Adapting Rational Rose RealTime for Target Environments

RATIONAL ROSE® REALTIME

VERSION: 2003.06.00

PART NUMBER: 800-026121-000

WINDOWS/UNIX

Rational
the software development company

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

Part Number: 800-026121-000
Version Number: 2003.06.00

Contents

Preface xiii

Audience. xiii

Other Resources xiv

Rational Rose RealTime Integrations With Other Rational Products xiv

Contacting Rational Customer Support xv

1 Using the TargetRTS Wizard. 17

Overview of the TargetRTS Wizard 18

Understanding the TargetRTS 18

Maintaining TargetRTS Libraries using the TargetRTS Wizard 19

Managing Your TargetRTS Configurations 21

Duplicating a Configuration 21

NoRTOS Target Base 25

Editing a Configuration 25

Understanding the makefiles 27

Editing the Target 28

Editing the Libset 30

Modifying a Configuration. 31

Building Configurations. 32

Deleting Configurations 34

Creating Ports between C and C++ 35

2 Introducing the TargetRTS 41

Overview. 41

Other Resources 42

3 Before Starting a Port 43

OS Knowledge and Experience 43

Toolchain Functionality 44

OS Capabilities. 44

Simple non-Rational Rose RealTime Program on Target. 45

TCP/IP Functionality 46
Floating Point Operations. 46
Standard Input/Output Functionality 46
Debugging 47
Training. 47
Support 47
What to do Before Calling Rational Customer Support 47
4 Porting the TargetRTS 49
Overview 49
Phases of a Port 50
Choose a Configuration Name. 50
Target Name 51
Libset Name 52
Building Rational Rose RealTime Applications for Targets without Operating Systems 52
Benefits of Using a NoRTOS Configuration 52
Using a NoRTOS Configuration 52
Verification. 53
Creating a Setup Script (setup.pl) 54
TargetRTS makefiles 56
Default makefile 59
Target makefile. 64
Libset makefile 65
Config makefile 65
5 Porting the TargetRTS for C 71
Configuring the TargetRTS. 71
Platform-specific Implementation 75
Method RTTimespec_clock_gettime(timespec) 76
Constructor RTThread_construct(this,job,priority,stacksize) 76
Class RTMutex 76
Class RTSyncObject. 77
main() function 78
Class RTMain 78
Method RTStdio_putString() 79

Method RTDebuggerInput_nextChar() 79
Class RTTcpSocket 79
Class RTIOMonitor 80
File main.c 80
Adding New Files to the TargetRTS 80
The MANIFEST.c File 80
Regenerating make Dependencies 81
6 Porting the TargetRTS for C++ 83
Configuring the TargetRTS 83
Platform-specific Implementation 88
Method RTTimespec::getclock() 89
Constructor RTThread::RTThread() 90
Class RTMutex 90
Class RTSyncObject 90
main() function 91
Class RTMain 92
Method RTDiagStream::write() 92
Method RTDebuggerInput::nextChar() 93
Class RTTcpSocket 93
Class RTIOMonitor 93
File main.cc 93
Adding New Files to the TargetRTS 93
The MANIFEST.cpp File 94
Regenerating make Dependencies 94
7 Modifying the Error Parser 95
Overview of the Error Parser 95
How the Error Parser Works 96
The Error Parsing Rules 96
How "rtcomp.pl" Integrates With the Compiler 97
Reusing an Existing Error Parser 98
Creating a New Error Parser 98
8 Testing the TargetRTS Port 101
Overview 101
HelloWorld Model 101
Other Test Models 102
Other Resources 102

9 **Tuning the TargetRTS 103**
Disabling TargetRTS Features for Performance 103
Target Compiler Optimizations 103
Target Operating System Optimizations 104
Specific TargetRTS Performance Enhancements .104
10 **Common Problems and Pitfalls 105**
Overview 105
Problems and Pitfalls with Target Toolchains 106
Compiler Optimizations 106
Linker Configuration File 106
System Include Files 106
Problems and Pitfalls with TargetRTS/RTOS Interaction 107
Return Codes for POSIX Function Calls 107
Thread Creation 107
Real-time Clock 107
Real-time Clock 107
Signal Handlers 109
RTOS Supplies `main()` Function 109
Default Command Line Arguments 109
Exiting Application 110
Problems and Pitfalls with Target TCP/IP Interfaces 111
gethostbyname() reentrancy 111
select() statement 111
11 **TargetRTS Porting Example 113**
Overview 113
Choosing the Configuration Name 113
Create Setup Script 114
Create makefiles 115
Libset makefile 115
Target makefile 117
Configuration makefile 118
TargetRTS Configuration Definitions 118
Code Changes to TargetRTS Classes 119
Building the New TargetRTS 122

12 Customizing for Target Control and Observability 123
Introduction 123
Model Compilation and Target Control. 124
Intended Audience 124
Target Control. 125
Target Control Modes. 125
Manual Mode 125
Basic Mode 125
Debugger Mode 126
Target Control Scripts. 126
Menu Commands. 127
Reset 127
Load 128
Unload 129
Execute 130
Terminate 131
General Issues. 132
Third-Party Source Code Debugger Integration. 133
Registering Threads on UNIX 133
Calling Sequence 133
Debugger DLL API 135
Get DLL Capabilities 136
Create Debug Session. 137
Destroy Debug Session 138
Initialize Debugger 138
Cleanup Debugger. 139
Start Debugger 140
Stop Debugger. 140
Set Callback. 141
Event Callback Function 142
Set Source Search Path 143
Set Breakpoint in File. 143
Set Breakpoint At Function 144
Clear Breakpoint 145
Set DllTrace 146

Index. 147

List of Tables

Table 1	Required Operating System Features for the C and C++ TargetRTS.	44
Table 2	Example Configuration Names	51
Table 3	Variables in the Setup.pl Script	55
Table 4	Make Macro Definitions	66
Table 5	Preprocessor Definitions	72
Table 6	Required TargetRTS Classes and Functions.	75
Table 7	Preprocessor Definitions	84
Table 8	Required TargetRTS Classes and Functions.	89
Table 9	Tools Required for Building the TargetRTS for C	115
Table 10	Important Toolchain Command Line Options	116
Table 11	Quick Summary of Common C TargetRTS Source File Changes	120
Table 12	Quick Summary of Common C++ TargetRTS Source File Changes	121

Preface

This manual describes how you can quickly and easily customize your existing TargetRTS libraries, and simplify the porting of the TargetRTS to new targets. With the **TargetRTS Wizard**, you can quickly create a new TargetRTS configuration, modify or duplicate an existing configuration, or delete an existing configuration that is no longer required.

Later chapters describe the properties for porting the TargetRTS to a new target environment.

This manual also describes how to add support to Rational Rose RealTime for target control and observability, and how to integrate Rational Rose RealTime with source code debuggers.

This manual is organized as follows:

- *Using the TargetRTS Wizard* on page 17
- *Introducing the TargetRTS* on page 41
- *Before Starting a Port* on page 43
- *Porting the TargetRTS* on page 49
- *Porting the TargetRTS for C* on page 71
- *Porting the TargetRTS for C++* on page 83
- *Modifying the Error Parser* on page 95
- *Testing the TargetRTS Port* on page 101
- *Tuning the TargetRTS* on page 103
- *Common Problems and Pitfalls* on page 105
- *TargetRTS Porting Example* on page 113
- *Customizing for Target Control and Observability* on page 123

Audience

This guide is intended for all readers including managers, project leaders, analysts, developers, and testers.

This guide is specifically designed for software development professionals familiar with the target environment they intend to port to.

Other Resources

- Online Help is available for Rational Rose RealTime.

  Select an option from the **Help** menu.

  All manuals are available online, either in HTML or PDF format. To access the online manuals, click **Rational Rose RealTime Documentation** from the **Start** menu.

- To send feedback about documentation for Rational products, please send e-mail to `techpubs@rational.com`.
- 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`.
- For articles, discussion forums, and Web-based training courses on developing software with Rational Suite products, join the Rational Developer Network by selecting **Start > Programs > Rational Suite > Logon to the Rational Developer Network.**

Rational Rose RealTime Integrations With Other Rational Products

Integration	Description	Where it is Documented
Rose RealTime–ClearCase	You can archive Rose RT components in ClearCase.	▪ *Toolset Guide: Rational Rose RealTime* ▪ *Guide to Team Development: Rational Rose RealTime*
Rose RealTime–UCM	Rose RealTime developers can create baselines of Rose RT projects in UCM and create Rose RealTime projects from baselines.	▪ *Toolset Guide: Rational Rose RealTime* ▪ *Guide to Team Development: Rational Rose RealTime*
Rose RealTime–Purify	When linking or running a Rose RealTime model with Purify installed on the system, developers can invoke the Purify executable using the **Build > Run with Purify** command. While the model executes and when it completes, the integration displays a report in a Purify Tab in RoseRealTime.	▪ Rational Rose RealTime Help ▪ *Toolset Guide: Rational Rose RealTime* ▪ *Installation Guide: Rational Rose RealTime*

Integration	Description	Where it is Documented
Rose RealTime–RequisitePro	You can associate RequisitePro requirements and documents with Rose RealTime elements.	▪ *Addins, Tools, and Wizards Reference: Rational Rose RealTime* ▪ *Using RequisitePro* ▪ *Installation Guide: Rational Rose RealTime*
Rose RealTime–SoDa	You can create reports that extract information from a Rose RealTime model.	▪ *Installation Guide: Rational Rose RealTime* ▪ *Rational SoDA User's Guide* ▪ SoDA Help

Contacting Rational Customer Support

If you have questions about installing, using, or maintaining this product, contact Rational Customer Support.

Your Location	Telephone	Facsimile	E-mail
North, Central, and South America	+1 (800) 433-5444 (toll free) +1 (408) 863-4000 Cupertino, CA	+1 (781) 676-2460 Lexington, MA	support@rational.com
Europe, Middle East, Africa	+31 20 4546-200 Netherlands	+31 20 4546-201 Netherlands	support@europe.rational.com
Asia Pacific	+61-2-9419-0111 Australia	+61-2-9419-0123 Australia	support@apac.rational.com

Note: When you contact Rational Customer Support, please be prepared to supply the following information:

- Your name, company name, telephone number, and e-mail address
- Your operating system, version number, and any service packs or patches you have applied
- Product name and release number
- Your Service Request number (SR#) if you are following up on a previously reported problem

When sending email concerning a previously-reported problem, please include in the subject field: "[SR#XXXXX]", where XXXXX is the Service Request number of the issue. For example, "[SR#0176528] - New data on rational rose realtime install issue ".

Using the TargetRTS Wizard

1

Contents

This chapter is organized as follows:

- *Overview of the TargetRTS Wizard* on page 18
- *Understanding the TargetRTS* on page 18
- *Maintaining TargetRTS Libraries using the TargetRTS Wizard* on page 19
- *Duplicating a Configuration* on page 21
- *NoRTOS Target Base* on page 25
- *Editing a Configuration* on page 25
- *Understanding the makefiles* on page 27
- *Editing the Target* on page 28
- *Editing the Libset* on page 30
- *Modifying a Configuration* on page 31
- *Building Configurations* on page 32
- *Deleting Configurations* on page 34
- *Creating Ports between C and C++* on page 35

Overview of the TargetRTS Wizard

The **TargetRTS Wizard** facilitates the management of the TargetRTS source tree, allows easy customization of existing TargetRTS libraries, and simplifies porting of the TargetRTS to new targets. With the **TargetRTS Wizard**, you can create a new TargetRTS configuration, modify or duplicate an existing configuration, or delete an existing configuration that is no longer required.

Note: Porting to a new operating system or a **libset** is not a trivial process, even with the help of the **TargetRTS Wizard**. You must be familiar with the operating system, the toolchain, the TargetRTS, and its layout.

Note: The figures for the **TargetRTS Wizard** dialogs are for the C++ language.

Understanding the TargetRTS

The TargetRTS is the set of run-time services that provide a framework in which a Rational Rose RealTime model can run. The **TargetRTS Wizard** simplifies the activities of building, configuring, managing, and customizing the TargetRTS libraries and build environment.

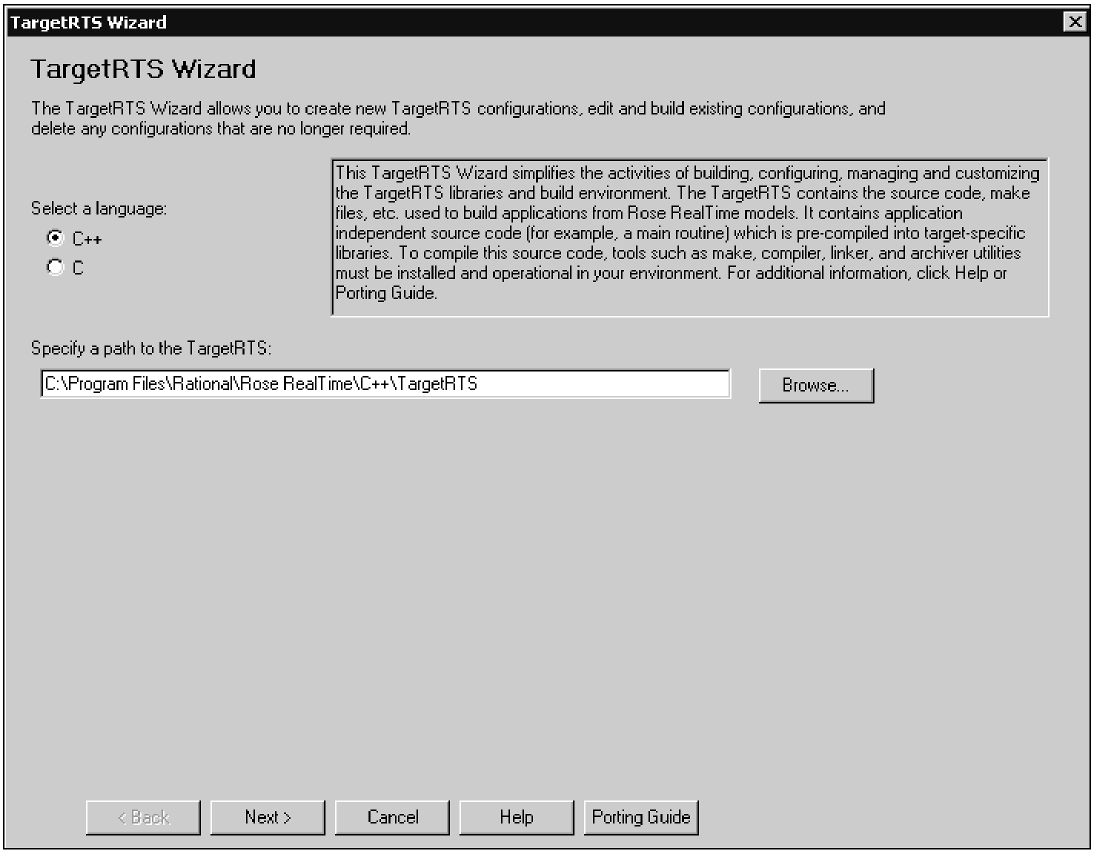
The TargetRTS contains the required parts, such as source code and **makefiles**, used to build applications from Rational Rose RealTime models. It contains application-independent source code which is pre-compiled into target-specific libraries.

To compile this source code, the tools (such as **make**, compiler, linker, and archiver utilities) must be installed and operational in your environment.

Maintaining TargetRTS Libraries using the TargetRTS Wizard

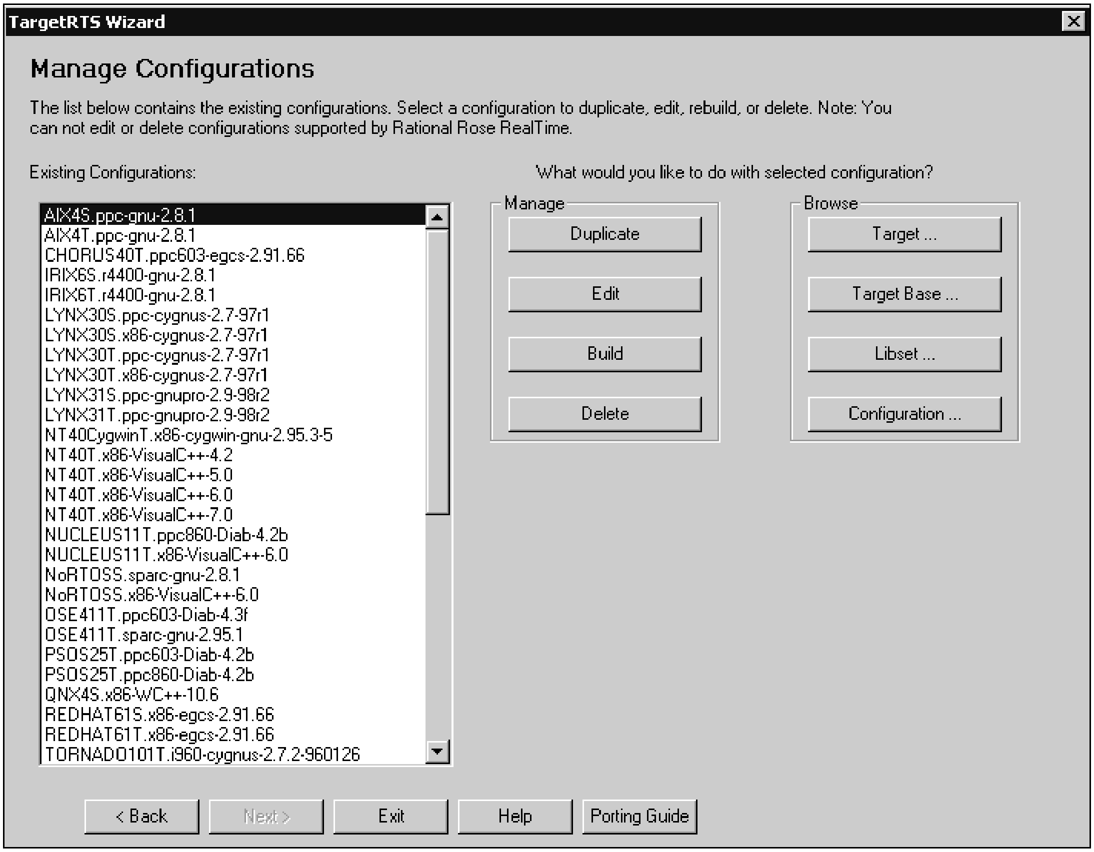
To access the TargetRTS Wizard, click **Tools > TargetRTS Wizard**. Figure 1 shows the first pane in the TargetRTS Wizard.

Figure 1 TargetRTS Wizard - First Pane



Use this pane to locate the TargetRTS tree for the **TargetRTS Wizard**, then click **Next**.

Figure 2 TargetRTS Wizard - Manage Configurations Panel



The **Existing Configurations** box contains a list of all your configurations. For some configurations, you can duplicate, edit, build, or delete them.

Note: Those configurations distributed with Rational Rose RealTime are read-only and cannot be edited or deleted. To modify a Rational Rose RealTime configuration that is read-only, select the configuration and click **Duplicate**.

For additional information on modifying a Rational Rose RealTime configuration, see *Duplicating a Configuration* on page 21.

Managing Your TargetRTS Configurations

When managing configurations with the **TargetRTS Wizard**, you can:

- Click **Duplicate** for *Duplicating a Configuration* on page 21
- Click **Edit** for *Editing a Configuration* on page 25
- Click **Build** for *Building Configurations* on page 32
- Click **Delete** for *Deleting Configurations* on page 34
- Click a browse option for browing directories

Browsing Directories

You can also browse other directories for configurations to quickly view the files necessary for each configuration. The **TargetRTS Wizard** opens the files in the external editor you specified in the **Path** box on the **Editor** tab by clicking **Tools > Options**.

Duplicating a Configuration

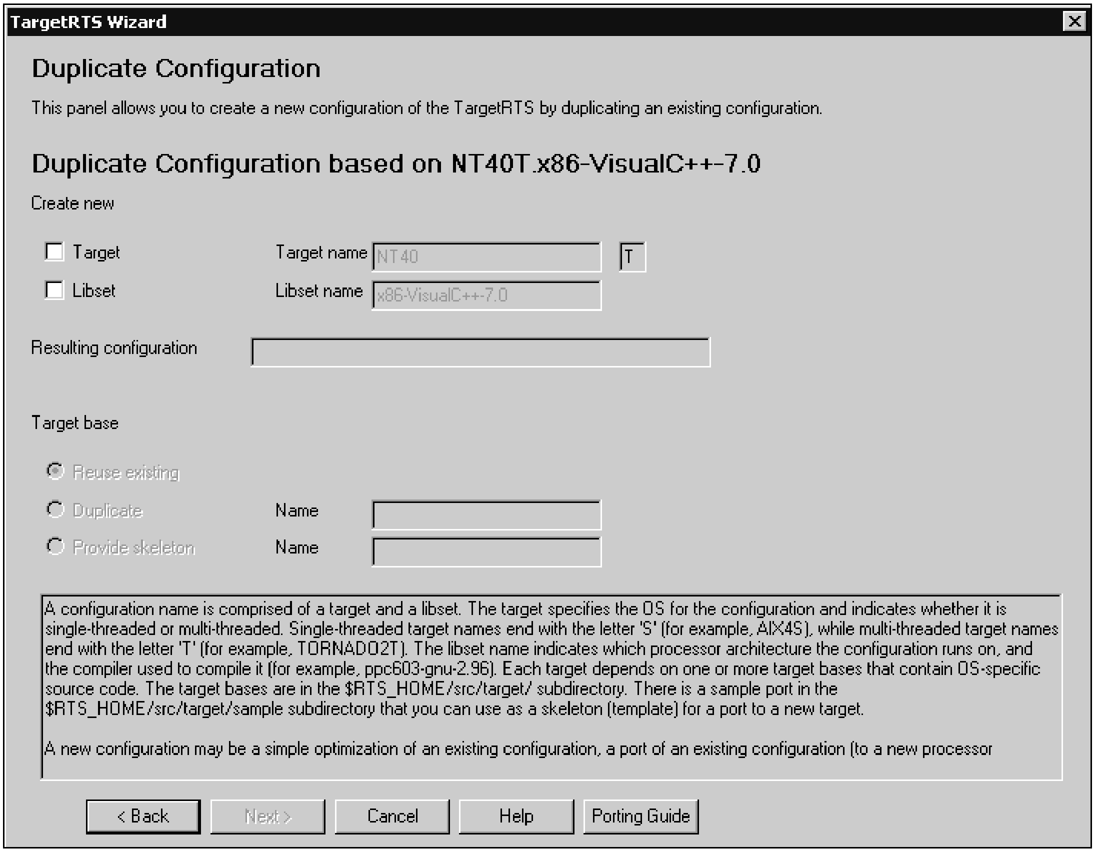
Duplicating an existing configuration is the first step to creating new configurations for new ports, or for a custom version of the same configuration.

Note: The configuration name is an important identifier of the TargetRTS. It identifies the operating system, hardware architecture, and compiler.

To duplicate a configuration:

1. From the **Existing Configuration** box on the **Manage Configuration** pane, select a configuration.
2. In the **Manage** box, click **Duplicate**.
3. Click **Next**.

Figure 3 TargetRTS Wizard - Duplicate Configuration Panel



A new configuration can be:

- A simple optimization of an existing configuration
- A port of an existing configuration (to a new processor architecture or to a new compiler)
- A port to an entirely new OS

Since the new configuration must have a new name, you must create a new **Target**, a new **Libset**, or both.

The **Target** specifies the OS for the configuration and indicates whether it is single-threaded or multi-threaded. Single-threaded target names end with the letter 'S' (for example, AIX4S), while multi-threaded target names end with the letter 'T' (for example, TORNADO2T). The **Libset** name indicates which processor architecture the configuration runs on, and the compiler used to compile it (for example, ppc603-gnu-2.96). Each target depends on one or more target bases that contain OS-specific source code. The **Target bases** are in the $ROSERT_HOME/src/target/ directory.

Note: There is a sample port in $ROSERT_HOME/src/target/sample that you can use as a skeleton (a template) for a port to a new target.

4 Under the **Create new** label, if you select **Target**, you can specify a new name in the **Target name** box.

The **Target name** represents the implementation-specific components of the TargetRTS. These components are generally specific to a given configuration, of a given version, of a given operating system. The **Target name** is also used to name the configuration of the target, such as single-threaded versus multi-threaded. The target name is defined as follows:

<target> ::= *<OS_name><OS_version><RTS_config>*

The components of `<target>` are defined as follows:

<*OS_name*> identifies the operating system (for example, SUN)

<*OS_version*> identifies the major version of that operating system.

Note: Do not use periods in the OS version because this will confuse the **make** utility when it attempts to build the TargetRTS.

<RTS_config> is a single letter that identifies the configuration; "**S**" for a single-threaded configuration, and "**T**" for a multi-threaded configuration.

For example:

```
SUN5T
```

If you select **Target**, the **Target base** area of the panel becomes enabled. The **Target base** controls the OS-specific source code used for the new target. If the duplicate configuration is a port to a different operating system, a new target base will be necessary. Duplicating a target base copies the target base used for the original target; you will likely have to modify the new base, as required. A skeleton target base contains only stubs for functions that are required for any target. These functions must be fully implemented and you will likely have to add additional functions.

You can specify a **NoRTOS** target base that does not use any OS-specific calls. For more information on using a **NoRTOS** target base, see *NoRTOS Target Base* on page 25.

Note: To reuse existing targets to create new configurations, you can specify the name of an existing **target** in the **Target name** box. The **TargetRTS Wizard** creates a new configuration (using the selected libset and the existing **target**), and the **target** will not be copied.

5 Under the **Create new** label, if you select **Libset**, you can specify a new name in the **Libset name** box.

Although the actual **libset** names can be chosen arbitrarily, by convention, those used by Rational Rose RealTime are defined as follows:

<*libset*> ::= <*processor*>-<*compiler_name*>-<*compiler_version*>

The components of <*libset*> are defined as follows:

<*processor*> identifies the processor architecture name

<*compiler_name*> identifies the compiler product name, or the vender for the compiler.

<*compiler_version*> identifies the compiler version. It is acceptable to use periods in the compiler version text.

For example:

```
sparc-gnu-2.8.1
```

Note: To reuse existing **libsets** to create new configurations, you can specify the name of an existing **libset** in the **Libset name** box. The **TargetRTS Wizard** creates a new configuration (using the selected target and the existing **libset**), and the **libset** will not be copied.

The **Resulting Configuration** box contains the name of the configuration.

6 Click **Next**.

The **TargetRTS Wizard** presents a **Summary** dialog that identifies all of the actions it will perform.

7 Click **Next**.

When appropriate, the **TargetRTS Wizard** displays a **Work Order** dialog containing a list of items that may require user intervention.

8 Click **Next**.

NoRTOS Target Base

Both the C and C++ TargetRTS have a NoRTOS target base that does not use any OS-specific calls. This means that a NoRTOS target base will work with any OS, or it will work without an OS. A single-threaded target (NoRTOSS) uses the NoRTOS target base.

Often, when porting to a new operating system, it is useful to create the **libset**, then use it with the NoRTOSS target to verify that the toolchain works properly. After the OS-independent version of the port is complete, you can use its **libset** with a new target to make the full port.

To create a configuration that uses a NoRTOS target base using the TargetRTS Wizard:

1. From the **Existing Configuration** box on the **Manage Configuration** pane, select a configuration that uses the **NoRTOSS** target.
2. In the **Manage** box, click **Duplicate**.
3. Under the **Create new** label, select **Libset**.
4. In the **Libset name** box, specify an appropriate name for the **libset**.

