Rational Software Corporation

C++ Porting Guide

RATIONAL ROSE® REALTIME

VERSION: 2002.05.20

PART NUMBER: 800-025104-000

WINDOWS/UNIX

Rational®
the software development company

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

Part Number: 800-025104-000

Version Number: 2002.05.20

Contents

Chapter 1

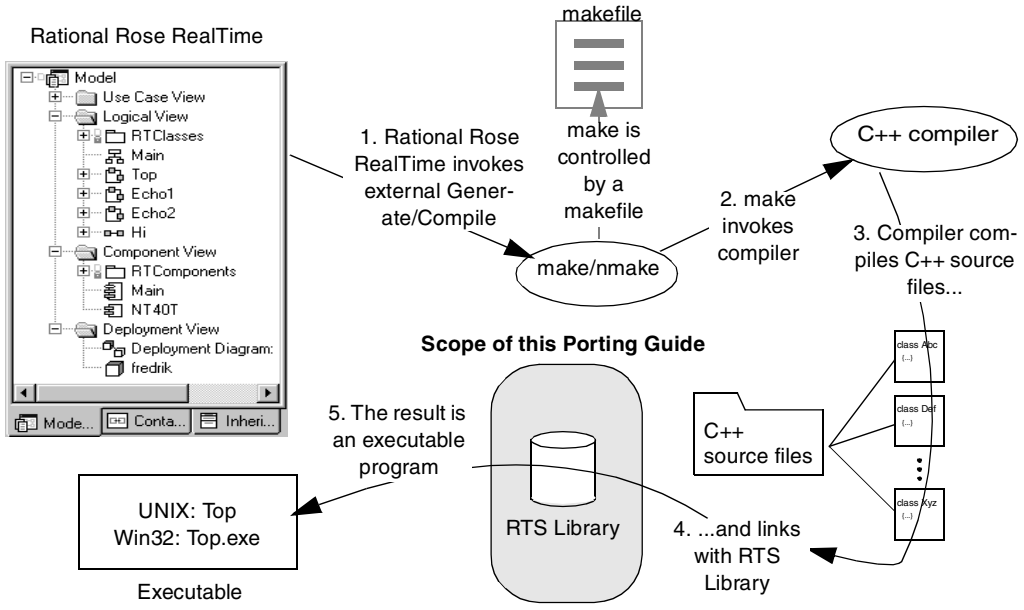
Introduction

The TargetRTS is the set of run-time services that provide a framework in which a Rational Rose RealTime model can run. It provides the run-time implementation of the UML-RT constructs used in the model. Figure 1 shows the context of the TargetRTS in building an executable program.

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

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

Figure 1 The TargetRTS in context



Other Resources

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

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

Chapter 2

Before starting a port

This chapter describes what you need to do before starting the port and it is organized as follows:

OS knowledge and experience

Knowledge and experience with the target operating system is key to a successful port. This knowledge should extend to the development environment and target hardware. The type of knowledge required includes such details as synchronization mechanisms, thread creation, memory management, timing, device drivers, board support packages, memory maps, TCP/IP support, priority and scheduling schemes, and so forth. See "OS capabilities" on page 4. for a list of OS capabilities required by the TargetRTS.

Experience with porting the TargetRTS to other platforms will aid greatly as the ports tend to follow a pattern. For each development environment and operating system there are bound to be a few surprises. See “Common Problems and Pitfalls” on page 51.

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 the creation of simple program, such as “Hello World”, which is compiled and run on the target. This step is described in “Simple non-Rational Rose RealTime Program on Target” on page 6.

OS capabilities

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

Table 1 Required operating system features for the C++ TargetRTS.

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::nextChar()` (input from console)	Standard input, as above. This can be removed from the C++ 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.

Simple non-Rational Rose RealTime Program on Target

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

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

Further testing is strongly recommended. This would include some basic RTOS services such as thread creation in your test program. Again, no TargetRTS code or libraries should be used. Many RTOSes provide example programs to compile and run. Try these out and verify the functionality of your setup. If you are using a source-level debugger, verify that you can step through the source code and examine variables. If the debugger is aware of operating system data structures, check if you can examine these. Another important test for C++ compilers is to include a static constructor in the test program. This will ensure that proper initialization is performed. 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.

TCP/IP functionality

In order to support Target Observability for the new port, the target operating system must provide a compatible TCP/IP stack. In general, the TCP/IP layer must support the BSD sockets interface, that is, the creation and deletion of sockets, functions such as **`socket()`**, **`connect()`**, **`bind()`**, **`listen()`**, **`select()`**, and so forth. Typically, RTOSes try to provide a BSD-compliant TCP/IP stack. TCP/IP functionality can be a common source of problems with new ports. See “Common Problems and Pitfalls” on page 51.

