

Rational® Test RealTime

Target Deployment Guide

VERSION: 2003.06.00

WINDOWS AND UNIX

Legal Notices

©2001-2003, Rational Software Corporation. All rights reserved.

Any reproduction or distribution of this work is expressly prohibited without the prior written consent of Rational.

Version Number: 2003.06.00

Rational, Rational Software Corporation, the Rational logo, Rational Developer Network, AnalystStudio, , ClearCase, ClearCase Attache, ClearCase MultiSite, ClearDDTS, ClearGuide, ClearQuest, ClearTrack, Connexis, e-Development Accelerators, DDTS, Object Testing, Object-Oriented Recording, ObjecTime, ObjecTime Design Logo, Objectory, PerformanceStudio, PureCoverage, PureDDTS, PureLink, Purify, Quantify, Rational Apex, Rational CRC, Rational Process Workbench, Rational Rose, Rational Suite, Rational Suite ContentStudio, , Rational Summit, Rational Visual Test, Rational Unified Process, RUP, RequisitePro, ScriptAssure, SiteCheck, SiteLoad, SoDA, TestFactory, TestFoundation, TestStudio, TestMate, VADS, and XDE, among others, are trademarks or registered trademarks of Rational Software Corporation in the United States and/or in other countries. All other names are used for identification purposes only, and are trademarks or registered trademarks of their respective companies.

Portions covered by U.S. Patent Nos. 5,193,180 and 5,335,344 and 5,535,329 and 5,574,898 and 5,649,200 and 5,675,802 and 5,754,760 and 5,835,701 and 6,049,666 and 6,126,329 and 6,167,534 and 6,206,584. Additional U.S. Patents and International Patents pending.

U.S. GOVERNMENT RIGHTS. All Rational software products provided to the U.S. Government are provided and licensed as commercial software, subject to the applicable license agreement. All such products provided to the U.S. Government pursuant to solicitations issued prior to December 1, 1995 are provided with "Restricted Rights" as provided for in FAR, 48 CFR 52.227-14 (JUNE 1987) or DFARS, 48 CFR 252.227-7013 (OCT 1988), as applicable.

WARRANTY DISCLAIMER. This document and its associated software may be used as stated in the underlying license agreement. Except as explicitly stated otherwise in such license agreement, and except to the extent prohibited or limited by law from jurisdiction to jurisdiction, Rational Software Corporation expressly disclaims all other warranties, express or implied, with respect to the media and software product and its documentation, including without limitation, the warranties of merchantability, non-infringement, title or fitness for a particular purpose or arising from a course of dealing, usage or trade practice, and any warranty against interference with Licensee's quiet enjoyment of the product.

Third Party Notices, Code, Licenses, and Acknowledgements

Portions Copyright ©1992-1999, Summit Software Company. All rights reserved.

Microsoft, the Microsoft logo, Active Accessibility, Active Client, Active Desktop, Active Directory, ActiveMovie, Active Platform, ActiveStore, ActiveSync, ActiveX, Ask Maxwell, Authenticode, AutoSum, BackOffice, the BackOffice logo, bCentral, BizTalk, Bookshelf, ClearType, CodeView, DataTips, Developer Studio, Direct3D, DirectAnimation, DirectDraw, DirectInput, DirectX, DirectXJ, DoubleSpace, DriveSpace, FrontPage, Funstone, Genuine Microsoft Products logo, IntelliEye, the IntelliEye logo, IntelliMirror, IntelliSense, J/Direct, JScript, LineShare, Liquid Motion, Mapbase, MapManager, MapPoint, MapVision, Microsoft Agent logo, the Microsoft eMbedded Visual Tools logo, the Microsoft Internet Explorer logo, the Microsoft Office Compatible logo, Microsoft Press, the Microsoft Press logo, Microsoft QuickBasic, MS-DOS, MSDN, NetMeeting, NetShow, the Office logo, Outlook, PhotoDraw, PivotChart, PivotTable, PowerPoint, QuickAssembler, QuickShelf, RelayOne, Rushmore, SharePoint, SourceSafe, TipWizard, V-Chat, VideoFlash, Virtual Basic, the Virtual Basic logo, Visual C++, Visual C#, Visual FoxPro, Visual InterDev, Visual J++, Visual SourceSafe, Visual Studio, the Visual Studio logo, Vizact, WebBot, WebPIP, Win32, Win32s, Win64, Windows, the Windows CE logo, the Windows logo, Windows NT, the Windows Start logo, and XENIX, are either trademarks or registered trademarks of Microsoft Corporation in the United States and/or in other countries.

Sun, Sun Microsystems, the Sun Logo, Ultra, AnswerBook 2, medialib, OpenBoot, Solaris, Java, Java 3D, ShowMe TV, SunForum, SunVTS, SunFDDI, StarOffice, and SunPCi, among others, are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

Purify is licensed under Sun Microsystems, Inc., U.S. Patent No. 5,404,499.

Licensee shall not incorporate any GLOBEtrouter software (FLEXIm libraries and utilities) into any product or application the primary purpose of which is software license management.

BasicScript is a registered trademark of Summit Software, Inc.

Design Patterns: Elements of Reusable Object-Oriented Software, by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. Copyright © 1995 by Addison-Wesley Publishing Company, Inc. All rights reserved.

Additional legal notices are described in the legal_information.html file that is included in your Rational software installation.

Target Deployment Guide Contents

Preface	ix
Audience	ix
Contacting Rational Technical Publications	ix
Other Resources	x
Customer Support	x
Target Deployment Port Tutorial.....	1
Tutorial Preparation	1
Tutorial Steps	1
Creating a New TDP	2
Editing a TDP	3
Validating a New TDP	5
Creating a New Configuration	5
Applying a Configuration to a Project	6
Validating the Compilation Procedure	6
Debugging a TDP	7
Customizing a TDP	8
Library Settings.....	8
Build Settings.....	9
User-defined I/O Primitives	10
Using a Debugger	11
Using the Debugger.....	11
Debug Results	12
Break Point Mode	12
Dumping the Buffer.....	13
Converting Data to ASCII	13
Planning a Target Deployment Port.....	15
Contents of a Target Deployment Port	15
Determining Target Requirements	16
Determining Target Requirements	16

Table Of Contents

Data Retrieval Capability	18
Free Data Space	18
Free Stack Space	18
Mutex	19
Thread Self and Private Data	19
Clock Interface.....	19
Heap Management	19
High-Speed Link.....	19
Task Management.....	20
BSD Socket Compliance	20
Thread Adaptation	20
Clock Adaptation	20
JVMPI Support.....	21
Heap Settings	21
Retrieving Data from the Target Host	21
Target System Categories.....	22
Determining Target Architecture Support.....	23
Data Retrieval Examples	24
Using the TDP Editor	29
Upgrading a Target Deployment Port	29
Opening a Target Deployment Port	29
Editing Customization Points	30
Creating a Target Deployment Port	31
Naming Conventions	31
Updating a Target Deployment Port	32
Using a Post-generation Script	32
Example.....	32
Migrating pre-V2002 TDPs to Present Format.....	33
unittest.ini.....	33
atuconf.h	34
attol_comm and attol_serv	37
private_io.ads	37
private_io.adb	38
attolcov_io.ads.....	39
attolcov.opp	39
atlcov.hpp	39
atlcov.def	39
atl_cc.def	40
atl_cc.def for C++	40
standard-ada95.ads.....	40
standard-ada83.ads.....	40
standard*.*	40

Perl Scripts	41
Index.....	45

Preface

Welcome to Rational Test RealTime.

This Reference Manual contains advanced information to help you use the product from the command line.

Test RealTime is a complete runtime analysis and testing solution for real-time and embedded systems. It addresses all runtime analysis needs and all test levels including component and system testing for the C, C++, Ada, and Java programming languages.

General information about using the product can be found in the *Test RealTime User Guide*.

If you are using the product for the first time, please take the time to go through the *Test RealTime Online Tutorial*.

If you are upgrading from a previous version of Test RealTime, please refer to *Upgrading a Target Deployment Port*.

Audience

This Target Deployment Guide is intended for advanced users of the product. Advanced knowledge of the target compiler, platform, development and test environment are required for Target Deployment Port customization tasks. Knowledge of Perl scripts is also required.

Contacting Rational Technical Publications

To send feedback about documentation for Rational products, please send e-mail to our technical publications department at techpubs@rational.com.

Keep in mind that this e-mail address is only for documentation feedback. For technical questions, please contact Customer Support.

Other Resources

All manuals are available online, either in HTML or PDF format. The online manuals are on the CD and are installed with the product.

For the most recent updates to the product, including documentation, please visit the Product Support section of the Web site at:

<http://www.rational.com/products/testrt/index.jsp>

Documentation updates and printable PDF versions of Rational documentation can also be downloaded from:

<http://www.rational.com/support/documentation/index.jsp>

For more information about Rational Software technical publications, see:

<http://www.rational.com/documentation>.

For more information on training opportunities, see the Rational University Web site:

<http://www.rational.com/university>.

Customer Support

Before contacting Rational Customer Support, make sure you have a look at the tips, advice and answers to frequently asked questions in Rational's Solution database:

<http://solutions.rational.com/solutions>

Choose the product from the list and enter a keyword that most represents your problem. For example, to obtain all the documents that talk about stubs taking parameters of type "char", enter "stub char". This database is updated with more than 20 documents each month.

