OBJECTIME ®

Porting Guide

Product Release: ObjecTime Developer 5.2
Document Version: 1.0
Release Date: October 1998
Part Number: OT-R520-PKG009

ObjecTime Limited
340 March Road
Kanata, Ontario
Canada K2K 2E4

Printed in Canada

Important Notice

Restricted Rights Legend

COMMERCIAL COMPUTER SOFTWARE — RESTRICTED RIGHTS

About this document

This guide is divided into the following four parts:

- **Part 1: Porting the TargetRTS for C workbook** leads you through the steps necessary to port the TargetRTS for C. Print this chapter so that you can capture information that you need to reference. The workbook begins with a simple model execution and moves progressively to more complex areas, including debugging, target observability, threading, error parsing, and timers.
- **Part 2: Porting guide reference** complements the first section. It provides reference material on how to port the TargetRTS to a new computing platform. This may involve a totally new compiler and target environment or may be a port to a newer version of the compiler/linker/debugger tool chain. Part 2 includes the following chapters:
  - The **Introduction** describes the purpose and audience for this document.
  - **Before starting the port** describes what you need to do before starting the port.
  - **Porting the TargetRTS** describes the common features of the TargetRTS and the requirements needed to port the TargetRTS to a new platform.
  - **Porting the TargetRTS for C++** describes porting the C++ version of the TargetRTS.
  - **Porting the TargetRTS for C** describes porting the C version of the TargetRTS.
  - **Modifying the error parser** describes how to modify the error parser.
  - **Testing the TargetRTS** describes how to test the TargetRTS.
  - **Tuning the TargetRTS** describes how to improve the performance of the TargetRTS.
  - **Common problems and pitfalls** provides information on common problems and pitfalls encountered when performing ports.
- **Part 3: Appendices** provides the following porting examples:
  - **TargetRTS for C++ porting example**
  - **TargetRTS for C Porting example**
- **Part 4: Index**

Note: It is intended that the next version of this guide will contain a Porting the TargetRTS for C++ workbook.

ObjecTime support

Your opinions and suggestions are both welcome and vital to the evolution of ObjecTime.

ObjecTime support hotline: (613) 591-3400

ObjecTime fax: (613) 591-3784

ObjecTime email: support@objectime.com

Table of Contents

Part 2: Porting guide reference 21

Part 1

Porting theTargetRTS for C workbook

Porting the TargetRTS for C workbook

1.0 Simple model execution phase

This chapter leads you through the steps required to create a simple, single-threaded, executable. Source code will most likely not need modification as this process primarily focuses on setting up compiler flags, link options, and creating your first C TargetRTS library. Print this chapter so that you capture the information you will need to reference.

1.1 Environment setup

1.1.1 Create target-specific environment variables

1 Confirm that basic environment variables specific to your target platform already exist on your host platform. If not, create them.

ex: Use the Windows NT Control Panel to create an environment variable called USR_MRI.

1.1.2 Note the values of existing environment variables

%OBJECTIME_HOME% =	
%RTS_HOME% =	
	%OBJECTIME_HOME%\C\TargetRTS

Note that %RTS_HOME% is a conceptual and internal environment variable that is used in this document. It does not need to be formally created as an operating system environment variable.

To find the value of %OBJECTIME_HOME% on Windows NT, open an ObjecTime Developer command prompt and type “set OB”.

1.1.3 Install all required target OS software on the host and target

Install and configure the target OS software on both the host and target platforms.

1.1.4 Compile and run hello.c for your target

☐ **1.** Confirm that all the required host tools are present and properly configured by compiling and linking the following hello.c program. This should be done with out the use of ObjecTime Developer.

☐ **2.** Confirm that all the required target tools are present and properly configured by loading the hello executable onto the target, and running it there. This should be done without the use of ObjecTime Developer.

```
#include <stdio.h>
main printf (“Hello World\n”)
{
printf (“Hello World\n”)
}
```

For more information see “Simple non-ObjecTime program on target” on page 27.

