The RT Services Library

How to manage it using the TargetRTS wizard

Author: Madhava Dama

Table of Contents

This document describes the TargetRTS wizard component, which is the wizard used by real-time applications developers to manage the targets from DevOps Model RealTime for TargetRTS.

Readers of this document are assumed to have read the document "Modeling Real-Time Applications in Model RealTime" which covers many of the concepts which are explained in more detail in this document.

The document was last updated for Model RealTime 10.2. All screen shots were captured on the Windows platform.

1. Using the TargetRTS Wizard

This chapter is organized as follows:

- *Overview of the TargetRTS Wizard*
- *Understanding the TargetRTS*
- *Maintaining TargetRTS Libraries using the TargetRTS Wizard*
- *Duplicating a Configuration*
- *NoRTOS Target Base*
- *Editing a Configuration*
- *Understanding the makefiles*
- *Editing the Target*
- *Editing the Libset*
- *Modifying a Configuration*
- *Building Configurations*
- *Deleting Configurations*

1.1 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 tool chain, the TargetRTS, and its layout.

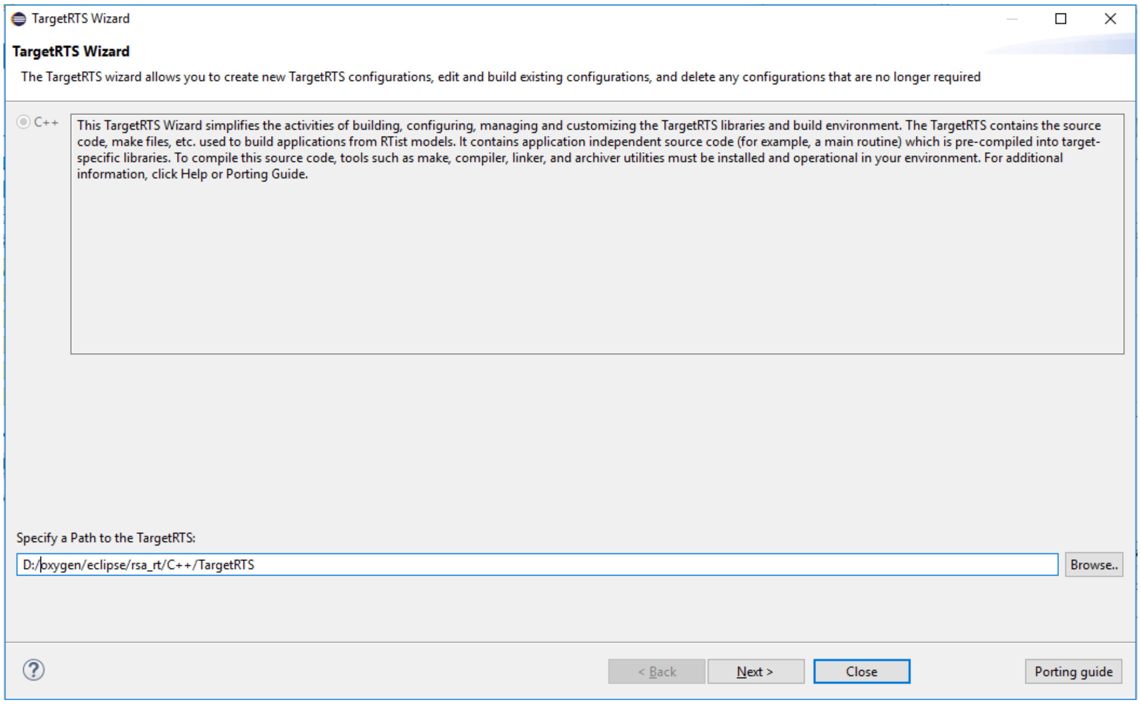
Note: The figures for the TargetRTS Wizard dialogs are for the C++ language.

1.2 Understanding the TargetRTS

The TargetRTS is the set of run-time services that provide a framework in which an M o d e l R e a l T i m e 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.

1.3 Maintaining TargetRTS Libraries using the TargetRTS Wizard

To access the TargetRTS Wizard invoke the command TargetRTS->TargetRTS Wizard. The picture below shows the first pane in the TargetRTS Wizard.



TargetRTS Wizard

TargetRTS Wizard

The TargetRTS wizard allows you to create new TargetRTS configurations, edit and build existing configurations, and delete any configurations that are no longer required

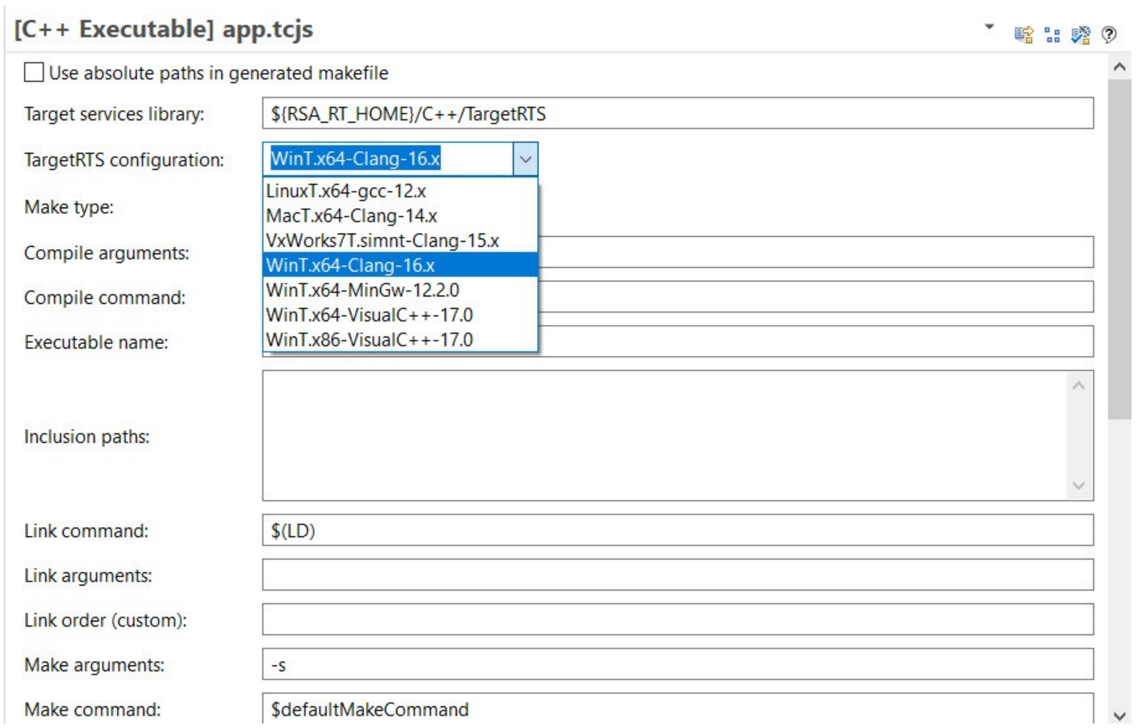
C++

This TargetRTS Wizard simplifies the activities of building, configuring, managing and customizing the TargetRTS libraries and build environment. The TargetRTS contains the source code, make files, etc. used to build applications from RTist models. It contains application independent source code (for example, a main routine) which is pre-compiled into target-specific libraries. To compile this source code, tools such as make, compiler, linker, and archiver utilities must be installed and operational in your environment. For additional information, click Help or Porting Guide.