When contacting Rational Customer Support, please be prepared to supply the following information:

- **About you:**
Name, title, e-mail address, telephone number
- **About your company:**
Company name and company address
- **About the product:**
Product name and version number (from the **Help** menu, select **About**).
What components of the product you are using
- **About your development environment:**
Operating system and version number (for example, Linux RedHat 8.0), target

compiler, operating system and microprocessor. If necessary, send the Target Deployment Port **.xdp** file

- **About your problem:**

Your service request number (if you are calling about a previously reported problem)

A summary description of the problem, related errors, and how it was made to occur

Please state how critical your problem is

Any files that can be helpful for the technical support to reproduce the problem (project, workspace, test scripts, source files). Formats accepted are **.zip** and compressed tar (**.tar.Z** or **.tar.gz**)

If your organization has a designated, on-site support person, please try to contact that person before contacting Rational Customer Support.

You can obtain technical assistance by sending e-mail to just one of the e-mail addresses cited below. E-mail is acknowledged immediately and is usually answered within one working day of its arrival at Rational. When sending an e-mail, place the product name in the subject line, and include a description of your problem in the body of your message.

Note When sending e-mail concerning a previously-reported problem, please include in the subject field: "**[SR# <number>]**", where **<number>** is the service request number of the issue. For example:

Re: [SR#12176528] New data on Rational Test RealTime install issue

Sometimes Rational technical support engineers will ask you to fax information to help them diagnose problems. You can also report a technical problem by fax if you prefer. Please mark faxes "**Attention: Customer Support**" and add your fax number to the information requested above.

Location	Contact
North America	Rational Software, 18880 Homestead Road, Cupertino, CA 95014 voice: (800) 433-5444 fax: (408) 863-4001 email: support@rational.com
Europe, Middle East, and Africa	Rational Software, Beechavenue 30, 1119 PV Schiphol-Rijk, The Netherlands voice: +31 20 454 6200 fax: +31 20 454 6201 email: support@europe.rational.com

Rational Target Deployment Guide

Asia Pacific

Rational Software Corporation Pty Ltd,
Level 13, Tower A, Zenith Centre,
821 Pacific Highway,
Chatswood NSW 2067,
Australia

voice: +61 2-9419-0111

fax: +61 2-9419-0123

email: support@apac.rational.com

Target Deployment Port Tutorial

The aim of this quick example is to demonstrate how to create and validate a new TDP on Windows. The same principles apply to other platforms: just replace Windows with the native or target platform of your choice.

Tutorial Preparation

An example project for this tutorial, names **add.rtp**, is provided with Test RealTime in the `/examples/TDP/tutorial` directory.

The TDP for this tutorial is based on the MinGW (Minimalist Gnu for Windows) C compiler distribution. MinGW is a collection of header files and import libraries that allow one to use GCC and produce native Windows32 programs that do not rely on any 3rd-party DLLs.

The MinGW distribution includes GNU Compiler Collection (GCC), GNU Binary Utilities (Binutils), GNU debugger (Gdb) , GNU make, and various other utilities.

To obtain a copy of the MinGW environment:

1. Connect to <http://www.mingw.org>
2. Locate and download the latest complete MinGW distribution.
3. Follow the instructions provided with the distribution for installation and configuration.

Tutorial Steps

This Tutorial will guide you through the steps of creating, modifying and debugging TDP, using custom I/O functions, a debugger and defining a break point strategy.

Creating a New TDP

In most cases, you will not create a TDP from scratch but rather base your new TDP on an existing TDP template. In this example, you will adapt an existing TDP `gccmingw_template.xdp` to your own environment.

The TDP file format is `.xdp`, as in XML Deployment Port. There are file-naming conventions when creating a new TDP:

- **c** for a C or C++ TDP, **j** for Java, **a** for an Ada TDP.
- An acronym for the target platform host, in this case call it **wingcc** for Windows GCC.
- The name of the development environment **mingw**

Therefore, our TDP filename shall be `cwingccmingw`.

All TDPs are located in the following directory:

```
<install_dir>/targets/<tdp_name>/<tdp_name>.xdp
```

where `<install_dir>` is the installation directory, and `<tdp_name>` is the name of the TDP.

To start the TDP Editor:

1. In Test RealTime, from the **Tools** menu, select **TDP Editor** and **Start**.
or
2. From the command line, type `tdpeditor`.

To open a TDP template:

1. In the **TDP Editor**, from the **File** menu, select **Open**.
2. In the **targets** subdirectory, select the `gccmingw_template.xdp` TDP file.
3. Right click the Top level node in the tree-view pane: **Gnu 2.95.3-5 (mingw)**.
4. Select **Rename**.and enter a new name for this TDP: **My_MinGW**.
This name identifies the TDP in the Test RealTime GUI.
5. In the **Comment for the root node** section, enter contact information such as your name and email address. This makes things easier when sharing the TDP with other users.

To save the new TDP:

1. From the **File** menu, select **Save xdp As**,
2. Save your new TDP as `cwingccmingw.xdp`.

- From the **File** menu, select **Save and Generate**. The TDP Editor automatically creates a directory named **cwingccmingw** and saves the **.xdp** file in that location.

Editing a TDP

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 obtained detailed reference information for that parameter in the **Help** Window.

The Navigation Tree covers all the customization points of the TDP. There are four main 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 Test RealTime 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 Test RealTime GUI integrated 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.
- **Library Settings:** This section describes a set of source code files as well as a dedicated customization file (**custom.h**), which adapt the TDP to target platform requirements. This section is definitively the most complex and usually only requires customization for specialized platform TDPs (unknown RTOS, no RTOS, unknown simulator, emulator, etc.)

- **Parser Settings:** This section modifies the behavior of the parser in order to address non-standard compiler extensions, such as for example, non-ANSI extensions. This section allows Test RealTime to properly parse your source code, either for instrumentation or code generation purposes.

On the right hand side of the TDP Editor window, the embedded Help provides contextual reference information for the part of the TDP that is selected in the tree-view pane.

To Edit the new TDP:

Use the TDP Editor's tree pane to navigate through the customization points of the TDP, and make the following changes:

1. Under **Basic Settings:** Change the **ENV_PATH** and customization points in both the **For C** and **For C++** nodes. **ENV_PATH** updates the **PATH** environment variable in order to invoke the **gcc** compiler directly. For example:

```
ENV_PATH          C:\Gcc\bin;$ENV{'PATH' }
```

Note When you modify a customization point in the TDP Editor, it is generally a good idea to add a note in the **Comment** box. This makes later modifications and TDP sharing much easier.

2. In the same manner, check all the other customization points to ensure that they reflect the correct path and filenames as provided with the MinGW distribution.
3. Under **Build Settings:** No changes should be required here, but have a look at the **Compilation Function**. Locate the corresponding Perl script and have a look at the Help window to understand how the **atl_cc** routine works. Next, look at the **Link Function** to understand the **alt_link** Perl routine.

Note All the parameters used by these Perl routines are set in the **Basic Settings** section of the TDP.

4. Under **Library Settings:** No changes are required at this point.
5. Under **Parser Settings:** In this section, you need to tell the Test RealTime code parser where the **std** GCC libraries are located. You must perform the same change as much as necessary for the features that you plan to use with this TDP.
6. Save the TDP.

Any changes made to the Basic Settings section of a TDP are read from the Test RealTime GUI and applied to the project. For this reason, whenever you modify the Basic Settings of a TDP that is currently used in a Test RealTime project, you must reload the TDP into the project.

To reload the TDP in Test RealTime:

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

You have created your first TDP. The next step is to validate the new TDP in Test RealTime.

Validating a New TDP

After a TDP has been created or modified, the first step is to validate that it works correctly on the target.

The first step is to change the TDP used by your project.

To make sure that your TDP is working properly, you must create a Component Testing test node and run it with all the relevant Runtime Analysis tools enabled. Once the following steps are covered, you can consider that your TDP is fully functional:

- Create a new Configuration Test RealTime
- Apply the new Configuration to a project
- Validate the compilation sequence with the new Configuration

Creating a New Configuration

In Test RealTime, the TDP is part of a Configuration. Each Configuration is based on a TDP, plus the particular Configuration Settings that are specific for each node of the project.

This means that you can base several slightly different Configurations on a single TDP.

To create a new Configuration in Test RealTime:

1. In Test RealTime, open the **add.rtp** example project.

This example project provides a series of test nodes for demonstration of Test RealTime features. For this tutorial, concentrate on the add test node, which contains a simple **add.c** source file as well as the corresponding **add.ptu** test script.

2. From the **Project** menu, select **Configurations**. Click **New**.

3. In the **New Configuration** box, enter a name for the new Configuration, and select the TDP on which it shall be based.

For our example, select your newly created MinGW TDP. Notice that two items appear in the list, one for C, another for C++ followed by the same name. Select the C version of the TDP

4. Click **OK, Close** and save the project. Update the TDP in the project.

Applying a Configuration to a Project

Now that the new Configuration has been created, based on your TDP, you need to select it for use in your project.

Although a project can use multiple Configurations, as well as multiple TDPs, there must always be at least one active Configuration.

TDP is used when selected from the Build combo-box, but remember that you have to be consistent between the TDP programming language selection and the source files used within your test environment.



To change the current Configuration of a project:


1. From the **Build** toolbar, select the Configuration you wish to use in the **Configuration** box.
2. Update any project settings if necessary.

Validating the Compilation Procedure

In order to validate the compilation sequence, the idea is to successfully compile the current project with the new Configuration.

To validate the compilation procedure:

1. In the Project Explorer, select a single source file.
2. From the Build toolbar, click the **Build Options**  button and clear all Runtime Analysis features (Memory Profiling, Performance Profiling, Code Coverage and Runtime Tracing) to ensure that these do not affect the build sequence.
3. Select the **add.c** source file.
4. From the Build toolbar, click **Build** .

The compilation should end with a **Passed**  status. If not, restart the TDP Editor and change the **atl_cc** Perl procedure accordingly.