   Note: For some situations where the new **libset** is similar to an already existing **libset**, it may be useful to specify the name of that existing **libset** into the **Libset name** box. The **TargetRTS Wizard** will then reuse that **libset** in the new configuration. The resulting configuration can then be duplicated to properly name the new **libset**. The **TargetRTS Wizard** will then use this **libset** with the new target to create the new configuration.

Editing a Configuration

After you duplicate a configuration, you can edit the new configuration. You can edit the **target**, the **libset**, or only the configuration itself.

Note: You cannot edit the configurations that are included with Rational Rose RealTime, nor the targets and **libsets** that these configurations use. You can only edit the configurations that you duplicated previously.

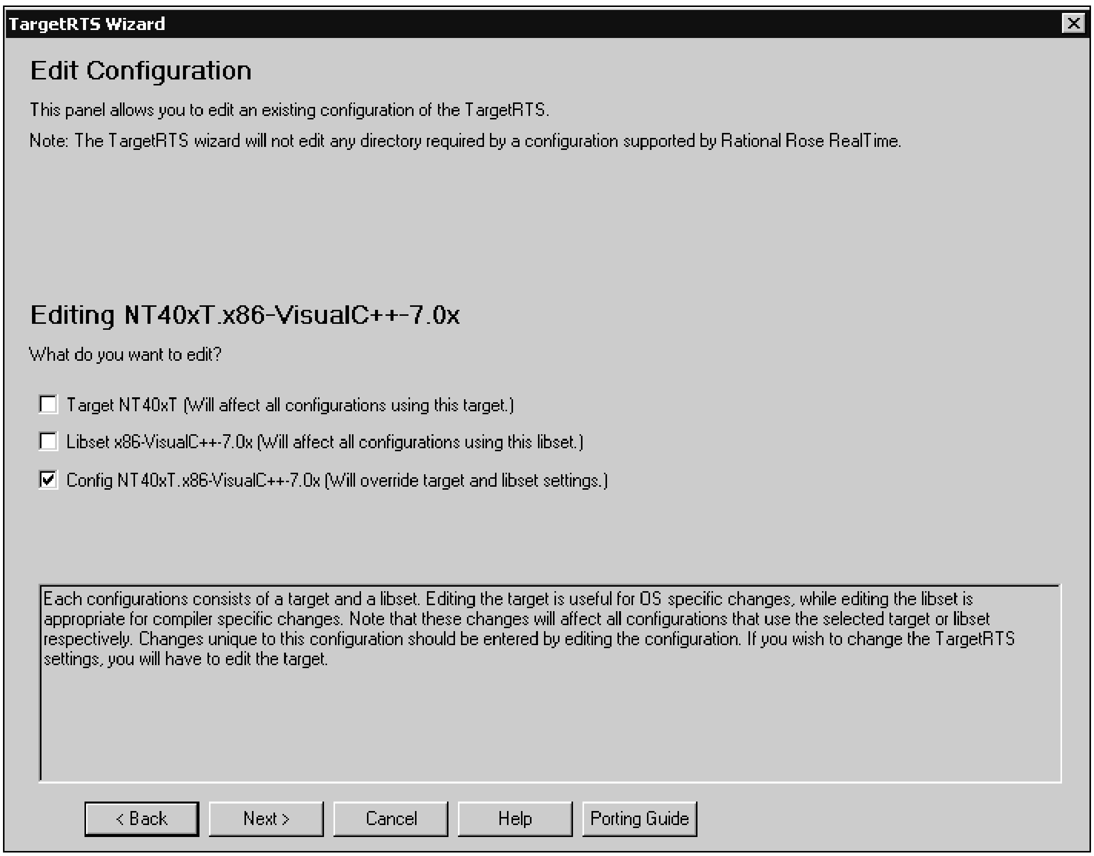
Every configuration is comprised of a target and a **libset**. Editing the target is useful for OS-specific changes, while editing the **libset** is appropriate for compiler-specific changes. To change the TargetRTS settings, you will need to edit the target.

Note: These changes affect all configurations that use the selected target or **libset**.

Figure 4 shows the **Edit Configuration** pane in the **TargetRTS Wizard**. From this pane, you can specify whether you want to edit a combination of the target, **libset**, or the configuration itself. For more information on editing, see the following:

- *Editing the Target* on page 28
- *Editing the Libset* on page 30
- *Modifying a Configuration* on page 31

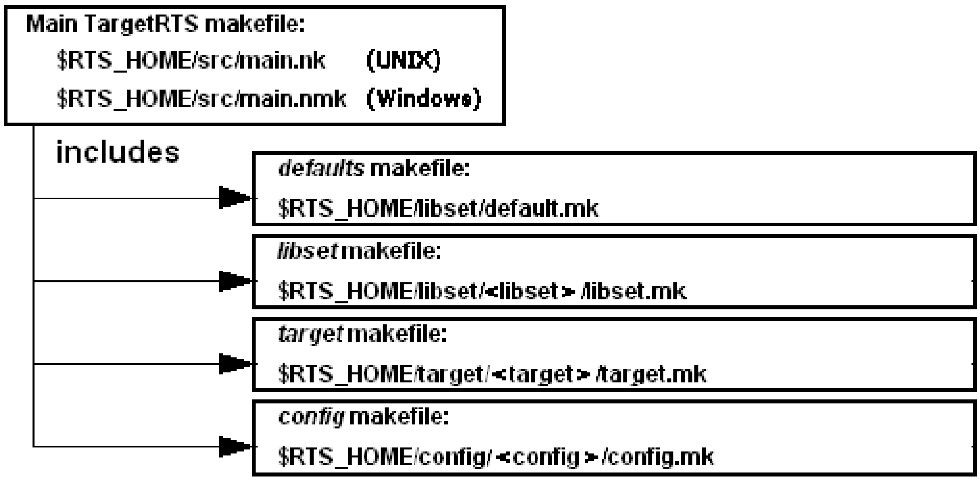
Figure 4 TargetRTS Wizard - Edit Configuration Panel



Understanding the makefiles

When you edit a configuration using the **TargetRTS Wizard**, you are modifying properties in one or more **makefiles**. Figure 5 shows the **makefiles** that you can update when specifying particular options while using the **TargetRTS Wizard**.

Figure 5 TargetRTS makefiles



The default.mk, libset.mk, target.mk, and config.mk **makefiles** are used to compile both the TargetRTS libraries and the model. The target.mk, libset.mk and config.mk **makefiles** override the defaults defined in $ROSERT_HOME/libset/default.mk. These are the **makefiles** that you can edit using the **TargetRTS Wizard**.

The main.nmk (**nmake** for Windows) or main.mk (**make** for UNIX) is the main definition for compiling the TargetRTS libraries. These **makefiles** should not be customized, and will not be discussed further in this document.

The default.mk file contains the default macro definitions that may be overridden by the platform-specific **makefiles**.

The target.mk file contains the definition specific to the target operating system.

The libset.mk file contains the definition specific to the compiler.

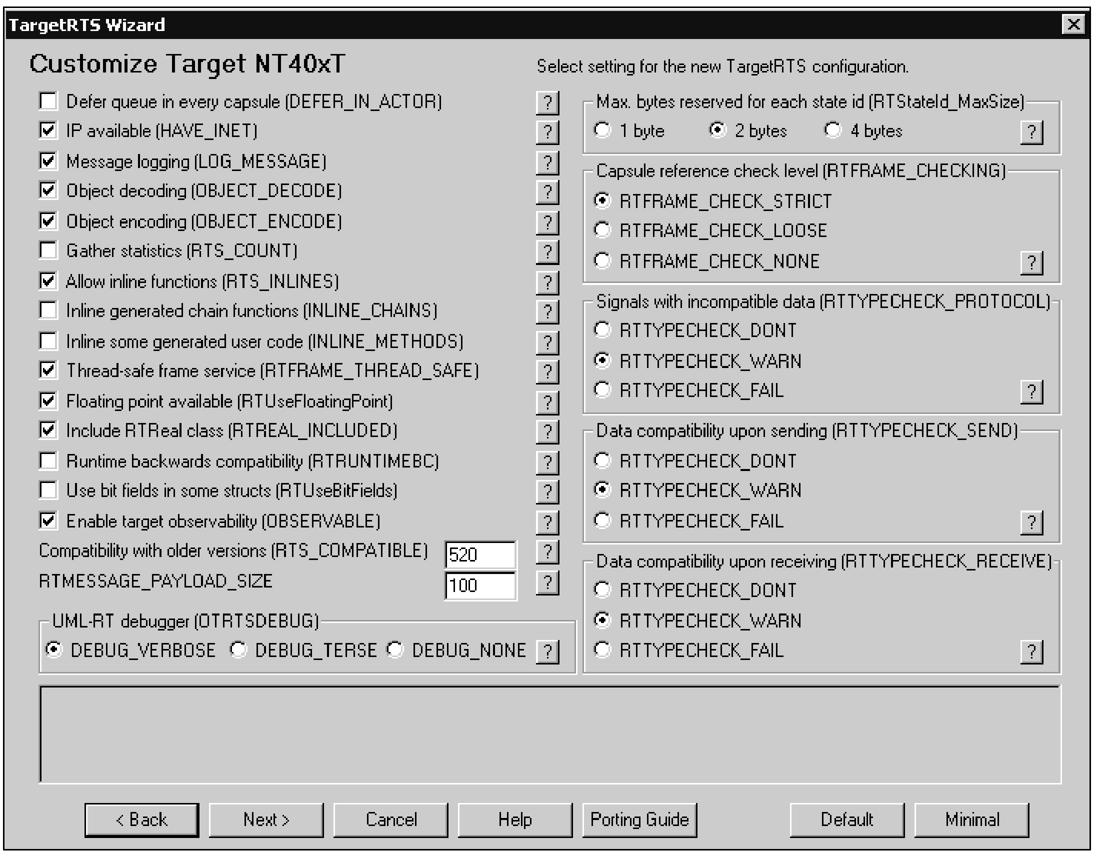
The config.mk file contains the definition specific to the combination of the compiler, operating system, and TargetRTS configuration.

Editing the Target

You can edit the target to create a custom TargetRTS library. Figure 6 shows the C++ options used to configure the run-time system.

Note: The Customize Target panel in the **TargetRTS Wizard** for C is similar to C++; however, some of the individual options differ. For additional information, click the question mark opposite each option.

Figure 6 TargetRTS Wizard - Customize Target Panel

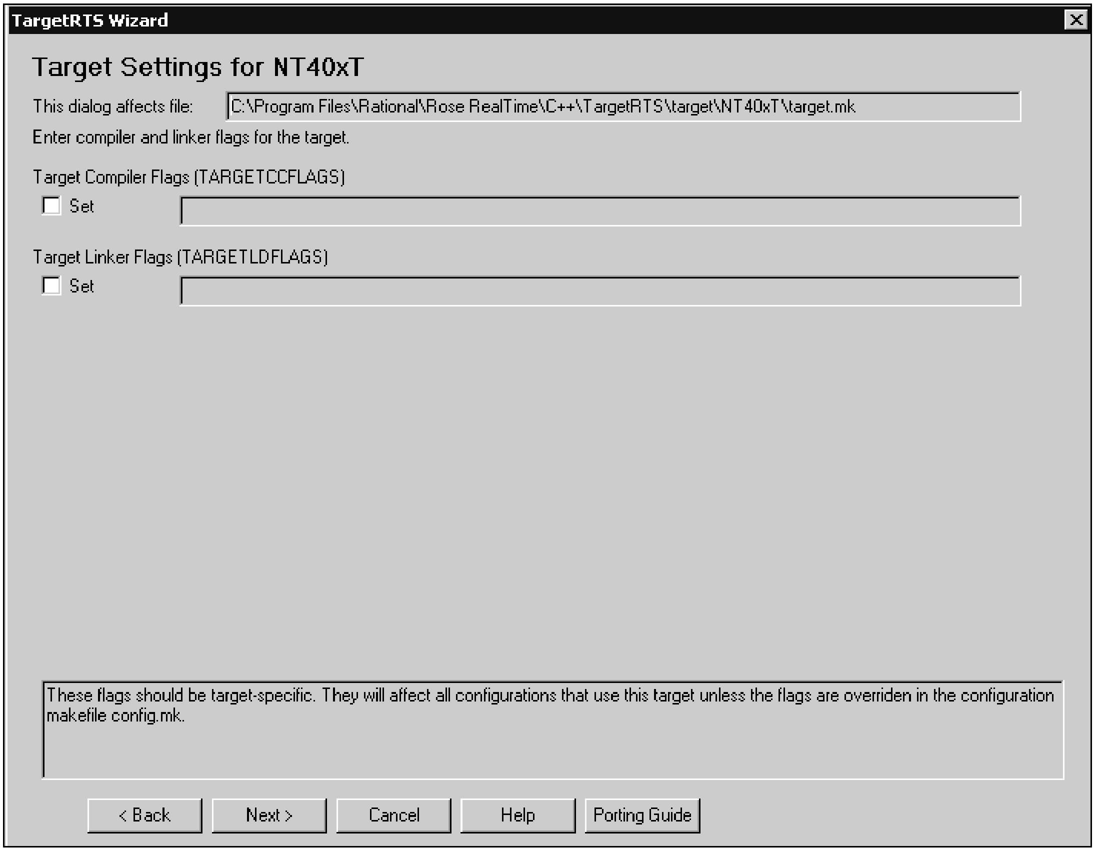


Note: Each entry is associated with a macro that controls that particular option in the TargetRTS source. Click **Default** to set all the options back to their defaults, and click **Minimal** to set the options for a much smaller and faster run-time system.

After you specify your required target options, click **Next**.

Figure 7 shows the **Target Settings** panel used to control compiler and linker flags for the target. The **Set** options control which variables are defined in the target.mk file for that particular target.

Figure 7 TargetRTS Wizard - Target Settings Panel



Target Compiler Flags (TARGETCCFLAGS)

Adds target-specific compilation flags in the file target.mk.

Target Linker Flags (TARGETLDFLAGS)

Redefines the target linker flags in the target.mk file.

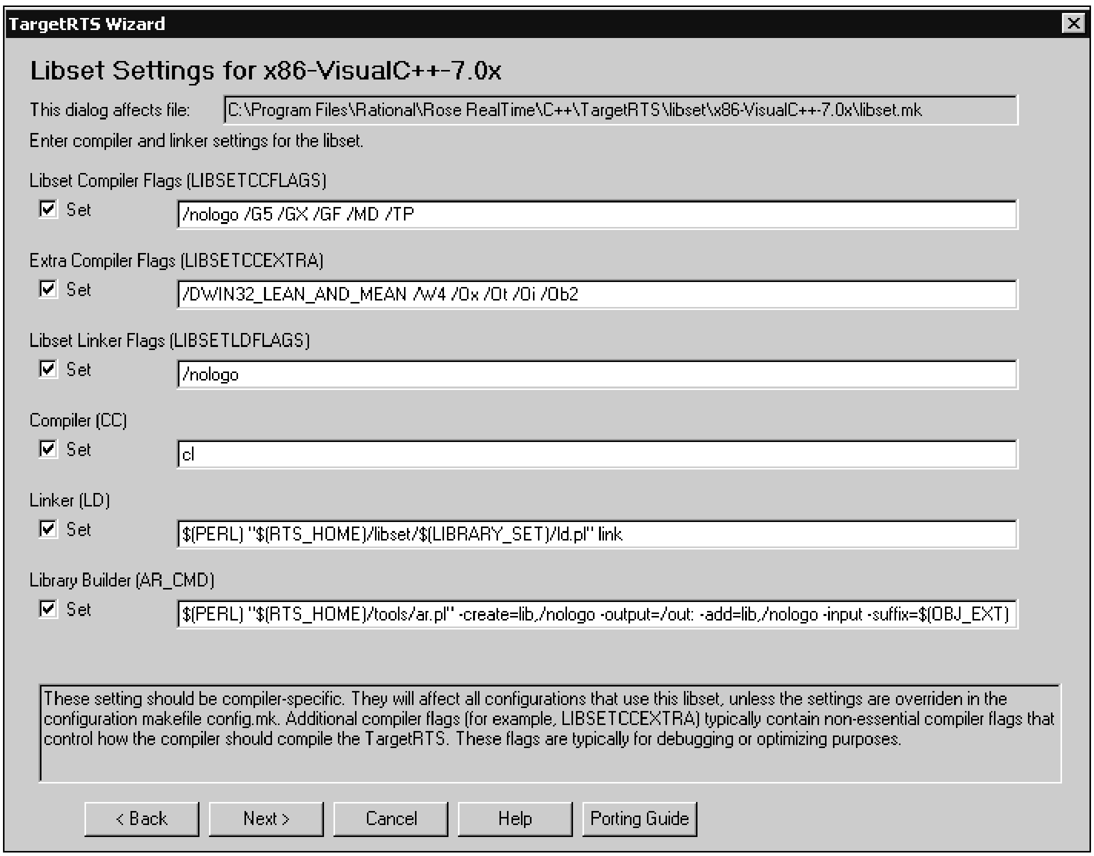
Note: These flags should be target-specific. They will affect all configurations that use this target unless you override them on the **Configuration Setting** panel of the **TargetRTS Wizard**.

Editing the Libset

You want to edit a **libset** to change the it to a different CPU architecture or a different compiler, or to change how the TargetRTS library is built (for example, changing compiler flags).

Figure 8 shows the options for configuring the **libset**. The **Set** options control which variables are defined in the libset.mk file for that particular **libset**. The text boxes to the right of the **Set** options contain their current values.

Figure 8 TargetRTS Wizard - Libset Settings Panel



Libset Compiler Flags (LIBSETCCFLAGS)

Adds compiler-specific compilation flags in the file libset.mk.

Extra Compiler Flags (LIBSETCCEXTRA)

Specifies any non-essential compiler flags that control how the compiler should compile the TargetRTS. These flags are used to compiles the TargetRTS library, but do not compile the models. Typically, you would specify optimization flags in this box.

Libset Linker Flags (LIBSETLDFLAGS

Adds compiler-specific linker flags in the libset.mk file.

Compiler (CC)

Specifies the name of the C or C++ compiler executable.

Linker (LD)

Specifies when a linker must be different from compiler (most compilers can invoke the linker), or if a preprocessing script is necessary.

Library Builder (AR_CMD)

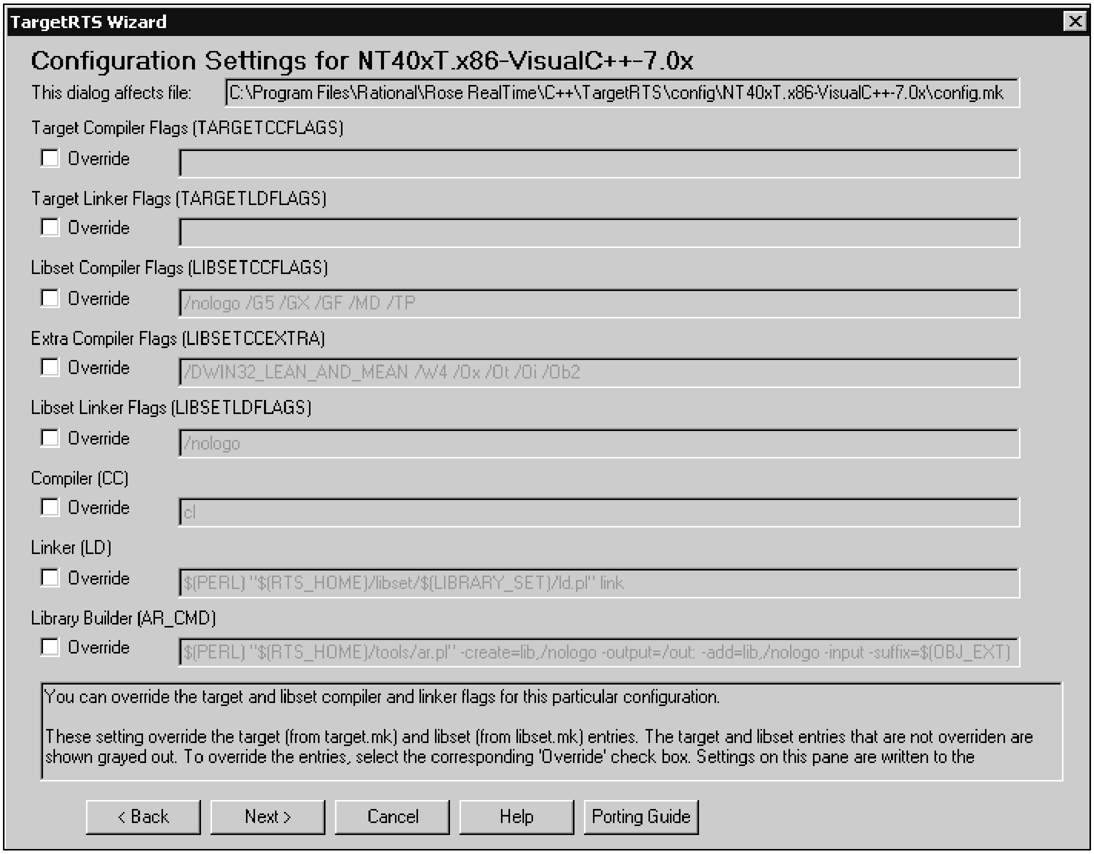
Specifies a command to run the library utility.

Modifying a Configuration

Editing a configuration overrides settings from the target.mk and libset.mk files. The overridden settings apply only to the selected configuration, and they are stored in that configuration's config.mk file.

Figure 9 shows the override options for the configuration. These are the same options that appear on the **Libset Settings** and the **Target Settings** panels in the **TargetRTS Wizard**.

Figure 9 TargetRTS Wizard - Configuration Settings Panel



Building Configurations

To build an existing configuration of the TargetRTS, you must specify the **make** command used by the build. Figure 10 shows the **Build Configuration** pane which you can use to compile the TargetRTS libraries.

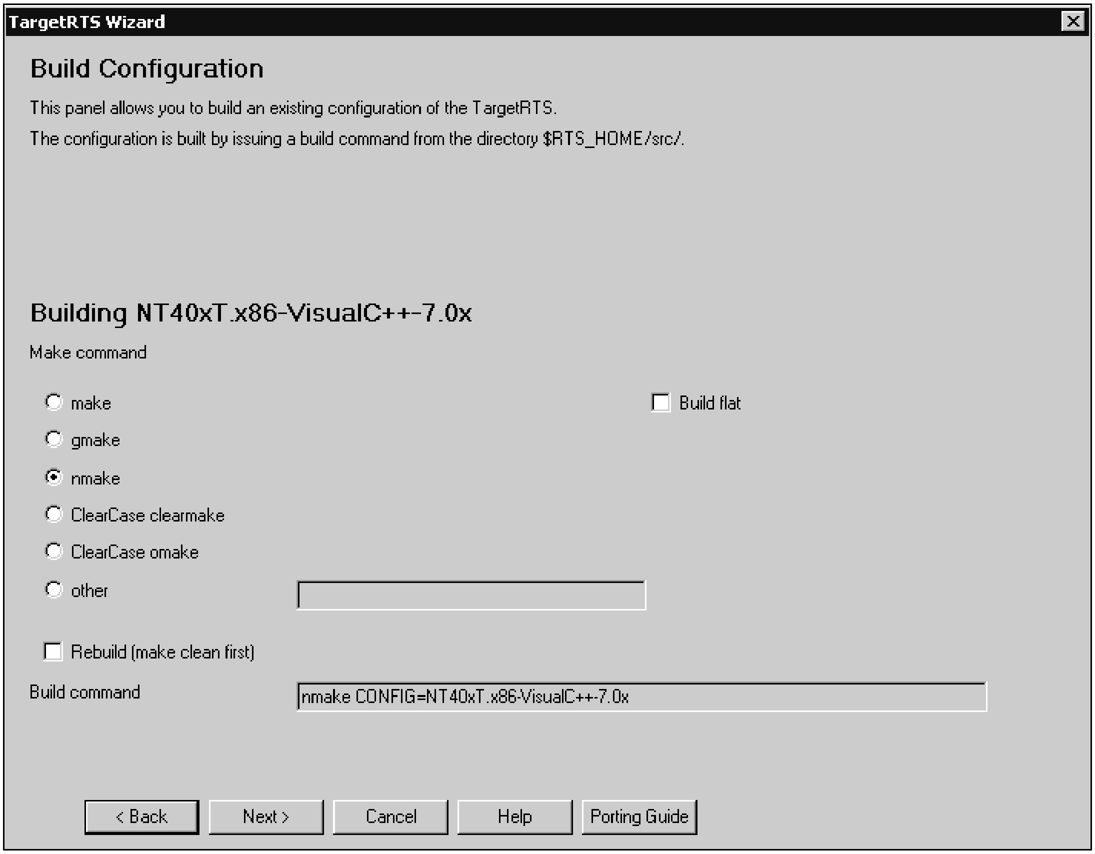
Building a selected configuration creates a directory with the following format:

$ROSERT_HOME/build-<*target*>-<*libset*>

This directory contains the dependency file and object files for the TargetRTS. When the build completes successfully, the resulting Rational Rose RealTime libraries save to a directory that uses the following format:

$ROSERT_HOME/lib/<*target*>.<*libset*>

Figure 10   TargetRTS Wizard - Build Configuration Panel



make

Specifies a UNIX implementation of a **make** utility (**make**).

gmake

Specifies the GNU implementation of **make**.

nmake

Specifies a Microsoft implementation of a **make** utility (**nmake**).

ClearCase clearmake

Specifies the UNIX implementation of a **make** utility for building software whose file are under ClearCase version control.

ClearCase omake

Specifies the Windows implementation of a **make** utility for building software whose files are under ClearCase version control.

other

Specify a alternate **make** utility to build the TargetRTS.

Rebuild (make clean first)

Ensures a clean build. When selected, all intermediate files are deleted first.

Build flat

Copies all source files into a single directory (one file per class) and builds the libraries from that location. This option is useful for debugging because some debuggers do not work properly with the TargetRTS source directory structure.

Note: Setting this option also decreases the build time considerably because fewer source files need to be opened and closed.