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.

Floating point operations

Some of the C++ TargetRTS classes (for example, **`RTReal`**) require the use of floating point operations. Investigate the support for floating point on your target system. It is possible to configure the support for RTReal from the TargetRTS via configuration options.

Standard input/output functionality

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

Debugging

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

Training

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

Support

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

What to do before calling Rational support

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

1. Get to know your compiler/linker/debugger toolchain. Be sure it is installed correctly, and that programs can be compiled, linked, downloaded to the target hardware and run successfully.
2. Get to know your target operating system. Be sure that an example multi-threaded program that exercises the various features of the RTOS is compiled, linked and downloaded to the target hardware and runs successfully. Do not use Rational Rose RealTime for this example program. This should be produced independently to verify toolchain and RTOS functionality.
3. Read this guide and the *C++ Reference* that is included with Rational Rose RealTime to understand the required capabilities of the RTOS needed to support the TargetRTS.
4. Ensure that the TCP/IP stack for your target platform is operational. In particular the sockets interface must be working, and additional utilities such as **`gethostbyname()`** must be functional.
5. Test the functionality of the standard input and output for your target. This will probably be verified in earlier steps.
6. Learn how to use the target debugger. This will be a useful tool when doing the port.
7. Get as much training on Rational Rose RealTime, the RTOS, and your toolchain as possible.

Chapter 3

Porting the TargetRTS

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

The ports supported by Rational Software and shipped with the TargetRTS source are a good place to begin considering design alternatives for a new port. The root directory for the TargetRTS source will be referred to from this point forward using the environment variable **`$RTS_HOME`**. It is usually defined as **`$ROSERT_HOME/C++/TargetRTS`**. For Windows, assume **`%ROSERT_HOME%\C++\TargetRTS`**. In the sections that follow, examples are extracted from this source.

Phases of a port

The major steps for implementing the port are as follows:

- Performing pre-port steps (see “Before starting a port” on page 3).
- Naming the platform (see “Choose a configuration name” on page 10).
- Defining the setup script (see “Create a setup script” on page 12).
- Defining the platform-specific **makefiles** (see “TargetRTS makefiles” on page 14).

- Defining the platform-specific header files (see “Porting the TargetRTS for C++” on page 27).
- Defining the platform-specific implementation of TargetRTS features (see “Preprocessor definitions” on page 28).
- Building the new TargetRTS and fixing compile and link problems (see “Building the new TargetRTS” on page 64).
- Testing the new TargetRTS using test model updates (see “Testing the TargetRTS Port” on page 47).
- Tuning the performance of the TargetRTS, if required (see “Tuning the TargetRTS” on page 49).

Choose a configuration name

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

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

Examples are given in Table 2

Table 2 Example configuration names.

Config Name	Description
`SUN4S.sparc-gnu-2.8.1`	SunOS 4.x Single-threaded on a Sparc processor using Free Software Foundation gnu version 2.8.1
`SUN5T.sparc-gnu-2.8.1`	Solaris 2.x Multi-threaded on a Sparc processor using Free Software Foundation gnu version 2.8.1

Config Name	Description
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.x Multi-threaded on a Motorola PowerPC processor using Cygnus C++ version 2.7.2-960126

Target name

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

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

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

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

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

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

Libset name

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

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

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

<processor> identifies processor architecture name

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

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

Create a setup script

The setup script is a file (**`setup.pl`**) containing Perl commands that set up the environment for the compilation of the TargetRTS for the platform. This file is contained in the **`$RTS_HOME/config/<config>`** directory. If the target toolchain environment variables are part of a user's standard environment, then the variables in the **`setup.pl`** file may not be necessary. These environment variables defined in the **`setup.pl`** file are **not** available when using the toolset to build user models.

The 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. This usually includes setting the shell environment variable **`PATH`** to point to the appropriate tools. Two variables must be defined for all targets, namely the **`$preprocessor`** variable and the **`$supported`** variable. The **`$preprocessor`** variable defines the C++ preprocessor command appropriate for the compilation environment, and is used to automatically generate source code dependencies for the TargetRTS. The **`$supported`** variable defines whether this target is supported by Rational Rose RealTime. Valid values for **`$supported`** are '`Yes`', '`No`' and 'Custom'. For a custom port, we recommend the value 'Custom'. This variable has no impact on how the port can be compiled or used.