You can repeat the same action for the following Perl procedures:

- **atl_cpp**: Preprocessing routine for Source Code Insertion

- **atl_link:** Link routine
- **atl_exec:** Execution routine
- **atl_execdbg:** Debugging routine

The compilation procedure is validated. You can now consider using the Test and Runtime Analysis features of Test RealTime on your project.

The next section provides help about debugging any compilation issues you may have encountered.

Debugging a TDP

If everything does not work as it should, the following method might help you troubleshoot TDP issues with Test RealTime.

To troubleshoot a TDP:

1. Set the **ATTOLSTUDIO_VERBOSE** environment variable to **1**. The exact procedure to do this depends on your operating system.
2. The Test RealTime GUI does not automatically inherit the Windows environment. Therefore you must save the project, close and relaunch the GUI.
3. Ensure that the correct TDP is selected. From the **Project** menu, select **Configurations** and click **New** to select the new TDP if necessary.
4. Decompose the complete build process into multiple steps. To do this, click the **Build Options** (▼) button, clear the **All** option and select only the first step of the compilation sequence (**Source compilation**). Clear any Runtime Analysis tools.
5. Select the source file under test (**add.c** in this example) and click **Build** (▶).
6. Repeat the same operation for each other compilation step and source file until the whole node can be successfully processed.

This should provide adequate feedback to help you debug each individual step of the compilation sequence.

In the current example, any problems encountered will usually be related to an incorrect file path in the **Basic Settings** of the TDP.

Customizing a TDP

This section of the Tutorial will demonstrate how to use the customize input-output (I/O) communication and break-point usage in order to address a target system without standard I/O functions.

First, create a new TDP based on the one created previously.

To create a new TDP:

1. Open the **cwingccmingw.xdp** TDP in the TDP Editor
2. Select the top-level node and rename it **My MinGW UserMode**.
3. From the **File** menu, select **Save xdp As** to save the new TDP as **cwingccmingw2.xdp**.
4. Collapse all the nodes in the Navigation window as this section concentrates only on the **Build Settings** and **Library Settings** nodes of the TDP Editor.

Library Settings

You first need to specify the I/O user mode, which means disabling the standard I/O mode for data retrieval on the target.

By default, when executing a program compiled with Test RealTime, the test data is dumped to a file on the file system by using the standard **fopen**, **fprintf** and **fclose** functions. On some platforms, these primitives are not available hence the need to use a set of user-defined I/O functions that allow the TDP to access the File System.

To change Library settings:

1. Expand **Library Settings, Data retrieval and error message output** and select **Data retrieval** to locate the **RTRT_IO** macro definition.
In the combo-box for **RTRT_IO** you can select:
 - **RTRT_NONE**: No I/O available
 - **RTRT_STD**: Standard I/O functions (**fopen**, **fprintf** and **fclose**)
 - **RTRT_USR**: User-defined I/O. This option enables the customization tabs.
2. Select **RTRT_USR**. Look at the user defined I/O primitives used to access the File System: **usr_open**, **usr_writeln** and **usr_close**.
Notice that **usr_writeln()** contains the following statement

```
printf("%s", s);
```
3. From the **File** menu, select **Save and Generate**.

- Update the Configuration in Test RealTime to use the **My MinGW UserMode TDP**, and **Build** your sample project.

This build should fail. The message console should display the following information, or similar:

```

Executing gcc_step1\Histo.exe ...
gcc_step1\Histo.exe
PU "Histo"
HO "... "
O1
NT "Initialization" 0 0
DT 0
...
A32 OK RA=T
NT "Termination" 61 41
DT 0
FT 91e544c5DC 0b72d3c1
PT "Termination"
PS 0 0 0
PY 0 0 0
QT "Termination"
QS 91e544c5 7965f082
NO "2 (Max Calling Level reached) "
CI 0h
Splitting 'gcc_step1\THisto.rio' traces file...
Traces file successfully split.
No RIO instruction found.
Errors have occurred.

```

This message shows that:

- ASCII character data was dumped from the program directly to the standard output of the executable through the **printf** directive.
- Test data output is encoded information that only the Test RealTime Report Generator is able to understand.
- The trace file is empty. Although the split is successful, no instructions are found and an error message is produced.

Therefore, for the build to be successful, you must provide the Report Generator with a valid trace file.

Build Settings

The Execution function is a basic command that produces an output file that redirects the standard output to **\$out**.

To change Build settings:

- In the TDP Editor, expand the **Build Settings** and select **Execution function**. The following code is displayed:

```

sub at1_exec($$$)
{

```

```
my ($exe,$out,$parameters) = @_;  
unlink($out);  
SystemP("$exe $parameters");  
}
```

2. Change the SystemP line to:
`SystemP("$exe $parameters >$out");`
3. Save the TDP, update the Configuration in Test RealTime and **Build** your sample project.

This time, the execution should run smoothly and produce complete reports. If not, rework the above functions until the execution is successful.

User-defined I/O Primitives

This section demonstrates how to define your own I/O primitives for the dump phase.

Again, create a new TDP based on the one created previously.

To create a new TDP:


1. Open the **cwingccmingw2.xdp** TDP in the TDP Editor
2. Select the top-level node and rename it **My MinGW UserMode2**
3. Save the current TDP as **cwingccmingw3.xdp**.
4. Collapse all the nodes in the Navigation window as this section concentrates only on the **Build Settings** and **Library Settings** nodes of the TDP Editor.

To set up user-defined I/O primitives:

1. Expand **Build Settings** and select the **Execution** function.
2. Delete the **>\$out** parameter that was added to the **SystemP** statement:
`SystemP("$exe $parameters");`
3. Expand **Library Settings, Data retrieval and error output** and select **Data retrieval** to locate the **RTRT_IO** macro definition.
4. Select the **RTRT_USR** entry.
5. On the **Settings** tab, in **RTRT_FILE_TYPE**, change **int** to **FILE***.
6. Add your own code for the **usr_open** function, such as:
`printf("...Opening file...\n");
return(fopen(fileName, "w"));`
7. Add your own code for the **usr_init** function:
`return(null);`

8. Add your own code for the **usr_writeln** function:


```
printf("...Dumping : %s\n",s);
fprintf(f, "%s", s);
```
9. Add your own code for the **usr_close** function:


```
printf("...Closing file...\n");
fclose(f);
```
10. Save the TDP, update the Configuration in Test RealTime and **Build**  the **add.c** example.



The examples described here make no sense in real life as they are functionally identical the standard I/O mechanism. However, they show how easy it is to map user-defined I/O primitives to the data retrieval mechanism implemented by the TDP.

Using a Debugger

Before moving to the next step we need to understand how Test RealTime uses the GDB debugger command. This function is called when the **Debug** build option is selected in the Test RealTime GUI.

Note This is NOT a break-point strategy. The Debug option merely allows you to manually inspect application execution.

To build a node in Test RealTime Debug mode:

1. In the Test RealTime Project Explorer, select the project node.
2. From the Build toolbar, click the **Build Options**  button and select Debug in the **Build Options** window.
3. In the Project Explorer, select the **add.c** node.
4. From the Build toolbar, click the **Build**  button.

This runs a command line window with the GDB up and running.

Using the Debugger

In the GDB window, type the following commands:

```
break priv_writeln
break priv_close
display atl_buffer
run
```

If you type **c** or **cont** for continue you should see the **atl_buffer** contents changing and showing information similar to what you obtained in the Message Console with the **printf** command.

Debug Results

The **priv_writeln** and **priv_close** primitives are implemented within the TDP. The former is interpreted as a dump request event, whereas the latter is an end of execution event.

The **atl_buffer** symbol (default size is 1024 bytes) dynamically gathers information from the test execution.

The objective is to produce a file on the file system just as we did with the standard I/O functions or the user-defined I/O functions. When a break point strategy is required, the manual process you have just accomplished must be somehow automated.

Break Point Mode

Using Break Point mode can be summed up as the following tasks:

- Compile, link and load the executable in the debugger. This is typically handled by the GUI, so no action is required.
- Dump the content of **atl_buffer** each time the break point on **priv_writeln** is met.
- Quit the debugger when the **priv_close** is reached
- Ensure sure that the file produced is ASCII

To do this, you must specify a break point for I/O. This means that you will no longer use the standard I/O or the user-defined I/O functions.

To disable I/O functions:

1. Expand **Library Settings, Data retrieval and error output** and select **Data retrieval** to locate the **RTRT_IO** macro definition. In the combo-box for **RTRT_IO** you can select:
 - **RTRT_NONE**: No I/O available
 - **RTRT_STD**: Standard I/O functions
 - **RTRT_USR**: User-defined I/O. Only this option allows you to access the customization tabs.
2. Select **RTRT_NONE**. This is the typical choice when on limited target platforms with no operating system and no file system.

Dumping the Buffer

You need to dump the content of **atl_buffer** each time the break point on **priv_writeln** is encountered. The way to do this, without a file system, is to specify how to use the **gdb** debugger command line in the **atl_exec** Perl script.

The debugger documentation explains how to call **gdb** and how to automate the use of the debugger through a command script.

To invoke the debugger from the **atl_exec**:

1. Expand **Build Settings** and select **Execution function** to locate the **atl_exec** Perl function.
2. Comment the existing command line with a **#** character.
3. Add the following lines to invoke **gdb**:

```
my $cmd="$TARGETDIT\cmd\run.cmd";
SystemP("gdb -se=$exe -command=$cmd > stdout.log");
```
4. Right-click **Build Settings** and select **Ascii File**. Rename the created file to **run.cmd**.
5. Copy the contents of the **run_example.cmd** file, provided in the **example** directory, into the **run.cmd** file.
6. Save the TDP, update the TDP in the project, and **Build** the **add.c** node.