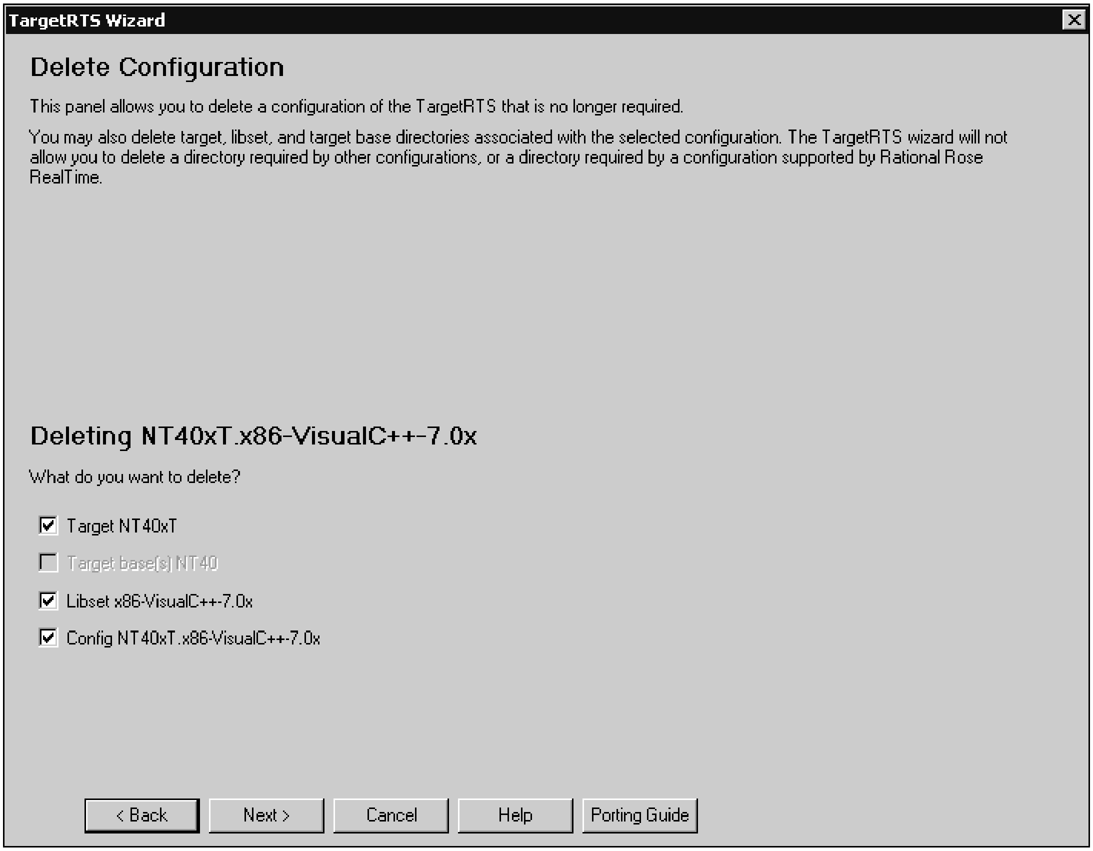
Deleting Configurations

For any duplicated configuration that you create, you can also delete those configurations.

Note: The configurations distributed with Rational Rose RealTime are read-only and cannot be deleted.

Figure 11 shows the **Delete Configuration** panel from which you can selectively delete the target, target base, **libset**, or the configuration-specific files for the selected configuration.

Figure 11 TargetRTS Wizard - Delete Configuration Panel



Creating Ports between C and C++

There is no automatic method of creating a C TargetRTS port form an existing C++ port to the same, or similar OS. You can use the existing port to identify how the OS-specific parts of the TargetRTS were implemented for the particular target. Because the C TargetRTS and C++ TargetRTS have a similar structure, this can save a lot of time.

To make a C TargetRTS port based on a C++ port for the same, or similar OS:

Note: The process of creating a C++ port from a C port is similar.

First, you want to create the directory structure for the new port.

1 Click **Tools > TargetRTS Wizard**.

2 Specify a language for the new port.

3 Verify that the path to the TargetRTS is correct, and click **Next**.

4 In the **Manage Configurations** panel, select a **NoRTOS** configuration from the **Existing Configurations** list.

5 Click **Duplicate**.

6 Click **Next**.

7 Create a port called:

<*new_target*>`S.`<*new_libset*>

where:

new_target is the name of the OS followed by its version.

Select **Target** and specify a name in the **Target name** box.

new_libset consists of the following format:

<*CPU_name*>-<*compiler_name*>-<*compiler_version*>

Select **Libset** and specify a name in the **Libset name** box.

Note: The "**S**" after the target name denotes a single-threaded configuration; the **TargetRTS Wizard** does not allow the creation of multi-threaded targets from single-threaded ones.

8 Under the **Target base** label, depending on your preferences, select either **Provide skeleton** or **Duplicate**.

9 In the **Name** box, specify a name for the target base.

Typically, the name is the name of the OS.

After the duplication process completes, you want to configure the new port for the intended toolchain.

10 In the **Manage Configurations** panel, select the new configuration.

11 Click **Edit**.

12 In the **Edit Configuration** pane, select the options to edit the **libset** and the **configuration**.

13 In the following panels, change the values as appropriate for the new toolchain.

14 You may have to edit the $ROSERT_HOME/libset/<new_libset>/libset.mk file to finish configuring the toolchain.

Note: You may have to create a file called $ROSERT_HOME/libset/RTLibSet.h to define compiler-specific macros.

Next, you want to configure the OS-specific parts of the port.

15 Because the **TargetRTS Wizard** does not permit the creation of a multi-threaded target from a single-threaded one, if the final port is for a multi-threaded environment, change the name of the following directory from:

$ROSERT_HOME/target/<*new_target*>S

to

$ROSERT_HOME/<*new_target*>T

and change the name of the following directory from:

$ROSERT_HOME/config/<*new_target*>S.<*new_libset*>

to

$ROSERT_HOME/config/<*new_target*>T.<*new_libset*>

16 To properly configure the $ROSERT_HOME//target directory, use the contents of the file $ROSERT_HOME/target/<*old_target*>T in the original port's TargetRTS to determine what the $ROSERT_HOME//target/<*new_target*>T file in the new port's TargetRTS should be.

Note: Some configuration macros are not the same in C and C++. However, all of the options are described in the file $ROSERT_HOME/include/RTPublic/Config.h. Also, you will want to review the contents of the file $ROSERT_HOME/target/<*new_target*>T/target.mk. If the new target is multi-threaded, the file $ROSERT_HOME/target/<new_target>T/RTTarget.h will require the **USE_THREADS** macro set to 1, and must also define default priorities for the main, debugger, and timer threads. Typically, you can obtain these values from the original port.

17 Some ports also require configuration-specific settings. These are defined in the file $ROSERT_HOME/config/<*new_target*>T.<*new_libset*>/config.mk. The file $ROSERT_HOME/config/<*new_target*>T.<new_libset>/setup.pl controls the environment configuration required for building the TargetRTS libraries (and possibly, the building of models) for the new platform. The setup.pl file from the old port may provide you with some assistance, but you will have to use your OS and compiler documentation to properly configure the environment.

18 You must write the OS-specific code for the new port. All such code resides in the following directory:

$ROSERT_HOME/src/target/<*new_target_base*>/

where:

new_target_base is the name assigned to the target base during the duplication process in the **TargetRTS Wizard**. This name is stored in the setup.pl script as a value of the **$target_base** variable. The skeleton implementation contains only stubs for functions necessary for all ports. This particular port will likely require you to define additional OS-specific functions. Use the target base from the original port to see how to implement these OS-specific functions. Almost every C TargetRTS "class" has a corresponding class in the C++ TargetRTS.

Note: Header files in the C target base must be in the RTPubl or RTPriv directories. Also, if some files appear only in this target base, they must be declared in the RTPriv/TGTMFEST.c file in the same manner as other files are declared in the file src/MANIFEST.c.

Note: It may be necessary to further configure that target, **libset**, or **config** settings.

After you finish configuring the target, **libset**, and **config**, you are ready to build the TargetRTS.

19 In the **Manage Configurations** panel, select the new configuration from the **Existing Configurations** list.

20 Click **Build**.

21 Specify a **make** utility and click **Next**.

22 Fix any errors encountered during the build process until the TargetRTS successfully builds, and the models link and run.

You may want to create Perl scripts for error parsing in the directory:

$RTS_HOME/codegen/compiler/<*vendor_name*>

<*vendor_name*> is defined in the $RTS_HOMElibset/<*new_libset*>/libset.mk file as a value of VENDOR.

Introducing the TargetRTS

2

Contents

This chapter is organized as follows:

- *Overview* on page 41
- *Other Resources* on page 42

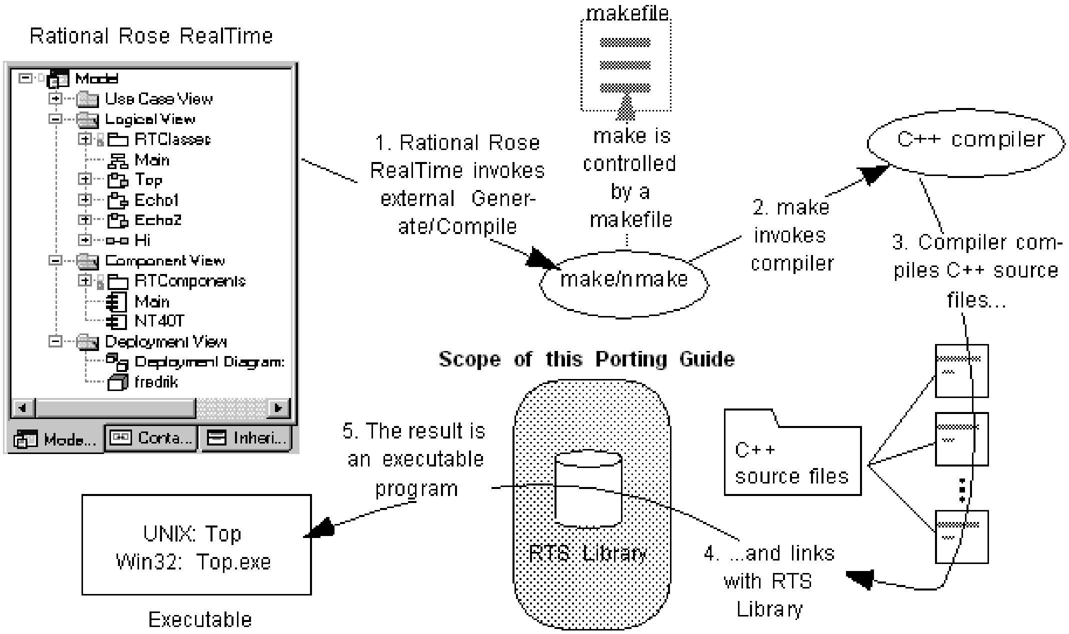
Overview

The TargetRTS is the set of run-time services that provide a framework in which a Rational Rose RealTime model can run. It provides the run-time implementation of the UML-RT constructs used in the model. Figure 12 shows the context of the TargetRTS in building an executable program.

This guide describes the steps required to port the TargetRTS to a new target environment. The new target may simply be a new version of an operating system or compiler on a UNIX host. In more complicated cases it may be a new operating system, compiler and target hardware. The latter scenario is of more interest to this guide, although all the information required for the former scenario is provided.

This guide is specifically designed for software development professionals familiar with the target environment they intend to port to. It is assumed that the reader has significant knowledge and experience with the development environment, operating system, and target hardware.

Figure 12 The TargetRTS in Context



Other Resources

Before starting a port, ensure that you have the following documents and material available:

- Operating system documentation (for system calls, available services)
- Compiler documentation
- Sample programs that come with compiler or operating system (use these to test your toolchain)
- Rational Rose RealTime *C Reference* or *C++ Reference*
- Rational Rose RealTime example models (to test the port)

Before Starting a Port

3

Contents

This chapter is organized as follows:

- *OS Knowledge and Experience* on page 43
- *Toolchain Functionality* on page 44
- *OS Capabilities* on page 44
- *Simple non-Rational Rose RealTime Program on Target* on page 45
- *TCP/IP Functionality* on page 46
- *Floating Point Operations* on page 46
- *Standard Input/Output Functionality* on page 46
- *Debugging* on page 47
- *Training* on page 47
- *Support* on page 47
- *What to do Before Calling Rational Customer Support* on page 47

OS Knowledge and Experience

Knowledge and experience with the target operating system is key to a successful port. This knowledge should extend to the development environment and target hardware. The type of knowledge required includes such details as synchronization mechanisms, thread creation, memory management, timing, device drivers, board support packages, memory maps, TCP/IP support, priority and scheduling schemes, and so forth. See *OS Capabilities* on page 44 for a list of OS capabilities required by the TargetRTS.

Experience with porting the TargetRTS to other platforms will aid greatly as the ports tend to follow a pattern. For each development environment and operating system there are bound to be a few surprises. See *Common Problems and Pitfalls* on page 105.

Toolchain Functionality

A functioning development environment must be in place before porting can begin. This includes the correct installation of tools such as linkers, compilers, assemblers and debuggers. To build the TargetRTS, you must have a working version of Perl for your development host (version 5.002 or greater). Perl is used extensively in the **makefiles** for the TargetRTS.

It is also important to initialize environment variables for inclusion of header files and location of library files. An easy way to test this is to create a simple program, such as "Hello World", and compile and run it on the target. This step is described in *Simple non-Rational Rose RealTime Program on Target* on page 45.

OS Capabilities

The target operating system must have a set of services that satisfy the requirements of the TargetRTS. In general, most commercial real-time operating systems (RTOS) have these services. Before starting a port, check for these basic capabilities in the target RTOS. Table 1 lists the TargetRTS feature and its corresponding RTOS service

Table 1 Required Operating System Features for the C and C++ TargetRTS.

C TargetRTS Feature	C++ TargetRTS Feature	Operating System Service
`RTTimespec_getclock()` (method required)	**`RTTimespec::getclock()`** (method required)	A function is required to return the current time. The more precision the better. In general, an RTOS will return time with precision of its internal timer.
`RTThread_construct()` (constructor required for threaded targets)	**`RTThread::RTThread()`** (constructor required for threaded targets)	Task creation function - must be able to create task or thread with specified stack size and priority. Be aware of priority scheme - some RTOSes use 0 as highest priority while others may use 0 for lowest priority.
`RTMutex` (all 4 methods required for threaded targets)	**`RTMutex`** (all 4 methods required for threaded targets)	A mutual exclusion mechanism. Some RTOSes provide optimized mutex service along with semaphores.
`RTSyncObject` (all 5 methods required for threaded targets)	**`RTSyncObject`** (all 5 methods required for threaded targets)	Semaphore, mailbox, signal - service must provide infinite and timed blocking.

Table 1 Required Operating System Features for the C and C++ TargetRTS.

C TargetRTS Feature	C++ TargetRTS Feature	Operating System Service
`RTStdio_putString()` (output to console)	**`RTDiagStream::write()`** (output to console)	Standard output - this may not be provided out-of-the-box. For embedded targets, device drivers added to the board support package may be required. Output is generally routed to external serial ports but TCP/IP or UDP/IP may be used instead.
`RTDebuggerInput_nextChar()` (input from console)	**`RTDebuggerInput::nextChar()`** (input from console)	Standard input, as above. This can be removed from the TargetRTS via configuration options.
Target Observability	Target Observability	TCP/IP support is required. This includes device drivers in the board support package for the ethernet hardware on the target. If not provided this is a substantial do-it-yourself project. Target Observability can be removed from the TargetRTS via configuration options.
`malloc`, **`free`**	**`new`**, **`delete`**	The RTOS must support some sort of memory management. In general, this is hidden from the user by the compiler as the RTOS resolves the new and delete symbols.
`main()` function	**`main()`** function	Some RTOSes have their own main function defined. If so, then the main function in the TargetRTS must be redefined.

Simple non-Rational Rose RealTime Program on Target

An easy way to test the toolchain functionality is to create a simple program that prints out “Hello World” on the console.

This program should not use any TargetRTS code or libraries. Compile and link the program outside of Rational Rose RealTime using your toolchain, and download the executable to the target. If it executes successfully, then your development environment is ready.

Further testing is strongly recommended. This would include some basic RTOS services such as thread creation in your test program. Again, no TargetRTS code or libraries should be used. Many RTOSes provide example programs to compile and run. Try these out and verify the functionality of your setup. If you are using a source-level debugger, verify that you can step through the source code and examine variables. If the debugger is aware of operating system data structures, check if you can examine these. The purpose of this testing to ensure that all of the required operating system features are operational and understood before attempting the port of the TargetRTS.

C++ Another important test for C++ compilers is to include a static constructor in the test program. This will ensure that proper initialization is performed.

TCP/IP Functionality

To support Target Observability for the new port, the target operating system must provide a compatible TCP/IP stack. In general, the TCP/IP layer must support the BSD sockets interface, that is, the creation and deletion of sockets, functions such as **`socket()`**, **`connect()`**, **`bind()`**, **`listen()`**, **`select()`**, and so forth. Typically, RTOSes try to provide a BSD-compliant TCP/IP stack. TCP/IP functionality can be a common source of problems with new ports. See *Common Problems and Pitfalls* on page 105.

If a TCP/IP stack is not provided, then you must implement one, which might require significant effort. Alternatively, the use of SLIP or PPP over a serial connection may be an option, but would require customizations. It would also affect the performance of Target Observability. Alternatively, you can choose not to use target observability.

Floating Point Operations

Some of the TargetRTS classes require the use of floating point operations. Investigate the support for floating point on your target system.

C++ It is possible to configure the support for RTReal from the TargetRTS via configuration options.

Standard Input/Output Functionality

The TargetRTS needs standard input and output to a console for log messages, panic messages, and debugger input/output. This may already be provided by the target development or operating system. Some embedded RTOS and development tools may not provide standard input and output, and instead require the addition of serial port device drivers to the board support package. The use of TCP/IP or UDP/IP to provided standard input/output is also an option.

Debugging

The use of a source-level debugger that provides some sort of operating system awareness is the best development tool for the port. This is the easiest way to examine source code, memory, variables, registers, stacks, and so forth.

Training

Training is an important component of a successful port. Rational offers training courses to help users understand, use, and port the TargetRTS. Your RTOS vendor may also offer training and this is recommended as well.

Support

Rational provides support for the standard ports as identified in the *Installation Guide*. All reported issues will be duplicated on one or more of the standard referenced configurations.

What to do Before Calling Rational Customer Support

The following steps should be followed before calling Rational Technial Support for help with a custom port of the TargetRTS.

1. Get to know your compiler/linker/debugger toolchain. Be sure it is installed correctly, and that programs can be compiled, linked, downloaded to the target hardware and run successfully.
2. Get to know your target operating system. Be sure that an example multi-threaded program that exercises the various features of the RTOS is compiled, linked and downloaded to the target hardware and runs successfully. Do not use Rational Rose RealTime for this example program. This should be produced independently to verify toolchain and RTOS functionality.
3. Read this guide and the *C Reference* or *C++ Reference* that is included with Rational Rose RealTime, to understand the required capabilities of the RTOS needed to support the TargetRTS.

4 Ensure that the TCP/IP stack for your target platform is operational. In particular the sockets interface must be working, and additional utilities such as **`gethostbyname()`** must be functional.

5 Test the functionality of the standard input and output for your target. This will probably be verified in earlier steps.

6 Learn how to use the target debugger. This will be a useful tool when doing the port.

7 Get as much training on Rational Rose RealTime, the RTOS, and your toolchain as possible.

Porting the TargetRTS

4

Contents

This chapter is organized as follows:

- *Overview* on page 49
- *Phases of a Port* on page 50
- *Choose a Configuration Name* on page 50
- *Building Rational Rose RealTime Applications for Targets without Operating Systems* on page 52
- *Creating a Setup Script (setup.pl)* on page 54
- *TargetRTS makefiles* on page 56
- *Default makefile* on page 59
- *Target makefile* on page 64
- *Libset makefile* on page 65
- *Config makefile* on page 65

Overview

The most common customization to the TargetRTS is porting it to a new platform. A platform is defined by the TargetRTS as the combination of the operating system, target hardware and the compiler/linker toolchain. A new operating system requires the most work since it often requires implementation changes. However, a new compiler may also require changes, in particular, to the configuration files.

The ports supported by Rational Software and shipped with the TargetRTS source are a good place to begin considering design alternatives for a new port. The root directory for the TargetRTS source will be referred to from this point forward using the environment variable **`$RTS_HOME`.**

C For C, it is usually defined as **$ROSERT_HOME/C/TargetRTS**. For Windows, assume **%ROSERT_HOME%\C\TargetRTS**.

C++ For C++, it is usually defined as **$ROSERT_HOME/C++/TargetRTS**. For Windows, assume **%ROSERT_HOME%\C++\TargetRTS**.

In the sections that follow, examples are extracted from this source.

Phases of a Port

The major steps for implementing the port are as follows:

- Performing pre-port steps (see *Before Starting a Port* on page 43).
- Naming the platform (see *Choose a Configuration Name* on page 50).
- Defining the setup script (see *Creating a Setup Script (setup.pl)* on page 54).
- Defining the platform-specific **makefiles** (see *TargetRTS makefiles* on page 56).
- Defining the platform-specific header files (see *Porting the TargetRTS for C++* on page 83).
- Defining the platform-specific implementation of TargetRTS features (see *Platform-specific Implementation* on page 88).
- Building the new TargetRTS and fixing compile and link problems (see *Building the New TargetRTS* on page 122).
- Testing the new TargetRTS using test model updates (see *Testing the TargetRTS Port* on page 101).
- Tuning the performance of the TargetRTS, if required (see *Tuning the TargetRTS* on page 103).

Choose a Configuration Name

The first step in implementing a port is picking the name for the configuration. This name and parts of it are used by the various **loadbuild** tools to find the files needed to build the TargetRTS for that configuration. It is also used during compilation of the Rational Rose RealTime models. There are two parts to the name: <**target**> and <**libset**>. The resulting names for TargetRTS configurations are defined as the concatenation of the target and **libset** names in the following pattern:

```
<config> ::= <target>.<libset>
```

Examples are given in Table 2.

Table 2 Example Configuration Names

Config Name	Description
SUN4S.sparc-gnu-2.8.1	SunOS 4.x Single-threaded on a Sparc processor using Free Software Foundation gnu version 2.8.1
SUN5T.sparc-gnu-2.8.1	Solaris 2.x Multi-threaded on a Sparc processor using Free Software Foundation gnu version 2.8.1
SUN5S.sparc-SunC++-4.2	Solaris 2.x Single-threaded on a Sparc processor using Sun Microsystems SPARCUtils C++ version 4.2
NT40T.x86-VisualC++-6.0	Windows NT 4.0 Multi-threaded on an x86 processor using Microsoft Visual C++ version 6.0
TORNADO2T.ppc-cygnus-2.7.2-960126	Tornado 2 Multi-threaded on a Motorola PowerPC processor using Cygnus C++ version 2.7.2-960126

Target Name

The target name presents the implementation-specific components of the TargetRTS. These components are generally specific to a given configuration, of a given version, of a given operating system. The target name is also used to name the configuration of the target, for example, single versus multi-threaded. The target name is defined as follows:

```
<target> ::= <OS name><OS version><RTS config>
```

For example: **SUN5T**. The components of **<target>** are defined as follows:

<OS name> identifies the operating system (for example, **SUN**)

<OS version> identifies the major version of that operating system (for example, **5** meaning SunOS 5.x, that is, Solaris 2.x). Do **not** use periods in the OS version, as this will confuse the make utility when trying to build the TargetRTS.

<RTS config> is a single letter to further identify the configuration. Currently only ‘S’ for single-threaded and T’ for multi-threaded configurations are supported.

Libset Name

Although the actual **libset** names can be chosen arbitrarily, by convention those used by Rational Rose RealTime are defined as follows:

```
<libset> ::= <processor>-<compiler name>-<compiler version>
```

For example: **`sparc-gnu-2.8.1`**. The components of **<libset>** are defined as follows:

<processor> identifies the processor architecture name

<compiler name> identifies the compiler product name or the vendor for the compiler

<compiler version> identifies the compiler version. It is acceptable to use periods in the compiler version text.

Building Rational Rose RealTime Applications for Targets without Operating Systems

You can configure the Rational Rose RealTime run-time libraries to build Rational Rose RealTime applications that run without an operating system. The resulting application that is generated will be a “main” program; you can build and run a main program on the target.

If there is no RTOS available on the target, or if the application will exist in a single thread, you can use a NoRTOS configuration.

Benefits of Using a NoRTOS Configuration

The benefits to using a NoRTOS configuration are:

- A NoRTOS configuration does not require any RTOS services.
- A NoRTOS configuration is useful in small footprint and simple device configurations, or in configurations where threading is not required.
- You can get started quickly by minimizing the effort required to make the initial port operational.

Using a NoRTOS Configuration

If you are creating a new target configuration, you can begin by creating a NoRTOS configuration, and later change it to a threaded configuration.

A NoRTOS does not have any RTOS dependencies; however, this does not prevent you from using RTOS services in your application.

To configure a NoRTOS configuration using the TargetRTS Wizard:

1 From the **Tools** menu, click **TargetRTS Wizard**.

2 Select a language and click **Next**.

3 In the **Manage Configurations** pane, select a NoRTOS configuration, such as **NoRTOSS.x86-VisualC++-6.0 NoRTOSS.sparc-gnu-2.8.1**.

4 Click **Duplicate** to modify the NoRTOS configuration for you requirements.

5 In the **Duplicate Configuration** pane, select **Libset**.

6 In the **Libset name** box, specify a new Libset, or if you want to reuse an existing libset, type the name of that libset. For additional information on creating a Libset name, see *Libset Name* on page 52.

7 Click **Next**.

8 In the **Summary** pane, review the information, and then click **Next**.

9 In the **Work Order** pane, review the information, and then click **Next**.

The resulting run-time libraries for this port have no dependencies on any operating services. They do expect console I/O if there is no **stdin**/**stdout** for your target that can easily be compiled. Linking your Rational Rose RealTime model with the NoRTOS library creates a program with a "main" entry function.

Although the resulting services library has no operating system dependencies, it does depend on the compiler used to build the program for a specific CPU. To complete a port, you will need to add the supporting compiler interfaces.

Verification

You should verify that you can:

- build and link against a services library
- compile and link for your target inside the toolset
- create an executable for your target.

Other things you may want to test are:

- error parsing (for example, you can add a syntax error, double-click on the resulting error in the **Build Errors** tab, then observe the error in the model to see if it is the correct error)
- timing services (for example, add a timing port and test the timing services).
- if you have interfaces to load, unload, reset your target from your host, you may want to create Perl script wrappers to make those capabilities accessible within Rational Rose RealTime. See $ROSERT_HOME/bin/tc/win32 for examples of these scripts.

Creating a Setup Script (setup.pl)

The setup script is a file, setup.pl, containing Perl commands that configure the environment for the compilation of the TargetRTS for the specified platform. This file is located in the directory $RTS_HOME/config/<config>.

Note: If the target toolchain environment variables are included in a user's standard environment, the variables in the setup.pl file may not be required. These environment variables defined in the setup.pl file are not available when using the toolset to build user models.

Commands in the setup.pl file are executed before any of the TargetRTS compilation tools are invoked. Typically, definitions for locations of files on the host platform are included in this file (such as setting the **shell** environment variable **PATH** to point to the appropriate tools).

Table 3 describes the variables in the setup.pl file that are specific to Rational Rose RealTime:

Table 3 Variables in the Setup.pl Script

Variables	Description
$preprocessor	Defines the C++ preprocessor command appropriate for the compilation environment, and automatically generates source code dependencies for the TargetRTS.
$supported	Defines whether Rational Rose RealTime supports this target. Valid values for **$supported** are **Yes**, **No**, and **Custom**. For a custom port, we recommend **Custom**. This variable has no impact on how the port is compiled or used.
$target_base	Indicates that the implementation of the target-specific features of the TargetRTS are rooted in the same source directory as the **$target_base** target. For example, for the **TORNADO2** targets, the **$target_base** is set to **TORNADO1**. As a result, **TORNADO2** implementations of TargetRTS classes are in the same source directory as those of the **TORNADO1** target, that is, $RTS_HOME/src/target/TORNADO1. This variable can contain multiple entries separated by a comma. When using multiple entries, the target source directories are searched in the specified order.
$postprocessor	An optional variable that runs after **$preprocessor**.

Note: The **$preprocessor** and **$supported** variables must be defined for all targets.