Specify a Path to the TargetRTS:

D:/oxygen/eclipse/rsa_rt/C++/TargetRTS

Use this pane to locate the TargetRTS source location for the TargetRTS Wizard, then click Next.


[C++ Executable] app.tcjs

- [ ] Use absolute paths in generated makefile

Target services library:	${RSA_RT_HOME}/C++/TargetRTS
TargetRTS configuration:	WinT.x64-Clang-16.x
Make type:	
Compile arguments:	
Compile command:	
Executable name:	
Inclusion paths:	
Link command:	$(LD)
Link arguments:	
Link order (custom):	
Make arguments:	-s
Make command:	$defaultMakeCommand

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 and supported by Model RealTime are read-only and cannot be edited or deleted. To modify an Model RealTime configuration that is read-only, select the configuration and click Duplicate.

For additional information on modifying an Model RealTime configuration, see Duplicating a Configuration.

1.3.1 Managing Your TargetRTS Configurations

When managing configurations with the TargetRTS Wizard, you can:

- Select Duplicate radio button for *Duplicating a Configuration*
- Select Edit radio button for *Editing a Configuration*
- Select Build radio button for *Building Configurations*
- Select Delete radio button for *Deleting Configurations*
- Select browse radio button option for 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.

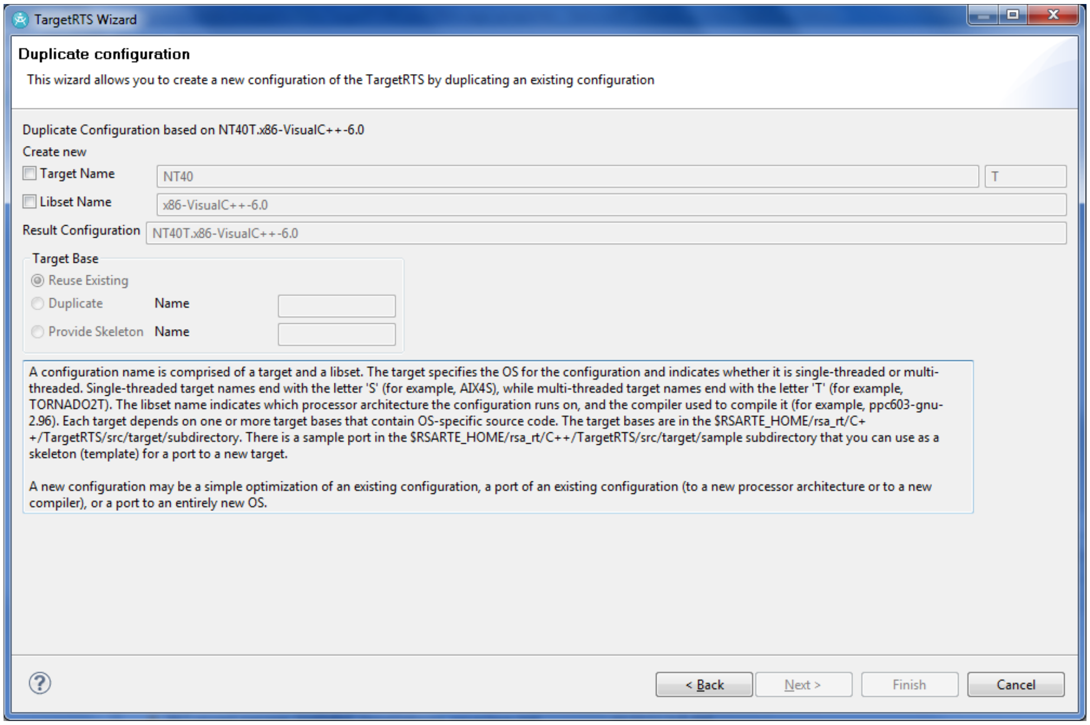
1.4 Duplicating a Configuration

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 radio button.
3. Click Next.



This wizard allows you to create a new configuration of the TargetRTS by duplicating an existing configuration

Duplicate Configuration based on NT40T.x86-VisualC++-6.0

Create new

- [ ] Target Name: NT40 | T
- [ ] Libset Name: x86-VisualC++-6.0

Result Configuration: NT40T.x86-VisualC++-6.0

Target Base
- [✓ Reuse Existing]
- Duplicate — Name
- Provide Skeleton — Name

A configuration name is comprised of a target and a libset. 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 $RSARTE_HOME/rsa_rt/C++/TargetRTS/src/target/subdirectory. There is a sample port in the $RSARTE_HOME/rsa_rt/C++/TargetRTS/src/target/sample subdirectory that you can use as a skeleton (template) for a port to a new target.

A new configuration may be a simple optimization of an existing configuration, a port of an existing configuration (to a new processor architecture or to a new compiler), or a port to an entirely new OS.

A new configuration can be:

- A simple optimization of an existing configuration
- A port of an existing configuration (to 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
<installdir>/rsa_rt/C++/TargetRTS/src/target/ directory.

Note: There is a sample port in <installdir>/rsa_rt/C++/TargetRTS/src/target/sample that you can use as a skeleton (a template) for a port to a new target.
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.

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.

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 Model 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.

Click Next.

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

Click Next.

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

Click Next.

1.5 NoRTOS Target Base

C++ TargetRTS has 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:

- From the Existing Configuration box on the Manage Configuration pane, select a configuration that uses the NoRTOSS target.
- In the Manage box, click Duplicate radio button.
- Under the Create new label, select Libset.
- 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.

1.6 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.

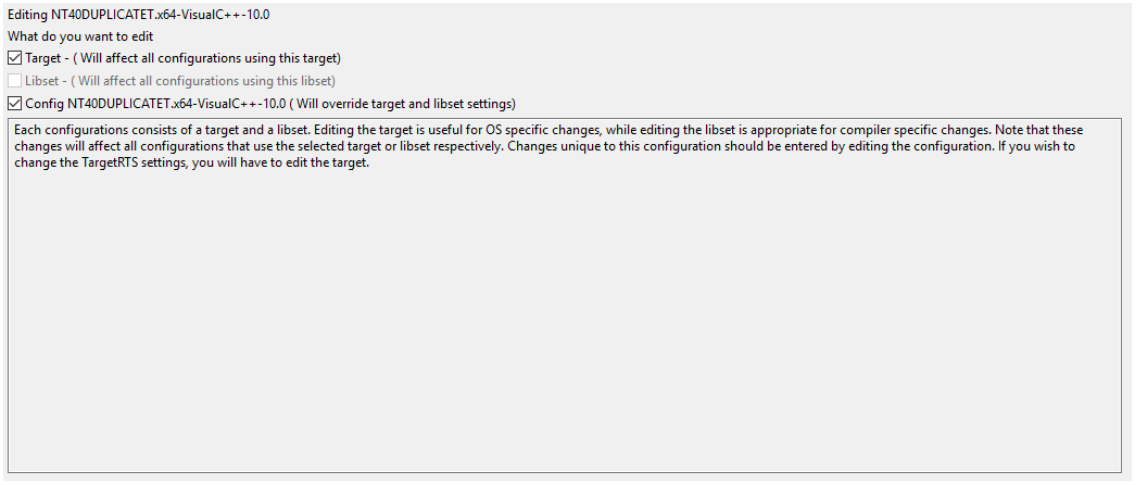
Note: You cannot edit the configurations that are included with Model RealTime, nor the targets and libsets that these configurations use. You can only edit the configurations that you duplicated previously.