Converting Data to ASCII

Depending on the cross development environment, the format of the dumped data can vary largely from one target to another. In most cases, the results must be decoded and converted to ASCII data in order to be processed by the Test RealTime Data Splitter and Report Generators.

You need to decode the dump data to ASCII with a Perl routine by using the Perl subroutine named **decode.pl**.

To decode dump data to ASCII:

1. Save the TDP, update the Configuration in Test RealTime and **Build** the **add.c** node.
 Test RealTime returns an error: the dump accomplished by the debugger does not produce a plain ASCII file as expected.
2. Look at the result file created by GDB. The relevant data is present but is represented in hex and mixed with other information.
3. Open the **decode.pl** Perl script in a text editor, provided with the example.

4. In the TDP Editor, expand **Build Settings** and select **Execution function** to locate the **atl_exec** Perl function.
5. Copy-paste the contents of the **decode.pl** Perl script into the **atl_exec** Perl function after execution of **gdb**.
6. Save the TDP update the TDP in the project, and **Build**  the **add.c** node.

This time, everything should work as expected and you should be able to view the reports generated by the execution.

Congratulations! You have completed what is probably the most complex part building a TDP.

Planning a Target Deployment Port

Rational's Target Deployment Technology extends Rational Test RealTime to provide support for your own target environment.

Setting up a Target Deployment Port (TDP) essentially involves the creation of a set of files and procedures that enable the execution of generated test programs or instrumented applications directly on your target host, as well as enabling the retrieval of test and runtime analysis results from the target host.

Due to the nature of the tasks at hand and to the characteristics of each target host, you may or may not be able to run certain features of the product on certain targets.

First, refer to Target Requirements for a list of minimum requirements that the target system must provide for each test and runtime analysis feature.

Contents of a Target Deployment Port

By default, the Target Deployment Ports available on your machine are located within the product installation folder, in the `\targets` directory:

Each Target Deployment Port is stored in its own directory. The directory name starts with a **c** for the C and C++ languages, **ada** for the Ada language or **j** for Java, followed by the name of the development environment, such as the compiler and target platform.

A Target Deployment Port 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 Test RealTime 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 Test RealTime GUI integrated 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.

- **Library Settings:** This section describes a set of source code files as well as a dedicated customization file (**custom.h**), which adapt the TDP to target platform requirements. This section is definitively the most complex and usually only requires customization for specialized platform TDPs (unknown RTOS, no RTOS, unknown simulator, emulator, etc.). These files are stored in the TDP **lib** subdirectory.
- **Parser Settings:** This section modifies the behavior of the parser in order to address non-standard compiler extensions, such as for example, non-ANSI extensions. This section allows Test RealTime to properly parse your source code, either for instrumentation or code generation purposes. The resulting files are stored in the TDP **ana** subdirectory.

Use the **Help** Window in the TDP Editor to obtain reference information about each setting.

Determining Target Requirements

Determining Target Requirements

The following tables lists the minimum requirements that your development environment must provide to enable use of each feature of Test RealTime:

- C, C++ and Ada requirements
- Java requirements

C, C++ and Ada Requirements

Each feature is listed as a column title.

	Component Testing for C and Ada	Component Testing for C++	System Testing for C #Virtual Testers=1	System Testing for C #Virtual Testers>1	Code Coverage	Runtime Tracing	Memory Profiling	Performance Profiling
Data Retrieval Capability	Required	Required	Required	Required	Required	Required	Required	Required
Free Data Space					For stand alone	For stand alone	For stand alone	For stand alone

Free Stack Space				For stand alone	For stand alone	For stand alone	For stand alone
Mutex	For MT			For MT	For MT	For MT	For MT
Thread Self and PrivateData	For MT				For MT	For MT	For MT
Clock Interface		Required	Required				Required
Heap Management		Required	Required			Required	
High Speed Link					Required		
Task Management	For MT		Required		For MT	For MT	For MT
BSD Sockets			Required				
Ada	N/A	N/A	N/A		N/A	N/A	N/A

- **For stand alone:** Required for stand alone use of a runtime analysis feature - i.e. used without a Test RealTime test feature
- **For MT:** Required if the application under test is a multi-threaded application based on a preemptive multi-tasking mechanism.

Note Only the Component Testing for C and Ada and Code Coverage features support the Ada language. System Testing for C can, however, be used to send messages to an Ada-written application if C bindings exist for that feature.

Java Requirements

	Component Testing for Java	Code Coverage	Runtime Tracing	Memory Profiling	Performance Profiling
Data Retrieval Capability	Required	Required	Required	Required	Required
Free Data Space		For stand alone	For stand alone	For stand alone	For stand alone
Free Stack Space		For stand alone	For stand alone	For stand alone	For stand alone
Thread		Required	Required		Required

Adaptation	
Clock Adaptation	Required
JVMPI Support	Required
Heap Settings	Required

- **For stand alone:** Required for stand alone use of a runtime analysis feature - i.e. used without Component Testing for Java.

Data Retrieval Capability

Test programs or instrumented applications need to generate a text file on the host - this is how information is gathered to prepare Test RealTime reports.

The Target Deployment Port gathers this report data by obtaining the value of a (char *) global variable, containing regular ASCII codes, from the application or test driver running on the target machine.

This retrieval can be accomplished in whichever way is most practical for the target. It could be through file system access, a socket, specific system calls or a debugger script. Most known environments allow at least some form of I/O.

At least one form of data retrieval capability is required.

Free Data Space

All runtime analysis features are based on Rational Source Code Insertion (SCI) technology. The overhead introduced by this technology is dependent both on the selected instrumentation level and on code complexity.

The Code Coverage feature requires the most free data space. The overhead for default Code Coverage levels (procedure / method entries and decisions) typically increases code size by 25%. Runtime Tracing, Memory Profiling and Performance Profiling introduce a significantly lower overhead (about 16 bytes per instrumented file).

The Component Testing features Test RealTime do not typically require additional free memory because it is rare for the entire application to be run on the target.

Free Stack Space

The stack size should not be optimized for the requirements of the original application. The Test RealTime instrumentation process adds a few bytes to the stack and inserts calls to the TDP embedded runtime library.

Since, based on experience, it is difficult to identify stack overflow, the user should assume that each instrumented function requires, on average, an extra 30 bytes for local data.

Mutex

This customization is required by all runtime analysis features of the product if the application under test uses a preemptive scheduling mechanism. A mutual exclusion mechanism is required to ensure uninterrupted operation of critical sections of the Target Deployment Port.

Thread Self and Private Data

It must be possible to retrieve the current identifier of a thread, and it must be possible to create thread-specific data (e.g. `pthread_key_create` for POSIX).

Clock Interface

A clock interface is not necessary for the Component Testing for C and Ada, for C++, Memory Profiling and Performance Profiling features, but it is required for Performance Profiling and System Testing for C. The goal is to read and return a clock value (Performance Profiling) and to provide time out values (System Testing for C).

If you are using Performance Profiling and System Testing for C with Component Testing and the clock interface does exist, then Component Testing indicates time measurements for each function under test and the Runtime Tracing feature timestamps all messages.

Heap Management

This customization is required by Memory Profiling and System Testing for C only.

Both Memory Profiling and System Testing for C need to allocate memory dynamically.

Memory Profiling also tracks and records memory heap usage, based on the standard **malloc** and **free** functions. However, it can also handle user-defined or operating system dependent memory usage functions, if necessary.

High-Speed Link

For Runtime Tracing On-the-Fly only.

To use the Runtime Tracing feature without a testing feature, a high-speed link between the host and target machine is required in order to take full advantage of the On-the-Fly tracing mode. This is because Runtime Tracing-instrumented code "writes

a line" to the host for each entry point and exit point of every instrumented function. This means that as the application is running, a continuous flow of messages is written to the host. Understandably, a 9600 bit rate, for example, would not be sufficient for use of the Runtime Tracing feature with an entire application.

Note that the Code Coverage, Memory Profiling and Performance Profiling features store their data in static target memory, and data is only sent back to the host at specified flush points (with the Runtime Tracing feature, static memory is also flushed when it becomes full). Technically, a Memory Profiling, Performance Profiling, and Code Coverage instrumented application can run for weeks without seeing a growth in consumed memory; nothing need be sent to the host until a user-defined flush point is reached.

Task Management

Runtime analysis features require task management capabilities when they are used to monitor multi-threaded applications.

When the System Testing feature for C executes more than one virtual tester, full task management capabilities must be available. In other words, System Testing for C should be able to start a task, stop a task, and get the status of a task.

BSD Socket Compliance

When the System Testing feature for C executes more than one virtual tester, the target must be BSD socket compliant. This is necessary because System Testing for C uses TCP/IP sockets to enable communication between System Testing Agents and the System Testing Supervisor, as well as to enable virtual tester RENDEZVOUS synchronization.

If, in fact, the target host is BSD socket-compliant, then it is guaranteed that you can address the Data Retrieval Capability and the High-Speed Link requirements.

Thread Adaptation

This is required by all Java runtime analysis features except Memory Profiling for Java.

The **waitForThreads** method must wait for the last thread to terminate before dumping results and exiting the application.

On J2ME platforms, this method is empty.

Clock Adaptation

This customization is required for the Performance Profiling feature

- The **getClock** method must return the clock value, represented as a *long*.

- The `getClockUnit` method must return an array of bytes representing the clock unit.

JVMPI Support

The Java Virtual Machine (JVM) must support the JVM Profiler Interface (JVMPI) technology used for memory monitoring.

This is required for Memory Profiling for Java.

Heap Settings

This customization is part of the JVMPI support settings.