The example file located in the directory:

$RTS_HOME/config/TORNADO2T.ppc-cygnus-2.7.2-960126/setup.pl

contains the following:

```
if( $OS_HOME = $ENV{'OS_HOME'} )
{
     $os = $ENV{'OS'} || 'default';

     if( $os eq 'Windows_NT' )
     {
          $wind_base        = $ENV{'WIND_BASE'};
          $wind_host_type   = 'x86-win32';
          $ENV{'PATH'} =
"$wind_base/host/$wind_host_type/bin;$ENV{'PATH'}";
     }
     else
     {
          $rosert_home      = $ENV{'ROSERT_HOME'};
```

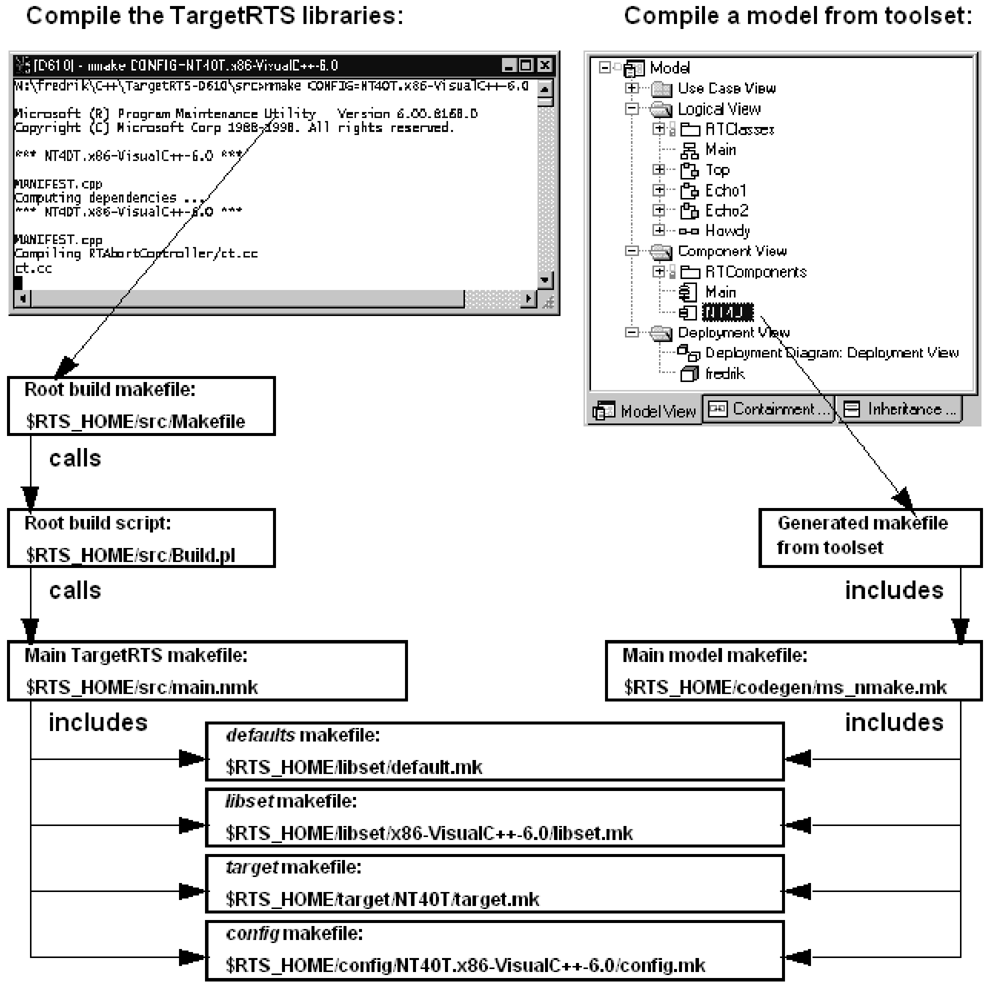
```
          chomp( $host      = `$rosert_home/bin/machineType` );

          $wind_base        = "$OS_HOME/wrs/tornado-2.0";
          if( $host eq 'sun5' )
          {
             $wind_host_type   = 'sun4-solaris2';
          }
          $ENV{'PATH'} =
"$wind_base/host/$wind_host_type/bin:$ENV{'PATH'}";
          $ENV{'WIND_BASE'} = "$wind_base";
     }

     $ENV{'GCC_EXEC_PREFIX'}
="$wind_base/host/$wind_host_type/lib/gcc-lib/";
     $ENV{'VXWORKS_HOME'}    = "$wind_base/target";
     $ENV{'VX_BSP_BASE'}     = "$wind_base/target";
     $ENV{'VX_HSP_BASE'}     = "$wind_base/target";
     $ENV{'VX_VW_BASE'}      = "$wind_base/target";
     $ENV{'WIND_HOST_TYPE'}  = "$wind_host_type";
}

$preprocessor = "ccppc -DPRAGMA -E -P >MANIFEST.i";
$target_base  = 'TORNADO1';
$supported    = 'Yes';
```

Note: The setup file is **not** used when compiling the generated source, neither from within the toolset, nor from the command-line. The environment variables defined in the setup file must instead be defined in the user's environment before starting the Rational Rose RealTime toolset. In the given example, the setup file assumes that the user's environment has the variable **`OS_HOME`** already defined as a partial path to where the RTOS is installed.

TargetRTS makefiles

Two types of builds are supported by the **makefiles** for the TargetRTS: compilation of the TargetRTS libraries and compilation of the generated code. The platform-specific definitions are required by both and are thus placed in separate files. The sequencing of the **makefiles** for the two paths are shown in Figure 13.

Figure 13 Sequencing of Makefiles



As shown, there is a **makefile** for each of the following:

- **`$RTS_HOME/src/Makefile`** is the root **makefile** for TargetRTS compilation. It invokes a Perl script called **`Build.pl`**. This script checks the dependencies for the TargetRTS source code and generates a **makefile** called **`depend.mk`** in the **`$RTS_HOME/build-<config>`** directory. It then builds the TargetRTS from this directory. This **makefile** and **`Build.pl`** should **not** be customized, and will not be discussed further in this document.

- **`$RTS_HOME/src/main.nmk`** (**`main.mk`** for UNIX) is the main definition for compiling the TargetRTS libraries. These **makefiles** should **not** be customized, and will not be discussed further in this document.

- The generated **makefile** for the model being compiled. See the *C Reference* or *C++ Reference* for more details on how this **makefile** is generated.
- **`$RTS_HOME/codegen/ms_nmake.mk`** (**`gnu_make.mk`** for Gnu, **`unix_make.mk`** for other Unix) is the main definition for compiling a model. These **makefiles** should **not** be customized, and will not be discussed further in this document.
- **`$RTS_HOME/libset/default.mk`**, the default macro definitions that may be overridden by the platform specific **makefiles**. See *Default makefile* on page 59.
- **`$RTS_HOME/target/<target>/target.mk`** is the definition specific to the target operating system. See *Target makefile* on page 64.
- **`$RTS_HOME/libset/<libset>/libset.mk`** is the definition specific to the compiler. See *Libset makefile* on page 65.
- **`$RTS_HOME/config/<config>/config.mk`** is the definition specific to the combination of the compiler, operating system and TargetRTS configuration. See *Config makefile* on page 65.

The default.mk, libset.mk, target.mk, and config.mk **makefiles** are used to compile both the TargetRTS libraries and the model.

Compilation of the model is usually performed by right-clicking on a component in the toolset and choosing **Build > Build... > Generate and compile** , or set the component as default and hit [F7]. It is, however, also possible to just generate the source and make files needed from within the toolset, and compile from the output directory by issuing the **make** command (**nmake** for Windows).

Compilation of the TargetRTS is performed from the $RTS_HOME/src directory by issuing the command

```
make CONFIG=<target>.<libset>
```

For example in UNIX:

```
make CONFIG=SUN5T.sparc-gnu-2.8.1
```

For example in Windows:

```
nmake CONFIG=NT40T.x86-VisualC++-6.0
```

Note: Some make utilities also allows the following:

```
make CONFIG=<target>.<libset>
```

For example:

```
make SUN5T.sparc-gnu-2.8.1
```

Default makefile

The **target.mk**, **libset.mk** and **config.mk** makefiles are expected to override defaults defined in **$RTS_HOME/libset/default.mk**. The defaults are as follows for each language.

For the C language:

C

```
# ======== General Defaults
============================================

CONFIG = $(TARGET).$(LIBRARY_SET)

# Defaults for macros which may be modified by
#    libset/$(LIBRARY_SET)/libset.mk
#    target/$(TARGET)/target.mk
# or config/$(CONFIG)/config.mk

PERL          = rtperl
FEEDBACK      = $(PERL) "$(RTS_HOME)/tools/feedback.pl"
MERGE         = $(PERL) "$(RTS_HOME)/tools/merge.pl"
NOP           = $(PERL) "$(RTS_HOME)/tools/nop.pl"
RM            = $(PERL) "$(RTS_HOME)/tools/rm.pl"
RMF           = $(RM) -f
TOUCH         = $(PERL) "$(RTS_HOME)/tools/touch.pl"

# codegen makefiles stuff

RTCOMP        = $(PERL) "$(RTS_HOME)/codegen/rtcomp.pl"
RTLINK        = $(PERL) "$(RTS_HOME)/codegen/rtlink.pl"
VENDOR        = generic

# Macros used when make must recurse

MAKEFILE      = Makefile

# Macros used when creating an object file from a C source file

CC            = $(FEEDBACK) -fail \
                    CC should be defined by libset.mk or generated
makefile
DEBUG_TAG     = -g
DEPEND_TAG    = -I
DEFINE_TAG    = -D
INCLUDE_TAG   = -I
LIBSETCCEXTRA =
LIBSETCCFLAGS =
OBJECT_OPT    = -c
OBJOUT_OPT    = -o
```

```
OBJOUT_TAG    =
SHLIBCCFLAGS  = -PIC
TARGETCCFLAGS =

# Macros used when creating an object library from a set of object
files

AR_CMD        = $(PERL) "$(RTS_HOME)/tools/ar.pl"
AR            = $(AR_CMD)
LIBOUT_OPT    =
LIBOUT_TAG    =
RANLIB        = $(NOP)

# Macros used when creating a shared library from a set of object
files

SHLIB_CMD     = $(FEEDBACK) -fail Shared libraries not supported.
SHLIBOUT_OPT  = -o
SHLIBOUT_TAG  =

# Macros used when creating an executable from a set of object files,
libraries

LD            = $(CC)
DIR_TAG       = -L
LIBSETLDFLAGS =
LIB_TAG       = -l
OT_LIB_TAG    = -l
TARGETLDFLAGS =
TARGETLIBS    =
EXEOUT_OPT    = -o
EXEOUT_TAG    =

# Macros used to construct names of various kinds of files

EXEC_EXT      =
LIB_PFX       = lib
LIB_EXT       = .a
C_EXT         = .c
OBJ_EXT       = .o
SHLIB_PFX     = lib
SHLIB_EXT     = .so

# ========= Shared Macros
===============================================

# RTCODEBASE can be overridden in the target/$(TARGET)/target.mk file
RTCODEBASE    = $(PLATFORM)
```

```
RTSYSTEM_INCPATHS = \
        $(INCLUDE_TAG)"$(RTS_HOME)/libset/$(LIBRARY_SET)" \
        $(INCLUDE_TAG)"$(RTS_HOME)/target/$(TARGET)" \
        $(INCLUDE_TAG)"$(RTS_HOME)/include"

RTS_LIBRARY   = $(RTS_HOME)/lib/$(CONFIG)

SYSTEM_LIBS   = $(DIR_TAG)"$(RTS_LIBRARY)" \
                $(OT_LIB_TAG)ObjecTimeC \
                $(OT_LIB_TAG)ObjecTimeCTransport \
                $(OT_LIB_TAG)ObjecTimeC \
                $(OT_LIB_TAG)ObjecTimeCTransport

# ========= Linking
====================================================

LD_OUT = $@

LD_HEAD = \
        $(EXEOUT_OPT) $(EXEOUT_TAG)$(LD_OUT) \
        $(LIBSETLDFLAGS) \
        "$(RTS_LIBRARY)/main$(OBJ_EXT)"

ALL_OBJS_LIST = $(ALL_OBJS)

LD_TAIL = \
        $(SYSTEM_LIBS) \
        $(TARGETLDFLAGS) \
        $(TARGETLIBS)

# ======== Compiling
===================================================

CC_HEAD = \
        $(OBJECT_OPT) $(OBJOUT_OPT) $(OBJOUT_TAG)$@ \
        $(LIBSETCCFLAGS) \
        $(TARGETCCFLAGS) \
        $(RTSYSTEM_INCPATHS)

CC_TAIL =

#
======================================================================
```

For the C++ language:

C++

```
# ======== General Defaults
=============================================

CONFIG = $(TARGET).$(LIBRARY_SET)

# Defaults for macros which may be modified by
#    libset/$(LIBRARY_SET)/libset.mk
#    target/$(TARGET)/target.mk
# or config/$(CONFIG)/config.mk

PERL          = rtperl
FEEDBACK      = $(PERL) "$(RTS_HOME)/tools/feedback.pl"
MERGE         = $(PERL) "$(RTS_HOME)/tools/merge.pl"
NOP           = $(PERL) "$(RTS_HOME)/tools/nop.pl"
RM            = $(PERL) "$(RTS_HOME)/tools/rm.pl"
RMF           = $(RM) -f
TOUCH         = $(PERL) "$(RTS_HOME)/tools/touch.pl"

# codegen makefiles stuff

RTGEN         = rtcppgen
RTCOMP        = $(PERL) "$(RTS_HOME)/codegen/rtcomp.pl"
RTLINK        = $(PERL) "$(RTS_HOME)/codegen/rtlink.pl"
VENDOR        = generic

# Macros used when make must recurse

MAKEFILE      = Makefile

# Macros used when creating an object file from a C++ source file

CC            = $(FEEDBACK) -fail \
                    CC should be defined by libset.mk or generated
makefile
DEBUG_TAG     = -g
DEPEND_TAG    = -I
DEFINE_TAG    = -D
INCLUDE_TAG   = -I
LIBSETCCEXTRA =
LIBSETCCFLAGS =
OBJECT_OPT    = -c
OBJOUT_OPT    = -o
OBJOUT_TAG    =
SHLIBCCFLAGS  = -PIC
TARGETCCFLAGS =
```

```
# Macros used when creating an object library from a set of object
files

AR_CMD         = $(PERL) "$(RTS_HOME)/tools/ar.pl"
AR             = $(AR_CMD)
LIBOUT_OPT     =
LIBOUT_TAG     =
RANLIB         = $(NOP)

# Macros used when creating a shared library from a set of object
files

SHLIB_CMD      = $(FEEDBACK) -fail Shared libraries not supported.
SHLIBOUT_OPT   = -o
SHLIBOUT_TAG   =

# Macros used when creating an executable from a set of object files,
libraries

LD             = $(CC)
DIR_TAG        = -L
LIBSETLDFLAGS =
LIB_TAG        = -l
OT_LIB_TAG     = -l
TARGETLDFLAGS =
TARGETLIBS     =
EXEOUT_OPT     = -o
EXEOUT_TAG     =

# Macros used to construct names of various kinds of files

EXEC_EXT       =
LIB_PFX        = lib
LIB_EXT        = .a
CPP_EXT        = .cc
OBJ_EXT        = .o
SHLIB_PFX      = lib
SHLIB_EXT      = .so

# ========= Shared Macros
===============================================

RTSYSTEM_INCPATHS = \
           $(INCLUDE_TAG)"$(RTS_HOME)/libset/$(LIBRARY_SET)" \
           $(INCLUDE_TAG)"$(RTS_HOME)/target/$(TARGET)" \
           $(INCLUDE_TAG)"$(RTS_HOME)/include"

RTS_LIBRARY    = $(RTS_HOME)/lib/$(CONFIG)
```

```
SYSTEM_LIBS   =      $(DIR_TAG)"$(RTS_LIBRARY)" \
                     $(OT_LIB_TAG)ObjectTime \
                     $(OT_LIB_TAG)ObjectTimeTypes

# ========= Linking
====================================================

LD_OUT = $@

LD_HEAD = \
        $(EXEOUT_OPT) $(EXEOUT_TAG)$(LD_OUT) \
        $(LIBSETLDFLAGS) \
        "$(RTS_LIBRARY)/main$(OBJ_EXT)"

ALL_OBJS_LIST = $(ALL_OBJS)

LD_TAIL = \
        $(SYSTEM_LIBS) \
        $(TARGETLDFLAGS) \
        $(TARGETLIBS)

# ======== Compiling
===================================================

CC_HEAD = \
        $(OBJECT_OPT) $(OBJOUT_OPT) $(OBJOUT_TAG)$@ \
        $(LIBSETCCFLAGS) \
        $(TARGETCCFLAGS) \
        $(RTSYSTEM_INCPATHS)

CC_TAIL =

#
======================================================================
```

Target makefile

The **$RTS_HOME/target/<target>/target.mk** makefile provides definitions specific to the operating system. The definitions in this makefile override the defaults in **$RTS_HOME/libset/default.mk**. An example target makefile file, **$RTS_HOME/target/SUN5T/target.mk**, contains the following:

```
TARGETCCFLAGS = $(DEFINE_TAG)_REENTRANT
TARGETLDFLAGS = $(LIB_TAG)nsl $(LIB_TAG)socket -R$(RTS_LIBRARY)
TARGETLIBS    = $(LIB_TAG)posix4 $(LIB_TAG)thread
```

Libset makefile

The **`$RTS_HOME/libset/<libset>/libset.mk`** makefile provides definitions specific to the compiler. The definitions in this makefile override the defaults in **`$RTS_HOME/libset/default.mk`**. An example **libset makefile** file, **`$RTS_HOME/libset/sparc-gnu-2.8.1/libset.mk`**, contains the following:

For the C language:

C

```
VENDOR        = gnu

CC            = g++
SHLIB_CMD     = $(CC) -shared -z text -o

LIBSETCCFLAGS = -V2.8.1
LIBSETCCEXTRA = -O4 -finline -finline-functions -Wall -Winline \
                -Wwrite-strings
SHLIBS        =
LIBSETLDFLAGS = -V2.8.1
```

For the C++ language:

C++

```
VENDOR        = gnu

CC            = g++

LIBSETCCFLAGS = -V2.8.1 -fno-exceptions -fno-rtti
LIBSETCCEXTRA = -O4 -finline -finline-functions -fno-builtin \
                -Wall -Winline -Wwrite-strings
SHLIBS        =
LIBSETLDFLAGS = -V2.8.1
```

Config makefile

The **`$RTS_HOME/config/<config>/config.mk`** makefile provides definitions specific to the combination of the compiler, operating system and TargetRTS configuration. This **makefile** is empty for most target/libset combinations. Usually this file will only be needed to work around issues that may not appear in either the target or libset alone.

Note: Definitions in this file override the definitions in the target.mk and libset.mk files.

C An example use of this file for the C language can be found in**$RTS_HOME/config/OSE401T.ppc603-Diab-4.1a/config.mk**:

```
EXEC_EXT = .elf

TARGETCCFLAGS = \
    $(DEFINE_TAG)BIG_ENDIAN \
    $(INCLUDE_TAG)$(OSE_ROOT)/powerpc/include \
    $(INCLUDE_TAG)$(OSE_ROOT)/powerpc/krn-603/include

TARGETLDFLAGS = \
    $(DIR_TAG)$(OSE_ROOT)/powerpc/lib \
        $(LIB_TAG)inett \
        $(LIB_TAG)inetutil \
        $(LIB_TAG)rtc \
    $(DIR_TAG)$(OSE_ROOT)/powerpc/krn-603/lib \
        $(LIB_TAG)krn1dpr \
        $(LIB_TAG)krnflib
```

C++ An example use of this file for the C++ language can be found in **$RTS_HOME/config/VRTX4T.ppc603-Microtec-1.3C/config.mk**:

```
EXEC_EXT   = .x
TARGETLIBS = $(USR_MRI)/lib/cppcb.lib
```

Table defines which make macros can be redefined and where they are set.

Table 4 Make Macro Definitions

Macro Name	Defined where	Note
TARGET	Defined in **ms_nmake.mk**, **gnu_make.mk** and **unix_make.mk**.	Redefinition **not** recommended.
CONFIG	Defined in **default.mk**.	Redefinition **not** recommended.
PERL	Default defined in **default.mk** as "**rtperl**"	Some compilation hosts may require an explicit path; if necessary, redefine in **libset.mk** or **config.mk**.
FEEDBACK	Defined in **default.mk**.	Redefinition **not** recommended.
MERGE	Defined in **default.mk**.	Redefinition **not** recommended.

Table 4 Make Macro Definitions

NOP	Default defined in **`default.mk`**.	Redefinition from Perl script to (faster) OS-dependent command is possible.
RM	Default defined in **`default.mk`**.	Redefinition from Perl script to (faster) OS-dependent command is possible.
RMF	Default defined in **`default.mk`**.	Redefinition from Perl script to (faster) OS-dependent command is possible.
TOUCH	Default defined in **`default.mk`**.	Redefinition from Perl script to (faster) OS-dependent command is possible.
RTGEN	Defined in **`default.mk`**.	Redefinition **not** recommended.
RTCOMP	Defined in **`default.mk`**.	Redefinition **not** recommended.
RTLINK	Defined in **`default.mk`**.	Redefinition **not** recommended.
VENDOR	Default defined in **`default.mk`** as "generic" and intended to be overridden in **`libset.mk`**.	During porting, this may be left as "generic". However, you should provide an error-parser script eventually. Since error formats are typically vendor-specific (independent of the version of the compiler or of the compilation host-type), scripts are identified by the vendor's name in **`libset.mk`**.
MAKEFILE	Defined in **`default.mk`**.	Redefinition **not** recommended.
CC	Default defined in **`default.mk`** to cause compile-time error; must be redefined in **`libset.mk`**.	Must be redefined in **`libset.mk`** before porting.
DEBUG_TAG	Default defined in **`default.mk`**.	Redefine in **`libset.mk`** if necessary for a compiler.
DEPEND_TAG	Default defined in **`default.mk`**.	Redefine in **`libset.mk`** if necessary for a compiler.
DEFINE_TAG	Default defined in **`default.mk`**.	Redefine in **`libset.mk`** if necessary for a compiler.
INCLUDE_TAG	Default defined in **`default.mk`**.	Redefine in **`libset.mk`** if necessary for a compiler.
LIBSETCCEXTRA	Default defined in **`default.mk`**.	Add compiler-specific compilation flags in **`libset.mk`**, if necessary.

Table 4 Make Macro Definitions

LIBSETCCFLAGS	Default defined in **`default.mk`**.	Add compiler-specific compilation flags in **`libset.mk`**, if necessary.
OBJECT_OPT	Default defined in **`default.mk`**.	Redefine in **`libset.mk`** if necessary for a compiler.
OBJOUT_OPT	Default defined in **`default.mk`**.	Redefine in **`libset.mk`** if necessary for a compiler.
OBJOUT_TAG	Default defined in **`default.mk`**.	Redefine in **`libset.mk`** if necessary for a compiler.
TARGETCCFLAGS	Default defined in **`default.mk`**.	Add target-specific compilation flags in **`target.mk`**, if necessary.
AR_CMD	Default defined in **`default.mk`**.	Redefine in **`libset.mk`** if necessary for a linker.
LIBOUT_OPT	Default defined in **`default.mk`**.	Redefine in **`libset.mk`** if necessary for a linker.
LIBOUT_TAG	Default defined in **`default.mk`**.	Redefine in **`libset.mk`** if necessary for a linker.
RANLIB	Default defined in **`default.mk`**.	Redefine in **`libset.mk`** or **`target.mk`** if necessary for a linker.
LD	Default defined in **`default.mk`**.	Redefine in **`libset.mk`** if linker must be different from compiler (most compilers can invoke the linker anyhow), or if a preprocessing script is necessary.
DIR_TAG	Default defined in **`default.mk`**.	Redefine in **`libset.mk`** if necessary for a linker.
LIBSETLDFLAGS	Default defined in **`default.mk`**.	Redefine in **`libset.mk`** if necessary for a linker.
LIB_TAG	Default defined in **`default.mk`**.	Redefine in **`libset.mk`** if necessary for a linker.
OT_LIB_TAG	Default defined in **`default.mk`**.	Redefine in **`libset.mk`** if necessary for a linker.
TARGETLDFLAGS	Default defined in **`default.mk`**.	Redefine in **`config.mk`** or **`target.mk`** if necessary for a linker.
TARGETLIBS	Default defined in **`default.mk`**.	Redefine in **`config.mk`** or **`target.mk`** if necessary for a linker.

Table 4 Make Macro Definitions

EXEOUT_OPT	Default defined in **default.mk**.	Redefine in **libset.mk** or **config.mk** if necessary for a linker.
EXEOUT_TAG	Default defined in **default.mk**.	Redefine in **libset.mk** if necessary for a linker.
EXEC_EXT	Default defined in **default.mk**.	Redefine in **config.mk**, **libset.mk** or **target.mk** if necessary for a linker.
LIB_PFX	Default defined in **default.mk**.	Redefine in **config.mk** or **libset.mk** if necessary for a linker.
LIB_EXT	Default defined in **default.mk**.	Redefine in **libset.mk** if necessary for a linker.
OBJ_EXT	Default defined in **default.mk**.	Redefine in **libset.mk** if necessary for a compiler/linker.
RTSYSTEM_INCPATHS	Defined in **default.mk**.	Redefinition **not** recommended.
RTS_LIBRARY	Defined in **default.mk**.	Redefinition **not** recommended.
SYSTEM_LIBS	Defined in **default.mk**.	Redefinition **not** recommended.
LD_OUT	Defined in **default.mk**.	Redefinition **not** recommended.
LD_HEAD	Default defined in **default.mk**.	Redefine in **config.mk**, **libset.mk** or **target.mk** if necessary for a linker.
ALL_OBJS_LIST	Default defined in **default.mk**. as the concatenation of all object files in the update.	Redefine in **libset.mk** to “%$(ALL_OBJS_LISTFILE)” to pass list of object files to linker (or linker script), if line length limitations forbid passing list via shell.
LD_TAIL	Default defined in **default.mk**.	Redefine in **config.mk**, **libset.mk** or **target.mk** if necessary for a linker.
CC_HEAD	Default defined in **default.mk**.	Redefine in **config.mk**, **libset.mk** or **target.mk** if necessary for a compiler.
CC_TAIL	Default defined in **default.mk**.	Redefine in **config.mk**, **libset.mk** or **target.mk** if necessary for a compiler.

Porting the TargetRTS for C

5

Contents

This chapter is organized as follows:

- *Configuring the TargetRTS* on page 71
- *Platform-specific Implementation* on page 75
- *Adding New Files to the TargetRTS* on page 80

Configuring the TargetRTS

Much of the configurability of the TargetRTS is done at the source code file level: target-specific source files override common source files. This is illustrated in the next section on platform-specific implementations. However, configurability is also available within a source file using preprocessor definitions. The configuration is set in two C header files:

- `$RTS_HOME/target/<target>/RTTarget.h` for specifying the operating system specific definitions.
- `$RTS_HOME/libset/<libset>/RTLibSet.h` for specifying the compiler specific definitions; this file does not exist by default.

Definitions made in these files override their default definitions in `$RTS_HOME/include/RTPubl/Config.h`. The symbols and their default values are listed in Table .

Note: In Table , in general, defining a symbol with the value 1 enables (= sets) the feature the symbol represents and defining it with the value 0 disables (= clears) the feature.