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.

The picture below 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*
- *Editing the Libset*
- *Modifying a Configuration*



Editing NT40DUPLICATET.x64-VisualC++-10.0

What do you want to edit

- [✓] Target - (Will affect all configurations using this target)
- [ ] Libset - (Will affect all configurations using this libset)
- [✓] Config NT40DUPLICATET.x64-VisualC++-10.0 (Will override target and libset settings)

Each configurations consists 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. Note that these changes will affect all configurations that use the selected target or libset respectively. Changes unique to this configuration should be entered by editing the configuration. If you wish to change the TargetRTS settings, you will have to edit the target.

1.7 Understanding the makefiles

When you edit a configuration using the TargetRTS Wizard, you are modifying properties in one or more makefiles. The makefiles that you can update when specifying particular options while using the TargetRTS Wizard are the following:

- **Main TargetRTS makefile:**
  $RTS_HOME/src/main.nk (Unix)
  $RTS_HOME/src/main.nmk (Windows)
  - **Default makefile:**
    $RTS_HOME/libset/default.mk
  - **libset makefile:**
    $RTS_HOME/libset/<libset>/libset.mk
  - **target makefile:**
    $RTS_HOME/target/<target>/target.mk
  - **config makefile:**
    $RTS_HOME/config/<config>/config.mk

The default.mk, libset.mk, target.mk, and config.mk makefiles are used to compile both the TargetRTS libraries and the code that is generated from the model. The target.mk, libset.mk and config.mk makefiles override the defaults defined in <installdir>/rsa_rt/C++/TargetRTS/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.

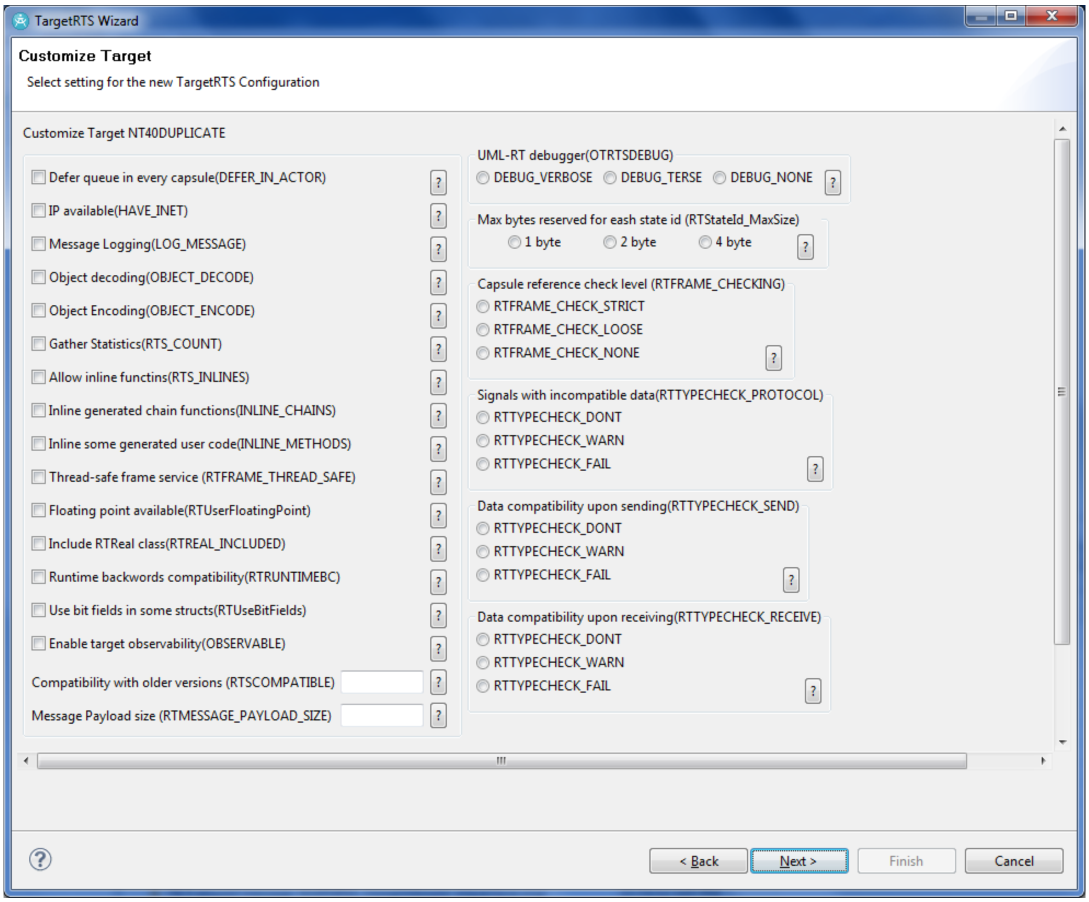
The target.mk file contains the definition specific to the target operating system.

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.

1.8 Editing the Target

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



TargetRTS Wizard

Customize Target

Select setting for the new TargetRTS Configuration

Customize Target NT40DUPLICATE

- Defer queue in every capsule(DEFER_IN_ACTOR)
- IP available(HAVE_INET)
- Message Logging(LOG_MESSAGE)
- Object decoding(OBJECT_DECODE)
- Object Encoding(OBJECT_ENCODE)
- Gather Statistics(RTS_COUNT)
- Allow inline functins(RTS_INLINES)
- Inline generated chain functions(INLINE_CHAINS)
- Inline some generated user code(INLINE_METHODS)
- Thread-safe frame service (RTFRAME_THREAD_SAFE)
- Floating point available(RTUserFloatingPoint)
- Include RTReal class(RTREAL_INCLUDED)
- Runtime backwords compatibility(RTRUNTIMEBC)
- Use bit fields in some structs(RTUseBitFields)
- Enable target observability(OBSERVABLE)

Compatibility with older versions (RTSCOMPATIBLE)

Message Payload size (RTMESSAGE_PAYLOAD_SIZE)

UML-RT debugger(OTRTSDEBUG): DEBUG_VERBOSE, DEBUG_TERSE, DEBUG_NONE

Max bytes reserved for eash state id (RTStateId_MaxSize): 1 byte, 2 byte, 4 byte

Capsule reference check level (RTFRAME_CHECKING): RTFRAME_CHECK_STRICT, RTFRAME_CHECK_LOOSE, RTFRAME_CHECK_NONE

