which are required by the Ada Test Report Generator.

Source File Under Test

-source= *<source file>*

This option specifies the name of the source file being tested, allowing the Test Script Compiler to:

- Maintain the source file name in the table of correspondence files so that the Test Report Generator can display this name in the header of the results obtained file.

Establish the list of include files in the tested source file.

The name of the tested source file may be specified with a relative or absolute directory in a syntax recognized by the operating system, or, in UNIX, by an environment variable.

By default, the list of include files in the tested source file and the source file name are not displayed in the Results Obtained file.

Condition Definition

-define= *<ident>[= <value>] {[, <ident>[= <value>]}*

This option specifies conditions to be applied when the Test Script Compiler starts. These conditions allow conditional test harness generation as well as identifier definition for Ada.

The identifiers specified by the **-define** option apply conditions to the generation of any **IF ... ELSE ... END IF** blocks in the test script.

If the option is used with one of the conditions specified in the IF instruction, the **IF ... ELSE** block (if **ELSE** is present) or the **ELSE ... END IF** block (if **ELSE** is not present) is analyzed and generated. The **ELSE ... END IF** block is eliminated.

If the option is not used or if none of the conditions specified in the IF instruction are satisfied, the **ELSE ... END IF** block is analyzed and generated.

All symbols defined by this option are equivalent to the following line in Ada

```
-define <ident> [<value>]
```

By default, the **ELSE ... END IF** blocks are analyzed and generated.

Specifying Tests, Families, and Services

-test= *<test>{[, <test>]}* | **-extest=** *<test>{[, <test>]}*

This option specifies a list of tests to be executed.

Use **-test** to only generate the source code related to the specified tests, and **-extest** to specify the tests for which you do not want to generate source code.

Both **-test** and **-extest** cannot be used together.

By default, all tests are selected.

-family=<family>{, <family>} | **-exfamily**=<family>{, <family>}

Use **-family** to only generate the source code related to the specified families, and **-exfamily** to specify the families for which you do not want to generate source code.

Both **-family** and **-exfamily** cannot be used together.

By default, all families are selected.

-service=<service>{[, <service>]} | **-exservice**= <service>{[, <service>]}

Use **-service** to only generate the source code related to the specified services, and **-exservice** family to specify the services for which you do not want to generate source code.

Both **-service** and **-exservice** cannot be used together.

By default, all services are selected.

Test Script Parsing

-fast | **-nofast**

The **-fast** option tells the Test Script Compiler to analyze only those tests that you want to generate. This setting considerably speeds up the Test Script Compiler when you use the **-service**, **-exservice**, **-family**, **-exfamily**, **-test**, or **-extest** options.

The **-fast** option is selected by default.

If you want a full test script analysis, this option can be de-selected using the **-nofast** option.

-noanalyse

This option disables the native language parser.

By default, native language lines are analyzed. This option enables you to disable this parsing.

-noedit

This option limits unit test code generation to the initialization of variables, making it possible to generate tighter code for special purposes such as debugging. If you specify the **-noedit** option, you cannot generate a test report.

By default, code is generated normally.

-nopath

Use this option if you do not want to generate long pathnames on the open and close execution trace file call, and on the Target Deployment Port header file include directive. This can be useful, for example, to preserve memory on embedded targets.

By default, full pathnames are generated.

-nosimulation

This option determines the conditional generation related to simulation in the source file generated by the Test Script Compiler. Blocks delimited by the keywords **SIMUL ... ELSE_SIMUL ... END SIMUL** can be included in the test scripts.

See [SIMUL on page 734](#) blocks in the [Ada Test Script Language on page 716](#).

-restriction=ANSI | KR | NOEXCEPTION | NOIMAGE | NOPOS | SEPAR | NOALLOC

This option lets you modify the behavior of test script parser.

- **noexception:** tells the Test Script Compiler to skip EXCEPTION blocks when generating a test harness. This allows the use of compilers that do not implement exception handling. By default, EXCEPTION blocks are generated in the test program.
- **noimage:** initialization, expected, and obtained values display as integers instead of character strings. By default, reports are generated with IMAGE attributes.
- **npos:** modifies the way enumerated variables are displayed in the test report by not generating any POS or IMAGE attributes. Initialization and expected values are displayed as they are written in the test script, whereas obtained values do not appear (although they are tested). Use this option to save memory on restricted target platforms. By default, reports are is generated with IMAGE attributes.
- **separ:** modifies the format of the generated test program. In place of a main procedure including a sub-procedure for each service, the Test Script Compiler generates one separate procedure for each service. With this restriction, the Test Script Compiler generates several compilation units and avoids overflow errors on compilation. By default, code is generated normally.
- **noalloc:** disables memory allocation for non-constraint types. When using this option, you must use the pragma ATTOL_RANGE to specify an alternate memory usage method.