Table 5 Preprocessor Definitions

Symbol	Default Value	Possible Values	Description
USE_THREADS	none, must be defined in the platform headers (usually `RTTarget.h`)	0 or 1	Determines whether the single-threaded or multi-threaded version of the TargetRTS is used. If USE_THREADS is 0, the TargetRTS is single-threaded. If USE_THREADS is 1, the TargetRTS is multi-threaded.
MESSAGE_ DEFERRAL	1	0 or 1	If 1, message deferral capabilities per controller will be present in the TargetRTS. If 0, no message deferral capabilities at all.
TIMING_SERVICE	1	0 or 1	If 1, timing service will be available in the TargetRTS.
TO_OVER_TCP	1if OBSERVABLE	0 or 1	Set to 1 if Target Observability over TCP/IP should be supported.
LOG_MESSAGE	1if OTRTSDEBUG != DEBUG_NONE	0 or 1	Sets whether the debugger can log the contents of messages.
LOG_SERVICE	1	0 or 1	Sets whether the RTLog_show_... methods should be available or not.
RTS_NAMES	1	0 or 1	Sets whether the name strings in the data **structs** should be present or not.
STDIO_ENABLED	1	0 or 1	Sets whether the **RTStdio_** and **RTLog_** methods should be available or not.
OBJECT_DECODE	1	0 or 1	Enables the conversion of strings to objects. Needed for Target Observability.
OBJECT_ENCODE	1	0 or 1	Enables the conversion of objects to strings. Needed for Target Observability.
SEND_BY_VALUE	1	0 or 1	If 1, send data using type descriptors. If 0, just send pointers.

Table 5 Preprocessor Definitions

Symbol	Default Value	Possible Values	Description
OTRTSDEBUG	DEBUG_VERBOSE	DEBUG_VERBOSE	Enables the TargetRTS debugger. It will make it possible to log all important internal events such as the delivery of messages, the creation and destruction of capsules, and so on. This is necessary for the target debug feature.
		DEBUG_NONE	Reduces the size of the resulting executable while increasing performance. However, the RTS debugger will not be available.
RTS_MEMORY_POLICY	RTS_CAN_ALLOCATE if OBSERVABLE or PURIFY, else RTS_NEVER_ALLOCATE	RTS_CAN_ALLOCATE	Dynamic memory allocation is always allowed.
		RTS_WARN_ALLOCATE	Dynamic memory allocation is always allowed, but a warning is printed on the console.
		RTS_NEVER_ALLOCATE	Dynamic memory allocation is not allowed at all after system initialization.
PURIFY	0	0 or 1	If 1, this flag indicates that the Purify tool is being used. This tells the TargetRTS to disable all object caching, which degrades performance but allows Purify to monitor **RTMessage** objects.
RTS_COMPATIBLE	521	521 or 610	If 521, obsolete features from ObjecTime Developer 5.2.1 of the TargetRTS will be present. Set to 610 to disable backwards compatibility.
RTS_INLINES	0	0 or 1	Controls whether TargetRTS header files define any inline functions.
RTMESSAGE_PAYLOAD_SIZE	36	any scalar value >= 0	Reserve this many bytes in **RTMessage** for small objects. When data must be copied, objects that are no larger than this will use that space in the message itself rather than allocated on the heap.

Table 5 Preprocessor Definitions

Symbol	Default Value	Possible Values	Description
INTERNAL_LAYER_ SERVICE	1	0 or 1	Should internal SAPs and SPPs be supported?
MAX_NUM_SPPS	10	any scalar value > 0	Maximum number of SAPs and SPPs that can be connected at any given time.
DEBUGGER_STACK_ SIZE	20480	any scalar value > 0	Stack size in bytes for the debugger ("main") thread.
MINIMUM_FREE_ MSGQ_SIZE	5	any scalar value > 0	When freeing a message, keep at least this many messages in the Controller's free list.
DEFAULT_FREE_ MSGQ_SIZE	10	any scalar value > MINIMUM_ FREE_ MSGQ_SIZE	When freeing a message, keep at most this many messages in the Controller's free list.
RTS_CLEANUP_ MECHANISM	1	0 or 1	If 1, provide destructors and call them on shutdown, etc. If 0, do not (this is a space optimization).
MULTIPLE_ PRIORITIES	1	0 or 1	If 1, there are 6 distinct priorities and 6 message queues per controller. If 0, there is only 1 priority and 1 queue per controller.
INLINE_CHAINS	<blank>	inline or <blank>	**Inlines** state machine chains for better performance at the expense of potentially larger executable memory size.
INLINE_METHODS	<blank>	inline or <blank>	**Inlines** user-defined capsule methods for better performance at the expense of potentially larger executable memory size.
OBSERVABLE	1 if debugger, decoding and encoding all are enabled.	0 or 1	The ability to use the Target Observability facilities.

Platform-specific Implementation

The implementation of the TargetRTS is contained in the $RTS_HOME/src directory. In this directory, there is a subdirectory for each class. In general, within each subdirectory there is one source file for each method in the class. Wherever possible, the name of the source file matches the name of the method.

To port the TargetRTS to a new platform, it may be necessary to replace some of these methods. Additionally, some of the methods that do not have default behaviors must be provided. The target-specific source is placed in a subdirectory of $RTS_HOME/src/target/<target_base>, where **<target_base>** is defined by **$target_base** variable in the file setup.pl file (see *Creating a Setup Script (setup.pl)* on page 54). The target name often appears with the trailing 'S' or 'T'. The name defaults to the target name without the "S" or "T" if the variable **$target_base** is not defined in the setup.pl file. For the remainder of this section, the target directory is referred to as $TARGET_SRC. For example, the target source directory for `<target> SUN5T` is $RTS_HOME/src/target/SUN5. This directory provides an overlay to the $RTS_HOME/src directory. When the TargetRTS **loadbuild** tools search for the source for a method, it searches first in the $TARGET_SRC directory, then in $RTS_HOME/src.

Note: There is only a single source directory for all configurations of the TargetRTS for a given platform. C preprocessor macros, such as **USE_THREADS**, may be used to differentiate code for specific configurations.

There is a sample port in the $RTS_HOME/target/sample subdirectory to use as a template for a port to a new target. These implementations can be incorporated into a target implementation by copying the contents of these subdirectories into the $TARGET_SRC directory. You may also want to search the other target subdirectories to verify that the implementation of various TargetRTS classes resembles your target RTOS. You can copy any required code to the new $TARGET_SRC directory.

Table 6 shows the functions that must be provided in any port of the TargetRTS. These are the minimum requirements for a new port, as most ports will include changes to more classes than those listed.

Table 6 Required TargetRTS Classes and Functions

Required TargetRTS Classes and Functions
`RTTimespec_clock_gettime()`
`RTThread_construct()`

Table 6 Required TargetRTS Classes and Functions

Required TargetRTS Classes and Functions
`RTMutex (all 4 methods)`
`RTSyncObject` (all 5 methods)

The remainder of this section discusses the most common required implementation code required for a new target.

Method RTTimespec_clock_gettime(timespec)

To implement the Timing service, the TargetRTS uses the time of day clock. The method `RTTimespec_clock_gettime()`, found in the file $TARGET_SRC/Timespec/getclock.c, gets the time of day from the operating system. There is **no** default implementation of this method and it **must be provided by the target**. The format of this time of day is the POSIX-style `struct timespec` which contains two fields: the number of seconds and the number of nanoseconds from some fixed point of time. This fixed point is usually the Universal Time reference point of January 1, 1970. This does not need to be the case. However, to support absolute time-outs, the TargetRTS assumes that the reference time is midnight of some day.

Constructor RTThread_construct(this,job,priority,stacksize)

To support multi-threading, the TargetRTS provides the class **RTThread**. The constructor should create a stack and start a new thread using RTThread_run(this) as its entry point. There is **no** default implementation; any multi-threaded target implementation must provide the constructor for this class in the file $TARGET_SRC/Thread/ct.c.

Class RTMutex

In the multi-threaded TargetRTS, shared resources are protected using **mutexes** implemented by the class **RTMutex**. There is no default declaration or implementation. The description of the **RTMutex** class should be placed in the file $TARGET_SRC/RTPriv/Mutex.h.

There are four methods to `RTMutex`:

- `RTMutex_construct(this)` - the constructor, in $TARGET_SRC/Mutex/ct.c, performs any initialization of the mutex.
- `RTMutex_destruct(this)` - the destructor, in $TARGET_SRC/Mutex/dt.c, performs any clean up when the mutex is no longer required.
- `RTMutex_enter(this)` - in $TARGET_SRC/Mutex/enter.c, locks the mutex if it is available, or blocks the current thread until it is available.
- `RTMutex_leave(this)` - in $TARGET_SRC/Mutex/leave.c, frees the mutex and unblocks the first thread waiting on the `RTMutex_enter()`.

Class RTSyncObject

An additional synchronization mechanism used by the TargetRTS is implemented by class **RTSyncObject**. Many operating systems provide what is known as a 'binary semaphore'. A synchronization object is essentially the same thing. Many implementations of a semaphore, however, do not provide a wait (or 'pend') with time-out. The lack of this time-out feature requires the use of a more heavyweight implementation using a mutex and a condition variable (POSIX condition variables have a '**timedwait**' feature). A description of each method can be found in the $RTS_HOME/src/target/sample/SyncObj directory. There is no default declaration or implementation. The description of the **RTSyncObject** class should be placed in the file $TARGET_SRC/RTPriv/SyncObj.h. The implementation of five methods is required:

- `RTSyncObject_construct(this)` - the constructor, in `$TARGET_SRC/SyncObj/ct.c`, performs any initialization required.
- `RTSyncObject_destruct(this)` - the destructor, in `$TARGET_SRC/SyncObj/dt.c`, performs any clean up given that the sync object is no longer required.
- `RTSyncObject_signal(this)` - in `$TARGET_SRC/SyncObj/signal.c`. Signal this synchronization object. If the owner is currently waiting, it should be readied. Otherwise the state of this object should be such that the next call to wait or timedwait made by the owner will not block. Signalling a second or subsequent time should have no effect.
- `RTSyncObject_timedwait(this, expiryTime)` - in `$TARGET_SRC/SyncObj/timewait.c`. Wait for this synchronization object to be signalled. Only the owning thread is permitted to use this function. If the object is in the 'signalled' state it should be reset to 'unsignalled' and the function

should return immediately. Otherwise the current thread should block until either the object is signalled by another thread or the absolute expiry time arrives, whichever occurs first. The object should always be left in the 'unsignalled' state.

- `RTSyncObject_wait(this)` - in `$TARGET_SRC/SyncObj/wait.c`. Wait for this synchronization object to be signalled. Only the owning thread is permitted to use this function. If the object is in the 'signalled' state it should be reset to 'unsignalled' and the function should return immediately. Otherwise the current thread should block until the object is signalled by another thread. The object should always be left in the 'unsignalled' state.

main() function

In order for the execution of the TargetRTS to begin, code must be provided to call `RTMain_entryPoint( int argc, const char * const * argv )`, passing in the arguments to the program. This code is placed in the file `$TARGET_SRC/Main/main.c`.

On many platforms, this is the code for the `main()` function, which simply passes `argc` and `argv` directly. However, on other platforms, these parameters must be constructed. For example, with Tornado, the arguments to the program are placed on the stack. An array of strings containing the arguments must be explicitly created.

If the platform does not provide a mechanism for passing arguments to an executable, default arguments for use by `RTMain_entryPoint()` can be defined in the toolset. These arguments are made available by the code generator, and can be used by overriding `main()` to call `RTMain_entryPoint( 0, (const char * const *)0 );` instead.

Class RTMain

`RTMain_entryPoint()` indirectly via `RTMain_mainLine()` calls a number of methods for target-specific initialization and shutdown. These methods are as follows:

- `RTMain_startup()` - in file `$TARGET_SRC/Main/startup.c`, it initializes the target in preparation for execution of the model. This includes things such as setting the priority of the main thread, calling static constructors, and initializing devices, for example, timers and consoles. Note that on most platforms this method is empty.
- `RTMain_shutdown()` - in file `$TARGET_SRC/Main/shutdown.c`, it generally undoes the initialization that was performed in `RTMain_startup()`, for example, calling static destructor and cleaning up operating resources such as file descriptors.

- RTMain_installHandlers() - in file $TARGET_SRC/Main/allHand.c. In addition to target start-up and shutdown, RTMain_mainLine() also calls this method to install Unix style signal handlers, where available. These signal handlers are used by the single threaded TargetRTS for timer and I/O interrupts. If the target OS does not implement signal handlers, this method can be overridden by an empty method.
- RTMain_installOneHandler() - in file $TARGET_SRC/Main/oneHand.c. This method is used by RTMain_installHandlers() to install the Unix style signal handlers. These signal handlers are used by the single threaded TargetRTS for timer and I/O interrupts. If the target OS does not implement signal handlers, this method can be overridden by an empty method.

Method RTStdio_putString()

The RTStdio class handles output of diagnostic messages to the standard error. If your target does not support the fputs() function, you must supply a replacement for the RTStdio_putString() method in $TARGET_SRC/Stdio/string.c. This method outputs a string to the standard error device.

Method RTDebuggerInput_nextChar()

The RTDebuggerInput class handles the input to the TargetRTS debugger. If your target system does not support the fgetc() function, then you must supply a replacement for the RTDebuggerInput_nextChar() method in $TARGET_SRC/DebugInp/nextChar.c. This method reads individual characters from the standard input device.

Class RTTcpSocket

The RTTcpSocket class provides an interface from the TargetRTS to the sockets library of the target operating system. Many operating systems provide the familiar BSD sockets interface. If this is the case then little modification is necessary. Typically, small changes to data types are needed to satisfy the sockets interface. If code changes are required, override the functions in RTinet.

Note: This class is not necessary if you do not use Target Observability (set the OBSERVABLE macro to 0), and if your application does not require TCP/IP networking.

Class RTIOMonitor

The `RTIOMonitor` class is used to monitor activity on a set of TCP/IP sockets. This class makes use of file descriptor sets and the `select()` function. There may be differences in the way these sets are implemented on your target operating system.

File main.c

The file `main.c` contains the `main` function for the TargetRTS and therefore the entire application. Some operating systems already have a `main` function defined. This file must be modified to take this into account. A typical solution is to create a root thread, which in turn calls the entry point to the TargetRTS, `RTMain_entryPoint()`.

Adding New Files to the TargetRTS

If you create a new method in a new file for an existing class, or you are adding a new class to the TargetRTS, then you must add the new file names to a manifest file. This must be done in order for the dependency calculations to include the new files and thus include them into the TargetRTS.

The MANIFEST.c File

This file lists all the elements of the run-time system. There is one entry per line, and each entry has two or more fields separated by white space. The first field is a directory name. The second field is the base name of a file. By convention the directory name and file name typically correspond to the class name and member name, respectively. The third and subsequent fields, if present, give an expression that evaluates to zero when the element should be excluded. Note that the expression is evaluated by Perl and so should be of a form that it can handle.

If you have added a new generic (non-target specific) source file to the TargetRTS, you must add an entry to the `$RTS_HOME/src/MANIFEST.c` file for this file. By convention, the entry should be placed next to the other files for the specific class that you have modified. If you are adding a whole class, then place the entries next to the super class if it exists, or next to similar classes in the `MANIFEST.c` file.

Be sure to associate the new entry with the proper `GROUP`, see `MANIFEST.c` for details.

A target base directory can optionally contain the file called RTPriv/TGTRFEST.c that uses the same format and services to specify file names to that particular target base.

Regenerating make Dependencies

If a file has been overridden in `$TARGET_SRC` directory or a new file has been added to the `MANIFEST.c`, you must regenerate the dependencies in order for the modification to be included in the new TargetRTS. This is done by removing the `depend.mk` file in the build directory, `$RTS_HOME/build-<config>`. This will cause the dependencies to be recalculated and a new `depend.mk` file to be created.

Porting the TargetRTS for C++ 6

Contents

This chapter is organized as follows:

- *Configuring the TargetRTS* on page 83
- *Platform-specific Implementation* on page 88
- *Adding New Files to the TargetRTS* on page 93

Configuring the TargetRTS

Much of the configurability of the TargetRTS is done at the source code file level: target-specific source files override common source files. This is illustrated in the next section on platform-specific implementations. However, configurability is also available within a source file using preprocessor definitions. The configuration is set in two C++ header files:

- `$RTS_HOME/target/<target>/RTTarget.h` for specifying the operating system specific definitions.
- `$RTS_HOME/libset/<libset>/RTLibSet.h` for specifying the compiler specific definitions; this file does not exist by default.

Definitions made in these files override their default definitions in `$RTS_HOME/include/RTConfig.h`. The symbols and their default values are listed in Table 7.

Note: In Table 7, in general, defining a symbol with the value 1 enables (= sets) the feature the symbol represents and defining it with the value 0 disables (= clears) the feature.

Table 7 Preprocessor Definitions

Symbol	Default Value	Possible Values	Description
USE_THREADS	none, must be defined in the platform headers (usually `RTTarget.h`)	0 or 1	Determines whether the single-threaded or multi-threaded version of the TargetRTS is used. If USE_THREADS is 0, the TargetRTS is single-threaded. If USE_THREADS is 1, the TargetRTS is multi-threaded.
DEFER_IN_ACTOR	0	0 or 1	If 1, there will be one defer queue in each capsule. If 0, there will only be one defer queue per controller. This is a size/speed trade-off. Separate queues for each capsule uses more memory but results in better performance.
HAVE_INET	1	0 or 1	Set to 1 if TCP/IP is supported.
INTEGER_POSTFIX	1	0 or 1	Sets whether the compiler understands the post increment operator on classes. i.e. Class x; x++;
LOG_MESSAGE	1	0 or 1	Sets whether the debugger can log the contents of messages.
OBJECT_DECODE	1	0 or 1	Enables the conversion of strings to objects, needed for Target Observability.
OBJECT_ENCODE	1	0 or 1	Enables the conversion of objects to strings. Needed for Target Observability.

Table 7 Preprocessor Definitions

Symbol	Default Value	Possible Values	Description
OTRTSDEBUG	DEBUG_VERBOSE	DEBUG_VERBOSE	Enables the TargetRTS debugger. It will make it possible to log all important internal events such as the delivery of messages, the creation and destruction of capsules, and so on. This is necessary for the target observability feature.
		DEBUG_TERSE	Reduces the size of the resulting executable at the expense of limiting the amount of debug information.
		DEBUG_NONE	Further reduces the executable size, while increasing performance. However, the RTS debugger will not be available.
PURIFY	0	0 or 1	If 1, this flag indicates that the Purify tool is being used. This tells the TargetRTS to disable all object caching, which degrades performance but allows Purify to monitor **RTMessage** objects.
RTS_COMPATIBLE	520	520, 600 or 620	If 520, obsolete features from ObjecTime Developer 5.2 of the TargetRTS will be present. If 600, obsolete features from version 6.0 of the TargetRTS will be present. Set to 620 to disable backwards compatibility.
RTS_COUNT	0	0 or 1	If this flag is 1, the TargetRTS will keep track of the number of messages sent, the number of capsules incarnated, and other statistics. Naturally, keeping track of statistics adds overhead.
RTS_INLINES	1	0 or 1	Controls whether TargetRTS header files define any inline functions.

Table 7 Preprocessor Definitions

Symbol	Default Value	Possible Values	Description
RTFRAME_ THREAD_SAFE	1	0 or 1	Setting this macro to 1 guarantees that the frame service is thread safe. This is an option because some applications may use the frame service in ways that don't require this level of safety.
RTFRAME_ CHECKING	RTFRAME_ CHECK_STRICT	RTFRAME_ CHECK_STRICT	The frame service is intended to provide operations on components of the capsules which have a frame SAP. Here, references must be in same capsule.
		RTFRAME_ CHECK_LOOSE	References must be in same thread (but not the same capsule).
		RTFRAME_ CHECK_NONE	No checking is done. This is compatible with ObjecTime Developer pre-5.2.
RTMESSAGE_ PAYLOAD_SIZE	100	any scalar value >= 0	Reserve this many bytes in **RTMessage** for small objects. When data must be copied, objects that are no larger than this will use that space in the message itself rather than allocated on the heap.
RTREAL_INCLUDED	1	0 or 1	Should the class **RTReal** be present? Target environments that don't support floating point data types, or can't afford them, should set it to 0.

Table 7 Preprocessor Definitions

Symbol	Default Value	Possible Values	Description
RTTYPECHECK_ PROTOCOL	RTTYPECHECK_ WARN	RTTYPECHECK_ FAIL	What to do about protocols which have signals of incompatible data types? Set error code, fail operation.
		RTTYPECHECK_ WARN	Set error code, but proceed.
		RTTYPECHECK_ DONT	No checking.
RTTYPECHECK_ SEND	RTTYPECHECK_ WARN	(see above)	What to do about send, invoke or reply when the signal or type is incompatible with the protocol?
RTTYPECHECK_ RECEIVE	RTTYPECHECK_ DONT or RTTYPE-CHECK_WARN (depending on the two above)	(see above)	Should signal be checked for signal and type compatibility as it is received?
RTQUALIFY_ NESTED	0	0 or 1	Some compilers have trouble with the class nesting for protocol backwards compatibility and require the class names to be fully qualified.
RTUseBitFields	0	0 or 1	Some structures can be made smaller through the use of bit-fields. This space savings often comes at the expense of greater code bulk.
SUSPEND	0	0	The ability to 'suspend' capsules is currently unsupported. Leave at 0.
RTStateId_MaxSize	2 bytes (< 65536 states)	1 byte (<256 states), 2 bytes, or 4 bytes (>=65536 states)	Maximum number of bytes allocated to store each state id.
RTStateId	This is a typedef calculated from the value of **RTStateId_MaxSize**. Do not modify directly, adjust **RTStateId_MaxSize** instead.		

Table 7 Preprocessor Definitions

Symbol	Default Value	Possible Values	Description
INLINE_CHAINS	<blank>	inline or <blank>	**Inlines** state machine chains for better performance at the expense of potentially larger executable memory size.
INLINE_METHODS	<blank>	inline or <blank>	**Inlines** user-defined capsule methods for better performance at the expense of potentially larger executable memory size.
OBSERVABLE	1 if debugger, inet, decoding and encoding all are enabled.	0 or 1	The ability to use the Target Observability facilities.
EXTERNAL_LAYER	0	0	The "els" connection service is not provided. Leave at 0.

Platform-specific Implementation

The implementation of the TargetRTS is contained in the `$RTS_HOME/src` directory. In this directory, there is a subdirectory for each class. In general, within each subdirectory there is one source file for each method in the class. Wherever possible, the name of the source file matches the name of the method.

To port the TargetRTS to a new platform, it may be necessary to replace some of these methods. Additionally, some of the methods that do not have default behaviors must be provided. The target-specific source is placed in a subdirectory of `$RTS_HOME/src/target/<target_base>`, where `<target_base>` is the target name without the trailing 'S' or 'T'. For the remainder of this section, the target directory is referred to as `$TARGET_SRC`. For example, the target source directory for `<target> PSOS2T` is `$RTS_HOME/src/target/PSOS2`. This directory provides an overlay to the `$RTS_HOME/src` directory. When the TargetRTS **loadbuild** tools search for the source for a method, it searches first in the `$TARGET_SRC` directory, then in `$RTS_HOME/src`.

Note: There is only a single source directory for all configurations of the TargetRTS for a given platform. C++ preprocessor macros, such as `USE_THREADS`, may be used to differentiate code for specific configurations.

There is a sample port in the `$RTS_HOME/src/target/sample` subdirectory to use as a template for a port to a new target. These implementations can be incorporated into a target implementation by copying the contents of these subdirectories into the `$TARGET_SRC` directory. You may also want to search the other target subdirectories to verify that the implementation of various TargetRTS classes resembles your target RTOS. You can copy any required code to the new `$TARGET_SRC` directory.

Table 8 shows the classes and functions that must be provided in any port of the TargetRTS. These are the minimum requirements for a new port, as most ports will include changes to more classes than those listed.

Table 8 Required TargetRTS Classes and Functions

Required TargetRTS Classes and Functions
`RTTimespec::getclock()`
`RTThread::RTThread()`
`RTMutex (all 4 methods)`
`RTSyncObject` (all 5 methods)

The remainder of this section discusses the most common required implementation code required for a new target.

Method RTTimespec::getclock()

To implement the Timing service, the TargetRTS uses the time of day clock. The method `RTTimespec::getclock()`, found in the file `$TARGET_SRC/RTTimespec/getclock.cc`, gets the time of day from the operating system. There is **no** default implementation of this method and it **must be provided by the target**. The format of this time of day is the POSIX-style `RTTimespec` which contains two fields: the number of seconds and the number of nanoseconds from some fixed point of time. This fixed point is usually the Universal Time reference point of January 1, 1970. This does not need to be the case. However, to support absolute time-outs, the TargetRTS assumes that the reference time is midnight of some day.

Constructor RTThread::RTThread()

To support multi-threading, the TargetRTS provides the class `RTThread`. The constructor should create a stack and start a new thread using `job->`**`mainLoop`**`()` as its entry point. There is **no** default implementation, the target implementation must provide the constructor for this class in the file `$TARGET_SRC/RTThread/ct.cc`.

Class RTMutex

In the multi-threaded TargetRTS, shared resources are protected using **mutexes** implemented by the class `RTMutex`. There is no default declaration or implementation. The description of the `RTMutex` class should be placed in the file `$TARGET_SRC/RTMutex.h`. There are four methods to `RTMutex`:

- `RTMutex()` - the constructor, in `$TARGET_SRC/RTMutex/ct.cc`, performs any initialization of the **mutex**.
- `~RTMutex()` - the destructor, in `$TARGET_SRC/RTMutex/dt.cc`, performs any clean up when the **mutex** is no longer required.
- `enter()` - in `$TARGET_SRC/RTMutex/enter.cc`, locks the **mutex** if it is available, or blocks the current thread until it is available.
- `leave()` - in `$TARGET_SRC/RTMutex/leave.cc`, frees the **mutex** and unblocks a thread waiting on the `enter()`.

Class RTSyncObject

An additional synchronization mechanism used by the TargetRTS is implemented by class `RTSyncObject`. Many operating systems provide what is known as a 'binary semaphore'. A synchronization object is essentially the same thing. Many implementations of a semaphore, however, do not provide a wait (or 'pend') with time-out. The lack of this time-out feature requires the use of a more heavyweight implementation using a **mutex** and a condition variable (POSIX condition variables have a '**timedwait**' feature). A description of each method can be found in the `$RTS_HOME/src/target/sample/RTSyncObject` directory. There is no default declaration or implementation. The description of the `RTSyncObject` should be in the file `$TARGET_SRC/RTSyncObject.h`.

The implementation of five methods is required:

- `RTSyncObject()` - the constructor, in `$TARGET_SRC/RTSyncObject/ct.cc`, performs any initialization required.
- `~RTSyncObject()` - the destructor, in `$TARGET_SRC/RTSyncObject/dt.cc`, performs any clean up given that the sync object is no longer required.
- `signal()` - in `$TARGET_SRC/RTSyncObject/signal.cc`. Signal this synchronization object. If the owner is currently waiting, it should be readied. Otherwise the state of this object should be such that the next call to wait or **timedwait** made by the owner will not block. Signalling a second or subsequent time should have no effect.
- `wait()` - in `$TARGET_SRC/RTSyncObject/wait.cc`. Wait for this synchronization object to be signalled. Only the owning thread is permitted to use this function. If the object is in the 'signalled' state it should be reset to 'unsignalled' and the function should return immediately. Otherwise the current thread should block until the object is signalled by another thread. The object should always be left in the 'unsignalled' state.
- `timedwait()` - in `$TARGET_SRC/RTSyncObject/timedwait.cc`. Wait for this synchronization object to be signalled. Only the owning thread is permitted to use this function. If the object is in the 'signalled' state it should be reset to 'unsignalled' and the function should return immediately. Otherwise the current thread should block until either the object is signalled by another thread or the absolute expiry time arrives, whichever occurs first. The object should always be left in the 'unsignalled' state.

main() function

In order for the execution of the TargetRTS to begin, code must be provided to call `RTMain::entryPoint( int argc, const char * const * argv )`, passing in the arguments to the program. This code is placed in the file `$TARGET_SRC/MAIN/main.cc`.

