Rational® PurifyPlus RealTime

Target Deployment Guide

VERSION: 2003.06.00

WINDOWS AND UNIX

Rational®
the software development company

support@rational.com
http://www.rational.com

Legal Notices

Version Number: 2003.06.00

Third Party Notices, Code, Licenses, and Acknowledgements

Target Deployment Guide Contents

Preface

Welcome to Rational Target Deployment Port technology.

This manual covers the Target Deployment Port (TDP) technology used by PurifyPlus RealTime to support embedded target platforms.

For general information on using PurifyPlus RealTime, please refer to one of these other documents provided with the product:

- **User Guide** for general information about using the product
- **Reference Manual** for scripting statement information and batch mode usage
- **Online Tutorial** to learn about using the product

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/pplus_rt.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 PurifyPlus 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 e-mail: 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

	e-mail: support@europe.rational.com
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 e-mail: support@apac.rational.com

About Target Deployment Port Technology

Rational Target Deployment Port (TDP) technology is a versatile, low-overhead method of enabling both limitless embedded target support and target-independent testing and runtime analysis capabilities. Used by all of the features of the product, the TDP technology is built to accommodate your compiler, linker, debugger, and target architecture. The TDP acts as a buffer between your target architecture and all runtime analysis assets, ensuring full product independence. The product assets don't have to change when the environment does.

Over twenty TDP configurations are shipped with PurifyPlus RealTime and additional TDPs are periodically made available on our website (this site can be accessed via the **Help** menu in the product). However, in order to help you achieve your test objectives, each Target Deployment Port can be tailored - via an interactive TDP editor - to match the specifics of the environment in which you are working.

The TDP Editor is accessible via the **Tools** menu in PurifyPlus RealTime.

Target Deployment Port technology key capabilities and benefits are:

- Compiler dialect-aware and linker-aware, for transparent test building.
- Painless retrieval of runtime analysis results from the target environment using JTAG probes, emulators or any available communication link, such as serial, Ethernet or file system.
- Powerful 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 TDP editor enabling simple, in-house TDP creation and customization

Planning a Target Deployment Port

Rational's Target Deployment Technology extends Rational PurifyPlus 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 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 PurifyPlus 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 PurifyPlus 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 PurifyPlus 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 PurifyPlus RealTime:

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

C, C++ and Ada Requirements

Each feature is listed as a column title.

	Code Coverage	Runtime Tracing	Memory Profiling	Performance Profiling
Data Retrieval Capability	Required	Required	Required	Required
Free Data Space	Required	Required	Required	Required
Free Stack Space	Required	Required	Required	Required
Mutex	For MT	For MT	For MT	For MT
Thread Self and PrivateData		For MT	For MT	For MT
Clock Interface				Required

Heap Management		Required	
High Speed Link	Required		
Task Management	For MT	For MT	For MT
Ada	N/A	N/A	N/A

- **For MT:** Required if the application under test is a multi-threaded application based on a preemptive multi-tasking mechanism.

Java Requirements

	Code Coverage	Runtime Tracing	Memory Profiling	Performance Profiling
Data Retrieval Capability	Required	Required	Required	Required
Free Data Space	Required	Required	Required	Required
Free Stack Space	Required	Required	Required	Required
Thread Adaptation	Required	Required		Required
Clock Adaptation				Required
JVMPI Support			Required	
Heap Settings			Required	

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 PurifyPlus 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).

Free Stack Space

The stack size should not be optimized for the requirements of the original application. The PurifyPlus 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 Memory Profiling and Performance Profiling features, but it is required for Performance Profiling. The goal is to read and return a clock value (Performance Profiling).

Heap Management

This customization is required by Memory Profiling only.

Memory Profiling needs 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, 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.

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 PurifyPlus 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 PurifyPlus 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, runtime analysis feature execution on the target machine will not be possible with PurifyPlus 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:

```
1 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 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 commmand indicates to continue the execution, whilest 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 than 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 obtained 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 PurifyPlus 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 PurifyPlus RealTime.

To start the TDP Editor:

1. In PurifyPlus 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 PurifyPlus 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 PurifyPlus 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 PurifyPlus 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 PurifyPlus 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 PurifyPlus 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 PurifyPlus 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 PurifyPlus 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 PurifyPlus 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

Upgrading from Pre-v2002 Target Deployment Ports

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.

unitest.ini

Template: any template

Copy all **unitest.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:

```
COMPILERVER=""
CCSCRIPT="atl_cc.pl"
LDSCRIPT="atl_link.pl"
EXESCRIPT="atl_exec.pl"
STDFILE="atl_cc.def"
```

atuconf.h

Original Location: lib folder

Template: templatec.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
`#define ANSI_C`	**Target Compiler Specifics** **Linkage Directives** **RTRT_KR** The default value is unselected. Keep this setting unselected if ANSI_C is defined.
`#define USE_OLD 1`	**Environmental Constraints** **sprintf function avaliability** **RTRT_SPRINTF** If **USE_OLD** is set to **1**, select **RTRT_SPRINTF**.
Either of the following statements:	**Environmental Constraints**

a. `#define ATL_EXIT exit(0)` b. `#define ATL_EXIT` c. `#define ATL_EXIT my_exit`	**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);`
`}`
	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.

`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 */`
`}`
c. Nothing defined for IO
	If no data retrieval function was defined, set RTRT_IO to RTRT_NONE.

#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 if 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 definitions->string_max_len**

constant :=100	
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: templatea.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

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

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 defintion 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

$

A

B

C

D

E

F

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X