1.2 Choose names

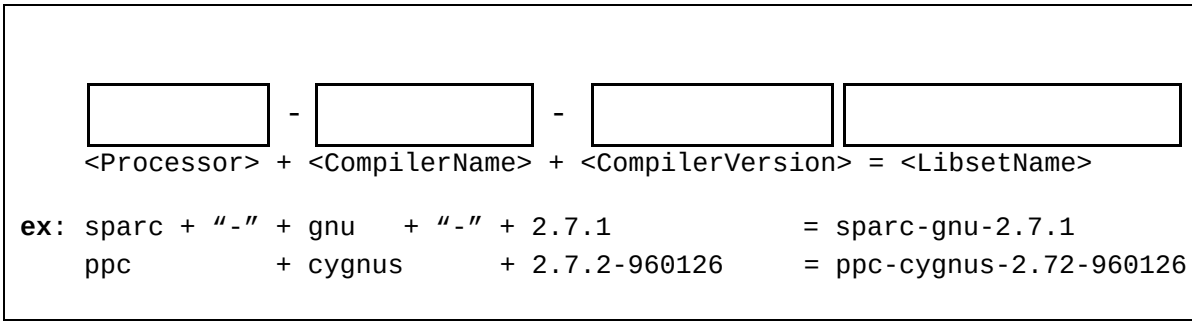
1.2.1 Choose a target name and target base name

	<OSname>	+ <OSversion>		=	<TargetBaseName>
	<OSname>	+ <OSversion>	+ <RTSconfig> (**S** or **T**)	=	<TargetName>
ex:	SUN	+ 5		=	SUN5
	SUN	+ 5	+ S	=	SUN5S
	TORNADO	+ 101		=	TORNADO101
	TORNADA	+ 101	+ S	=	TORNADO101S

For more information, see “Target name” on page 32.

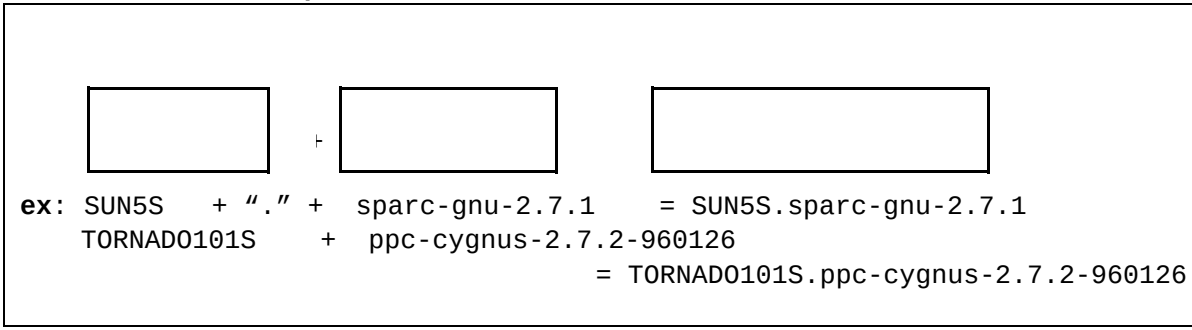
Note: By ObjecTime convention, ‘S’ denotes a single-threaded executable while ‘T’ denotes a multi-threaded executable. For more information about multi-threaded run-time systems see “Threaded phase” on page 15.

1.2.2 Choose a libset name



For more information, see "Libset name" on page 33.

1.2.3 Determine the platform name



For more information, see "Choose a platform name" on page 31.

1.3 Create directories

1. Open an ObjecTime Developer command prompt.

2. Create the following directories:

☐ `%RTS_HOME%\config\` ☐ `<PlatformName>`

☐ `%RTS_HOME%\lib\` ☐ `<PlatformName>`

☐ `%RTS_HOME%\libset\` ☐ `<LibsetName>`

☐ `%RTS_HOME%\src\target\` ☐ `<TargetBaseName>`

☐ `%RTS_HOME%\target\` ☐ `<TargetName>`

3. If you need to override any of the standard source code, replicate the directory structure in `%RTS_HOME\src` under `%RTS_HOME%\src\target\<TargetBaseName>`.