If available, the dynamic memory allocation required by the feature is made through standard `malloc` and `free` functions.

If the use of such routines is not allowed on the target, fill `JVMPI_SIZE_T`, `jvmpi_usr_malloc` and `jvmpi_usr_free` types and functions with the appropriate code.

Retrieving Data from the Target Host

All test and runtime analysis features of the product must be able to retrieve the value of a global (`char *`) variable from an application running on the target machine and then write that value to a text file on the host machine. (The variable will contain only ASCII values).

This retrieval may be the result of a specific program running on the target, of an adapted execution procedure on the host, or both.

To perform data retrieval, the program generated or instrumented by the product is linked with the Target Deployment Port data retrieval functions and type definition.

For example, in the C language, the type definition and data retrieval functions are:

```
#define RTRT_FILE <Type>
RTRT_FILE priv_init(char *fName);          /* fName: file name to
be written on the host */
RTRT_FILE priv_open(char *fName);         /* fName: file name to
be written on the host */
void priv_writeln(RTRT_FILE f, char *data); /* data is the data
that should be printed in the file */
void priv_close(RTRT_FILE f);             /* Close the host file
*/
```

These data retrieval functions are called by the Target Deployment Port library.

Depending on the nature of the target platform, some or all of these routines may be empty.

Target System Categories

Target platforms can be classified into three categories, characterized by their data-retrieval method:

- Standard Mode
- User Mode
- Breakpoint Mode

Standard Mode

This kind of target system allows use of a regular **FILE *** data type and of the **fopen**, **fprintf** and **fclose** functions found in the standard C library. Such systems include, for example, all UNIX or Windows platforms, as well as LynxOS or QNX.

If the standard C library is usable on the target, use these regular **fopen/fprintf/fclose** functions for TDP data retrieval. This is by far the easiest data retrieval option.

- If your target system is compliant with the Standard Mode category, data retrieval is assured.

User Mode

On *User Mode* systems, the standard C library calls described above are not available but other calls that send characters to the host machine are available. This could be a simple *putchar*-like function sending a character to a serial line, or it could be a method for sending a string to a simulated I/O channel, such as in the case of a microprocessor simulator.

- If your target system is using an operating system, there are usually functions that enable communication between the host machine and the target. Therefore, data retrieval capability is assured.
- If your target system allows use of a standard socket library, User Mode is always possible - thus data retrieval is assured.

Breakpoint Mode

On breakpoint mode systems, no I/O functions are available on the target platform. This is usually the case with small target calculators, such as those used in the automotive industry, running on a microprocessor simulator or emulator with no operating system.

If no communication functions are available on the target platform, the best alternative is to use a debugger logging mechanism, assuming one exists.

To retrieve data using in breakpoint mode:

1. set a breakpoint on the **priv_writeln** function

2. at this breakpoint, have the debugger retrieve the value of **atl_buffer** and write it to a host-based file
3. continue the execution

Note In breakpoint mode, some compilers and linkers ignore empty functions and remove them from the final **a.out** binary. As the debugger must use these routines to set breakpoints, you must ensure that the linker includes these functions - any associated symbols must be in the map file. Currently, all of the **priv_** functions for C and C++ contain a small amount of dummy code to avoid this issue; however, you might need to add dummy code for Ada.

Determining Target Architecture Support

If your target can be used in Standard or User Mode, then it is fully supported by Test RealTime.

However, if your target can only be used in Breakpoint Mode, then you must ask yourself the following questions to determine if your target platform has enough data retrieval capability to be supported by Test RealTime:

- Does this debugger provide access to symbols?
- Is there a command language?
- Is there a way to run commands from a file?
- Can a command file be executed automatically when the debugger starts, either from a particular filename or from an option of the command line syntax.
- Is there a command to stop the debugger? (The execution process must be blocked until execution is terminated and the trace file is generated.)
- Is there a way to set software breakpoints?
- Is there a way to log what happens into a file?
- Is there a way to dump the contents of a variable in any format, or to dump a memory buffer and log the value?
- Can the debugger automatically run other debugger commands when a breakpoint is reached, such as a variable dump and resume; or, alternatively, does the debugger command language include loop instructions?

If the answer to any of these questions is "No", then no data retrieval capability exists. Therefore, test and runtime analysis feature execution on the target machine will not be possible with Test RealTime.

Data Retrieval Examples

Data Retrieval is accomplished through the association of the Target Deployment Port library functions with an execution procedure.

The following examples demonstrate the Standard, User, and Breakpoint Modes, based on a simple program which writes a text message to a file named "cNewTdp\atl.out".

Standard Mode Example: Native

```
#define RTRT_FILE FILE *
RTRT_FILE usr_open(char *fileName)
{ return((RTRT_FILE)(fopen(fileName,"w"))); }
void usr_writeln(RTRT_FILE f,char *s)
{ fprintf(f,"%s",s); }
void usr_close(RTRT_FILE f)
{ fclose(f); }
char atl_buffer[100];
void main(void)
{
RTRT_FILE f ;
strcpy(atl_buffer,"Hello World ");
f=usr_open("cNewTdp\\atl.out");
usr_writeln(f,atl_buffer);
usr_close(f);
}
```

Execution command : a.out

When executing a.out, cNewTdp\atl.out will be created, and will contain "Hello World".

User Mode Example: BSO-Tasking Crossview

Source code of the program running on the target:

```
#define RTRT_FILE int
RTRT_FILE usr_open(char *fName) { return(1); }
void usr_writeln(RTRT_FILE f,char *s) { _simo(1,s,80); }
void usr_close(RTRT_FILE f) { ; }
char atl_buffer[100];
void main(void)
{
RTRT_FILE f ;
strcpy(atl_buffer,"Hello World");
f=usr_open("cNewTdp\\atl.out");
usr_writeln(f,atl_buffer);
usr_close(f);
}
```

Execution command from host:

```
xfw166.exe a.out -p TestRt.cmd
```

Content of TestRt.cmd:

```
l sio o atl.out
r
q y
```

In this example, `usr_open` and `usr_close` functions are empty. `Priv_writeln` uses a BSO-Tasking function, `_simo`, which allows to send the content of the `s` parameter on the channel number 1 (an equivalent of a file handle).

On another side, on the host machine, the Crossview simulator (launched by the `xfw166.exe` program) is configured by the command

```
1 sio o atl.out
```

indicating to the simulator running on the host, that any character being written on the channel number 1 should be logged into a file name `atl.out`

The next command is to run the program, and quit at the end.

The original needs, which was to have `cNewTdp\atl.out` file be written on the host has to be completed by a script on the host machine, consisting in moving the `atl.out` generated in the current directory into the `cNewTdp` directory. The complete execution step would be in Perl:

```
SystemP("xfw166.exe a.out -p TestRt.cmd");
If ( ! -r atl.out ) { Error... return(1); }
move("atl.out", "cNewTdp/atl.out");
```

Breakpoint-Mode :

In all the breakpoint mode examples, the `usr_` functions are empty.

Breakpoint Mode Example: Keil MicroVision

Source code of the program running on the target:

```
#define RTRT_FILE int
RTRT_FILE usr_open(char *fName) { return(1); }
void usr_writeln(RTRT_FILE f, char *s) {;}
void usr_close(RTRT_FILE f) { ; }
char atl_buffer[100];
void main(void)
{
    RTRT_FILE f ;
    strcpy(atl_buffer, "Hello World");
    f=usr_open("cNewTdp\\atl.out");
    usr_writeln(f,atl_buffer);
    usr_close(f);
}
```

Execution command from host:

```
uv2.exe -d TestRt.cmd
```

Content of `TestRt.cmd`:

```
load a.out
func void out(void) {
int i=0;
while(atl_buffer[i]) printf("%c",atl_buffer[i++]);
printf("\n");
}
bs usr_writeln, "out() "
```

```
bs usr_close
reset
log > Tmpatl.out
g
exit
```

In this example, all the `usr_` functions are empty. The intelligence is deported into the `TestRt.cmd` script which a command file for the debugger.

It first loads `a.out` executable program. It then defines a function, which prints the value of `atl_buffer` in the MicroVision command window. Then it sets two breakpoints. The first one in `usr_writeln`, and the second one in `usr_close`. When `usr_writeln` is reached, the program halts, and the debugger automatically runs his `out()` function, which print the value of `atl_buffer` into its command window. When `usr_close` is reached, the program halts.

Then, the debugger scripts resets the processor, and logs anything that happens in the debugger command window into a file named `Tmpatl.out`. It then starts the execution, (which finally halts when `usr_close` is reached as no action is associated with this breakpoint) and exits.

The final result is contained into `Tmpatl.out`, which should be cleanup by the host (a little decoder in Perl for example) to give the final expected `cNewTdp\atl.out` file containing "Hello World". The global execution step in Perl would be:

```
SystemP("uv2.exe -d TestRt.cmd") ;
# Decode and clean Tmpatl.out and write the results in
# cNewTdp\atl.out
Decode_Tmpatl.out_Into_Final_Intermediate_Report();
```

Breakpoint Mode Example: PowerPC-SingleStep

Source code of the program running on the target:

```
#define RTRT_FILE int
RTRT_FILE usr_open(char *fName) { return(1); }
void usr_writeln(RTRT_FILE f, char *s) { _simo(1,s,80); }
void usr_close(RTRT_FILE f) { ; }
char atl_buffer[100];
void main(void)
{
RTRT_FILE f ;
strcpy(atl_buffer, "Hello World");
f=usr_open("cNewTdp\\atl.out");
usr_writeln(f,atl_buffer);
usr_close(f);
}
```

Execution command from host:

```
simppc.exe TestRt.cmd
```

Content of `TestRt.cmd`:

```
debug a.out
break usr_close
```

```
break usr_writeln -g -c "read atl_buffer >> Tmpatl.out"  
go  
exit
```

As in the previous example, all the `usr_` functions are empty. The intelligence is deported into the `TestRt.cmd` script which a command file executed when the `SingleStep` debugger is launched.