Signals with incompatible data(RTTYPECHECK_PROTOCOL): RTTYPECHECK_DONT, RTTYPECHECK_WARN, RTTYPECHECK_FAIL

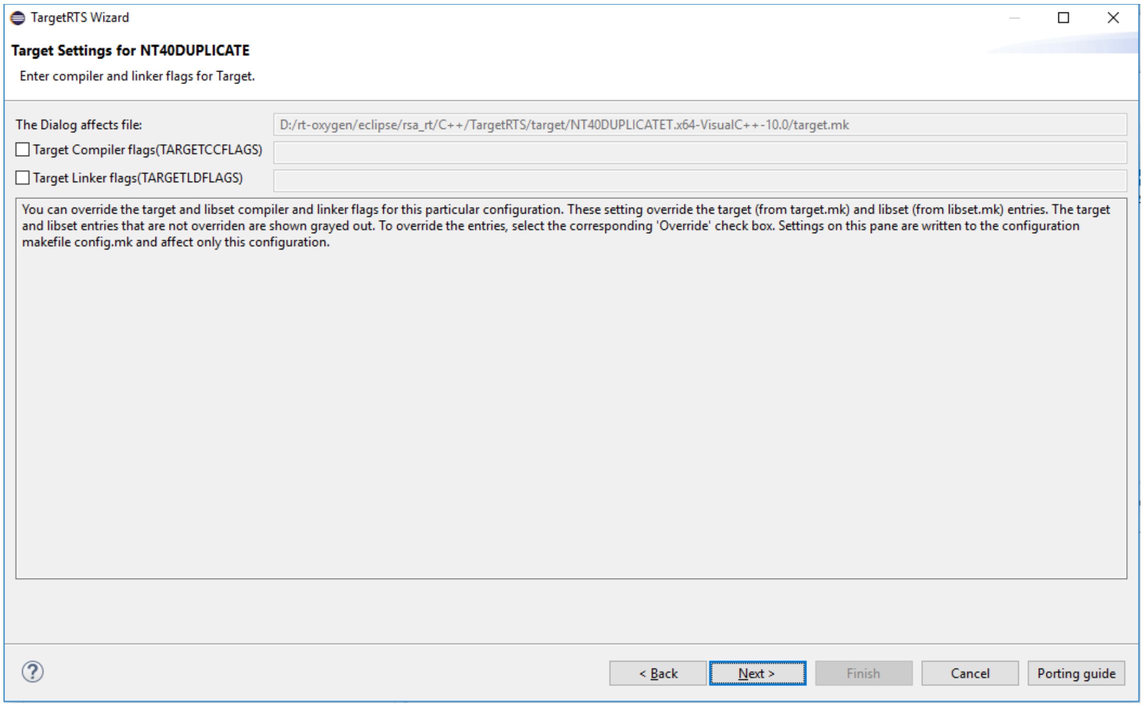
Data compatibility upon sending(RTTYPECHECK_SEND): RTTYPECHECK_DONT, RTTYPECHECK_WARN, RTTYPECHECK_FAIL

Data compatibility upon receiving(RTTYPECHECK_RECEIVE): RTTYPECHECK_DONT, RTTYPECHECK_WARN, RTTYPECHECK_FAIL

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.

The picture below 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.


TargetRTS Wizard
Target Settings for NT40DUPLICATE
Enter compiler and linker flags for Target.

The Dialog affects file: D:/rt-oxygen/eclipse/rsa_rt/C++/TargetRTS/target/NT40DUPLICATET.x64-VisualC++-10.0/target.mk

☐ Target Compiler flags(TARGETCCFLAGS)

☐ Target Linker flags(TARGETLDFLAGS)

You can override the target and libset compiler and linker flags for this particular configuration. These setting override the target (from target.mk) and libset (from libset.mk) entries. The target and libset entries that are not overriden are shown grayed out. To override the entries, select the corresponding 'Override' check box. Settings on this pane are written to the configuration makefile config.mk and affect only this configuration.

Target Compiler Flags (TARGETCCFLAGS)

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

Target Linker Flags (TARGETLDFLAGS)

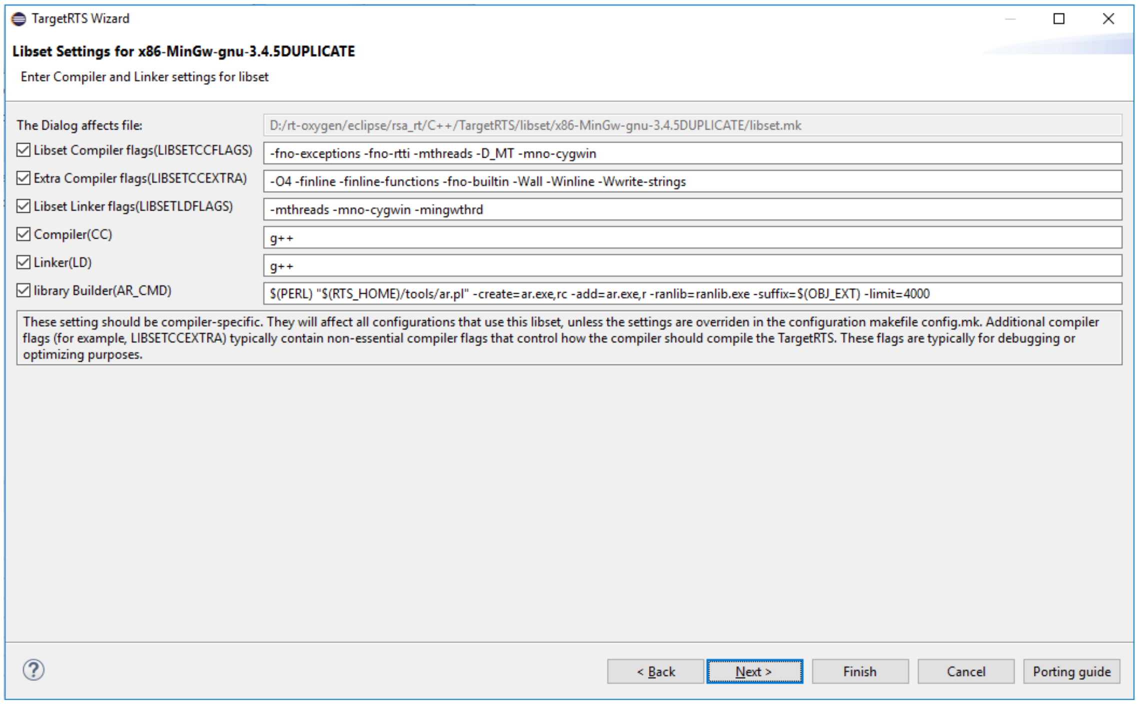
Redefines the target linker flags in the target.mk file.

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.

1.9 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).

The picture below 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.