For example:

`%RTS_HOME%\src\target\<TargetBaseName>\MAIN\main.c`

overrides

`%RTS_HOME%\src\MAIN\main.c`

You will most likely find it unnecessary to make modifications or additions to the source code for the simple model execution phase.

1.4 Create and edit Perl scripts

1.4.1 Create an environment variable setup script

☐ Create and edit the following file:

```
%RTS_HOME%\config\<PlatformName>\setup.pl
```

1. Edit the path environment variable in setup.pl.

ex:
```
$usr_mri = $ENV{'USR_MRI'};
$ENV('PATH') = "$usr_mri/bin;$ENV{'PATH'}";
```

2. Create the preprocessor environment variable.

ex:
```
$preprocessor = "cccppc -E >MANIFEST.i";
```

3. Create the supported environment variable.

ex:
```
$supported = 'Yes';
```

4. Create other new environment variables required by the target.

ex:
```
$include_opt = 'J';
$target_base = 'VRTX3';
```

For more information, see "Create a setup script" on page 33.

1.4.2 Make copies of Perl scripts

If the compiler does not have options that correspond to the regular -L and -l options, it will probably be necessary to copy and then modify `ld.pl` from `%RTS_HOME%\libset\x86-VisualC++-5.0` to `%RTS_HOME%\libset\<LibsetName>`.

In addition, if your system does not provide a suitable ar command copy `ar.pl` from `%RTS_HOME%\tools` to `%RTS_HOME%\libset\<LibsetName>` and modify as necessary.

1.5 Create and edit makefiles

For more information, see "TargetRTS makefiles" on page 34.

1.5.1 Create the config makefile

☐ `%RTS_HOME%\config\<PlatformName>\config.mk`

For more information, see "Config makefile" on page 39.

1.5.2 Create the libset makefile

☐ %RTS_HOME%\libset\<LibsetName>\libset.mk

For more information, see “Libset makefile” on page 39.

1.5.3 Create the target makefile

☐ %RTS_HOME%\target\<TargetName>\target.mk

For more information, see “Target makefile” on page 38.

1.6 Create and edit C source and header files

1.6.1 Create RTTarget.h

☐ Create and edit the following file:

%RTS_HOME%\target\<TargetName>\RTTarget.h

```
#ifndef __RTTarget_h__
#define __RTTarget_h__ included

#define USE_THREADS 0

#define RSLMULTITHREADED RSLFALSE
#define RSLTO RSLFALSE
#define RSLDEBUG RSLFALSE

#endif // __RTTarget_h__
```

For the simple model execution phase, disable Multi-threading (RSLMULTITHREADED), Target Observability (RSLTO) and Debugging (RSLDEBUG).

1.6.2 Creating platform-specific source files

If there is a need to override a portion of the standard code for the port, contained in the $RTS_HOME/src directory, copy the file that you want to override to the corresponding directory in $RTS_HOME/src/target.

For example:

On Unix:

```
$RTS_HOME/src/target/<TargetBaseName>/MAIN/main.c
```

overrides

```
$RTS_HOME/src/MAIN/main.c
```

Then when the compilation of the libraries is performed, the Perl scripts will choose the target specific source file rather than the standard source file.

1.6.3 Creating platform-specific include files

To override include files, such as `RT_Time.h`, contained in the `$RTS_HOME/src/include` directory, copy them to `$RTS_HOME/src/target/<TargetBaseName>` and then modify them.