Several **-restriction** options can be used on the same command line. The ANSI and KR parameters, however, cannot be used together.

Other options

-studio_log

This option is for internal usage only.

-target

This option is deprecated.

Using an Option File

@<parameter file>

This syntax allows the compiler to pass options to the preprocessor through a file. The parameter file name can be written in absolute or relative format.

The format of the file must follow these rules:

- One or more options can occur per line.

Each option must follow the same syntax as the command line version, with the character that usually introduces the option being '-' under UNIX and '/' under Windows.

You may not use both an option file and command line options.

By default, no file is taken into account.

If the option file is not found, a fatal error is generated and the preprocessor stops.

Examples

```
attolprepro C add.ptu Tadd.cpp -service=add -test=1,2,3 -family=nominal
```

```
attolprepro CPP @add.opt
```

In this case, the parameter file **add.opt** would contains:

```
add.ptu Tadd.cpp
```

```
-service=add
```

```
-test=1,2,3
```

```
-family=nominal
```

Return Codes

After execution, the program exits with the following return codes

Code	Description
0	End of execution with no errors
3	End of execution with one or more warning messages

- 5 End of execution with one or more errors
- 7 End of execution because of fatal error
- 9 End of execution because of internal error

These codes help you decide on a course of action once the Test Script Compiler has finished test execution. For example, if the return code in the command file shows there have been incorrect tests, you can save certain files in order to analyze them later.

All messages are sent to the standard error output device.

Related Topics

[C++ Test Script Compiler on page 1130](#) | [System Testing Script Compiler on page 1112](#)

attolpostproada

Ada Test Script Compiler - attolpostproada

Purpose

The Ada Test Report Generator processes a trace file produced during test execution, and generates a test report.

Syntax

attolpostproada *<trace_filename>* *<report_filename>* [*<options>*]

attolpostproada @ *<option file>*

where:

- *<trace_filename>* specifies the root (filename without extension) of the trace file that is generated when the program runs.
- <report_filename>* specifies the name of the **.rod** compact report file produced by the Test Report Generator.
- <options>* can be any of the optional parameters specified below.
- <option_file>* is the name of a plain-text file containing a list of options.

Description

The Test Report Generator uses *<trace_filename>* to find the names of both the **.rio** trace file and the **.tdc**, **.ddt** and **.mdt** files that are generated by the Test Script Compiler.

If *<report_filename>* is provided without an extension, the Test Report Generator attaches **.rod**.

If either `<trace_filename>` or `<report_filename>` are omitted, the Test Report Generator produces an error message and terminates.

If any of the required files (`.rio`, `.tdc`, `.ddt` or `.mdt`) do not exist, cannot be read, or contain synchronization errors, the Test Report Generator produces an error message and terminates.

If the Test Report Generator cannot create the `.rod` compact report file, generation of the report is terminated. If the file already exists, the newly generated file replaces the existing report.

The `.rod` compact report file is an intermediate low-footprint format that can be stored on remote targets. The `.rod` files must be converted to the `.xrd` report file format to be displayed by the HCL OneTest™ EmbeddedGUI with the `rod2xrd` command line tool.

Options

Options can be in any order. They may be upper or lowercase and can be abbreviated to their shortest unambiguous number of characters.

-cio= *<coverage result file>*

This option allows you to insert coverage results in the report file. This option must be used only in conjunction with the Code Coverage feature.

-ra[=test | error]

This option specifies the form of the output report generated by the Test Report Generator.

Use **-ra** with no parameter, to display ALL test variables and mark any variables that are incorrect for a given test. This option is used by default.

Use the **-ra=test** option to display ALL test variables, with incorrect variables marked. This option provides a comprehensive display of variables for an incorrect test, which can prove useful in a complex test environment.

Use **-ra=error** to display only erroneous test variables.

For both **-ra=test** and **-ra=error**, if no errors are detected in the test, only general information about the test is produced.

-va=eval | noeval | combined

This option lets you specify the way in which initial and expected values of each variable are displayed in the test report.