Libset Settings for x86-MinGw-gnu-3.4.5DUPLICATE

Enter Compiler and Linker settings for libset

The Dialog affects file:	D:/rt-oxygen/eclipse/rsa_rt/C++/TargetRTS/libset/x86-MinGw-gnu-3.4.5DUPLICATE/libset.mk
☑ Libset Compiler flags(LIBSETCCFLAGS)	-fno-exceptions -fno-rtti -mthreads -D_MT -mno-cygwin
☑ Extra Compiler flags(LIBSETCCEXTRA)	-O4 -finline -finline-functions -fno-builtin -Wall -Winline -Wwrite-strings
☑ Libset Linker flags(LIBSETLDFLAGS)	-mthreads -mno-cygwin -mingwthrd
☑ Compiler(CC)	g++
☑ Linker(LD)	g++
☑ library Builder(AR_CMD)	$(PERL) "$(RTS_HOME)/tools/ar.pl" -create=ar.exe,rc -add=ar.exe,r -ranlib=ranlib.exe -suffix=$(OBJ_EXT) -limit=4000

These setting should be compiler-specific. They will affect all configurations that use this libset, unless the settings are overriden in the configuration makefile config.mk. Additional compiler flags (for example, LIBSETCCEXTRA) typically contain non-essential compiler flags that control how the compiler should compile the TargetRTS. These flags are typically for debugging or optimizing purposes.

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++ 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.

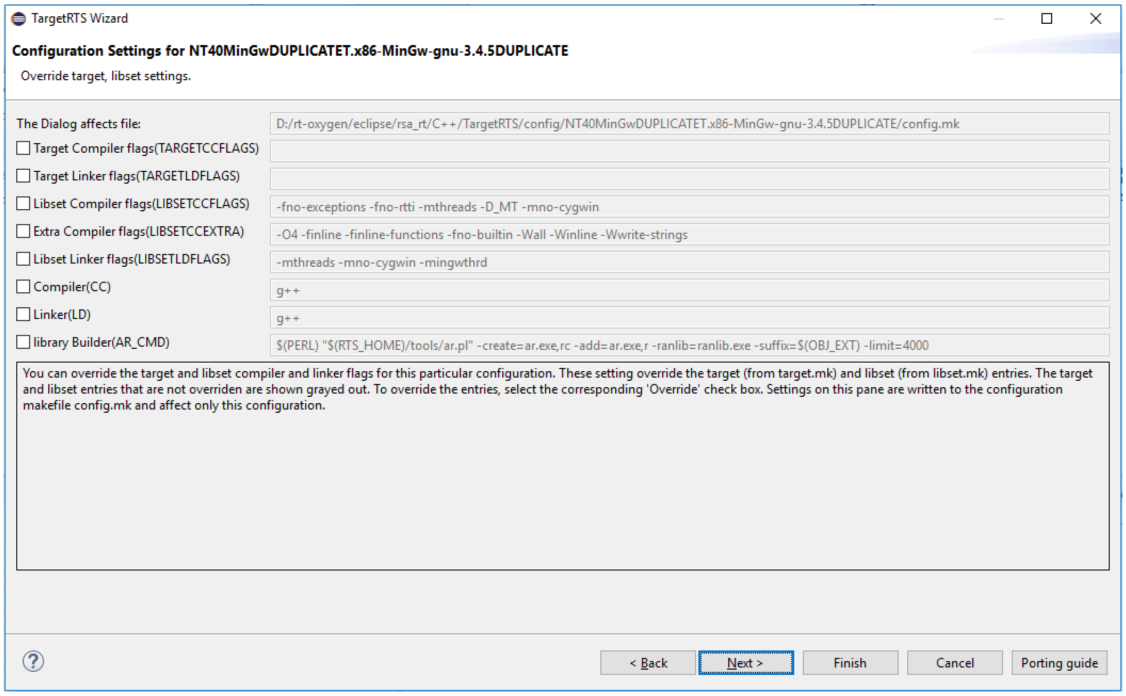
Library Builder (AR_CMD)

Specifies a command to run the library utility.

1.10 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.

The picture below 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.



TargetRTS Wizard

Configuration Settings for NT40MinGwDUPLICATET.x86-MinGw-gnu-3.4.5DUPLICATE

Override target, libset settings.

The Dialog affects file:	D:/rt-oxygen/eclipse/rsa_rt/C++/TargetRTS/config/NT40MinGwDUPLICATET.x86-MinGw-gnu-3.4.5DUPLICATE/config.mk
☐ Target Compiler flags(TARGETCCFLAGS)	
☐ Target Linker flags(TARGETLDFLAGS)	
☐ Libset Compiler flags(LIBSETCCFLAGS)	-fno-exceptions -fno-rtti -mthreads -D_MT -mno-cygwin
☐ Extra Compiler flags(LIBSETCCEXTRA)	-O4 -finline -finline-functions -fno-builtin -Wall -Winline -Wwrite-strings
☐ Libset Linker flags(LIBSETLDFLAGS)	-mthreads -mno-cygwin -mingwthrd
☐ Compiler(CC)	g++
☐ Linker(LD)	g++
☐ library Builder(AR_CMD)	$(PERL) "$(RTS_HOME)/tools/ar.pl" -create=ar.exe,rc -add=ar.exe,r -ranlib=ranlib.exe -suffix=$(OBJ_EXT) -limit=4000

You can override the target and libset compiler and linker flags for this particular configuration. These setting override the target (from target.mk) and libset (from libset.mk) entries. The target and libset entries that are not overriden are shown grayed out. To override the entries, select the corresponding 'Override' check box. Settings on this pane are written to the configuration makefile config.mk and affect only this configuration.

1.11 Building Configurations

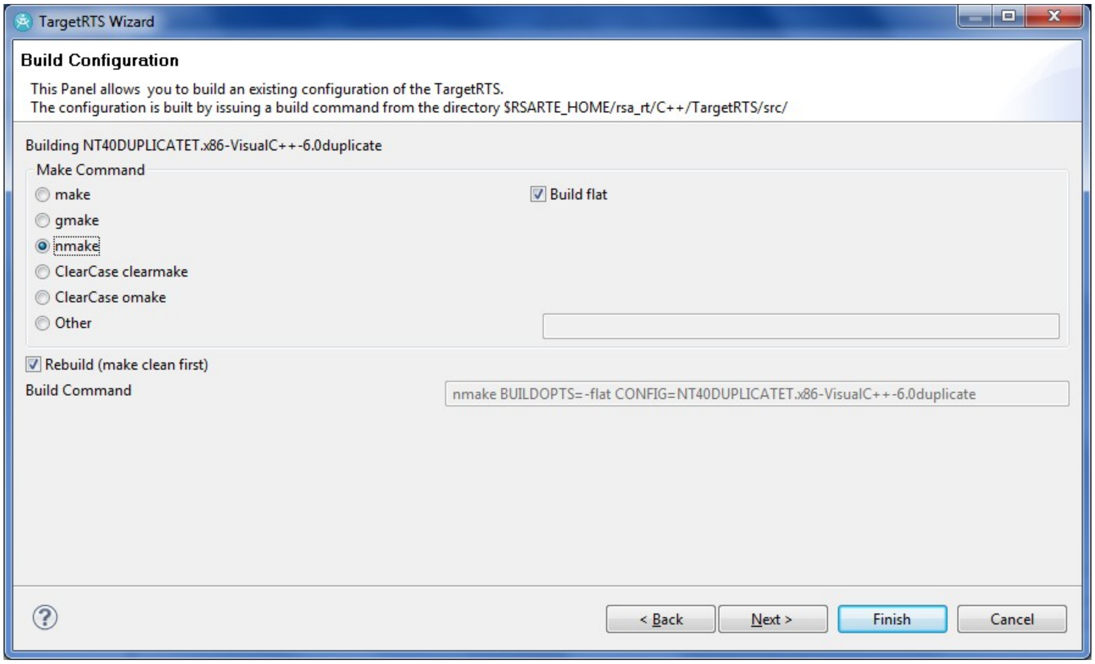
To build an existing configuration of the TargetRTS, you must specify the make command used by the build. The picture below shows the Build Configuration pane which you can use to compile the TargetRTS libraries.