Another variable to note is **$target_base**. This variable indicates that the implementation of the target-specific features of the TargetRTS are rooted in the same source directory as the **$target_base** target. For example, for the TORNADO2 targets, the **$target_base** is set to 'TORNADO1'. Therefore, TORNADO2 specific implementations of TargetRTS classes are found in the same source directory as those of the TORNADO1 target, that is, **$RTS_HOME/src/target/TORNADO1**.

The example file, **$RTS_HOME/config/TORNADO2T.ppc-cygnus-2.7.2-960126/setup.pl**, contains the following:

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

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

            $wind_base        = "$OS_HOME/wrs/tornado-2.0";
            if( $host eq 'sun5' )
            {
               $wind_host_type   = 'sun4-solaris2';
            }
            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";
```

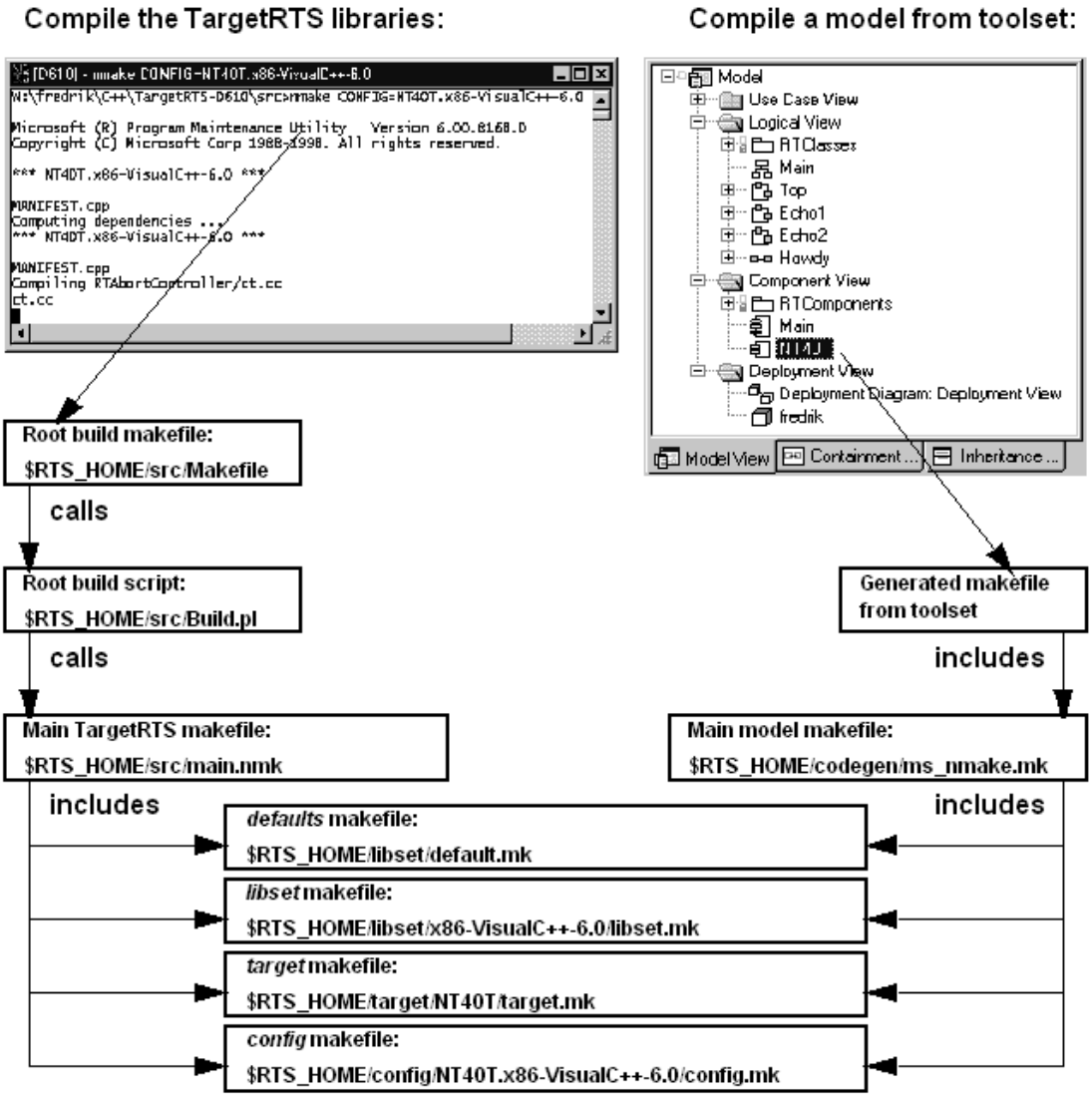
```
    $ENV{'VX_VW_BASE'}      = "$wind_base/target";
    $ENV{'WIND_HOST_TYPE'}  = "$wind_host_type";
}