1.7 Compile the TargetRTS for your target

1. Open an ObjecTime Developer Command Prompt.
2. Change to directory `%RTS_HOME%\src\`
3. Compile the TargetRTS by entering the following command:

```
make
nmake CONFIG=
```

```
<PlatformName>
```

ex: `make SUN5S.sparc-gnu-2.7.1`
`nmake CONFIG=NT40S.x86-VisualC++-5.0`

4. This creates a set of object and library files in the following directory:
`%RTS_HOME%\lib\<PlatformName>`
ex: If `<PlatformName> = SUN5S.sparc-gnu-2.7.1`, then the following files should be created in directory `%RTS_HOME%\lib\SUN5S.sparc-gnu-2.7.1` :
`libObjecTimeC.a`
`libObjecTimeCTransport.a`
`main.o`

1.8 Compile and run the C_HelloWorld model for your target

1.8.1 Activate C_HelloWorld within ObjecTime

1 Start an ObjecTime session. Do NOT Enable Target Observability.

2 Open the Workspace Browser.

3 Open a Directory Browser on %OBJECTIME_HOME%\ModelExamples\C.

4 Drag C_HelloWorld.update from the Directory Browser to the Workspace Browser.

5 Open a Model Browser on C_HelloWorld.

1.8.2 Create a new configuration

1. Right-click on Configuration in the Model Browser. Select New Configuration...
2. Name the new configuration → `_config`

   ex: `SUN5S_config`

   `TargetName`
3. Double-click on the name of the new configuration.
4. Activate the C Target TargetRTS version.
5. Right-click on Language Options. Select New Language Option...
6. Name the new language option →

   ex: `SUN5`

   `TargetBaseName`
7. Select C as the type. Press the OK button.
8. Activate the new language option.
9. Double-click on the name of the new language option.
   Set the Library to →

   ex: `sparc-gnu-2.7.1`

   `LibsetName`
10. If necessary, edit the various other values in the language options window, and then close it.
11. Close the Configuration Browser.
12. Activate the new configuration.

1.8.3 Edit the threads configuration

1 Right-click on the Update menu in the Model Browser. Select Open Threads Browser.

2 Activate the SingleThread threads configuration.

3 Close the Threads Browser.

1.8.4 Compile and run

1 Compile the HelloST actor. This will create an executable `%PWD%\C_HelloWorld\build\<TargetName>_config\HelloST.exe`.

2 Run the model on the target using the same approach that you used in step 1.1.4 when you compiled and ran `hello.c` on the target.

3 Confirm that it ran correctly by observing the output in the target console window. The output should look like:

```
C_HelloWorld-v1.0-OT5.2x-started:
Hello World
```

```
C_HelloWorld v1.0-OT5.2x-finished!
```

Compilation of the update is usually performed by compiling an actor in the ObjecTime toolset; however, an update can also be compiled from the `$UPDATE_DIR` by issuing the `make` command.

The sections that follow provide detail on additional procedures to create threaded executables, as well as more complex areas such as debugging, target observability, compiler/linker error parsing and timers.

> **Note:** Perl needs to be installed or the use of Perl disabled to avoid compilation failure in this simple model execution phase. For more information, please refer to section “If the compilation platform does not have Perl” on page 165.

2.0 Debug phase

2.1 Modify $RTS_HOME\target\<TargetName>\RTTarget.h

☐

1. Modify `$RTS_HOME\target\<TargetName>\RTTarget.h` to have the following define:

```
#define RSLDEBUG RSLTRUE
```

2. Compile the libraries again and see if any further code modifications are required. If standard Unix-like I/O capabilities are available on your target, it is likely that no modifications are required.

2.2 Debug run-time testing

☐

1. Compile the `C_HelloWorld` model against the new libraries.
2. Run it from the command line, type stats, and then 'info 0' and 'info 1', and then verify the information is sane.
3. Optional: If the stats and info commands are working correctly, it's most likely that all the debugging features will be functional. However, you may want to test them at this point. See the debugging test plan.