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

<installdir>/rsa_rt/C++/TargetRTS/build-<*target*>-<*libset*>

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

<installdir>/rsa_rt/C++/TargetRTS/lib/<*target*>.<*libset*>


TargetRTS Wizard

Build Configuration

This Panel allows you to build an existing configuration of the TargetRTS.
The configuration is built by issuing a build command from the directory $RSARTE_HOME/rsa_rt/C++/TargetRTS/src/

Building NT40DUPLICATET.x86-VisualC++-6.0duplicate

Make Command

- ( ) make
- ( ) gmake
- (●) nmake
- ( ) ClearCase clearmake
- ( ) ClearCase omake
- ( ) Other

[✓] Build flat

[✓] Rebuild (make clean first)

Build Command: nmake BUILDOPTS=-flat CONFIG=NT40DUPLICATET.x86-VisualC++-6.0duplicate

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 an 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.

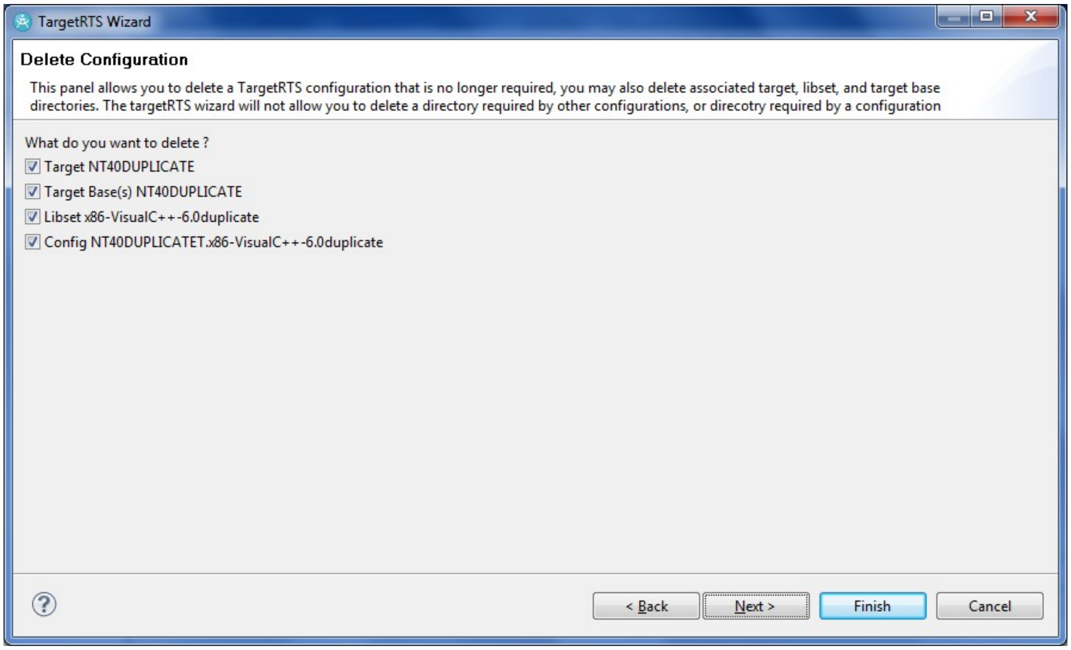
Note: Setting this option also decreases the build time considerably because fewer source files need to be opened and closed.

1.12 Deleting Configurations

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

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

The picture below 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.



Delete Configuration

This panel allows you to delete a TargetRTS configuration that is no longer required, you may also delete associated target, libset, and target base directories. The targetRTS wizard will not allow you to delete a directory required by other configurations, or direcotry required by a configuration

What do you want to delete ?

- [x] Target NT40DUPLICATE
- [x] Target Base(s) NT40DUPLICATE
- [x] Libset x86-VisualC++-6.0duplicate
- [x] Config NT40DUPLICATET.x86-VisualC++-6.0duplicate

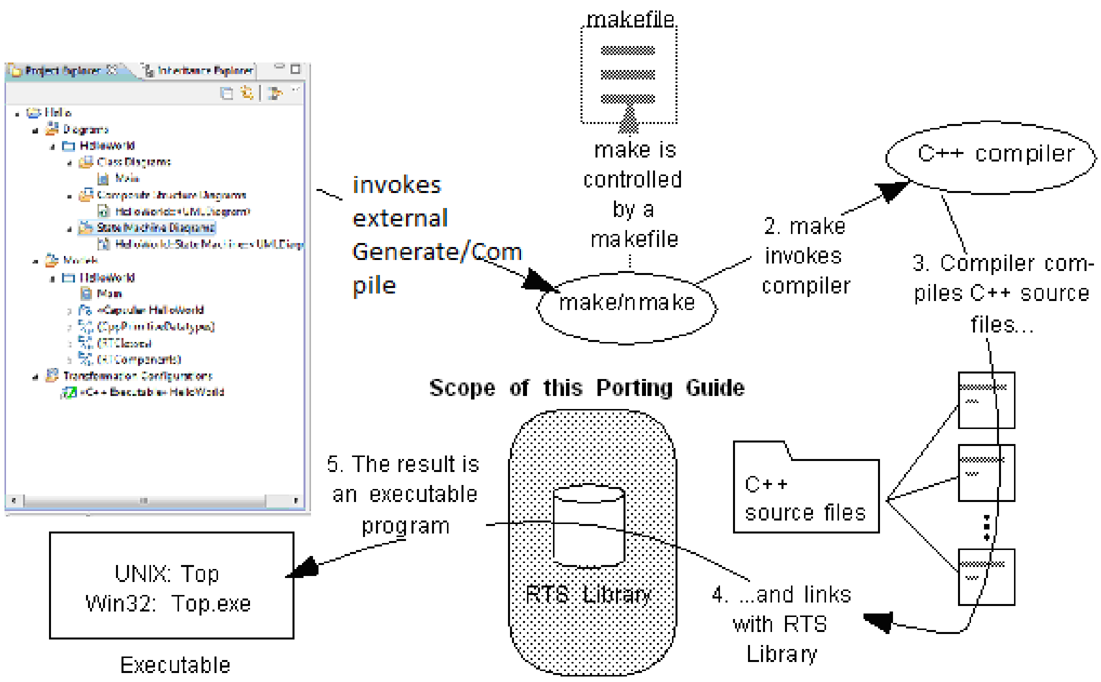
2. Introducing the TargetRTS

This chapter is organized as follows:

- *Overview*
- *Other Resources*

2.1 Overview

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


makefile
make is controlled by a makefile
C++ compiler
invokes external Generate/Compile
2. make invokes compiler
3. Compiler compiles C++ source files...
make/nmake
Scope of this Porting Guide
5. The result is an executable program
C++ source files
RTS Library
UNIX: Top
Win32: Top.exe
Executable
4. ...and links with RTS Library

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.