It first loads the executable program, `a.out` by the `debug` command.

Then it sets a breakpoint at `usr_close` function, which serves as an exit-point, then set a breakpoint in the `usr_writeln` function. The `-g` flag of the `break` command indicates to continue the execution, whilst the `-c` specifies a command that should be executed before continuing. This command (`read`) writes the value of the `atl_buffer` variable into `Tmpatl.out`.

The `SingleStep` debugger then starts the execution. When it stops, it means that `usr_close` has been reached. It then executes the `exit` command, to terminate the debugging session.

The final result is contained into `Tmpatl.out`, and should be cleaned-up by the host (a little decoder in Perl for example) to produce the final expected `cNewTdp\atl.out` file containing "Hello World".

Based on the "Hello World" program, we should now focus on automating the execution step and having `atl.out` being written.

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.

Upgrading a Target Deployment Port

If you are upgrading from a previous version of Rational Test RealTime you must update all existing Target Deployment Ports to use them with the new version.

To Update a Target Deployment Port:

1. Start the TDP Editor and open the **.xdp** Target Deployment Port file.
2. Save the **.xdp** Target Deployment Port file.

Opening a Target Deployment Port

Target Deployment Ports can be viewed and edited with the TDP Editor supplied with Test RealTime.

To start the TDP Editor:

1. In Test RealTime, from the **Tools** menu, select **TDP Editor** and **Start**.
or
2. From the command line, type **tdpeditor**.

To open a TDP:

1. From the **File** menu, select **Open**.
2. In the targets directory, select an **.xdp** file and click **Open**.

Editing Customization Points

Use the Navigation Tree on the left to select customization points. A Target Deployment Port 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 Test RealTime 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 Test RealTime GUI integrated 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 TDP **cmd** subdirectory
- **Library Settings:** This section describes a set of source code files as well as a dedicated customization file (**custom.h**), which adapt the TDP to target platform requirements. This section is definitively the most complex and usually only requires customization for specialized platform TDPs (unknown RTOS, no RTOS, unknown simulator, emulator, etc.). These files are stored in the TDP **lib** subdirectory.
- **Parser Settings:** This section modifies the behavior of the parser in order to address non-standard compiler extensions, such as for example, non-ANSI extensions. This section allows Test RealTime to properly parse your source code, either for instrumentation or code generation purposes. The resulting files are stored in the TDP **ana** subdirectory.

To edit a customization point

1. In the Navigation Tree, 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. As a good practice, enter any remarks or comments in the Comments window.

After making any changes to a TDP, you must update the TDP in Test RealTime to apply the changes to a project.

Creating a Target Deployment Port

To create a new Target Deployment Port (TDP), the best method is to make a copy of an existing TDP that requires minimal modifications.

Naming Conventions

By convention, the TDP directory name starts with a **c** for the C and C++ languages, **ada** for the Ada language or **j** for Java, followed by the name of the development environment, such as the compiler and target platform.

The name of the **.xdp** file generally follows the same convention.

The name of the top-level node can be a user-friendly name, as it is to be displayed in the Test RealTime GUI.

To create a new TDP:

1. In the **TDP Editor**, from the **File** menu, select **New**.
2. In the **Language Selection** box, select the language used for this TDP.
The TDP Editor uses this information to create a template, which already contains most of the information required for the TDP.
3. Right click the top level node in the tree-view pane, which contains the name of the TDP. Select **Rename**. and enter a new name for this TDP. This name identifies the TDP in the Test RealTime GUI and can be more explicit than the TDP filename (see Naming Conventions).
4. As a good practice, in the **Comment** section, enter contact information such as your name and email address. This makes things easier when sharing the TDP with other users.

To save the new TDP:

1. From the **File** menu, select **Save As**.

2. Save your new TDP with a filename that follows the naming conventions described above. The actual location of the **.xdp** file is not relevant. The TDP Editor automatically creates a directory with the same name as the **.xdp** file and saves the **.xdp** file at that location.

Updating a Target Deployment Port

The Target Deployment Port (TDP) settings are read or loaded when a Test RealTime 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 Test RealTime:

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

Using a Post-generation Script

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

Migrating pre-V2002 TDPs to Present Format

This section describes the conversion of TDPs built for older versions of Rational Test RealTime to the new, unified format.

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

- ATTOL Coverage, UniTest and SystemTest
- Rational Test RealTime v2001A

TDPs created for later versions of Rational Test RealTime or Rational PurifyPlus RealTime 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.

unittest.ini

Template: any template

Copy all **unittest.ini** settings into the **Basic Settings** section of the TDP Editor.

Environment Variables

In the old TDP, the following line inserted the string "Value;" in the front of the current value of X:

```
ENV_X="Value; "
```

In the new TDP, the same syntax would set x equal to "Value; ". The new, proper syntax for insertion or concatenation is either:

```
ENV_X="Value;$ENV{ 'X' }"
```

or

```
ENV_X="$ENV{ 'X' };Value"
```

This concatenation and insertion algorithm is also valid for simple \$Ini fields.

Additionally, the following line now sets <Value> to X if X is not defined in the environment:

```
ENV_SET_IF_NOT_SET_X="<Value>"
```

Other Changes

The following fields are no longer used and can be deleted:

```
COMPIVER=" "  
CCSCRIPT="atl_cc.pl"  
LDSCRIPT="atl_link.pl"  
EXEScript="atl_exec.pl"  
STDFILE="atl_cc.def"
```

atuconf.h

Original Location: lib folder

Template: template.xdp

The following list is not exhaustive but it contains most of the typical TDP settings found in earlier Target Deployment Port releases. All TDP Editor references are located in the **Library Settings** section.

Old TDP Settings	New Customization Points
<code>#define ANSI_C</code>	Target Compiler Specifics Linkage Directives RTRT_KR The default value is unselected. Keep this setting unselected if ANSI_C is defined.
<code>#define USE_OLD 1</code>	Environmental Constraints sprintf function availability RTRT_SPRINTF If USE_OLD is set to 1, select RTRT_SPRINTF .

```
#define ATTOL_HEADER_MAIN int
main(void) {
empty_func();
}
```

For Test RealTime Testing Features
Test program entry point prototype and termination instruction
RTRT_MAIN_HEADER

RTRT_MAIN_HEADER equals **ATTOL_HEADER_MAIN**.

Note **empty_func()** was a function used to initialize a set of unused variables. This function is no longer needed. As a result, it is not necessary to redefine **main()** unless 'main' is not the name of the entry function.

```
Copy #define ATTOL_RETURN_MAIN
return (0);
```

For Test RealTime Testing Features
Test program entry point prototype and termination instruction
RTRT_MAIN_RETURN

RTRT_MAIN_RETURN equals the value of **ATTOL_RETURN_MAIN**.

```
#define USE_STRING 0
```

For Test RealTime Testing Features
String support
RTRT_STRING

If **USE_STRING** is set to **0**, deselect **RTRT_STRING**.

```
#define USE_FLOAT 0
```

For Test RealTime Testing Features
Floating-point number support
RTRT_FLOAT

If **USE_FLOAT** was set to **0**, deselect **RTRT_FLOAT**.

Either of the following statements:

- a. `#define ATL_EXIT exit(0)`
- b. `#define ATL_EXIT`
- c. `#define ATL_EXIT my_exit`

Environmental Constraints
exit function availability
RTRT_EXIT

Set **RTRT_EXIT** to **RTRT_STD** if **ATL_EXIT** is set to **exit(0)**.

Set **RTRT_EXIT** to **RTRT_NONE** if **ATL_EXIT**

is undefined.

Set **RTRT_EXIT** to **RTRT_USR** if **ATL_EXIT** was defined to a user-defined function, and copy the code from this function to the **usr_exit** section.

Either of the following statements:

- a. `#define STD_TIME_FUNC`
- b. `#define USR_TIME_FUNC`

```
int usr_time() {
    /* Return current clock value*/
    return(-1);
}
```
- c. No clock interface defined.

Clock Interface

RTRT_CLOCK

If **STD_TIME_FUNC** is defined, set **RTRT_CLOCK** to **RTRT_STD**.

If **USR_TIME_FUNC** is defined, set **RTRT_CLOCK** to **RTRT_USR** and copy the code from **usr_time** to the **usr_clock** section.

If no clock interface was defined, set **RTRT_CLOCK** equal to **RTRT_NONE**.

Either of the following statements:

- `#define STD_DATE_FUNC`
- b. `#define USR_DATE_FUNC`

```
void usr_date(char *s) {
    /* Sets s to the current date */
    s[0]=0;
}
```
- c. Nothing date interface defined

No longer needed; dates are supplied by the host.

Either of the following statements:

- a. `#define STD_IO_FUNC`
- b. `#define USR_IO_FUNC`

```
typedef int usr_file;
usr_file usr_open(char *name) {
    /* Open the file named name */
    usr_file x=1;
    return(x);
}

void usr_writeln(usr_file
file,char *str) {
    /* Print str into file and add \n
*/
    printf("%s",str);
}

void usr_close(usr_file file) {
    /* Close the file */
```

Data Retrieval and Error Output

Test and runtime analysis results output

RTRT_IO

If **STD_IO_FUNC** is defined, set **RTRT_IO** to the **RTRT_STD** value.

If **USR_IO_FUNC** is defined, set **RTRT_IO** to **RTRT_USR**, set **RTRT_FILE_TYPE** to the **usr_file** type, and write code for the functions **usr_open**, **usr_writeln** and **usr_close** into the corresponding **usr_open**, **usr_writeln** and **usr_close** sections.