3.0 Target Observability phase

3.1 Modify $RTS_HOME\target\<TargetName>\RTTarget.h

☐

1. Modify `$RTS_HOME\target\<TargetName>\RTTarget.h` to have the following define:

```
#define RSLTO RSLTRUE
```

2. Compile the libraries again and see if any further code modifications are required. If your OS supports BSD-style sockets, it is likely that no modifications are required.

3.2 Target Observability run-time testing

☐

1. Compile the C_StateChanger update against the new libraries.
2. Start up the executable to connect with Target Observability.
   You can use any of the provided connection methods, such as manual or basic. It might also be necessary to embed the connection arguments as default arguments if the target does not accept arguments upon process startup.
   For more information on how to connect to a C target with Target Observability, please refer to the C Language Guide.
3. Open up a behavior editor on the top-level actor.
4. Click Run and ensure the animation works correctly when you inject a message with the port probe.
5. Optional: If the animation is working correctly, it is most likely that all the Target Observability (TO) features will be functional. However, you may want to test them at this point. See the TO test plan.

4.0 Threaded phase

4.1 Creating the multi-threaded libraries

Essentially, you will be completing most of the steps from the “Simple model execution phase” on page 3; however, you will be choosing a multi-threaded target. To save time, the steps you need to redo are included below.

☐

1. Consider that the new `<TargetName>` will be `<TargetBaseName>T`
2. Since the other files will be similar, you will only have to create:

```
$RTS_HOME\config\<TargetName>.<LibsetName>\config.mk
$RTS_HOME\config\<TargetName>.<LibsetName>\setup.pl
$RTS_HOME\target\<TargetName>\RTTarget.h
$RTS_HOME\target\<TargetName>\target.mk
```

3. Copy your single-threaded files and modify them to include the thread information described below:

a. Modify `$RTS_HOME\target\<TargetName>\target.mk` to include the thread library names in the `TARGETLIBS` macro.

For example, the line might look like the one below before the modification:

```
TARGETLIBS     = $(LIB_TAG)posix4
```

and like the one below after the modification:

```
TARGETLIBS     = $(LIB_TAG)posix4 $(LIB_TAG)thread
```

b. Modify `$RTS_HOME\target\<TargetName>\RTTarget.h` to have the following define:

```
#define RSLMULTITHREADED RSLTRUE
```

c. As well you will probably have to create

```
$RTS_HOME\src\target\<TargetBaseName>\THREAD\RTThread.c
```

4. Compile the new threaded libraries and see if any further code modifications are required.

4.2 Threaded phase run-time testing

☐

1. Compile the C_Multithreads model against the threaded TargetRTS.
2. Run it and observe the output to see if it is processing inter-thread messages.

5.0 Porting the error parser phase

The majority of the concepts involved in this section are explained in detail in “Modifying the error parser” on page 91. Using the concepts from that section, you should do the following:

☐

1. Determine if the compiler output for the new target’s libset is similar to the output for a reference port.

   For example, sparc-gnu-2.7.1 and sparc-gnu-2.8.1 have similar error formatting rules.

2. If yes, just set the VENDOR make macro in the `libset.mk` file to reference the existing vendor, and the error parsing port is done.
3. If no, see “Creating a new error parser” on page 89.

5.1 If the compilation platform does not have Perl

If the compilation platform does not have Perl, the following solutions are available:

5.1.1 Short-term solutions

(Both the following solutions disable the use of Perl for the compilation phase, and consequently, error parsing. You will, however, be able to compile without getting a fatal error indicating Perl is not present.)

Solution 1: Make use of a make overrides file, as described in the chapter “Makefiles” in the *C Language Guide*. Set the following variables to nil within that file:

- `OTCOMPILE_CMD =`
- `OTLINK_CMD =`

Solution 2: Modify the `$RTS_HOME/target/<TargetName>/target.mk` file to set the following variables to nil:

- `OTCOMPILE_CMD =`
- `OTLINK_CMD =`

5.1.2 Long Term Solution

Download and compile Perl for the compilation platform.