Use **-va=eval** if you want the test report to show the initial and expected value of each variable evaluated during execution of the test. This is only relevant for variables whose initial or expected value expressions are not reducible in the test script.

Note: For arrays and structures in which one of the members is an array, the initial values are not evaluated. For the expected values, only incorrect elements are evaluated.

Use **-va=noeval** if you want the test report to show the initial and expected values described in the test script.

The **-va=combined** option combines both **eval** and **noeval** parameters. For each variable, the Report Generator includes the initial and expected values described in the test script, as well as the initial and expected values evaluated during execution, if these values differ.

By default, the **-va=eval** parameter is used.

-studio_log

This option is for internal usage only.

Return Codes

After execution, the program exits with the following return codes

Code	Description
0	End of execution with no errors
3	End of execution with one or more warning messages
5	End of execution with one or more errors
7	End of execution because of fatal error
9	End of execution because of internal error

All messages are sent to the standard error output device.

Ada Link File Generator - **attolalk**

Purpose

The Ada Link File Generator (**attolalk**) feature automatically generates link files. It uses file name extensions that you allow or disallow, and on the file list found in the specified directories.

Syntax

```
attolalk [<options>] <link file name> <directory> [<directory> ... <directory>]
```

where:

- *<link file name>* is the name of the generated link file. If **attolalk** cannot write this file a fatal error is generated.
- <directory>* is a directory name. If **attolalk** cannot read file from this directory, a fatal error is generated.
- <options>* is a set of optional command line parameters as specified in the following section.

Description

The Link File Generator requires that the **LD_LIBRARY_PATH** environment variable is set to the **/lib** directory in the product installation directory.

File Extensions

A file extension is a character string of unconstrained positive length (greater than zero). A file name matches an extension if its length is greater than the length of extension, and if the N last characters of the file name are identical to the characters of the extension (N is the length of the extension). For example, **source.ada** matches the **.ada** extension but not **.1.ada** extension.

Permitted and Forbidden Extensions:

Permitted and forbidden file extensions for the Link File Generator are specified by the **ATTOLALK_EXT** and **ATTOLALK_NOEXT** environment variables and are separated by the ':' character on UNIX systems and ';' on Windows.

Example, on UNIX:

```
ATTOLALK_EXT=".ada;.a;.am"
```

```
ATTOLALK_NOEXT=".1.ada"
```

Example, on Windows:

```
ATTOLALK_EXT=".ad6;.adc;.ads;.adb"
```

By default, the allowed extension list is **".ada;.ads;.adb"** and the forbidden extension list is empty. These default values are overwritten by the value of the **ATTOLALK_EXT** variable.

Link File Generation

For each given directory, the contained file name list is loaded. Each file name is compared with the allowed extensions. If a match is found, the file name is compared with forbidden extension. If there is no match, the file is taken as an Ada source file. Each Ada source file is analyzed and may produce one or more lines in the generated link file (with the syntax described above).

Command Line Parameters

Options can be in any order. They may be upper or lowercase and can be abbreviated to their shortest unambiguous number of characters.

-r

Relative paths. With the **-r** option, all filenames are generated with relative paths.

-s

Silent mode. With the **-s** option, only errors are displayed.

-f

Force all Ada files. By default, the Link File Generator only analyzes Ada source files that were changed since the last analysis. Use the **-f** option to force the analysis of all Ada source files, regardless of when they were modified.

Return Codes

After execution, the program exits with the following return codes

Code	Description
0	End of execution with no errors
3	End of execution with one or more warning messages
5	End of execution with one or more errors
7	End of execution because of fatal error
9	End of execution because of internal error

All messages are sent to the standard error output device.

Related Topics

[Environment Variables on page 1035](#) | [Ada Unit Maker on page 1149](#) | [Ada Instrumentor on page 1088](#)

Ada Unit Maker - `attolchop`

Purpose

The Instrumentor generates several compilation units in the same file. Some compilers require a separate file for each compilation unit.

To achieve this, the Ada Unit Maker feature generates one file for each compilation unit found in a specified Ada source file as the `gnatchop` command, provided with the GNAT Ada compiler, does. You can choose the name of the generated files from several naming conventions.

Syntax

```
attolchop [<options>] <source file name> [..<source file name>] [<directory>]
```

where:

- *<source file name>* is the source file name to analyze. If this file cannot be read or contains lexical or syntax errors, a fatal error is generated.
- *<options>* is a set of optional command line parameters as specified in the following section.
- *<directory>* is the optional output directory.