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

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

TargetRTS makefiles

Two types of builds are supported by the **makefiles** for the TargetRTS: compilation of the TargetRTS libraries and compilation of the generated code. The platform-specific definitions are required by both and are thus placed in separate files. The sequencing of the **makefiles** for the two paths are shown in Figure 2, "Sequencing of Makefiles," on page 15.

Figure 2 Sequencing of Makefiles



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

- **`$RTS_HOME/src/Makefile`** is the root **makefile** for TargetRTS compilation. It invokes a Perl script called **`Build.pl`**. This script checks the dependencies for the TargetRTS source code and generates a **makefile** called **`depend.mk`** in the **`$RTS_HOME/build-<config>`** directory. It then builds the TargetRTS from this directory. This **makefile** and **`Build.pl`** should **not** be customized, and will not be discussed further in this document.
- **`$RTS_HOME/src/main.nmk`** (**`main.mk`** for Unix) is the main definition for compiling the TargetRTS libraries. These **makefiles** should **not** be customized, and will not be discussed further in this document.
- The generated **makefile** for the model being compiled. See the *C++ Reference* for more details on how this **makefile** is generated.
- **`$RTS_HOME/codegen/ms_nmake.mk`** (**`gnu_make.mk`** for Gnu, **`unix_make.mk`** for other Unix) is the main definition for compiling a model. These **makefiles** should **not** be customized, and will not be discussed further in this document.
- **`$RTS_HOME/libset/default.mk`**, the default macro definitions that may be overridden by the platform specific **makefiles**. See “Default makefile” on page 17.
- **`$RTS_HOME/target/<target>/target.mk`** is the definition specific to the target operating system. See “Target makefile” on page 20.
- **`$RTS_HOME/libset/<libset>/libset.mk`** is the definition specific to the compiler. See “Libset makefile” on page 20.
- **`$RTS_HOME/config/<config>/config.mk`** is the definition specific to the combination of the compiler, operating system and TargetRTS configuration. See “Config makefile” on page 21.

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

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

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

```
make <target>.<libset>
```

For example in Unix:

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

For example in Windows NT:

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

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:

```
# ======== 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)ObjecTime \
                      $(OT_LIB_TAG)ObjecTimeTypes

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

LD_OUT = $@

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

ALL_OBJS_LIST = $(ALL_OBJS)

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

```
# ======== Compiling ================================================

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

CC_TAIL =

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

Target makefile

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

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

Libset makefile

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

```
VENDOR         = gnu

CC             = g++

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

Config makefile

The **$RTS_HOME/config/<config>/config.mk** makefile provides definitions specific to the combination of the compiler, operating system and TargetRTS configuration. This **makefile** is empty for most target/libset combinations. Usually this file will only be needed to work around issues that may not appear in either the target or libset alone. An example use of this file can be found in **$RTS_HOME/config/VRTX4T.ppc603-Microtec-1.3C/config.mk**:

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

Table 3 defines which make macros can be redefined and where they are set.

Table 3 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 “perl”	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.

Chapter 4

Porting the TargetRTS for C++

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

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

Definitions made in these files override their default definitions in **`$RTS_HOME/include/RTConfig.h`**. The symbols and their default values are listed in Table 4.

Note: *In Table 4, in general, defining a symbol with the value 1 enables (= sets) the feature the symbol represents and defining it with the value 0 disables (= clears) the feature.*

Table 4 Preprocessor definitions

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

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

Symbol	Default Value	Possible Values	Description
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.

Symbol	Default Value	Possible Values	Description
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 two above)	(see above)	Should signal be checked for signal and type compatibility as it is received?

Symbol	Default Value	Possible Values	Description
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.

Platform-specific implementation

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

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

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

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

Table 5 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.

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

Table 5 Required TargetRTS Classes and Functions

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

Method `RTTimespec::getclock()`

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

Constructor `RTThread::RTThread()`

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

Class `RTMutex`

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

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

Class `RTSyncObject`

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

The implementation of five methods is required:

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

`main()` function

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

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

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

Class `RTMain`

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

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

- **`installOneHandler()`** - in file **`$TARGET_SRC/RTMain/installOneHandler.cc`**. This method is used by **`RTMain::installHandlers()`** to install the Unix style signal handlers. These signal handlers are used by the single threaded TargetRTS for timer and I/O interrupts. If the target OS does not implement signal handlers, this method can be overridden by an empty method.

Method `RTDiagStream::write()`

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

Method `RTDebuggerInput::nextChar()`

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

Class `RTTcpSocket`

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

Class `RTIOMonitor`

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

File `main.cc`

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

Adding new files to the TargetRTS

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

The `MANIFEST.cpp` file

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

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

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

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

Regenerating make dependencies

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

Chapter 5

Modifying the Error Parser

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

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

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

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

How the error parser works

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

The error parsing rules

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

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

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

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

How "rtcomp.pl" integrates with the compiler

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

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

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

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

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

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

Reusing an existing error parser

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

Creating a new error parser

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

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

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

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