If no data retrieval function was defined, set **RTRT_IO** to **RTRT_NONE**.

}

c. Nothing defined for IO

#define BUFFERED_IO

No longer necessary; this is the default mode.

attol_comm and attol_serv**Original Location:** lib folder**Template:** templatea.xdp

These files contained the implementation of any Ada restrictions made by target environment.

If your Ada environment implements the entire Ada standard, select the setting **Library Settings->Ada restrictions->std**

If your Ada environment does not allow the use of image attributes and of Ada exceptions, select the setting

Library Settings->Ada restrictions->smart

If your Ada environment does not allow the use of image attributes and Ada exceptions, and if the floating-point numbers were written from the target in hexadecimal mode, select the setting **Library Settings->Ada restrictions->dump**

private_io.ads**Original Location:** lib folder**Template:** templatea.xdp

Old settings are listed in the left column, updated settings in the right. All TDP Editor references are located in the **Library Settings** section.

With clauses;	with clauses for package specification
Affichage_chaine : constant :=100	Constant definitions->string_max_len
subtype priv_file is something;	Data types->PRIV_FILE
subtype longest_integer is something;	Data types->LONGEST_INTEGER
subtype longest_float is float;	Data types->LONGEST_FLOAT

Subtype priv_int is longest_integer;	Data types->INTEGER_32B
clock_present : constant boolean := FALSE ;	No longer used.
clock_offset : constant priv_int := 0;	Constant definitions->clock_offset This constant has been changed from integer to float.
clock_divide : constant priv_int := 1;	Constant definitions->clock_divide
clock_multiply : constant priv_int := 1;	Constant definitions->clock_multiply
clock_unit: constant string := "D0 "; -- D0 ms, D1 micro s, D2 cycles, D3 tops	Constant definitions->clock_unit
access_size : constant := 32;	Constant definitions->access_size
access_max : constant := (2**(access_size-1))-1;	Constant definitions->access_max
access_min : constant := - (2**(access_size-1));	Constant definitions->access_min
Any additional function/procedure specifications other than those for user_open, user_close, priv_open, priv_close, priv_writeln, priv_clock, priv_date.	User-defined function specifications

private_io.adb

Original Location: lib folder

Template: template.xdp

The code for the procedures priv_clock, priv_open, priv_close and priv_writeln must be reported with no modification in the settings

Library settings->Function bodies->Clock function/Open function/Close function/Write function

Be aware that some parameter names may have changed; for example, the parameter "fichier" is now "file".

Any additional **with** clauses that were written in private_io.adb have to be reported in the setting **Library settings->Function bodies->with clauses for package body**

Any other functions that were written in private_io.adb have to be reported in the setting

Library settings->Function bodies->User-defined function bodies

attolcov_io.ads

Original Location: lib folder

Template: templatea.xdp

Report the value of the constant atc_nb_bit_branch into the setting

Library Settings->Constants definitions->atc_nb_bit_branch

attolcov.opp

Original Location: <OldInstallDir>/.../atc/target/oldTdp

Template: templatec.xdp

Copy the contents of this file into the TDP editor in the section

Parser Settings->Component Testing and runtime analysis features for C++->Analyzer file configuration

atlcov.hpp

Original Location: <OldInstallDir>/.../atc/target/oldTdp

Template: templatec.xdp

Copy the contents of the old file into the TDP editor in the section

Parser Settings->Component Testing and runtime analysis features for C++->Header file configuration

atlcov.def

Original Location: <OldInstallDir>/.../atc/target/oldTdp

Template: templatec.xdp

Copy the contents of this file into the TDP editor in the section

Parser Settings->Runtime analysis features for C

atl_cc.def

Location: <OldInstallDir>/.../atu/target/oldCTdp/cmd

Template: templatec.xdp

Copy the contents of this file into the TDP editor in the section **Parser Settings->Component Testing and System Testing for C**

atl_cc.def for C++

Location: <OldInstallDir>/.../atu/target/oldCTdp/cmd

Template: templatec.xdp

Copy the contents of this file into the TDP editor in the section **Parser Settings->System Testing for C++**

standard-ada95.ads

Location: <OldInstallDir>/.../atc/target/oldTdp)

Template: templatea.xdp

Copy the contents of this file into the TDP editor to the **Parser Settings - Standard specification for Ada** section.

standard-ada83.ads

Location: <OldInstallDir>/.../atc/target/oldTdp)

Template: templatea.xdp

Copy the contents of this file into the TDP editor in the section **Parser Settings - Standard specification for Ada83**

standard*.*

Location: <OldInstallDir>/.../atu/target/oldTdp/ana)

Template: templatea.xdp

Here is the list of adaptations that must be reported in the TDP editor in the section **Parser Settings->Standard specification for Ada83**

These settings correspond to the previous use of Ada83 with the old Analyzer (without using Code Coverage).

- replace the boolean type definition with

```
type Boolean is _internal(BOOLEAN);
```

- replace the character type definition with
- ```
type Character is _internal(CHARACTER_8);
```
- delete the universal\_integer and universal\_float type definitions
  - delete all function definitions for all types.
  - add after the FLOAT type definition:

```
type _INTERNAL_INTEGER is _internal(INTERNAL_INTEGER);
```

```
type _INTERNAL_FLOAT is _internal(INTERNAL_FLOAT83);
```

The first is preferred; the second one corresponds to the case where Code Coverage is not available.

## Perl Scripts

### atl\_cc.pl

**Original Location:** cmd folder

**Template:** either

This file contained 2 functions.

Copy the **atl\_cc** function into the **Build Settings->Compilation function** section of the TDP Editor.

Copy the **atl\_cpp** function into the **Build Settings->Preprocessing function** section of the TDP Editor.

## Function prototypes

The function prototypes have changed. Old prototypes were:

```
sub atl_cc {
 my ($SourceFile, $OutputFile, $Includes, $AdditionalOptions)=@_;
}
```

and

```
sub atl_cpp {
 my ($SourceFile, $OutputFile, $Includes, $AdditionalOptions)=@_;
}
```

These are replaced by:

```
sub atl_cc ($$$$\@\@) {
 my ($lang, $src, $out, $cflags, $Defines, $Includes) = @_;
}
```

and

```
sub atl_cpp ($$$$\@\@) {
 my ($lang, $src, $out, $cppflags, $Defines, $Includes) = @_;
}
```

where

\$Defines and \$Includes are Perl references to arrays.

\$lang contains C, CPP, ADA or ADA83, based on the source file extension.

\$src and \$out are the source file and the output file to generate.

These functions must now compile both C or C++ source code. In fact, the same TDP should support both C and C++. To accomplish this dual functionality, simply make the appropriate edits for C++ in the Parser Settings section of the TDP Editor.

### atl\_link.pl

**Original Location:** cmd folder

**Template:** either

Copy the **atl\_cc** function into the **Build Settings->Link function** section of the TDP Editor.

Any other files required for the link phase, such as linker command files, boot assembly startup code, etc., should be added to the **Build Settings** section of the TDP editor by right-clicking the **Build Settings** node and selecting **Add Child->ASCII File**.

### Function prototype

The function prototype has changed. The old prototype was:

```
sub atl_link() {
 my ($ListObject, $OutputFile, $AdditionalFiles)=@_
}
```

This has been replaced by:

```
sub atl_link ($\@$\$) {
 my ($OutputFile, $Objects, $LdFlags, $LibPath, $Libraries)=@_ ;
}
```

where

\$Objects, \$LibPath are now given as references to Perl arrays.

All other parameters are scalar.

### atl\_exec.pl

**Original Location:** cmd folder

**Template:** either

Copy the **atl\_exec** function into the **Build Settings->Execution function** section of the TDP Editor.

Any other files required for the link phase, such as debugger scripts, mapping definitions, etc., should be added to the **Build Settings** section of the TDP editor by right-clicking the **Build Settings** node and selecting **Add Child->ASCII File**.

## Function prototype

The function prototype remains unchanged:

```
sub atl_exec($$$) {
 my ($exe, $out, $params) = @_;
}
```

## Other Perl Scripts

Any file other than **atl\_cc.pl**, **atl\_link.pl** or **atl\_exec.pl** must be added to the **Build Settings** section of the TDP editor by right-clicking the **Build Settings** node and selecting **Add Child->ASCII File**.