2.2 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)
The document “Building C++ Applications with Model RealTime”
Model RealTime example models (to test the port)

3. Before Starting a Port

This chapter is organized as follows:

- OS Knowledge and Experience
- Toolchain Functionality
- OS Capabilities
- Simple non-Model RealTime Program on Target
- TCP/IP Functionality
- Floating Point Operations
- Standard Input/Output Functionality
- Debugging
- Training
- Support
- What to do Before Calling Customer Support

3.1 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 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.

3.2 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-RT Program on Target*.

3.3 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. The table below lists the TargetRTS features and their corresponding RTOS services:

C++ TargetRTS Feature	**Operating System Service**
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::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)	A mutual exclusion mechanism. Some RTOSes provide optimized mutex service along with semaphores.
RTSyncObject (all 5 methods required for threaded targets)	Semaphore, mailbox, signal - service must provide infinite and timed blocking.
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::nex tChar() (input from console)	Standard input, as above. This can be removed from the TargetRTS via configuration options.

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.
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	Some RTOSes have their own main function defined. If so, then the main function in the TargetRTS must be redefined.

3.4 Simple non-RT 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 Model 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.

3.5 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.

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.

3.6 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.

3.7 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.

3.8 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.

3.9 Training

Training is an important component of a successful port. HCL 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.

3.10 Support

HCL 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.

3.11 What to do Before Calling Customer Support

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

- 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.
- 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 Model RealTime for this example program. This should be produced independently to verify toolchain and RTOS functionality.
- Read this guide and the C Reference or C++ Reference that is included with Model RealTime, to understand the required capabilities of the RTOS needed to support the TargetRTS.
- 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.

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

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

- Get as much training on Model RealTime, the RTOS, and your toolchain as possible.

4. Porting the TargetRTS

This chapter is organized as follows:

- *Overview*
- *Phases of a Port*
- *Choose a Configuration Name*
- *Building Model RealTime Applications for Targets without Operating Systems*
- *Creating a Setup Script (setup.pl)*
- *TargetRTS makefiles*
- *Default makefile*
- *Target makefile*
- *Libset makefile*
- *Config makefile*

4.1 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 that are 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 <installdir>/rsa_rt/C++/TargetRTS/C++/TargetRTS. In the sections that follow, examples are extracted from this source.

4.2 Phases of a Port

The major steps for implementing the port are as follows:

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

4.3 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 Model 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>
```

Some examples are given below.

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

4.4 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.

4.5 Libset Name

Although the actual libset names can be chosen arbitrarily, by convention those used by Model 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.

4.6 Building RT Applications for Targets without Operating Systems

You can configure the Model RealTime run-time libraries to build Model 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.

4.7 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.

4.8 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:

- From the Tools menu, click TargetRTS Wizard.
- Select a language and click Next.
- In the Manage Configurations pane, select a NoRTOS configuration, such as NoRTOSS.x86-VisualC++-6.0 NoRTOSS.sparc-gnu-2.8.1.
- Click Duplicate to modify the NoRTOS configuration for you requirements.
- In the Duplicate Configuration pane, select Libset.
- 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.
- Click Next.
- In the Summary pane, review the information, and then click Next.
- 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 Model 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.

4.9 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 Model RealTime. See <installdir>/rsa_rt/C++/TargetRTS/bin/tc/win32 for examples of these scripts.

4.10 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).

Note: Ensure correct tool chain paths are used in setup.pl, if required change them to appropriate paths/registry variables.

The table below describes the variables in the setup.pl file that are specific to Model RealTime:

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 Model 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.

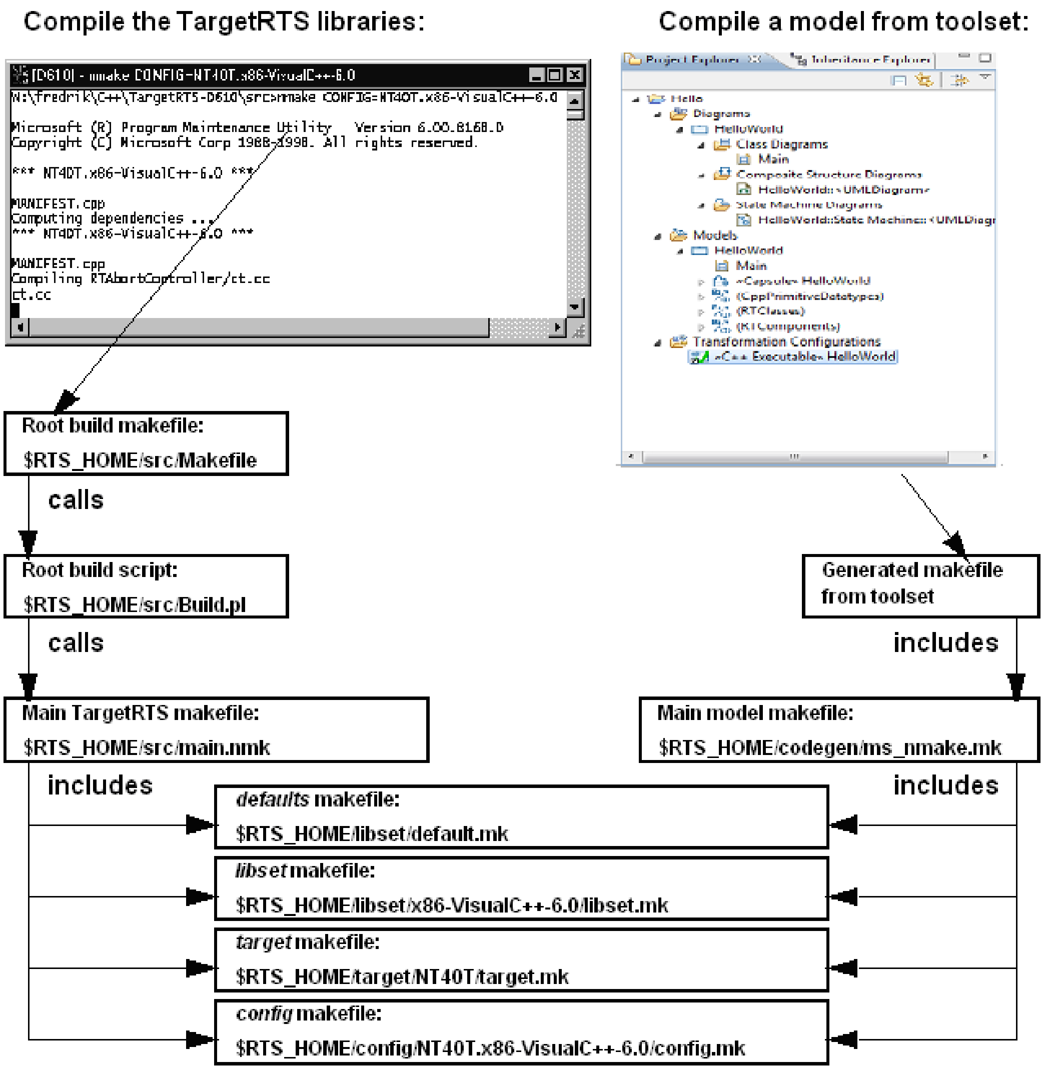
Note: The $preprocessor and $supported variables must be defined for all targets. As an example look at one of the setup.pl files located in the directory:

$RTS_HOME/config/<target>

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 Model RealTime toolset. 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.

4.11 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 the picture below.