```
package config;

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

return 1;
```

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

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

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

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

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

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

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

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

Takes no parameters, does nothing.

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

1. Write a model that contains a representative set of compilation errors, compile it, and observe the output for the errors it generates. Add expressions one at a time and recompile until you have successfully captured all the errors.
2. Use programs that search the actual compiler or linker executable for strings. Then manually examine the output and intelligently determine which of the strings look like error statements.

Chapter 6

Testing the TargetRTS Port

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

HelloWorld model

This model is available in:

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

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

Other test models

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

Other resources

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

Chapter 7

Tuning the TargetRTS

This chapter briefly describes areas in the TargetRTS that can be tuned to improve performance. This chapter is organized as follows:

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* describes how to create such a version of the TargetRTS. The concepts of a "minimal TargetRTS" disables Target Observability, logging service and the RTS debugger. The minimal TargetRTS should provide significant performance gains over the fully featured version.

Target compiler optimizations

Most compilers provide optimizations at the object code generation stage that can produce faster running code. In general, if your compiler supports such optimizations, they should be used. Be sure to remove all debug options at the same time since they may cancel out certain or all optimizations. Some optimizations may come at the cost of code size. If application code size is a factor for your target then the benefit of optimization versus code size will have to analyzed. Many compilers

may have different levels of optimization, which may produce differing degrees of code size and performance enhancements. It is hard to predict the outcome of such optimizations in C++. Using a performance testing model which measures the speed of certain operations may prove useful.

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

Target operating system optimizations

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

Specific TargetRTS performance enhancements

In C++, 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.

Chapter 8

Common Problems and Pitfalls

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

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`**

Problems and pitfalls with target toolchains

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

Compiler optimizations

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

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

Make sure the test models you use exercise each of the target OS primitives used by the TargetRTS. See Table 1, “Required operating system features for the C++ TargetRTS.,” on page 5 for a list of these features.

Linker configuration file

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

System include files

The structure and content of include files can be a challenge when moving to a new toolchain. In the TargetRTS an attempt is made to isolate the nuances of include files for each RTOS into a few specific include files that can be used by all the target-specific code. In general, all RTOS-specific definitions should be combined into a file called **`RT<os_name>.h`** in the **`$TARGET_SRC`** directory. This way all include files needed to access OS functions can be found in this one file. In the C++ TargetRTS, for TCP/IP specific include files, a file called **`RTtcp.h`**

should be created in the **$TARGET_SRC** directory. This file should contain all the necessary include files required for TCP/IP functions. Other, more specific, header files may be required to isolate unique interfaces for your RTOS. These may be added to the **$TARGET_SRC** directory as needed, and are typically prefixed by “RT”.

Problems and pitfalls with TargetRTS/RTOS interaction

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

Return codes for POSIX function calls

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

Thread creation

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

Real-time clock

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

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

For example:

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

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

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

    delta  = new_time;
    delta -= old_timer;

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

Signal handlers

Many RTOSs do not use signals that are typical of UNIX operating systems. If your RTOS does not provide signals, be sure to override the C++ TargetRTS code in **`RTMain::installHandlers()`** and **`RTMain::installOneHandler()`**.

RTOS supplies `main()` function

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

Default command line arguments

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

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

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 **`$TARGET_SRC/MAIN/main.cc`**. The modification needed is that instead of calling **`RTMain::entryPoint()`** with the arguments **`argc`** and **`argv`**, like in this default **`$RTS_HOME/src/MAIN/main.cc`**:

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

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

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

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

Exiting application

In the C++ TargetRTS, the **`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/RTDiag/panic.cc`** to use the exit function specific to your RTOS.

Problems and pitfalls with target TCP/IP interfaces

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

`gethostbyname()` reentrancy

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

Chapter 9

TargetRTS Porting Example

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. This is a currently supported platform.

Note: *The printed version of this document contains a workbook that can be used to capture the information you will need to reference while performing a port.*

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
```