6.0 Porting timers phase

The majority of the concepts involved in this section are explained in “Implementing timer services in the C TargetRTS” on page 77.

Using the concepts from this section, you should decide what type of timers you want to implement—local timers or actor timers. It is suggested that for RTOS applications, local timers be used; otherwise, actor timers can be used.

6.1 Local timers

☐

1. In `$RTS_HOME/target/<TargetName>/RTTarget.h` define the following:

```
#define RSLTIMERS RSLTRUE
#define RSLACTOR_TIMERS RSLFALSE
```

2. Compile the TargetRTS.
3. Decide whether to use Integrated timers, or Integrated IPC and timers. Integrated IPC's use another task for the timing function calls.

6.1.1 Integrated timers

☐ Take an existing example of an update with integrated timers
(such as C_TornadoQueuesWithTimers, Sender and Replyer Actors)
and modify it to use the OS specific functionality for the new OS.

Specifically, re-implement in the timer actor:

1. The signalling function
2. ExternalInterface (timer) function.
3. Initialize transition code for an actor in each thread that requires timer services. This initialization transition registers the functions and creates the data structures they will use.
4. Once the existing modified example is working, insert this functionality into an actor in each of your target threads.

6.1.2 Integrated IPC and timers

☐ Take an existing example of an update with integrated IPC timers
(such as C_TornadoQueuesWithTimers, Behavior and Responder actors)
and modify it to use the OS specific functionality for the new OS.

Specifically, re-implement in the timer actor:

1. ExternalInterface (Timer) function.
2. Initialize transition code for an actor in each thread that requires timer services. This initialization transition registers the functions and creates the data structures they will use. It also creates a message queue object and stores the handle to this in an actor ESV.
3. Once the existing modified example is working, insert this functionality into an actor in each of your target threads.

6.2 Actor timers

1. In `$RTS_HOME/target/<TargetName>/RTTarget.h` define the following:

```
#define RSLTIMERS RSLTRUE
#define RSLACTOR_TIMERS RSLTRUE
```

2. Compile the TargetRTS

3. Take an existing example of an actor timer, for example, Timer Solaris MT, from the Model `ModelExample/C/C_Timers` update.
   Merge it into another blank update, rename it, and merge it back into the `C_Timer` update.
   Modify it to use the OS specific functionality for the new OS.
   Specifically re-implement in the actor timer:

   a. External interface (Timer) function.
   b. Signalling function.
   c. Initialize transition code that creates a condition variable and stores it as an actor ESV.
   d. Initialize transition code that registers this actor as the provider of all timing services.