```
N:\fredrik\C++\TargetRTS-D610\src>nmake CONFIG=NT40T.x86-VisualC++-6.0

Microsoft (R) Program Maintenance Utility   Version 6.00.8168.0
Copyright (C) Microsoft Corp 1988-1998. All rights reserved.

*** NT40T.x86-VisualC++-6.0 ***

MANIFEST.cpp
Computing dependencies ...
*** NT40T.x86-VisualC++-6.0 ***

MANIFEST.cpp
Compiling RTAbortController/ct.cc
ct.cc
```

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* 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.
- $RTS_HOME/target/<target>/target.mk is the definition specific to the target operating system. See Target makefile.
- $RTS_HOME/libset/<libset>/libset.mk is the definition specific to the compiler. See Libset makefile.
- $RTS_HOME/config/<config>/config.mk is the definition specific to the combination of the compiler, operating system and TargetRTS configuration. See Config makefile.
- 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 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
```

4.12 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

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

4.13 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
```

4.14 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++
LIBSETCCFLAGS  = -V2.8.1 -fno-exceptions  -fno-rtti
LIBSETCCEXTRA  = -O4 -finline  -finline-functions  -fno-builtin \ -Wall  -Winline  -Wwrite-strings
SHLIBS                    =
LIBSETLDFLAGS  = -V2.8.1
```

4.15 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.

The table below defines which make macros can be redefined and where they are set.

Table 1 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.
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.
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.
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.

5. Porting the TargetRTS for C++

This chapter is organized as follows:

- *Configuring the TargetRTS*
- *Platform-specific Implementation*
- *Adding New Files to the TargetRTS*

5.1 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 the table below.

Note: In the 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.

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.
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.
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.
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	(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.		
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.

5.2 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.

The table below 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.

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.

5.2.1 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.

5.2.2 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.

5.2.3 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().

5.2.4 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.

5.2.5 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.

5.2.6 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.

5.2.7 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.

5.2.8 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.

5.2.9 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.

5.2.10 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.

5.2.11 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().

5.3 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.

5.3.1 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.

5.3.2 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.

6. Modifying the Error Parser

When code generated by Model RealTime is compiled and linked various messages may be produced by the compiler and linker. This includes errors, warnings and informational messages. Model RealTime relies on functionality from Eclipse CDT for parsing these messages. Refer to the CDT documentation for how to customize the error parser if you find that messages produced by your compiler or linker are not correctly parsed.

7. Testing the TargetRTS Port

A port to a new platform requires testing the TargetRTS. There are some standard Model 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 basic level of assurance that the port was successful.

Create the test models from the New Model Wizard. There is one very small HelloWorld sample, and a slightly bigger TrafficLight sample that can be used.

Running applications generated from these sample models, validates the TargetRTS initialization and startup, log service and console output and basic capsule functionality.

8. Tuning the TargetRTS

This chapter is organized as follows:

- *Disabling TargetRTS Features for Performance*
- *Target Compiler Optimizations*
- *Target Operating System Optimizations*
- *Specific TargetRTS Performance Enhancements*

8.1 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" disable Target Observability, logging service and the RTS debugger. The minimal TargetRTS should provide significant performance gains over the fully featured version.

8.2 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.

8.3 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.

8.4 Specific TargetRTS Performance Enhancements

In 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.

9. Common Problems and Pitfalls

This chapter is organized as follows:

- *Overview*
- *Problems and Pitfalls with Target Tool chains*
- *Problems and Pitfalls with TargetRTS/RTOS Interaction*
- *Problems and Pitfalls with Target TCP/IP Interfaces*

9.1 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. 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)). 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.

9.2 Problems and Pitfalls with Target Toolchains

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

9.2.1 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.

9.2.2 Linker Configuration File

When linking an application to an 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.

9.2.3 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 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, RTtcp.h 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 directory as needed, and are typically prefixed by “RT” in the C++ version.

9.3 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.

9.3.1 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.

9.3.2 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 Model RealTime application.

9.3.3 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 Model 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 Model 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

}
```

9.3.4 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().

9.3.5 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 Model RealTime application. The documentation for the RTOS will describe how to start your application in this manner.

9.3.6 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 Model RealTime using the transformation configuration property called *Default Arguments*.

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, like it is done by the default implementation in $RTS_HOME/src/MAIN/main.cc:

```
int main( int argc, const char * const * argv ) // Standard main

{

return RTMain::entryPoint( argc, argv );

}
```

...you should instead 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.

9.3.7 Exiting Application

In the 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.

9.4 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 using the Model Debugger) will be disabled.

9.4.1 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++ TargetRTS.

10. TargetRTS Porting Example

10.1 Overview

This chapter provides an example of porting the TargetRTS for C++ to a new platform. This is an example port rather than customization of an existing port. See 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++ Compiler version 2.7.2-960126 for Motorola PowerPC microprocessors.

10.2 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++ 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
```

10.3 Create Setup Script

The setup script 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
{
$RSA_RT_HOME/rsa_rt/C++/TargetRTS                 = $ENV{'ROSERT_HOME'};
chomp(  $host                                     =
`$RSA_RT_HOME/rsa_rt/C++/TargetRTS/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.

10.4 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 Model RealTime models on this new TargetRTS and platform.

10.4.1 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 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. The table below illustrates these required commands.

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 Model RealTime toolset.

Another important role of the libset makefile is the definition of command line options.
The table illustrates the typical subset of 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:

```
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)
```

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

10.4.3 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.

10.5 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/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>
```

10.6 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. See chapter Porting the TargetRTS for C++ for 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::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 contain a summary of the required changes to each file.

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.

10.7 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 Model RealTime libraries will be placed in the $RTS_HOME/lib/TORNADO2T.ppc-cygnus-2.7.2-960126 directory.

11. Known problems / Issues

- Re-size of the 2nd TargetRTS wizard page on Linux will overlap the **Browse target heading** label and Path text box, which results in that the **Browse target heading** label is not visible clearly
- Groups on right hand side does not have the same length on TargetRTS wizard Customize target page, hence particular wizard page looks like not aligned the groups to same length
- On Windows there will be a conflict between Cygwin and MinGW paths, Developer required to use any one of them by setting particular path to PATH environment variable(i.e. Not to set both simultaneously to PATH environment variable)