Description

The Ada Unit Maker feature can generate file names for Rational Apex or Gnat naming standards. To choose the naming standard, either set the **ATTOLCHOP** environment variable to **GNAT** or **APEX** or use the **-n** command line parameter. By default, the Ada Unit Maker uses the Gnat naming convention.

Gnat Naming

The full compilation unit name is set to lower case and all dot characters (".") are replaced with hyphens ("-"). The feature appends the **.ads** extension to the name if the unit is an extension or the **.adb** extension if the unit is a body. The Krunch Gnat short name mode is not supported by the Ada Unit Maker. Please refer to your Gnat documentation for further information.

Rational Apex Naming

The full compilation unit name is set to lower case; then the feature appends a **.1.ada** extension to the filename if the unit is a specification, or a **.2.ada** extension if the unit is a body. Please refer to your Apex documentation for further information.

Options

Options can be in any order. They may be upper or lowercase and can be abbreviated to their shortest unambiguous number of characters.

-l

This option must be placed first if it is used. This tells the Ada Unit Maker feature to send the name of the generated file, and no other messages, to the standard output.

-w

Overwrite. By default, the Ada Unit Maker produces an error if a filename already exists. Use the **-w** option to overwrite any existing files.

-v

This option returns the version number of the product.

-n APEX|GNAT

Naming standard. Use the **-n** option to select either the Rational Apex or Gnat naming convention. This parameter overrides the default setting (Gnat) as well as the **ATTOLCHOP** environment variable if set.

Return Codes

After execution, the program exits with the following return codes:

Code	Description
0	End of execution with no errors
3	End of execution with one or more warning messages
5	End of execution with one or more errors
7	End of execution because of a fatal error
9	End of execution because of an internal error

All messages are sent to the standard error output device.

Output window preferences

The general colors and font preferences panel allows you to specify the colors and fonts used in the output window.

This panel opens from menu **Edit > Preferences**. You can choose **Output Window** style or **Output Window Error** style.

Output window/Output window errors

In this panel, you can change the color and the font style used to display the build output messages or the standard error messages in the Output window. This windows opens from the menu **View > Other windowsOutput Window**.

Notices

This document provides information about copyright, trademarks, terms and conditions for the product documentation.

© Copyright IBM Corporation 2000, 2016 / © Copyright HCL Technologies Limited 2016, 2021

This information was developed for products and services offered in the US.

HCL® may not offer the products, services, or features discussed in this document in other countries. Consult your local HCL® representative for information on the products and services currently available in your area. Any reference to an HCL® product, program, or service is not intended to state or imply that only that HCL® product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any HCL® intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-HCL® product, program, or service.

HCL® may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

HCL
330 Potrero Ave.
Sunnyvale, CA 94085
USA
Attention: Office of the General Counsel

For license inquiries regarding double-byte character set (DBCS) information, contact the HCL® Intellectual Property Department in your country or send inquiries, in writing, to:

HCL
330 Potrero Ave.
Sunnyvale, CA 94085
USA
Attention: Office of the General Counsel

HCL TECHNOLOGIES LTD. PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. HCL® may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-HCL® websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this HCL® product and use of those websites is at your own risk.

HCL® may use or distribute any of the information you provide in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

HCL

330 Potrero Ave.

Sunnyvale, CA 94085

USA

Attention: Office of the General Counsel

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by HCL® under terms of the HCL® Customer Agreement, HCL® International Program License Agreement or any equivalent agreement between us.

The performance data discussed herein is presented as derived under specific operating conditions. Actual results may vary.

Information concerning non-HCL® products was obtained from the suppliers of those products, their published announcements or other publicly available sources. HCL® has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-HCL® products. Questions on the capabilities of non-HCL® products should be addressed to the suppliers of those products.

Statements regarding HCL®'s future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to HCL®, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. HCL®, therefore, cannot guarantee or imply reliability,

serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. HCL® shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work must include a copyright notice as follows:

© (your company name) (year).

Portions of this code are derived from HCL Ltd. Sample Programs.

© Copyright HCL Ltd. 2000, 2022.

Trademarks

HCL®, the HCL® logo, and ibm.com® are trademarks or registered trademarks of HCL Technologies Ltd., registered in many jurisdictions worldwide. Other product and service names might be trademarks of HCL® or other companies.