# Index

## \$

Value ..... 37

## A

A.out ..... 26, 28  
Access\_max ..... 41  
Access\_min ..... 41  
Access\_size ..... 41  
Ada ..... 19, 20, 26, 35, 41, 44, 47  
Ada TDPs  
    templatea.xdp ..... 19, 37  
Ada TDPs ..... 19, 37  
ADA83 ..... 44, 47  
Ada-written application ..... 20  
AdditionalOptions ..... 47  
Affichage\_chaine ..... 41  
ANSI\_C ..... 38  
ASCII ..... 22, 25  
Atc\_nb\_bit\_branch ..... 43  
Atl.out ..... 28  
Atl\_buffer ..... 26, 28  
Atl\_cc ..... 47  
Atl\_cc.def  
    C ..... 44  
Atl\_cc.def ..... 37, 44  
Atl\_cc.pl ..... 37, 47  
Atl\_cpp ..... 47  
Atl\_exec ..... 47  
Atl\_exec.pl ..... 37, 47  
ATL\_EXIT ..... 38  
ATL\_EXIT my\_exit ..... 38

Atl\_link ..... 47  
Atl\_link.pl ..... 37, 47  
Atlcov.def ..... 44  
Atlcov.hpp ..... 43  
Attol\_comm ..... 41  
ATTOL\_HEADER\_MAIN ..... 38  
ATTOL\_HEADER\_MAIN int ..... 38  
ATTOL\_RETURN\_MAIN ..... 38  
Attol\_serv ..... 41  
Attolcov.opp ..... 43  
Attolcov\_io.ads ..... 43  
Atuconf.h ..... 38

## B

B ..... 38  
BOOLEAN ..... 44  
Breakpoint-Mode ..... 28  
Bs\_priv\_close ..... 28  
BSD Socket Compliance ..... 24  
BSO-Tasking ..... 28  
BUFFERED\_IO ..... 38

## C

C  
    atl\_cc.def ..... 44  
    C ..... 19, 35  
    templatec.xdp ..... 19, 37  
C ..... 19, 20, 25, 26, 28, 35, 37, 38, 44, 47  
CCSCRIPT ..... 37  
CHARACTER\_8 ..... 44  
Characteristics ..... 19  
Child->ASCII File ..... 47

|                                 |        |                        |            |
|---------------------------------|--------|------------------------|------------|
| Clock Interface .....           | 23     | Func.....              | 28         |
| Clock_divide .....              | 41     | Function.....          | 41         |
| Clock_multiply .....            | 41     |                        |            |
| Clock_offset .....              | 41     | <b>H</b>               |            |
| Clock_present.....              | 41     | Heap Management.....   | 23         |
| Clock_unit.....                 | 41     | Hexa .....             | 41         |
| Cmd .....                       | 47     | High-Speed Link .....  | 23         |
| Commmand.....                   | 28     |                        |            |
| COMPILERVER.....                | 37     | <b>I</b>               |            |
| CPP .....                       | 47     | I/O.....               | 22, 26     |
| Crossview .....                 | 28     | Instrumentation .....  | 22         |
|                                 |        | Int.....               | 28         |
| <b>D</b>                        |        | INTERNAL_FLOAT .....   | 44         |
| D TestRt.cmd.....               | 28     | INTERNAL_FLOAT83 ..... | 44         |
| Data .....                      | 25     | INTERNAL_INTEGER.....  | 44         |
| Data Retrieval Capability ..... | 22     | IO .....               | 38         |
| Data Retrieval Examples.....    | 28     |                        |            |
| Definition.....                 | 44     | <b>K</b>               |            |
| Do.....                         | 27     | Keil MicroVision.....  | 28         |
|                                 |        |                        |            |
| <b>E</b>                        |        | <b>L</b>               |            |
| E-mail .....                    | 1      | Lang.....              | 47         |
| ENV .....                       | 37     | Ld.....                | 28         |
| ENV_SET_IF_NOT_SET_X .....      | 37     | LDSCRIPT .....         | 37         |
| ENV_X.....                      | 37     | LibPath.....           | 47         |
| Execution .....                 | 28     | Longest_float.....     | 41         |
| EXESCRPT.....                   | 37     | Longest_integer .....  | 41         |
|                                 |        | LynxOS .....           | 26         |
| <b>F</b>                        |        |                        |            |
| Fclose.....                     | 26, 28 | <b>M</b>               |            |
| Fichier.....                    | 43     | Malloc.....            | 23         |
| FILE .....                      | 26     | Memory Profiling.....  | 20, 22, 23 |
| FileName.....                   | 28     | MicroVision.....       | 28         |
| FLOAT.....                      | 44     | MT .....               | 20         |
| FName.....                      | 25, 28 | Mutex .....            | 23         |
| Fopen.....                      | 26, 28 |                        |            |
| Fprintf.....                    | 26, 28 | <b>O</b>               |            |
| Free Data Space.....            | 22     | OutputFile .....       | 47         |
| Free Stack Space.....           | 22     |                        |            |

**P**

P TestRt.cmd ..... 28  
 Performance Profiling ..... 20, 22, 23  
 Perl ..... 28, 47  
 POSIX  
     pthread\_key\_create ..... 23  
 POSIX ..... 23  
 Printf ..... 28, 38  
 Priv ..... 26, 28  
 Priv\_clock ..... 41, 43  
 Priv\_close ..... 25, 28, 41, 43  
 Priv\_date ..... 41  
 Priv\_file ..... 41  
 Priv\_int ..... 41  
 Priv\_writeln ..... 25, 26, 28, 41, 43  
 Private Data ..... 23  
 Private\_io.adb ..... 43  
 Private\_io.ads ..... 41  
 Procedure/method ..... 22  
 Pthread\_key\_create  
     POSIX ..... 23  
 Pthread\_key\_create ..... 23  
 PurifyPlus RealTime ... 1, 19, 20, 22, 23,  
 25, 27, 33  
 PurifyPlus RealTime Support ..... 27  
 Puchar-like ..... 26

**Q**

QNX ..... 26

**R**

Rational PurifyPlus RealTime 1, 19, 22,  
 27, 37  
 Rational Software Corporation ..... 1  
 Rational Test RealTime 1, 19, 22, 27, 37  
 Recode ..... 37  
 Recompilation ..... 33  
 Release ..... 19, 37  
 RENDEZVOUS synchronization ..... 24

Retrieval ..... 38  
 RTRT\_CLOCK  
     RTRT\_STD ..... 38  
     RTRT\_USR ..... 38  
 RTRT\_CLOCK ..... 38  
 RTRT\_EXIT  
     RTRT\_NONE ..... 38  
     RTRT\_STD ..... 38  
     RTRT\_USR ..... 38  
 RTRT\_EXIT ..... 38  
 RTRT\_FILE ..... 25, 28  
 RTRT\_FILE f ..... 25, 28  
 RTRT\_FILE f,char ..... 25, 28  
 RTRT\_FILE priv\_append ..... 25  
 RTRT\_FILE priv\_open ..... 25, 28  
 RTRT\_FILE\_TYPE  
     usr\_file ..... 38  
 RTRT\_FILE\_TYPE ..... 38  
 RTRT\_FLOAT ..... 38  
 RTRT\_IO  
     RTRT\_NONE ..... 38  
     RTRT\_STD ..... 38  
     RTRT\_USR ..... 38  
 RTRT\_IO ..... 38  
 RTRT\_MAIN\_HEADER ..... 38  
 RTRT\_MAIN\_RETURN ..... 38  
 RTRT\_NONE  
     RTRT\_EXIT ..... 38  
     RTRT\_IO ..... 38  
 RTRT\_NONE ..... 38  
 RTRT\_SPRINTF ..... 38  
 RTRT\_STD  
     RTRT\_CLOCK ..... 38  
     RTRT\_EXIT ..... 38  
     RTRT\_IO ..... 38  
 RTRT\_STD ..... 38  
 RTRT\_STRING ..... 38  
 RTRT\_USR  
     RTRT\_CLOCK ..... 38  
     RTRT\_EXIT ..... 38

RTRT\_IO .....38  
 RTRT\_USR.....38

**S**

S TestRt.cmd.....28  
 SCI.....22  
 Simo.....28  
 SingleStep .....28  
 Sio o atl.out.....28  
 SourceFile.....47  
 Src .....47  
 Standard  
     Standard-ada83.ads .....44  
     Standard-ada95.ads .....44  
 Standard.....44  
 STD\_DATE\_FUNC.....38  
 STD\_IO\_FUNC .....38  
 STD\_TIME\_FUNC.....38  
 STDFILE.....37  
 STDOUT.....28  
 Str.....38  
 Strcpy .....28  
 Subdirectory ..... 19, 33  
 SystemP .....28

**T**

Target  
     Target Deployment Port ..... 19, 35  
     Target Host .....25  
     Target Requirements .....20  
     Target System Categories .....26  
 Target .....27  
 Task Management .....24  
 TCP/IP .....24  
 TDP 19, 22, 26, 33, 35, 37, 38, 41, 43, 44,  
 47  
 TDP Editor  
     Using.....33  
 TDP Editor .....33  
 TDP Migration .....37

Technical Support ..... 1  
 Templatea.xdp  
     Ada TDPs..... 19, 37  
 Templatea.xdp ..... 19, 37, 41, 43, 44  
 Templatec.xdp  
     C ..... 19, 37  
 Templatec.xdp ..... 19, 37, 38, 43, 44  
 Test RealTime... 1, 19, 20, 22, 23, 25, 27,  
 33, 38  
 Test RealTime Support ..... 27  
 TestRt.cmd..... 28  
 Thread Self ..... 23  
 Thread-specific ..... 23  
 Timestamp..... 23  
 Tmain .....28  
 Tmpatl.out.....28  
 Typedef int usr\_file ..... 38

**U**

Unitest.ini ..... 37  
 Universal\_float..... 44  
 Universal\_integer ..... 44  
 UNIX .....26  
 USE\_FLOAT..... 38  
 USE\_OLD ..... 38  
 USE\_STRING ..... 38  
 User\_close..... 41  
 User\_open..... 41  
 Using  
     TDP Editor ..... 33  
 Using ..... 33  
 Usr\_clock..... 38  
 Usr\_close..... 38  
 Usr\_date..... 38  
 USR\_DATE\_FUNC ..... 38  
 Usr\_file  
     RTRT\_FILE\_TYPE ..... 38  
 Usr\_file..... 38  
 Usr\_file file ..... 38  
 Usr\_file file,char..... 38

Usr\_file usr\_open.....38  
USR\_IO\_FUNC .....38  
Usr\_open.....38  
Usr\_time.....38  
USR\_TIME\_FUNC.....38  
Usr\_writeln.....38

**V**

v2002 Release 2 ..... 19, 37  
VxWorks .....28

**W**

Whilest .....28  
WindShell tool .....28

**X**

Xdp ..... 19, 33  
Xdp file.....35  
Xml .....35  
XML-based TDP ..... 19