Create setup script

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

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

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

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

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

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

The setup script must contain the mandatory definitions for the **$preprocessor** and **$supported** flags. The toolchain environment variables are usually required for cross compiler tools such as Microtec, 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 that the **$target_base** variable is set to **TORNADO1**. This means that the **TORNADO2T** target uses the same code base for the TargetRTS classes as the **TORNADO1** target.

Create makefiles

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

Libset makefile

The **libset makefile** is used to make specific definitions for the compiler. The command line interface for C++ compilers can differ significantly, particularly for cross-compilers such as the Cygnus C++ compiler. It is in this file that we make definitions for command line options for the compiler and linker and override other definitions made in **$RTS_HOME/libset/default.mk**. See "Default makefile" on page 17 for details. In any port of the TargetRTS there are certain commands required in the toolchain in order to support the building of the TargetRTS. Table 6 illustrates these required commands, the Unix equivalent, and the Microtec variant.

Table 6 Tools required for building the TargetRTS.

Command	Unix	Microtec
library archive	$(PERL)	$RTS_HOME/tools/ar.pl (script)
C++ Compiler	CC	ccppc
Linker	ld	lnkppc
VENDOR	n/a	Microtec

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

Another important role of the **libset makefile** is the definition of command line options. Table 7 illustrates the typical subset of command line options, the Unix equivalent, and the Cygnus variant.

Table 7 Important toolchain command line options

Option	Unix	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

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

```
VENDOR          = cygnus

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

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

ALL_OBJS_LIST = %$(ALL_OBJS_LISTFILE)
```

Target makefile

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

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

Configuration makefile

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

TargetRTS configuration definitions

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

These definitions are used to enable and disable various features in the TargetRTS. By default almost all of the TargetRTS features are enabled (for example, Target Observability). The porting effort may be made easier if some of these features are disabled. See section "TargetRTS Customization Example" in the *C++ Reference* for instructions on how to build a minimal TargetRTS. The content 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>** not available.

```
ppc-cygnus-2.7.2-960R6
#include <new.h>
```

Code changes to TargetRTS classes

Most ports to new targets require some minor changes to the TargetRTS code. These changes typically apply to operating system features for thread (task) creation and destruction, mutual exclusion and synchronization and time services. Table 4, “Preprocessor definitions,” on page 28 gives a description of TargetRTS classes that might require changes.

The required changes to the TargetRTS source for TORNADO2 and the Cygnus compiler are located in the **`$RTS_HOME/src/target/TORNADO1`** directory. See the discussion for the setup script above for an explanation of why the directory is called **`TORNADO1`** rather than **`TORNADO2`**. For the remainder of this section, this directory is referred to as **`$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 **`$TARGET_SRC`**. For example, the new definition for **`RTTimespec::getclock()`** requires a subdirectory **`$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 8 contains a summary of the required changes to each file.

Table 8 Quick summary of common TargetRTS source file changes

Class	File	Change
MAIN	main.cc	**`main`** already defined by RTOS, use **`rtsMain`** with nonstandard argument handling instead.
RTDiag	panic.cc	Modified version since there is no **`exit()`** method
RTMain	targetStartup.cc	Modify main thread priority to that specified in the toolset
RTMutex (required)	ct.cc dt.cc enter.cc leave.cc	**Required implementation** using Tornado specific calls to **`semMCreate`**, **`semDelete`**, **`semTake`** and **`semGive`**.

Class	File	Change
RTSyncObject (required)	ct.cc dt.cc signal.cc timedwait.cc wait.cc	**Required implementation** using Tornado specific calls to **`semBCreate`**, **`semDelete`**, **`semGive`** and **`semTake`**.
RTThread (required)	ct.cc	**Required implementation** using Tornado specific calls to **`taskSpawn`** and **`taskSuspend`**, etc.
RTTimespec (required)	getclock.cc	**Required implementation** using Tornado specific call to **`clock_gettime`**.
RTinet	lookup.cc	Modified version, uses **`hostGetByName`** instead of **`gethostbyname`**.

Building the new TargetRTS

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

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

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

Index

E

F

G

H

I

L

M

N

O

P

R

S

T

U