Terms and conditions for product documentation

Permissions for the use of these publications are granted subject to the following terms and conditions.

Applicability

These terms and conditions are in addition to any terms of use for the HCL® website.

Personal use

You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of HCL®.

Commercial use

You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of HCL®.

Rights

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

HCL® reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by HCL®, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

HCL® MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

Index

Numerics

- 2D and 3D charts
 - generating 924
 - viewing 938

A

- Activating runtime analysis 50
- adding to data dictionary 201
- additional software 15
- administrator 15
- Analyzing source code 47

B

- Build
 - configuration settings 947

C

- call graph
 - preferences 939
- changing settings
 - code coverage 63
 - code review 91
 - memory profiling 74
 - runtime tracing 86, 89
- code coverage
 - configuration settings 63
- code review
 - configuration settings 91
 - overview 90
- command line
 - test suites 213
- configuration settings
 - build 947
 - target deployment port 944
- configurations
 - creating 209
 - switching 210, 661
- configure
 - Rational
 - Quality Manager 30
- configuring 23
- create
 - stub 222
- Create
 - Test case 195
- creating
 - data pools 202
 - test configurations 209
 - test suites 207
- Cygwin 17

D

- data dictionary 201
 - adding data sets 201
- data pools
 - creating 202
 - using 200
- data sets 201, 201
- definitions
 - test assets 189
- dictionary
 - overview 201

E

- Eclipse IDE 15
- editing
 - running
 - target deployment port editor 24

- target deployment ports 24
- Editors
 - Test case editor 195, 223
 - Test harness editor 205
- Enabling runtime analysis 50
- errors
 - memory profiling 958
- errors and warnings
 - preferences 940
- Exuberant Ctags 17

G

- generating
 - 2D and 3D charts 924
- getting started 45
- guide 9
- getting started 9

H

- HCL
- OneTest
- Embedded
 - preferences 939

I

- importing
 - Rational
 - Quality Manager 31
- initializations
 - multiple values 197
- installation
 - HCL
 - OneTest
 - Embedded 15
 - installation log files
 - verifying software installation 20
- installing
 - Cygwin 17
 - Exuberant Ctags 17
 - Instrumentation 11

M

- memory profiling
 - configuration settings 74
 - errors 958
 - warnings 960
- multiple values
 - initializations 197

N

- No Change 193
- No Check 193

O

- output window preferences 1010, 1151
- overview
 - code review 90
- Overview
 - Instrumentation 11
 - Runtime analysis 47
 - Source code insertion 11
 - Test cases 192

P

- performance profiling
 - overview 77
- preferences

- call graph 939
- errors and warnings 940
- HCL
- OneTest
- Embedded
 - 939
 - report generation 941
 - report viewers 940, 943
 - syntax coloring 940
 - target deployment port 941
 - test generation 941

R

- Rational
- Quality Manager
 - configuring 30
 - importing 31
 - starting 30
- Rational Quality Manager adapter
 - overview 29
 - reference 939
- report generation
 - preferences 941
- report viewers
 - preferences 940, 943
- reports
 - generating 924
- running
 - command line 213
 - test suites 212
- Runtime analysis 47
 - Enabling 50
- runtime tracing
 - advanced information 84
 - configuration settings 86, 89

S

- Same As Init 193
- settings
 - build 947
 - target deployment port 944
- software installation
 - verification 20
- source files
 - definition 189
- start
 - Rational
 - Quality Manager 30
- stub
 - Create 222
- stubs
 - definition 189
- Studio 224
- switching
 - test configurations 210, 661
- syntax coloring
 - preferences 940

T

- target deployment port
 - configuration settings 944
 - editor 24
 - preferences 941
- Target Deployment Port 1039
- TDP
 - preferences 941
- Test assets
 - Definition 189

- Test case
 - Create 195
- Test cases
 - Definition 189
 - Editing 195, 223
 - Structure 192
- test configurations
 - creating 209
 - switching 210, 661
- test generation
 - preferences 941
- Test harnesses
 - Definition 189
 - Editing 205
- test reports
 - generating 924
- test suites
 - command line 213
 - creating 207
 - definition 189
 - running 212
- testing 45

U

- user privileges 15
- using
 - data pools 200

V

- viewing
 - 2D and 3D charts 938

W

- warnings
 - memory profiling 960
- workbench
 - command line 213