On many platforms, this is the code for the `main()` function, which simply passes **`argc`** and **`argv`** directly. However, on other platforms, these parameters must be constructed. For example, with Tornado, the arguments to the program are placed on the stack. An array of strings containing the arguments must be explicitly created.

If the platform does not provide a mechanism for passing arguments to an executable, default arguments for `entryPoint()` can be defined in the toolset. These arguments are made available by the code generator, and can be used by overriding `main()` to call `RTMain::entryPoint( 0, (const char * const *)0 );` instead.

Class RTMain

`RTMain::mainLine()` indirectly calls a number of methods for target-specific initialization and shutdown. These methods are as follows:

- `targetStartup()` - in file `$TARGET_SRC/RTMain/targetStartup.cc`, it initializes the target in preparation for execution of the model. This includes things such as initializing devices, for example, timers and consoles.
- `targetShutdown()` - in file `$TARGET_SRC/RTMain/targetShutdown.cc`, it generally undoes the initialization that was performed in `targetStartup()`, for example, cleaning up operating resources such as file descriptors.
- `installHandlers()` - in file `$TARGET_SRC/RTMain/installHandlers.cc`. In addition to target start-up and shutdown, `RTMain::mainLine()` also calls this method to install Unix style signal handlers, where available. These signal handlers are used by the single threaded TargetRTS for timer and I/O interrupts. If the target OS does not implement signal handlers, this method can be overridden by an empty method.
- `installOneHandler()` - in file `$TARGET_SRC/RTMain/installOneHandler.cc`. This method is used by `RTMain::installHandlers()` to install the Unix style signal handlers. These signal handlers are used by the single threaded TargetRTS for timer and I/O interrupts. If the target OS does not implement signal handlers, this method can be overridden by an empty method.

Method RTDiagStream::write()

The `RTDiagStream` class handles output of diagnostic messages to the standard error. If your target does not support the `fputs()` function, you must supply a replacement for the `RTDiagStream::write()` method in `$TARGET_SRC/RTDiagStream/write.cc`. This method outputs a string to the standard error device.

Method RTDebuggerInput::nextChar()

The `RTDebuggerInput` class handles the input to the TargetRTS debugger. If your target system does not support the `fgetc()` function, then you must supply a replacement for the `RTDebuggerInput::nextChar()` method in `$TARGET_SRC/RTDebuggerInput/nextChar.cc`. This method reads individual characters from the standard input device.

Class RTTcpSocket

The `RTTcpSocket` class provides an interface from the TargetRTS to the sockets library of the target operating system. Many operating systems provide the familiar BSD sockets interface. If this is the case then little modification is necessary. Typically, small changes to data types are needed to satisfy the sockets interface. If code changes are required, override the functions in `RTinet`.

Note: This class is not necessary if you do not plan to use Target Observability (Set the OBSERVABLE macro to 0), and if your application does not require TCP/IP networking.

Class RTIOMonitor

The `RTIOMonitor` class is used to monitor activity on a set of TCP/IP sockets. This class makes use of file descriptor sets and the `select()` function. There may be differences in the way these sets are implemented on your target operating system. Only `RTIOMonitor::wait` should need modification.

File main.cc

The file `main.cc` contains the `main` function for the TargetRTS and therefore the entire application. Some operating systems already have a `main` function defined. This file must be modified to take this into account. A typical solution is to create a root thread, which in turn calls the entry point to the TargetRTS, `RTMain::entryPoint()`.

Adding New Files to the TargetRTS

If you create a new method in a new file for an existing class, or you are adding a new class to the TargetRTS, then you must add the new file names to a manifest file. This must be done in order for the dependency calculations to include the new files and thus include them into the TargetRTS.

The MANIFEST.cpp File

This file lists all the elements of the run-time system. There is one entry per line, and each entry has two or more fields separated by white space. The first field is a directory name. The second field is the base name of a file. By convention the directory name and file name typically correspond to the class name and member name, respectively. The third and subsequent fields, if present, give an expression that evaluates to zero when the element should be excluded. Note that the expression is evaluated by Perl and so should be of a form that it can handle.

If you have added a new generic (non-target specific) source file to the TargetRTS, you must add an entry to the `$RTS_HOME/src/MANIFEST.cpp` file for this file. By convention, the entry should be placed next to the other files for the specific class that you have modified. If you are adding a whole class, then place the entries next to the super class if it exists, or next to similar classes in the `MANIFEST.cpp` file.

If the added file is target specific, add an entry to `$TARGET_SRC/TARGET-MANIFEST.cpp` instead (create this file if it doesn't exist already).

In both cases, be sure to associate the new entry with the proper `GROUP`, see `MANIFEST.cpp` for details.

Regenerating make Dependencies

If a file has been overridden in `$TARGET_SRC` directory or a new file has been added to the `MANIFEST.cpp`, you must regenerate the dependencies in order for the modification to be included in the new TargetRTS. This is done by removing the `depend.mk` file in the build directory, `$RTS_HOME/build-<config>`. This will cause the dependencies to be recalculated and a new `depend.mk` file to be created.

Modifying the Error Parser 7

Contents

This chapter is organized as follows:

- *Overview of the Error Parser* on page 95
- *How the Error Parser Works* on page 96
- *Reusing an Existing Error Parser* on page 98
- *Creating a New Error Parser* on page 98

Overview of the Error Parser

The error parser is intended to convert specific compiler (or linker) error messages into a format that can be browsed by the modeling user from the Build Errors tab within the toolset. Whenever possible, the format identifies a browseable model element, as well as including the description and the severity of the compiler message.

Typically, compilers cite a particular line-number of a source file when producing an error or warning message. Since the source files are generated by the code-generator, the line numbers are meaningless to the modeling user. The error parser provides a mechanism to translate a line-number from an arbitrary source file into a reference to a particular model element. The intention is that the modeling user can double-click a compiler message and see where the problem occurred in the model: for example which transition, or which member definition. The user can then take corrective action and compile the model again. Unfortunately (as with hand-written source files), the corrective action is not always necessary where the problem occurred, but it is usually a good start.

Most linker messages do not cite a particular line-number, since their problems are typically about undefined symbols, multiply defined symbols or misuses of the command-line options. In these cases, the errors can be resolved by modifying a component within the model. It is not possible to always correctly determine which component property, or even which component produced the message (typically the executable component is tagged).

The error parser is intended as a convenience to the model designer, but it cannot correctly identify the source model-element for all errors, including compiler command-line errors, compilation errors caused by external header files or linkage errors. In these cases, no model-element is given, but an error message should still be returned to the toolset.

How the Error Parser Works

Before modifying the error parser, it is important to understand how it works.

The Error Parsing Rules

The error parsing rules are considered vendor-specific; they do not vary dramatically between compilation host platforms or between subsequent compiler-version releases. Each **libset** references its associated error parser via the **`VENDOR`** make macro in the **`$RTS_HOME/libset/<libset>/libset.mk`** file. For each vendor name **`<vendor>`**, there is a corresponding subdirectory **`$RTS_HOME/codegen/compiler/<vendor>`**. In each of these directories there are two Perl scripts, **`comp.pl`** and **`link.pl`**. These two files contain a set of regular expressions (**regexps**), along with a handler function pointer for each **regexp**.

Each **regexp** used is a Perl regular expression. If you are not familiar with Perl or regular expressions in general, it is suggested that you obtain a Perl book or find an equivalent reference online. As an example, the two O'Reilly books *Programming Perl* and *Mastering Regular Expressions* are excellent sources of Perl and **regexp** information.

When the code that was generated from the Rational Rose RealTime toolset is compiled, it is done via the main compilation controller script **`$RTS_HOME/codegen/rtcomp.pl`**. This script loads the vendor-specific regular expressions in **`$RTS_HOME/codegen/compiler/<vendor>/comp.pl`** and applies these **regexps** to each line printed by the compiler.

The same procedure is done while linking, but it's done by the main linking controller script **`$RTS_HOME/codegen/rtlink.pl`** which loads the vendor-specific regular expressions in **`$RTS_HOME/codegen/compiler/<vendor>/link.pl`** instead.

How "rtcomp.pl" Integrates With the Compiler

Once issued by the make utility, every compilation command-line is wrapped in a call to a perl script "rtcomp.pl". For example, if working in C++,

C++

```
> rtperl "C:\RoseRT6.2/C++/TargetRTS/codegen/rtcomp.pl" \
   -vendor VisualC++ -spacify dq \
   -I ../src -componentname NewComponent1 \
   -src NewCapsule1 ../src/NewCapsule1.cpp --  \
   cl /c  /FoNewCapsule1.OBJ  /nologo /G5 /GX /GF /MD /TP \
   /I"C:\RoseRT6.2/C++/TargetRTS/libset/x86-VisualC++-6.0"  \
   /I"C:\RoseRT6.2/C++/TargetRTS/target/NT40T" \
   /I"C:\RoseRT6.2/C++/TargetRTS/include" /Zi /I../src \
   ../src/NewCapsule1.cpp
!> Compiling NewCapsule1
NewCapsule1.cpp
../src/NewCapsule1.cpp(25) : error C2065: 'i' : undeclared identifier
GES capsuleClass 'NewCapsule1' transition ':TOP:Initial:Initial' line
'1' description 'C2065: ''i'' : undeclared identifier' severity
'error'
```

The perl script "rtcomp.pl" has the following functions:

- It explicitly provides feedback on the current activity ("`!> Compiling NewCapsule1`")
- If necessary, it creates GES (Generic Error Stream) errors based on incorrect command-line usage (typically these are tagged to the component).
- It runs the compiler, using the command-line arguments following the -- argument. Compiler output is captured for error parsing and conversion to GES.
- Assuming the compilation was successful, the perl script performs compilation dependency analysis and stores the results in local .dep files for future build-avoidance. (This step is skipped when the Compilation Make Type is "ClearCase_clearmake" or "ClearCase_omake".)
- It returns an exit code (back to the **Makefile**) indicating the compilation's success or failure, depending on the existence of any errors.

While parsing the errors, any reference to a source-file line-number is converted into a model element reference by scanning through the offending file to see if the offending line-number is embedded within a pair of RME (Referable Model Element) labels. These RME labels are provided by the code generator for exactly this purpose.

The resulting message is printed out in GES (Generic Error Stream) format, an internal format. GES format must start with "GES" and must contain a description and severity field. Other fields identifying the model element will only be provided if they can be found.

Reusing an Existing Error Parser

If you are porting to a new **libset**, but using an existing compiler vendor, just set the **`VENDOR`** make macro in the **`$RTS_HOME/libset/<libset>/libset.mk`** file to reference the existing vendor, and the error parsing port is done.

Creating a New Error Parser

If you are porting to a new vendor, you will first need to pick a vendor name **`<vendor>`**. Then create the directory **`$RTS_HOME/codegen/compiler/<vendor>`** and the two files **`comp.pl`** and **`link.pl`** in this directory.

Each of the files should contain the following (reading this requires some knowledge of Perl):

- The package identifier: **`package config;`** first in the file.
- An array, **`@handlers`**, where each element is a reference to an array with two elements: the **regexp** matching string, and a reference to the associated handler routine.
- A line saying **`return 1;`** (or just **`1;`**) at the end of the file, to indicate to Perl that this file was loaded and initialized OK.

A typical **`comp.pl`**, for the vendor VisualC++ (Microsoft Visual C++), contains the following:

```
package config;

@handlers =
(
    [ '^(.*)\((\d+)\)\s+:\s+fatal error (.*)',
        sub { rterror::action_print( $1, $2, $3, 0 ); } ],
    [ '^(.*)\((\d+)\)\s+:\s+error (.*)',
        sub { rterror::action_print( $1, $2, $3, 0 ); } ],
    [ '^(.*)\((\d+)\)\s+:\s+warning (.*)',
        sub { rterror::action_print( $1, $2, $3, 1 ); } ],
    [ '(warning.*)',     sub { rterror::action_message( $1, 1 ); } ],
    [ '(fatal error.*)', sub { rterror::action_message( $1, 0 ); } ]
);

return 1;
```

In this example you can see that each of the five elements in the `@handlers` array is a reference to another array with two elements (as indicated by the `[ , ]` notation). The first of these two elements is a string containing the **regexp** we're trying to match, and the second element contains a reference to the handler routine. The **regexps** are written so that they'll save (as indicated by the `( )` notation) the file name, the line number and the descriptive message in the variables `$1`, `$2` and `$3` respectively. These variables are used in the call to the Perl handler routines `rterror::action_print()` and `rterror::action_message()`.

When compiling the generated code (or linking, in which case the script `link.pl` is used), each line printed by the compiler (linker) is matched against the regular expressions in the `@handlers` array, starting with the first (topmost) **regexp**. If there is no match, the next **regexp** below is tried and so on, until there either was a match, or we've come to the end of the `@handlers` array. The default behavior for an unmatched compiler message is to ignore the message.

The following three handler methods can be used inside the `sub { ... }` part:

```
rterror::action_print( $fileName, $lineNr, $msg, $severity );
```

If **fileName** exists, it prints the RME tag from the file, along with line number, message and the severity text (0 for 'error', 1 for 'warning'). If **fileName** wasn't found, it prints the file name, line number, message and severity text.

```
rterror::action_message( $msg, $severity );
```

Prints the message and the severity text, optionally prepended by the component name, if known. This is particularly useful when the error is likely in a component (such as errors during linking, or problems with compiler flags).

```
rterror::action_ignore();
```

Does not take parameters and does nothing.

You will need to figure out what error expressions your compiler and linker generate, and populate the `@handlers` array in `comp.pl` or `link.pl` with appropriate regular expressions. There are a couple of ways to efficiently determine what the errors your compiler generates looks like:

1. Write a model that contains a representative set of compilation errors, compile it, and observe the output for the errors it generates. Add expressions one at a time and recompile until you have successfully captured all the errors.

2. Use programs that search the actual compiler or linker executable for strings. Then manually examine the output and intelligently determine which of the strings look like error statements.

Testing the TargetRTS Port 8

Contents

This chapter is organized as follows:

- *Overview* on page 101
- *HelloWorld Model* on page 101
- *Other Test Models* on page 102
- *Other Resources* on page 102

Overview

A port to a new platform requires testing the TargetRTS. There are some standard Rational Rose RealTime models that are part of the installation and can be used to test the functionality of the TargetRTS. These tests are not comprehensive but provide some assurance that the port was successful.

HelloWorld Model

C++ This model is available in:

```
$ROSERT_HOME/Tutorials/gstarted/QuickstartTutorial.rtmdl
```

The HelloWorld model is a single capsule model that uses the Log service to output “Hello World” to the target console. It makes use of the Log service to output the message. The HelloWorld model, if functional, validates the TargetRTS initialization and startup, log service and console output and basic capsule functionality.

Other Test Models

More test models are available in the online tutorials and examples. Please take a look at **`$ROSERT_HOME/Examples/Models/C++`** or **`$ROSERT_HOME/Examples/Models/C`** and **`$ROSERT_HOME/Tutorials`** for information on what's available.

Other Resources

We suggest that you visit the Rational Rose RealTime product support web site for the latest updates, models and patches. The URL is http://www.rational.com/support/.

Tuning the TargetRTS 9

Contents

This chapter is organized as follows:

- *Disabling TargetRTS Features for Performance* on page 103
- *Target Compiler Optimizations* on page 103
- *Target Operating System Optimizations* on page 104
- *Specific TargetRTS Performance Enhancements* on page 104

Disabling TargetRTS Features for Performance

The TargetRTS can be modified to exclude many of its features to provide a minimum high performance feature set. The section "Configuring and customizing the Services Library" in the *C Reference* or *C++ Reference* describes how to create such a version of the TargetRTS. The concepts of a "minimal TargetRTS" disables Target Observability, logging service and the RTS debugger. The minimal TargetRTS should provide significant performance gains over the fully featured version.

Target Compiler Optimizations

Most compilers provide optimizations at the object code generation stage that can produce faster running code. In general, if your compiler supports such optimizations, they should be used. Be sure to remove all debug options at the same time since they may cancel out certain or all optimizations. Some optimizations may come at the cost of code size. If application code size is a factor for your target then the benefit of optimization versus code size will have to analyzed. Many compilers may have different levels of optimization, which may produce differing degrees of code size and performance enhancements. It is hard to predict the outcome of such optimizations in C or C++. Using a performance testing model which measures the speed of certain operations may prove useful.

Note: Optimizations can cause errors in the running application that were not present before optimizations were enabled. Be sure to fully test the TargetRTS after enabling any optimizations.

Target Operating System Optimizations

The Target operating system may provide optimizations. For example, it may be possible to link in a non-debug version of the OS with the application. These optimizations are specific to each RTOS. Refer to the documentation for your specific RTOS.

Specific TargetRTS Performance Enhancements

In C or C++, one key area that can improve performance in the TargetRTS is in inter-thread message passing. The TargetRTS make use of two synchronization mechanisms for much of its message passing, namely, the **`RTMutex`** and **`RTSyncObject`** classes. Some operating systems provide heavy-weight and light-weight synchronization mechanisms. The light-weight version has less features but higher performance; whereas, the heavy-weight version may have more features but poorer performance. Your choice of implementation for the **`RTMutex`** and **`RTSyncObject`** may affect the performance of inter-thread message passing, so be sure to investigate and determine the lightest-weight mechanism necessary to satisfy the requirements of these classes.

Common Problems and Pitfalls 10

Contents

This chapter is organized as follows:

- *Overview* on page 105
- *Problems and Pitfalls with Target Toolchains* on page 106
- *Problems and Pitfalls with TargetRTS/RTOS Interaction* on page 107
- *Problems and Pitfalls with Target TCP/IP Interfaces* on page 111

Overview

This chapter contains information on common problems and pitfalls that we have encountered with previous ports. The TargetRTS is supported on a number of platforms and has been verified on each of these platforms. In general, the problems and pitfalls encountered are mainly due to RTOS and toolchain differences from those verified in the standard platforms - for a complete list, please see the *Rational Rose RealTime Installation Guide*. Other problems arise from lack of support for certain features required by the TargetRTS and thus require a custom workaround to satisfy the TargetRTS.

The target-specific source is placed in a subdirectory of $RTS_HOME/src/target/<target_base>, where **<target_base>** is defined by **$target_base** variable in the file setup.pl file (see *Creating a Setup Script (setup.pl)* on page 54). The target name often appears with the trailing ‘S’ or ‘T’. The name defaults to the target name without the "S" or "T" if the variable **$target_base** is not defined in the setup.pl file.

Problems and Pitfalls with Target Toolchains

This section describes possible problems with the tools used to build the TargetRTS and the model.

Compiler Optimizations

Compiler optimizations, in general, either help speed up the application, or make the footprint of the executable smaller. Some optimizations can unfortunately cause errors in the application. One such problem occurs when the compiler optimizes references to a memory location that is not modified by the application. It assumes that because the application does not modify the contents of the address, it is never modified. In a multi-threaded environment, some compiler optimizations might not yield the desired result, so be cautious.

Optimizations vary from compiler to compiler, so refer to the documentation for your specific toolchain. Review the optimizations that are available and be aware that some may cause errors in the application. Running a set of test models is a good way to ensure the optimizations have not broken the TargetRTS.

Make sure the test models you use exercise each of the target OS primitives used by the TargetRTS.

Linker Configuration File

When linking an application to a embedded target, there is usually some sort of linker configuration file that defines where in memory each section of the application will go. Many default linker configuration files are included without the user's knowledge and may cause strange linking errors as applications grow larger. Be sure to define your own linker configuration file appropriate for your target.

System Include Files

The structure and content of include files can be a challenge when moving to a new toolchain. In the TargetRTS an attempt is made to isolate the nuances of include files for each RTOS into a few specific include files that can be used by all the target-specific code. In general, all RTOS-specific definitions should be combined into a file called <os_name>.h in the `$TARGET_SRC/RTPriv` directory in the C TargetRTS, RT<os_name>.h in the `$TARGET_SRC` directory in the C++ TargetRTS. This way all include files needed to access OS functions can be found in this one file. In the C TargetRTS, for TCP/IP specific include files, a file called `Tcp.h,` in the C++ TargetRTS, `RTtcp.h` ,should be created in the `$TARGET_SRC/RTPriv` directory (C), or `$TARGET_SRC` directory (C++). This file should contain all the necessary include files required for TCP/IP functions. Other, more specific, header files may be required to isolate unique interfaces for your RTOS. These may be added to the `$TARGET_SRC/RTPriv` or `$TARGET_SRC` directory as needed, and are typically prefixed by "RT" in the C++ version.

Problems and Pitfalls with TargetRTS/RTOS Interaction

This section describes the possible problems between the operating system and the system calls that are part of the TargetRTS.

Return Codes for POSIX Function Calls

Even though POSIX is a standard, there are still some discrepancies in the implementation of the interface. Some implementations of the POSIX function calls return an error code, while others return -1 and store the result in global variable `errno`. Check your specific RTOS to see how error conditions are reported.

Thread Creation

Thread creation has caused problems in the past. One specific problem is the lack of free space on the heap to allocate the stack for the new thread. This causes a system crash with no error message or exception raised. Other potential pitfalls arise with thread priorities. Do not alter the relative priorities of the C TargetRTS or C++ TargetRTS threads (main thread), timer thread and debugger thread). Incorrect priorities may effect the functioning of timers, the debugger or even the Rational Rose RealTime application.

Real-time Clock

C Most RTOSes provide a function to retrieve the current system time. Typically it may return clock ticks, milliseconds or even nanoseconds. In the C TargetRTS, a conversion from the RTOS time to `RTTimespec` is typically required in order to satisfy the requirements of the `RTTimespec_clock_gettime()` function. Some RTOSes may provide a macro or function to resolve the number of ticks per second and thus make conversion to `RTTimespec` straightforward. Others may require hard-coded conversion based on the known tick rate for the RTOS. If this rate is later changed then the conversion will fail. This results in incorrect behavior for all timers in the Rational Rose RealTime model.

Real-time Clock

C++ Most RTOSes provide a function to retrieve the current system time. Typically it may return clock ticks, milliseconds or even nanoseconds. In the C++ TargetRTS, a conversion from the RTOS time to `RTTimespec` is required in order to satisfy the requirements of the `RTTimespec::getclock()` function. Some RTOSes may provide a macro or function to resolve the number of ticks per second and thus make conversion to `RTTimespec` straightforward. Others may require hard-coded conversion based on the known tick rate for the RTOS. If this rate is later changed then the conversion will fail. This results in incorrect behavior for all timers in the Rational Rose RealTime model.

In the C++ TargetRTS, when changing the system clock, note that if the time returned by the `RTTimespec::getclock()` function is affected by changes in the system clock, the function call that adjusts the time must be located between calls to the `Timing::Base` methods `adjustTimeBegin()` and `adjustTimeEnd()`. If, however, system clock changes do not affect the `RTTimespec::getclock()` function, do not use the `Timing::Base` methods `adjustTimeBegin()` and `adjustTimeEnd()`. Timers will fail in this case and cause unwanted behavior in your Rational Rose RealTime application.

For example:

```
void AdjustTimeActor::setclock( constRTTimespec & new_time )
{
    RTTimespec old_time;
    RTTimespec delta;

    timer.adjustTimeBegin();  // stop Rose RealTime timer service

    sys_getclock( old_time ); // an OS-specific function
    sys_setclock( new_time ); // an OS-specific function

    delta  = new_time;
    delta -= old_timer;

    timer.adjustTimeEnd( delta ); // resume Rose RealTime timer
service
}
```

Signal Handlers

Many RTOSs do not use signals that are typical of UNIX operating systems. If your RTOS does not provide signals, be sure to override the C TargetRTS code in

C `RTMain_installHandlers()` and `RTMain_installOneHandler()`.

C++ TargetRTS code in

C++ `RTMain::installHandlers()` and `RTMain::installOneHandler()`.

RTOS Supplies `main()` Function

The TargetRTS assumes that it defines the `main()` function for an application. Some RTOSs may provide their own `main()` function, which causes a duplicate reference error at link time. If this is the case for your RTOS, you have to modify the code in `$TARGET_SRC/MAIN/main.c or $TARGET_SRC/MAIN/main.cc`. Typically, you have to start a thread that contains the `main()` function for the Rational Rose RealTime application. The documentation for the RTOS will describe how to start your application in this manner.

Default Command Line Arguments

Embedded targets do not usually have access to command line arguments, so RTOSs rarely provide a way to pass command line arguments to a running application. If your RTOS does not support command line arguments, you can use the default argument mechanism in the toolset. This feature lets you enter a set of default arguments for each component, and these arguments will appear in the generated code.

These arguments can be specified in the toolset via *Component Specification* > *C Executable* > *DefaultArguments* or *Component Specification* > *C++ Executable* > *DefaultArguments*.

Note: These arguments will appear in the generated code verbatim, so use quotes around, and commas between, your arguments to avoid compilation errors.

You will also have to create a slightly modified `main()` function and put it into `$TARGET_SRC/MAIN/main.c or $TARGET_SRC/MAIN/main.cc`. The modification needed is that instead of calling `RTMain_entryPoint()` or `RTMain::entryPoint()` with the arguments **argc** and **argv**,

C like in this default $RTS_HOME/src/Main/main.c:

```
int main( int argc, const char * const * argv ) /* Standard main */
{
    return RTMain_entryPoint( argc, argv );
}
```

...you should call RTMain_entryPoint() with two null arguments, like this:

```
int main() /* This main takes no arguments */
{
    return RTMain_entryPoint( 0, (const char * const *)0 );
}
```

C++ or, like in this default $RTS_HOME/src/MAIN/main.cc:

```
int main( int argc, const char * const * argv ) // Standard main
{
    return RTMain::entryPoint( argc, argv );
}
```

...you should call RTMain::entryPoint() like this:

```
int main() // This main takes no arguments
{
    return RTMain::entryPoint( 0, (const char * const *)0 );
}
```

This will cause the TargetRTS to use the default arguments instead. Please note that default arguments behave just like "real" command line arguments; the first argument, `RTMain_argv()[0]` or `RTMain::argStrings()[0]` is the name of the program. Your arguments are available in position `[1]` and onwards.

Exiting Application

In the C or C++ TargetRTS, the `RTStdio_panic() or RTDiag::panic()` function requires a way to terminate the application. This is generally achieved by exiting the application. If your RTOS does not support the `exit()` function, you have to override the code in `$TARGET_SRC/Main/exit.c or $TARGET_SRC/RTDiag/panic.cc` to use the exit function specific to your RTOS.

Problems and Pitfalls with Target TCP/IP Interfaces

This section describes the possible problems with OS specific TCP/IP interfaces. Your model can still run without TCP/IP support in the TargetRTS, however Target Observability (for example, observing a running model from the toolset) will be disabled.

gethostbyname() reentrancy

A problem was found on some UNIX targets when trying to use the `gethostbyname()` function in a multi-threaded application. The call was replaced with a call to the `gethostbyname_r()` function, which is re-entrant and thread safe. If this is the case for your target OS, change the code for `RTinet_lookup()` in `$TARGET_SRC/Inet/lookup.c or $TARGET_SRC/RTinet/lookup.cc` in the C or C++ TargetRTS.

select() statement

C Some implementations of the `select()` statement do not correctly use the value set in the width parameter. Consequently the function thinks the file descriptor sets are larger than they really are. This can cause memory corruption and, consequently, serious failures in the running application. To overcome this problem in the C TargetRTS, some targets (OSE) override the `RTIOMonitor_min_size()` function in `$TARGET_SRC/IOMonit/min_size.c.` In these cases, the minimum size is assumed to be the maximum file descriptor set size.

TargetRTS Porting Example

11

Contents

- *Overview* on page 113
- *Choosing the Configuration Name* on page 113
- *Create Setup Script* on page 114
- *Create makefiles* on page 115
- *TargetRTS Configuration Definitions* on page 118
- *Code Changes to TargetRTS Classes* on page 119
- *Building the New TargetRTS* on page 122

Overview

This chapter provides an example of porting the TargetRTS for C or C++ to a new platform. This is an example port rather than customization of an existing port. See the *C Reference* or the *C++ Reference* for a customization example. This porting example should help implement the information presented in previous sections. The target platform for this example is the Tornado 2 real-time operating system using the Cygnus C or C++ Compiler version 2.7.2-960126 for Motorola PowerPC microprocessors. This is a currently supported platform.

Choosing the Configuration Name

The configuration name is an important identifier of the TargetRTS. It identifies the operating system, hardware architecture and (cross) compiler. In this example, the operating system is Tornado 2. The hardware architecture is Motorola PowerPC (ppc). The compiler is the Cygnus C or C++ Compiler version 2.7.2-960126. For this example we will only consider the multi-threaded version of the TargetRTS since this provides the most interesting porting challenges. The resulting configuration name is as follows:

```
<target> = TORNADO2T
<libset> = ppc-cygnus-2.7.2->960126
<config> = <target>.<libset>= TORNADO2T.ppc-cygnus-2.7.2-960126
```

Create Setup Script

The setup script is in the file **$RTS_HOME/config/TORNADO2T.ppc-cygnus-2.7.2-960126/setup.pl**. This file is a Perl script that defines environment variables for the compilation of the TargetRTS:

```
if( $OS_HOME = $ENV{'OS_HOME'} )
{
    $os = $ENV{'OS'} || 'default';

    if( $os eq 'Windows_NT' )
    {
        $wind_base        = $ENV{'WIND_BASE'};
        $wind_host_type   = 'x86-win32';
        $ENV{'PATH'} =
"$wind_base/host/$wind_host_type/bin;$ENV{'PATH'}";
    }
    else
    {
        $rosert_home      = $ENV{'ROSERT_HOME'};
        chomp( $host      = `$rosert_home/bin/machineType` );

        $wind_base        = "$OS_HOME/wrs/tornado-2.0";
        if( $host eq 'sun5' )
        {
            $wind_host_type   = 'sun4-solaris2';
        }
        elsif( $host eq 'hpux10' )
        {
            $wind_host_type   = 'parisc-hpux10';
        }
        $ENV{'PATH'} =
"$wind_base/host/$wind_host_type/bin:$ENV{'PATH'}";
        $ENV{'WIND_BASE'} = "$wind_base";
    }

    $ENV{'GCC_EXEC_PREFIX'}
="$wind_base/host/$wind_host_type/lib/gcc-lib/";
    $ENV{'VXWORKS_HOME'}    = "$wind_base/target";
    $ENV{'VX_BSP_BASE'}     = "$wind_base/target";
    $ENV{'VX_HSP_BASE'}     = "$wind_base/target";
    $ENV{'VX_VW_BASE'}      = "$wind_base/target";
    $ENV{'WIND_HOST_TYPE'}  = "$wind_host_type";
}

$preprocessor = "ccppc -DPRAGMA -E -P >MANIFEST.i";
$target_base  = 'TORNADO1';
$supported    = 'Yes';
```

The setup script must contain the mandatory definitions for the **$preprocessor** and **$supported** flags. The toolchain environment variables are usually required for cross compiler tools, since it is not typically part of a user's command path, and the environment variable definitions are probably not already defined in most users' environments.

Note: The **$target_base** variable is set to **TORNADO1**. This means that the **TORNADO2T** target uses the same code base for the TargetRTS classes as the **TORNADO1** target.

Create makefiles

The next step in porting the TargetRTS is to create various makefiles needed to build the TargetRTS for the platform and to build Rational Rose RealTime models on this new TargetRTS and platform.

Libset makefile

The **libset makefile** is used to make specific definitions for the compiler. The command line interface for C and C++ compilers can differ significantly, particularly for cross-compilers such as the Cygnus C or C++ compiler. It is in this file that we make definitions for command line options for the compiler and linker and override other definitions made in **$RTS_HOME/libset/default.mk.** See *Default makefile* on page 59 for details. In any port of the TargetRTS, there are certain commands required in the toolchain in order to support the building of the TargetRTS. Table 9 illustrates these required commands.

Table 9 Tools Required for Building the TargetRTS for C

Command	GNU CC on Solaris	Cygnus cross-compiler for VxWorks
library archive	$RTS_HOME/tools/ar.pl	$RTS_HOME/tools/ ar.pl -create=arppc,rc
C Compiler	g++ or gcc	ccppc
Linker	g++ or gcc	$RTS_HOME/target/TORNADO2T/link.pl ARCH=ppc
VENDOR	gnu	cygnus

The library archive command (**`ar`**) for the Cygnus toolchain requires the use of a script to work the way the TargetRTS build requires. The **libset makefile** must define the **`VENDOR`** macro that instructs the error parser which type of compiler is being used. The error parser uses this information to decode error messages returned by the compiler to a format compatible with the Rational Rose RealTime toolset.

Another important role of the **libset makefile** is the definition of command line options. Table illustrates the typical subset of command line options.

Table 10 Important Toolchain Command Line Options

Option	GNUcc on Solaris	Cygnus
LIBSETCCFLAGS		-DPRAGMA -ansi -nostdinc -DCPU=PPC603
LIBSETCCEXTRA		-O4 -finline -finline-functions -Wall

The compiler options may vary greatly from one platform to another, but must support some basic features. Read the compiler documentation carefully and review some of the **`libset.mk`** files for other TargetRTS platforms for guidance. A list of required features follows:

- to compile source files into object files only (that is, not to proceed to the link phase), typically the '-c' option
- to place the object file in a desired directory and file name, typically the '-o' option
- to link and place the executable in a desired directory and file name, typically the '-o' option for the link phase
- to turn on debugging information in the compiled code, typically the '-g' option
- to specify the pathname of include files, typically the '-I' option
- to specify the pathname of libraries, typically the '-L' option
- to specify the libraries to link, typically the '-l' (ell) option
- to turn on code optimization, typically '-O' option and sub-options

C The contents of the C version of the **libset makefile**, **$RTS_HOME/libset/ppc-cygnus-2.7.2-960126/libset.mk** , is as follows:

```
AR_CMD        = $(PERL) $(RTS_HOME)/tools/ar.pl -create=arppc,rc
CC            = ccppc
LD            = ldppc
RANLIB        = ranlibppc

VENDOR        = cygnus

LIBSETCCFLAGS = -DPRAGMA -nostdinc -DCPU=PPC603
SHLIBS        =
```

C++ The contents of the C++ version of the **libset makefile**, **$RTS_HOME/libset/ppc-cygnus-2.7.2-960126/libset.mk** is as follows:

```
VENDOR        = cygnus

AR_CMD        = $(PERL) $(RTS_HOME)/tools/ar.pl -create=arppc,rc -
ranlib = ranlibppc
CC            = ccppc
LD            = $(PERL) "$(RTS_HOME)/target/$(TARGET)/link.pl"
ARCH=ppc
RANLIB        = ranlibppc

LIBSETCCFLAGS = -DPRAGMA -ansi -nostdinc -DCPU=PPC603
LIBSETCCEXTRA = -O4 -finline -finline-functions -Wall
SHLIBS        =

ALL_OBJS_LIST = %$(ALL_OBJS_LISTFILE)
```

Target makefile

The target **makefile** is used to make definitions specific to the target operating system and the TargetRTS configuration. These are usually specific command line options for the compiler and linker to define such things as include directories for the target OS and libraries and their *pathnames*. These definitions must be common to all TORNADO2T targets, regardless of **libsets**.

C The contents of the target C **makefile**, **$RTS_HOME/target/TORNADO2T/target.mk**, is as follows:

```
TARGETCCFLAGS = $(DEFINE_TAG)_REENTRANT \
     $(INCLUDE_TAG)$(VXWORKS_HOME)/h -fno-builtin
TARGETLDFLAGS = -r
RTCODEBASE    = TORNADO101
```

C++ The contents of the target C++ **makefile**, **$RTS_HOME/target/TORNADO2T/target.mk**, is as follows:

```
TARGETCCFLAGS = $(INCLUDE_TAG)$(VXWORKS_HOME)/h
```

Configuration makefile

The configuration **makefile** is used to make definitions required by the operating system and compilation environment together. In this particular case, the configuration **makefile**, $RTS_HOME/config/TORNADO2T.ppc-cygnus-2.7.2-960126/config.mk, is empty because there is no need for any definitions specific to the compiler and operating system combination.

TargetRTS Configuration Definitions

The default configuration definitions for the TargetRTS are found in the include file $RTS_HOME/include/RTConfig.h. The definitions in this file can be overridden by $RTS_HOME/target/TORNADO2T/RTTarget.h and possibly $RTS_HOME/libset/ppc-cygnus-2.7.2-960126/RTLibSet.h.

These definitions are used to enable and disable various features in the TargetRTS. By default almost all of the TargetRTS features are enabled (for example, Target Observability). The porting effort may be made easier if some of these features are disabled. See section "TargetRTS Customization Example" in the *C++ Reference* for instructions on how to build a minimal TargetRTS.

C The content of the C version of the file **$RTS_HOME/target/TORNADO2T/RTTarget.h** is as follows:

```
#ifndef __RTTarget_h__
#define __RTTarget_h__ included

#define USE_THREADS 1

#define DEFAULT_DEBUG_PRIORITY 60
#define DEFAULT_MAIN_PRIORITY  75
#define DEFAULT_TIMER_PRIORITY 70

#endif /* __RTTarget_h__ */
```

C++ The content of the C++ version of the file **$RTS_HOME/target/VRTX4T/RTTarget.h** is as follows:

```
#ifndef __RTTarget_h__
#define __RTTarget_h__ included

#define TARGET_TORNADO 1
```

```
#define USE_THREADS        1
#define PERFORM_CTOR_DTOR  0

#define DEFAULT_DEBUG_PRIORITY  60
#define DEFAULT_MAIN_PRIORITY   75
#define DEFAULT_TIMER_PRIORITY  70

#endif // __RTTarget_h__
```

There is no need for the file $RTS_HOME/libset/ppc-cygnus-2.7.2-960126/RTLibSet.h since no compiler-specific compile-time features need to be modified.

`RTnew.h` may be necessary in **`libset/-`** if **`<new>`** is not available.

$RTS_HOME/libset/ppc-cygnus-2.7.2-960126/RTRTnew.h is as follows:

```
#include <new.h>
```

Code Changes to TargetRTS Classes

Most ports to new targets require some minor changes to the TargetRTS code. These changes typically apply to operating system features for thread (task) creation and destruction, mutual exclusion and synchronization and time services. Table 6 on page 75 and Table 8 on page 89give a description of TargetRTS classes that might require changes.

The required changes to the TargetRTS source for TORNADO2 and the Cygnus compiler are, for C++, located in the $RTS_HOME/src/target/TORNADO1 directory. See the discussion for the setup script above for an explanation of why the directory is called **`TORNADO101`** for C, rather than **`TORNADO2`**. For the remainder of this section, this directory is referred to as **`$TARGET_SRC`**.

The files in the **`$TARGET_SRC`** directory each override their counterpart in **`$RTS_HOME/src`**. To override a definition from the source directory, a new subdirectory should be created in **`$TARGET_SRC`**.

C For example, for C, the new definition for **`RTTimespec_clock_gettime()`** requires a subdirectory $TARGET_SRC/Timespec. The new file containing **`RTTimespec_clock_gettime()`** would be $TARGET_SRC/Timespec/getclock.c.

The required changes to the TargetRTS are too large to include in this document. Table 11 and Table 12 contain a summary of the required changes to each file.

Table 11 Quick Summary of Common C TargetRTS Source File Changes

Class	File	Change
RTInet (dir Inet)	async.c	Modified version since FIOASYNC was not defined.
RTInet (dir Inet)	lookup.c	**`gethostbyname`** not available, use **`hostGetByName`** instead
main (dir Main)	main.c	**`main`** already defined by RTOS, use **`rtsMain`** with nonstandard argument handling instead.
RTMutex (dir Mutex) (required)	ct.c dt.c enter.c leave.c	**Required implementation** using Tornado specific calls to **`semMCreate`**, **`semDelete`**, **`semTake`** and **`semGive`**.
RTSyncObject (dir SyncObj) (required)	ct.c dt.c signal.c wait.c timewait.c	**Required implementation** using Tornado specific calls to **`semBCreate`**, **`semDelete`**, **`semGive`** and **`semTake`**.
RTThread (dir Thread) (required)	ct.c	**Required implementation** using Tornado specific calls to **`taskSpawn`** and **`taskDelete`**.
RTTimespec (dir Timespec) (required)	getclock.c	**Required implementation** using Tornado specific call to **`clock_gettime`**.

C++ For example, for C++, the new definition for **`RTTimespec::getclock()`** requires a subdirectory $TARGET_SRC/RTTimespec. The new file containing **`RTTimespec::getclock()`** would be $TARGET_SRC/RTTimespec/getclock.cc.

The required changes to the TargetRTS are too large to include in this document. Table 12 contains a summary of the required changes to each file.

Table 12 Quick Summary of Common C++ TargetRTS Source File Changes

Class	File	Change
MAIN	main.cc	**`main`** already defined by RTOS, use **`rtsMain`** with nonstandard argument handling instead.
RTDiag	panic.cc	Modified version since there is no **`exit()`** method
RTMain	targetStartup.cc	Modify main thread priority to that specified in the toolset
RTMutex (required)	ct.cc dt.cc enter.cc leave.cc	**Required implementation** using Tornado specific calls to **`semMCreate`**, **`semDelete`**, **`semTake`** and **`semGive`**.
RTSyncObject (required)	ct.cc dt.cc signal.cc timedwait.cc wait.cc	**Required implementation** using Tornado specific calls to **`semBCreate`**, **`semDelete`**, **`semGive`** and **`semTake`**.
RTThread (required)	ct.cc	**Required implementation** using Tornado specific calls to **`taskSpawn`** and **`taskSuspend`**, etc.
RTTimespec (required)	getclock.cc	**Required implementation** using Tornado specific call to **`clock_gettime`**.
RTinet	lookup.cc	Modified version, uses **`hostGetByName`** instead of **`gethostbyname`**.

Building the New TargetRTS

After the setup script, makefiles, and source are complete, the TargetRTS is ready to be built. To build the TargetRTS for the Tornado 2 Cygnus target, type the following in the **$RTS_HOME/src** directory:

```
make TORNADO2T.ppc-cygnus-2.7.2-960126
```

This will create the directory $RTS_HOME/build-TORNADO2T.ppc-cygnus-2.7.2-960126 which will contain the dependency file and object files for the TargetRTS. If the build completes successfully the resulting Rational Rose RealTime libraries will be placed in the $RTS_HOME/lib/TORNADO2T.ppc-cygnus-2.7.2-960126 directory.

Customizing for Target Control and Observability

12

Contents

This chapter isorganized as follows:

- *Introduction* on page 123
- *Model Compilation and Target Control* on page 124
- *Target Control* on page 125
- *Menu Commands* on page 127
- *Third-Party Source Code Debugger Integration* on page 133

Introduction

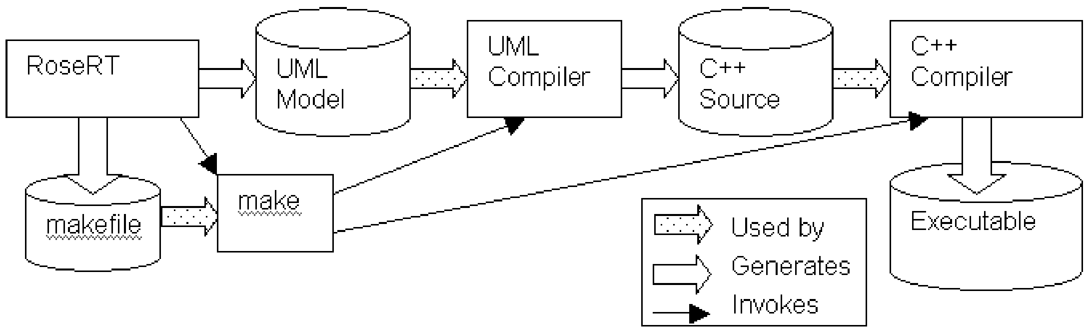
Rational Rose RealTime is a comprehensive visual modeling environment that delivers a powerful combination of notation, processes, and tools optimized to meet the challenges of real-time software development. The Rational Rose RealTime UML model compiler converts models directly into executable applications. Those executables can be controlled and debugged at run-time under the control of the toolset. Rational Rose RealTime integrates with source debuggers providing the developer with the choice of debugging at the UML and source code level. A combination of UML editors, a model compiler, and run-time debugging tools address the complete life-cycle of a project from early use case analysis through design, implementation, and testing.

This document describes how to add support to Rational Rose RealTime 6.0 and later for target control and observability, and how to integrate Rational Rose RealTime with source code debuggers.

Model Compilation and Target Control

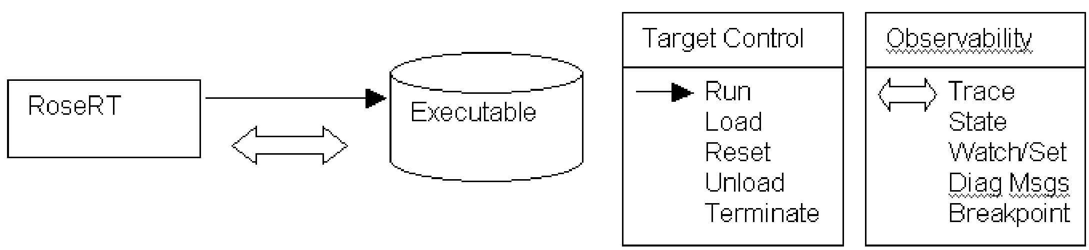
Rational Rose RealTime models are compiled seamlessly into applications ready for execution on the host or target operating systems. Figure 14 provides a high level overview of model compilation.

Figure 14 UML Model Compilation



Rational Rose RealTime also has the ability to control the executing application at run-time (for example, during debugging). Target Observability provides the ability to observe and debug the executing application at the UML level. Figure 15 shows a simplified high-level overview of Target Control and Observability.

Figure 15 Target Control and Observability



Rational Rose RealTime also supports inter-working with traditional source code debuggers. This enables developers to control, observe, and debug the application at the UML level and detailed source code level simultaneously.

Intended Audience

This guide is specifically designed for technical staff responsible for enabling these capabilities for a specific target execution environment. It is assumed that the reader has significant knowledge and experience with the development environment, operating system, and target hardware.

Target Control

Target Control refers to the Rational Rose RealTime toolset features that load, unload, execute, and terminate a Rational Rose RealTime-generated application, as well as the ability to reset a remote target platform.

Target Control is not the same feature as Target Observability. Target Observability allows the observation of the application executing on a target from the UML level (such as state change, state machine breakpoints, event tracing, and so on) on the host-based toolset. Target Control interacts with the APIs of the target execution environment to load, run, and terminate the application, whereas Target Observability communicates directly with the running application.

Target Control Modes

Rational Rose RealTime supports three different Target Control modes:

- Manual Mode
- Basic Mode
- Debugger Mode

Manual Mode

In **Manual mode,** Rational Rose RealTime does not provide any Target Control functionality. The user is responsible for performing Target Control operations (such as loading and executing). After the target application starts, the user can direct the Rational Rose RealTime toolset to connect to the executing target application for Target Observability.

Basic Mode

In **Basic mode**, Rational Rose RealTime uses the target environment's APIs to control the execution of the target application. Rational Rose RealTime supports automatic target control for a number of host and target platform combinations. Users deploy on a number of other target environments as well.

Rational Rose RealTime uses Perl scripts to perform the Target Control operations. These scripts can call the target APIs directly or can call some intermediary helper application to control the execution on the target.

There are five Target Control scripts:

- reset.pl
- load.pl
- unload.pl
- execute.pl
- terminate.pl

Debugger Mode

Debugger mode provides the same capabilities as Basic mode and, in addition, provides the ability to inter-work with a C or C++ source debugger (for example, Visual C++) to set source code level breakpoints from within the UML model. When these source breakpoints are hit at run-time, control of the executable is passed to the source debugger. When the application is continued, control of the executable is passed back to the Rational Rose RealTime toolset. Debugger mode provides an integrated debug environment that permits a simultaneous use of source code and UML debugging styles.

Target Control Scripts

When you open the Specification dialog for a **Processor** in the **Deployment View**, the **Load Scripts** text box specifies the path to the Target Control scripts (for example, $TARGET_PATH/win32/, $TARGET_PATH/tornado2/). This directory contains a maximum of five Target Control scripts, each of which has a different function:

- **reset.pl** - Resets the target processor. See Reset.
- **load.pl** - Loads a Component onto a target. See Load.
- **unload.pl** - Unloads a Component from a target. See Unload.
- **execute.pl** - Executes a Component. See Execute.
- **terminate.pl** - Terminates the execution of a Component. See Terminate

The Target Control Scripts determine the Target Control capabilities for the Processor. If a script exists in the Target Control Scripts directory, then the toolset assumes that the corresponding capability exists. Whenever a Component Instance is created on a Processor (that is, a Component in the **Component View** is assigned to a Processor in the **Deployment View**), the toolset checks to see which scripts are available and enables those capabilities in the toolset menus that are accessible by right-clicking on a Component Instance. These menu options are now available to the user.

The presence of the scripts is not their only purpose. Each existing Target Control script must also provide the associated capability. For example, the load script must load the corresponding component onto the target specified by the Processor, and so on. The scripts use information from the Processor and Component Instances specifications, but note that the scripts do not need to use all the parameters that are passed to them. Any script needs to process only those arguments that allow it to perform its intended operation.

These scripts are written in Perl, but they may spawn other executables needed to provide the desired capability. Every script also indicates whether it was successful.

Menu Commands

If the path to the Target Control scripts contains the following scripts, that corresponding menu command will become active on the **Processor** menu:

- **reset.pl** - Resets the target processor and activates the **Reset** command.
- **load.pl** - Loads a Component onto a target and activates the **Load** command.
- **unload.pl** - Unloads a Component from a target and activates the **Unload** command.
- **execute.pl** - Executes a Component and activates the **Run** menu option (**Execute**).
- **terminate.pl** - Terminates the execution of a Component and activates the **Shutdown** menu option (**Terminate**).

Reset

Description

The reset.pl script resets a target processor. If this script exists, the **Reset** command will be active on the corresponding Processor menu.

Command Line

Rtperl reset.pl –ip *target* **–server** *targetServer* **–os** *targetOS* **–cpu** *targetCPU*

Arguments

-ip ***target***	Target name or address
-server ***targetServer***	Target server name or address
-os ***OS***	OS executing on target
-cpu ***CPU***	CPU on the target

Returns

::Ok::	String indicating success
Error String	Error string to be displayed in error message box in the toolset

Note: The data for the script arguments are retrieved from the **Processor Specification** dialog.

Load

Description

The load.pl script loads a component onto the corresponding target processor. If this script exists, the **Load** command is available on the corresponding Component Instance menu when the Component Instance is in a "loadable" state.

Command Line

Rtperl load.pl –ip *target* **–server** *targetServer* **–os** *targetOS* **–cpu** *targetCPU*

-exe *componentDir* –**prio** *priority* –**port** *Toport*

Arguments

-ip ***target***	Target name or address
-server ***targetServer***	Target server name or address
-os ***OS***	OS executing on target
-cpu ***CPU***	CPU on the target
-exe ***executable***	6.1 and later: Fully qualified executable name
-prio ***priority***	Priority to run the component instance
-port ***Toport***	Target Observability port

Returns

::Ok:: [-warning 'xxx'] [-passback xxx]	6.1 and later: String indicating success. Now two option parameters may follow the **::Ok:: string: -warning** and -**passback**. See General Issues.
Error String	Error string to be displayed in error message box in the toolset

Note: The data for the options are retrieved from the **Processor** and **Component Instance Specification** dialog.

Unload

Description

The unload.pl script removes a component from the corresponding target processor. If this script exists, the **Unload** command is available on the corresponding Component Instance menu when the Component Instance is in an "unloadable" state.

Command Line

Rtperl unload.pl –ip *target* **–server** *targetServer* **–os** *targetOS* **–cpu** *targetCPU*

-exe *componentDir* –**prio** *priority* –**port** *TOport* **paramsFromLoad**

Arguments

-ip ***target***	Target name or address
-server ***targetServer***	Target server name or address
-os ***OS***	OS executing on target
-cpu ***CPU***	CPU on the target
-exe ***executable***	6.1 and later: Fully qualified executable name
-prio ***priority***	Priority to run the component instance
-port ***Toport***	Target Observability port
ParamsFromLoad	Any parameters that were returned from a successful **Load** operation.

Returns

::Ok:: [-warning 'xxx']	6.1 and later: String indicating success. Now, one option parameter may follow **::Ok:: string: -warning**. See General Issues
Error String	Error string to be displayed in error message box in the toolset

Note: The data for the options are retrieved from the **Processor** and **Component Instance Specification** dialog.

Execute

Description

The execute.pl script starts execution of a component instance on the corresponding target processor. If this script exists, the **Run** command is available on the Component Instance menu when the Component Instance is in a "runable" state.

Command Line

Rtperl execute.pl –ip *target* **–server** *targetServer* **–os** *targetOS* **–cpu** *targetCPU*

-exe *componentDir* –**prio** *priority* –**port** *Toport*

-args *commandLineArgs*

Arguments

-ip ***target***	Target name or address
-server ***targetServer***	Target server name or address
-os ***OS***	OS executing on target
-cpu ***CPU***	CPU on the target
-exe ***componentDir***	6.0.x: Path to Component directory. It is used to locate the component
-exe ***executable***	6.1 and later: Fully qualified executable name

-prio *priority*	Priority to run the component instance
-port *Toport*	Target Observability port
-args *commandLineArgs*	Command Line arguments that are to be used when starting the target application. Parameters that follow the **-args** tag are all passed to the target application

Returns

:Ok:: paramsFromExecute	String indicating success. Any strings passed back after the **::Ok::** will be based to the terminate.pl script when the user invokes the *Shutdown* command
::Ok:: [-warning 'xxx'] [-passback xxx]	6.1 and later: String that represents the operation was successful. Now two option parameters may follow the **::Ok::** string: -**warning** and -**passback**. See General Issues
Error String	Error string to be displayed in error message box in the toolset

Note: The data for the options are retrieved from the **Processor** and **Component Instance Specification** dialog.

An example of **paramsFromExecute** is a handle that identifies the process that was created. For example, on Windows we return **–pid nnnnnn.** This allows us to pass back the PID (Process ID) to the Terminate script.

Terminate

Description

The terminate.pl script is used to kill a component instance on the corresponding target processor. If this script exists, the **Shutdown** command is available on the corresponding Component Instance menu when the Component Instance is in a "killable" state.

Command Line

Rtperl terminate.pl –ip *target* **–server** *targetServer* **–os** *targetOS* **–cpu** *targetCPU*

-exe *componentDir* –**prio** *priority* –**port** *TOport*
paramsFromExecute

Arguments

-ip ***target***	Target name or address
-server ***targetServer***	Target server name or address
-os ***OS***	OS executing on target
-cpu ***CPU***	CPU on the target
-exe ***executable***	6.1 and later: Fully qualified executable name
-prio ***priority***	Priority to run the component instance
-port ***Toport***	Target Observability port
ParamsFromExecute	Any parameters that were returned from a successful **Run** operation

Returns

::Ok:: [-warning 'xxx']	6.1 and later: String indicating success. Now optional parameter may follow the ::**Ok:: string: -warning**. See General Issues
Error String	Error string to be displayed in error message box in the toolset

Note: The data for the options are retrieved from the **Processor** and **Component Instance Specification** dialog.

General Issues

- In releases **6.0.x**, the **–exe** option is followed by the Component Directory. The **Load** and **Execute** scripts call a Perl script (findexe.pl) to find the corresponding executable.
- In releases **6.1** and later, the **–exe** option is followed by the fully qualified executable name.
- Release **6.1** formalized what comes after the **::Ok::** string. The **Load**, **Unload**, **Execute**, and **Terminate** can succeed (in other words, return **::Ok::**) but may return a warning. The warning is identified by the parameter **–warning** followed by a string enclosed in single quotes ('). The toolset will display a dialog box specifying that a warning occurred. The string returned in quotes is appended to the toolset logs. Anything appearing after the **-passback** parameter will be returned to the originating call.

Third-Party Source Code Debugger Integration

The format for the Debugger Mode is **Debugger-X** where **X** is the name of the debugger DLL. This DLL must exist in the $ROSERT_HOME/bin/$ROSERT_HOST directory and is called **libX.dll**.

Registering Threads on UNIX

When building a debugger integration DLL without MainWin and using **callback** functions, additional steps are required to ensure that Rational Rose RealTime knows about the **callback** thread. The following steps are necessary for a thread-safe interface:

- Call **tcThreadInit()** from the **callback** thread before doing any callbacks.
- The **callback** thread must call **tcThreadCleanup()** before terminating.

There is a header file for this service in $ROSERT_HOME/bin/tc/tcsetup.h and a supporting dynamic library (for Solaris) in $ROSERT_HOME/bin/tc/sun5/libtcsetup.so.

You may call **tcThreadInit** (init) and **tcThreadCleanup** (cleanup) any number of times, as long as the **tcThreadInit** is always followed by a **tcThreadCleanup** before the next **init** occurs. This is useful if you wanted to do a similar function to the following: **tcThreadInit**, **callback**, **tcThreadCleanup**, for each **callback** instead of **tcThreadInit** at thread startup, and **tcThreadCleanup** at thread termination. However, we recommend that the **tcThreadInit** and **tcThreadCleanup** fuctions be called only once (**tcThreadInit** at startup and **cleanup** at termination) since this approach is less error prone.

Calling Sequence

Source code debuggers come with a variety of capabilities. For the toolset to use the debugger DLL in the best possible way, the DLL must provide a list of its capabilities. The following are capabilities of the debugger DLL that are available to Rational Rose RealTime:

Capability	Description
Function Breakpoints	The DLL uses the function name to set a breakpoint.
Line Breakpoints	The DLL uses a file name and line number to set a breakpoint.
Detects Breakpoint Hits	The DLL calls the callback function when a breakpoint is hit.

Capability	Description
User Termination Detected	The DLL calls the callback function when it detects that the user terminated the debugger manually.
Debugger Loads Target	The DLL must be called to load the target. If not, the toolset uses the Basic mode mechanism, if one exists.
Debugger Unloads Target	The DLL must be called to unload the target. If not, the toolset uses the Basic mode mechanism, if one exists.
Debugger Executes Component	The DLL must be called to start the Component Instance. If not, the toolset uses the Basic mode mechanism, if one exists.
Debugger Terminates Component Instance	The DLL must be called to terminate a component instance. If not, the toolset will use the Basic mode mechanism, if one exists.
Supports Search Paths	The DLL can use a given search path to search for source code.
Reload Before Restarting	The target must be reloaded before it is restarted.

The values of these flags determine how and which debugger DLL functions are called. The rules of operation are:

- The debugger DLL is loaded after the user applies the change to the Operation Mode in the Component Instance specification for the Component Instance. The debugger DLL is loaded only once per toolset session.
- If the DLL is loaded successfully, the toolset obtains the debugger DLLs capabilities and saves them.
- The toolset calls the **tcCreateDebugSession** function to create a new session.

  Note: A new session is created for each Component Instance that uses the debugger DLL.
- The Target Control capabilities (**Load**, **Unload**, **Run**, **Shutdown**) are determined using the debugger DLL capabilities as well as the Target Control scripts. The debugger DLL capabilities take precedence over the Target Control scripts.
- If a target must be loaded, it can be loaded in one of two ways: using the debugger or the Basic mode Target Control script. If the "Debugger Loads Target" flag is set, the debugger DLL is expected to load the target in the **tcInitializeDebugger** function. Otherwise, the Target Control load script is used to load the target, and then the **tcInitializeDebugger** function is called.
- If the target is not loadable, then the **tcInitializeDebugger** function is called when the user invokes the **Run** command.

- If the "Debugger Executes Component" flag is set, then the **tcStartDebugger** function is called. If not set, the Target Control execute script is called and then followed by a call to the **tcStartDebugger** function.

  Note: **Note**: The breakpoint functions may be called before the **tcStartDebugger** function if breakpoints were set in the previous debug session.

- When the user invokes the **Shutdown** command, all breakpoints are removed, and the **tcStopDebugger** function is called. If the "Debugger Terminates Component Instance" is set, the **tcStopDebugger** must terminate the Component Instance. If not set, then the Target Control terminate script is called. If the target does not need to be unloaded, then the **tcCleanupDebugger** function is also called.
- When the user invokes the **Unload** command and the "Debugger Unloads Component" flag is set, the **tcCleanupDebugger** function is called. This function must unload the component from the target. If not set, the Target Control unload script is called.
- When the Debugger DLL is unloaded from the toolset (that is, when the Component Instance Operation mode is changed or when the toolset is shut down) **tcDestroyDebugSession** is called. This function is responsible for releasing any resources associated with this debugger DLL session.

Debugger DLL API

This section describes the API that must be implemented by a debugger DLL. The file, tcdllinterface.h, contains all the required type declarations and function prototypes. The functions are:

- Get DLL Capabilities
- Create Debug Session
- Destroy Debug Session
- Initialize Debugger
- Cleanup Debugger
- Start Debugger
- Stop Debugger
- Set Callback
- Event Callback Function
- Set Source Search Path
- Set Breakpoint in File
- Set Breakpoint At Function
- Clear Breakpoint
- Set DllTrace

Note: Several functions have parameters of type **TC_TCHAR**. This type corresponds to **TCHAR** type familiar to Windows developers. It is either a regular character (**char**) or a wide character (**wchar_t**). By default, **TC_TCHAR** is type defined to **char** in the file tcdllinterface.h.

Get DLL Capabilities

```
TCRET
tcGetDllCapabilities(
                TCDLLCAPS * pCaps/* Pointer to struct to get the
capabilites */
) ;
```

Description

This function populates in the given capability structure with the capabilities of the corresponding DLL. This is the first function that is called in the debugger DLL.

Arguments

TCDLLCAPS * pCaps	Structure to receive the DLL capabilities

Returns

TC_OK	Operation was successful
TC_FAILED	Operation failed. Missing capability structure.

Create Debug Session

```
TCHANDLE
tcCreateDebugSession(
                    const TC_TCHAR * szServerName, /* Name of Target
Server */
                    const TC_TCHAR * szTargetName, /* Name of Target*/
                    const TC_TCHAR * szArchitecture,/* Processor
Architecture */
                    const TC_TCHAR * szOS,          /* Operating System
*/
                    TCDEBUGFLAG      eFlag           /*
Enables/disables Tracing*/
) ;
```

Description

This function is called to create a debug session. It is called after the debugger DLL is loaded. It returns a DLL-specific handle that represents the newly created session. This handle is passed back to all other calls except the **tcGetDllCapabilities**. Typically, the handle is a pointer to a DLL-specific structure that maintains session-specific information.

Arguments

const TC_TCHAR * szServerName	Name or address of a Target Server
const TC_TCHAR * szTargetName	Name or address of the target
const TC_TCHAR * szArchitecture	Type of CPU on the target
const TC_TCHAR * szOS	OS running on the target
TCDEBUGFLAG eFlag	Enables/Disables Debug output from the DLL. See Note below.

Returns

TCHANDLE	DLL-specific handle identifying the newly created session.
(TCHANDLE)0	Unable to create a session.

Note: Currently, the toolset does not provide any means to set or clear the debug flag.

Destroy Debug Session

```
TCRET
tcDestroyDebugSession(
                TCHANDLE    hSession  /* Session to terminate */
) ;
```

Description

This function is called before the Debugger DLL is unloaded. It must release all session-specific resources that were allocated during the session.

Arguments

TCHANDLE hSession	A handle identifying a particular debug session

Returns

TC_OK	Operation was successful
TC_FAILED	Operation failed

Initialize Debugger

```
TCRET
tcInitializeDebugger(
                TCHANDLE       hSession,   /* Debugger Session */
                const TC_TCHAR * szComponent/* Location/name of the
component */
) ;
```

Description

This function is called to identify the component that the debugger is to work with. In some environments, this function will load the component onto the target.

Arguments

TCHANDLE hSession	A handle identifying a particular debug session
const TC_TCHAR * szComponent	The fully qualified name of the component

Returns

TC_OK	Operation was successful
TC_FAILED	Operation failed

Cleanup Debugger

```
TCRET
tcCleanupDebugger(
            TCHANDLE        hSession /* Debugger Session */
) ;
```

Description

This function is called to undo the activities of the tcInitializeDebugger function. In some environments, this function will unload the component from the target.

Arguments

TCHANDLE hSession	A handle identifying a particular debug session

Returns

TC_OK	Operation was successful
TC_FAILED	Operation failed

Start Debugger

```
TCRET
tcStartDebugger(
            TCHANDLE           hSession,/* Debugger Session */
            const TC_TCHAR * pszArgs,/* Command line arguments for
comp */
            int              nPriority   /* start up priority */
) ;
```

Description

This function is called to start the Component Instance. If the debugger does not start the Component instance, this is the point where the debugger should attach to it.

Arguments

TCHANDLE hSession	A handle identifying a particular debug session
const TC_TCHAR * pszArgs,	Command-line arguments for the Component Instance
int nPriority	Priority to run the application

Returns

TC_OK	Operation was successful
TC_FAILED	Operation failed

Stop Debugger

```
TCRET
tcStopDebugger(
            TCHANDLE        hSession/* Loader.Debugger Session */
) ;
```

Description

This function is called to terminate the Component Instance. If the debugger does not terminate the Component instance, this is the point where the debugger should detach from it.

Arguments

TCHANDLE hSession	A handle identifying a particular debug session

Returns

TC_OK	Operation was successful
TC_FAILED	Operation failed

Set Callback

```
TCRET
tcSetCallback(
            TCHANDLE      hSession,      /* Debugger Session */
            CALLBACKFNC pfncCallback,/* function to call on event */
            USERDEFINED lUserDefined1,/* toolset defined data */
            USERDEFINED lUserDefined2 /* toolset defined data */
) ;
```

Description

This function is called during the Target Observability session if the debugger DLL can detect breakpoint hits or user termination. It is used to set or clear a Toolset defined function.

Arguments

TCHANDLE hSession	A handle identifying a particular debug session
CALLBACKFNC pfncCallback,	Pointer to function the debugger DLL is to call when a breakpoint hit or user termination is detected
USERDEFINED lUserDefined1	Toolset information that must be passed back in the callback function
USERDEFINED lUserDefined2	Toolset information that must be passed back in the callback function

Returns

TC_OK	Operation was successful
TC_FAILED	Operation failed

Event Callback Function

```
void
fncCallback(
                  TCDLLEVENT* pEvent, /* identifies what event
occurred */
                  USERDEFINED  data1, /* data from SetCallback */
                  USERDEFINED  data2 /* data from SetCallback */
) ;
```

Description

This is the prototype of the callback function that is to be called by the debugger DLL when a breakpoint hit or user termination is detected.

Arguments

TCDLLEVENT * pEvent	Identifies the type of event the Debugger DLL is notifying the toolset of.
USERDEFINED lUserDefined1	Toolset information from the last tcSetCallback.
USERDEFINED lUserDefined2	Toolset information from the last tcSetCallback.

Returns

void	Nothing

Set Source Search Path

```
TCRET
tcSetSearchPath(
            TCHANDLE            hSession, /* Debugger Session */
            int                 nEntries, /* number of paths */
           const TC_TCHAR ** ppszSearchPaths/* list of search paths */
) ;
```

Description

This function is called by the toolset to specify the directories that contain the generated source code.

Arguments

TCHANDLE hSession	A handle identifying a particular debug session
int nEntries	The number of paths specified in the next parameter
const TC_TCHAR ** ppszSearchPaths	A list of search paths

Returns

TC_OK	Operation was successful
TC_FAILED	Operation failed

Set Breakpoint in File

```
unsigned long
tcSetBreakpointInFile(
            TCHANDLE           hSession, /* Debugger Session */
            const TC_TCHAR * szFileName,/* File to set breakpoint in
*/
            int              nLineNo  /* line number in file */
) ;
```

Description

This function is called when a breakpoint is required and the Debugger DLL supports breakpoints using file name and line number. This function may be called before the **tcStartDebugger**.

Arguments

TCHANDLE hSession	A handle identifying a particular debug session
const TC_TCHAR * szFileName	Name of file where you want to set the breakpoint
int nLine	The line number in the file where the breakpoint is to be set

Returns

unsigned long	A number uniquely identifying the corresponding breakpoint
0	Unable to set the breakpoint

Set Breakpoint At Function

```
unsigned long
tcSetBreakpointAtFnc(
            TCHANDLE            hSession,      /* Debugger Session */
            const TC_TCHAR    * szFunctionName/* fully qualified name
*/
) ;
```

Description

This function is called when a breakpoint is required and the Debugger DLL supports breakpoints using function names. This function may be called before the **tcStartDebugger.**

Arguments

TCHANDLE hSession	A handle identifying a particular debug session
const TC_TCHAR * szFunctionName	The fully qualified name of the function

Returns

unsigned long	A number uniquely identifying the corresponding breakpoint
0	Unable to set the breakpoint

Clear Breakpoint

```
TCRET
tcClearBreakpoint(
            TCHANDLE          hSession,       /* Debugger Session */
            unsigned long    nBreakpointId   /* breakpoint to remove
*/
) ;
```

Description

This function removes the specified breakpoint for the given session.

Arguments

TCHANDLE hSession	A handle identifying a particular debug session
unsigned long nBreakpointId	Identifier of the breakpoint to remove. Returned by a set breakpoint function

Returns

TC_OK	Operation was successful
TC_FAILED	Unable to remove the breakpoint

Set DllTrace

```
void
tcSetDllTrace(
            TCHANDLE        hSession,/* Debugger Session */
            TCDEBUGFLAG     eFlag   /* enables/disables trace output
*/
) ;
```

Description

This function enables or disables the Debugger DLL output for the given session.

Arguments

TCHANDLE hSession	A handle identifying a particular debug session
TCDEBUGFLAG eFlag	Specifies whether to enable or disable output

Returns

TC_OK	Operation was successful
TC_FAILED	Operation failed.

Note: This function is not currently used by the toolset, but it must exist. If this function is omitted from the debugger DLL, the toolset will not load the DLL successfully.

Index

A

adapting for target environments 41
adding
    new files to TargetRTS 80, 93
AR_CMD 31
arguments 78, 92
audience for target control 124

B

basic mode 125
build 52
    configurations 32
Build flat 34
building
    applications with no OS 52
    debugger integration DLL without MainWin 133
    the new TargetRTS 122

C

callback
    functions 133
    thread 133
calling sequence 133
Class RTMain 92
Class RTMutex 90
Class RTTcpSocket 93
Classes
    RTCondVar
        extending the Mutex 77, 90
    RTDiagStream 78, 91
    RTIOMonitor 80, 93
    RTMain 78, 92
        target-specific methods 78, 92
    RTMutex 76, 90
        protecting shared resources 76, 90
    RTSyncObject 77, 90
    RTTcpSocket 79, 93
Cleanup Debugger 139
Clear Breakpoint 145
ClearCase clearmake 33
ClearCase omake 34
clearmake 33
code changes to TargetRTS classes 119
command line arguments 109
commands
    Execute 130
    Load 128
    Reset 127
    Terminate 131
    Unload 129
common overrides required for a new target 76, 89
Compiler (CC flag) 31
compiler optimizations 106
Config makefile 65
Configuration 118
configuration
    building 32
    deleting 34
    duplicating 21
    editing 25, 31
    managing 21
    types 22
configuration makefile 118
configuration name, choosing a 113
configuring
    NoRTOS configuration 53
Constructor RTThread
    RTThread() 90
contacting Rational customer support xv
COUNT 85
Create Debug Session 137
creating
    ports between C and C++ 35

D

Debugger 73, 85
debugger DLL API 135
- Cleanup Debugger 139
- Clear Breakpoint 145
- Create Debug Session 137
- Destroy Debug Session 138
- Event Callback Function 142
- Get DLL Capabilities 136
- Initialize Debugger 138
- Set Breakpoint At Function 144
- Set Breakpoint in File 143
- Set Callback 141
- Set DllTrace 146
- Set Source Search Path 143
- Start Debugger 140
- Stop Debugger 140

debugger integration DLL 133
debugger mode 126
debugger statistics 85
debugging 47
default makefile 59
DEFER_IN_ACTOR 84
deleting
- configurations 34

Destroy Debug Session 138
disabling TargetRTS features for performance 103
DLL functions 134
duplicating
- configurations 21

E

editing
- configuration 31
- configurations 25
- libset 30
- target 28

entryPoint function 92
error parser 96
- creating new 98
- modifying 95
- reusing 98

error parsing rules 96
Event Callback Function 142
Execute command 130
execute.pl 127
exiting application 110
EXTERNAL_LAYER 88
Extra Compiler Flags 31

F

File main.cc 93
file main.cc 80
floating point operations 46
functions
- entryPoint 92
- gethostbyname() reentrancy 111
- Main 91
- main() 78, 91
- RTMain_mainline() 79
- RTMain_targetShutdown() 78
- RTMain_targetStartup() 78
- RTOS supplies main() 109
- targetShutdown 92
- targetStartup 92

G

generated code
- compilation supported by makefiles 56

get DLL Capabilities 136
gethostbyname() reentrancy 111
gmake 33

H

HelloWorld model 101

I

implementation
- platform-specific 84

Initialize Debugger 138
INTEGER_POSTFIX 84

L

libraries
  maintaining TargetRTS 19
Library Builder 31
Libset 22, 115
libset
  editing 30
  makefiles 65
  name, components of 52
  platform name, part of 50
Libset Compiler Flags 31
Libset Linker Flags 31
libset makefile 65, 115
Libset name 24, 52
libset name 52
LIBSETCCEXTRA 31
LIBSETCCFLAGS 31
LIBSETLDFLAGS 31
Linker (LD) 31
linker configuration file 106
linking problems 106
Load command 128
load.pl 127
LOG_MESSAGE 72, 84

M

main 78
Main function 91
main function 91
main() function 78, 109
main.c 80
main.cc 93
Make
  macro definitions 66
make 27, 33
make dependencies
  regenerating 81, 94
makefiles 27, 56
  Config 65
  config, template 65
  creating 115
  default 59
  libset 65
  libset,template 65
  sequencing of 57
  target 64
  TargetRTS 56
  typical target, template 64
makefiles, creating 115
MANIFEST.c file 80
MANIFEST.cpp 94
MANIFEST.cpp file 94
manual mode 125
menu commands 127
Method RTDebuggerInput
  nextChar() 93
Method RTDiagStream
  write() 92
Method RTTimespec
  getclock() 89
Method RTTimespec::getclock() 89
model compilation 124
modes
  basic 125
  debugger 126
  manual 125
  target control 125
multi-threaded configuration 23
multi-threaded mode
  support for 76, 90
Mutex
  methods to protect shared resources 77, 90

N

new error parser, creating a 98
new files, adding to the TargetRTS 93
nmake 27, 33
no RTOS 52
NoRTOS 24, 52
NoRTOS Configuration 52
NoRTOS Target Base 25
NoRTOS target base 24

O

OBJECT_DECODE 72, 84
OBJECT_ENCODE 72, 84
observability
  adding support for xiii, 123
omake 34
OS capabilities 44
OS knowledge and experience 43
OTRTSDEBUG 73, 85

P

PATH variable 54
phases of a port 50
platform
  two-part name
    target and libset 45, 50
platform name, choosing a 50
platform-specific
  implementation 75, 84
port, major steps for implementing the 50
porting
  adding new files to TargetRTS 80, 93
  before starting 43
  building new TargetRTS 122
  Class RTIOMonitor 80
  Class RTMain 78
  Class RTMutex 76
  Class RTSyncObject 77
  Class RTTcpSocket 79
  Config makefiles 65
  configuration name 50
  configuring TargetRTS 103
  Constructor
    RTThread_construct(this,job,priority,stacksize) 76
  creating new error parser 98
  default command line arguments 109
  default makefiles 59
  disabling TargetRTS features for performance 103
  File main.c 80
  floating point operations 46
  libset makefiles 65
  Libset name 52
  linker configuration 106
  linker configuration files 106
  main() function 78, 91
  MANIFEST.c file 80
  Method RTDebuggerInput_nextChar() 79
  Method RTStdio_putString() 79
  Method RTTimespec_clocl_gettime() 76
  modifying error parser 95
  Platform-specific implementation 75
  platform-specific implementation 88
  preprocessor definitions 72, 84
  problems 105
  Regenerating make dependencies 81
  regenerating make dependencies 94
  Required operating system features for the C and C++ TargetRTS 44
  Required TargetRTS Classes and Functions 75
  script for porting 54
  system include files 106
  target makefiles 64
  target name 51
  target TCP/IP interfaces 111
  target toolchains 106
  TargetRTS 49
  TargetRTS classes 119
  TargetRTS configuration definitions 118
  TargetRTS example 113
  TargetRTS for C 71
  TargetRTS for C++ 83
  TargetRTS makefiles 56
  TargetRTS performance enhancements 104
  TargetRTS/RTOS interaction 107
  TCP/IP functionality 46
  test models 102
  testing TargetRTS 101
  Tool chain functionality (C and C++) 44
porting phases 50
ports
  creating between C and C++ 35
POSIX function calls 107
Problems 105

problems and pitfalls
target toolchains 106
TargetRTS/RTOS interaction 107
TCP/IP interfaces 111
problems and pitfalls with target TCP/IP interfaces 111
problems and pitfalls with target toolchains 106
problems and pitfalls with TargetRTS/RTOS interaction 107

R

Rational customer support
contacting xv
Rational Technical Support
what to do before calling 47
Real-time clock 107
Rebuild (make clean first) 34
reentrancy 111
regenerating make dependencies 81, 94
Registering Threads on Unix 133
Reset command 127
reset.pl 127
reusing an existing error parser 98
rtcomp.pl 97
RTDebuggerInput 93
RTDebuggerInput_nextChar 79
RTDiagStream 92
RTIOMonitor 80, 93
RTMain 78, 92
RTMain_mainline() 79
RTMain_targetShutdown() function 78
RTMain_targetStartup() function 78
RTMutex 76, 90
RTOS
building with none 52
supplies main() function 109
RTOS supplies main() function 109
RTREAL_INCLUDED 86
RTS_config 23
RTStdio_putString 79
RTSyncObject 77, 90
RTTcpSocket 79, 93
RTThread 90
RTTimespec 89
rules of operation 134

S

script
create setup script for porting 54
TargetRTS porting example 114
scripts 126
target control 126
Select() statement 111
Set Breakpoint At Function 144
Set Breakpoint in File 143
Set Callback 141
Set DllTrace 146
Set Source Search Path 143
setup script 54
creating a 54, 114
TargetRTS compilation to the platform 54
setup.pl 54
signal handlers 109
simple non-ObjecTime program on target 45
simple non-Rose RealTime program on target 45
single-threaded configuration 23
source code debugger integration 133
standard input/output functionality 46
Start Debugger 140
Stop Debugger 140

T

Target 22, 64, 117
target
building with no OS 52
name, components of 51
no RTOS 52
platform name, part of 50
Target bases 23
Target Compiler Flags 29
Target compiler optimizations 103
target compiler optimizations 103

target control
  adding support for xiii, 123
  audience 124
  basic mode 125
  calling sequence 133
  debugger mode 126
  defined 125
  Execute command 130
  general issues 132
  Load command 128
  manual mode 125
  menu commands 127
  model compilation 124
  modes 125
  overview 124
  Reset command 127
  rules of operation 134
  scripts 126
  Terminate command 131
  third-party source code debugger integration 133
  Unload command 129
target control scripts 126
  defined 126
Target Linker Flags 29
target makefile 64, 117
Target name
  definition 23
target observability
  defined 125
  overview 124
target operating system optimizations 104
Target Settings 29
target TCP/IP interfaces 111
TARGETCCFLAGS 29
TARGETLDFLAGS 29
TargetRTS
  adding new files 80, 93
  building configurations 32
  building new 122
  configuration definitions 118
  configuration types 22
  configuring 103
  creating ports between C and C++ 35
  deleting configurations 34
  description 18
  disabling features for performance 103
  duplicating a configuration 21
  editing a configuration 25, 31
  editing the libset 30
  editing the target 28
  existing configurations 20
  libraries
    compilation supported by makefiles 56
  libset 22
  libset name 24
  maintaining libraries 19
  makefiles 27
  managing configurations 21
  NoRTOS Target Base 25
  overview regarding porting 42
  performance enhancements 104
  porting example 113
  porting for C 71
  porting for C++ 83
  required operating system features 44
  specific performance enhancements 104
  Summary 24
  target 22
  Target bases 23
  target name 23
  testing 101
  Work Order 24
TargetRTS classes, code changes to 119
TargetRTS configuration definitions 118
TargetRTS features, disabling for performance 103
TargetRTS makefiles 56
targetShutdown function 92
targetStartup function 92
TCP/IP functionality 46
  porting 46
tcThreadCleanup 133
tcThreadInit 133
Terminate command 131
terminate.pl 127
The 80
threads
  registering on UNIX 133
thread-safe interface 133

Training 47
troubleshooting
Compiler optimizations 106
Default command line arguments 109
linker configuration file (C++) 106
linker configuration files 106
porting 105
problems with porting 105
Real-time clock 107
Return codes for POSIX function calls 107
return codes for POSIX function calls 107
signal handlers 109
system include files 106
target TCP/IP interfaces 111
target toolchains 106
TargetRTS/RTOS interaction 107
thread creation 107

U

UNIX
Registering threads 133
Unload command 129
unload.pl 127
USE_THREADS 37, 72, 84