4. Drag the new actor timer into the top level actor of any update you want to provide timing services for.

Part 2
Porting guide reference

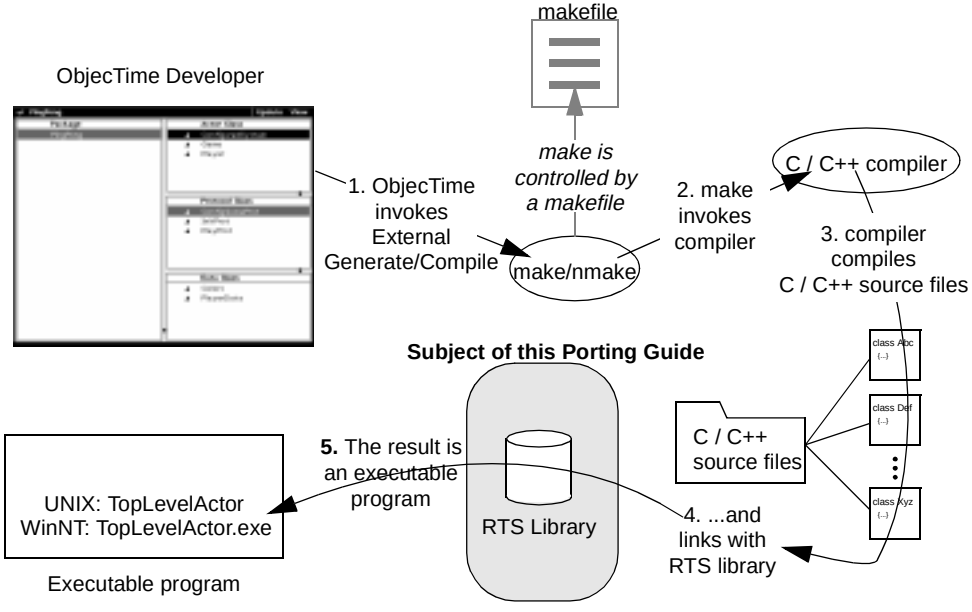
Introduction

The TargetRTS is the set of run-time services that provide a virtual machine on which an ObjecTime model can run. It provides the run-time implementation of the ROOM 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 ObjecTime staff and consultants. 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



Chapter 3

Before starting the port

This chapter describes what you need to do before starting the port.

OS knowledge and experience

Knowledge and experience with the target operating system is key to a successful port. This knowledge should extend to 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 25 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 99.

Tool chain 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 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-ObjecTime program on target” on page 27.

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 Services for C++

TargetRTS Feature	Operating System Service
RTSyncObject	Semaphore, mailbox, signal, condition variable — service must provide infinite and timed blocking.
RTTimespec::getclock	A function is required to return the current time. The more precision the better. In general, RTOS will return time with precision of its internal timer.
RTThread	Task creation function — must be able to create task or thread with specified stack size and priority. Be aware of priority scheme — some RTOS use 0 as highest priority while others may use 0 for lowest priority.
RTMutex	Semaphore, mutex, etc. — a mutual exclusion mechanism. Some RTOS provide optimized mutex service along with semaphores.
RTDiagStream (output to console)	Standard input and 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 (input from console)	As above. This can be removed from the C++ TargetRTS via configuration options. For more information, see “Customizing the Target Services Library” on page 47 of the *C++ Target Guide*.
Target Observability & External Layer	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. This can be removed from the TargetRTS via configuration options. For more information, see “Customizing the Target Services Library” on page 47 of the *C++ Target Guide*.
new, delete	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 symbol.
main function	Some RTOS have their own main function defined. If so, then the main function in the TargetRTS must be redefined.

Table 2 Required Operating System Services for a Multi-threaded C TargetRTS

TargetRTS Feature	Operating System Service
RSL_semaphore	Semaphore, condition variable — service must provide infinite and timed blocking.
RSL thr_create	Task creation function — must be able to create task or thread with specified stack size and priority. Be aware of priority scheme — some RTOS use 0 as highest priority while others may use 0 for lowest priority.
RSL_mutex	Semaphore, mutex, etc. — a mutual exclusion mechanism. Some RTOS provide optimized mutex service along with semaphores.
RSL_nextChar (input from console)	Standard character input. 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. This can be removed from the TargetRTS via configuration options.
malloc, free	RTOS should support some sort of memory management. If it is not supported, it may be possible to write your own.
main function	Some RTOS have their own main function defined. If so, then the main function in the TargetRTS must be redefined

Simple non-ObjecTime program on target

An easy way to test the tool chain 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, link, and download the program 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 the program. Again, no TargetRTS code or libraries should be used. Many RTOS provide example programs to compile and run. Try these out and verify the functionality. 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, then check to see if this functions. 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 BSD sockets interface, that is, the creation and deletion of sockets, functions such as `open()`, `connect()`, `bind()`, `listen()`, `select()`, and so forth. Typically, RTOSs try to provide a BSD-compliant TCP/IP stack. This is a common source of problems with new ports. See “Common problems and pitfalls” on page 99.

If a TCP/IP stack is not provided, then you must implement one, which requires significant effort. The lack of ethernet hardware may require the use of SLIP or PPP over a serial port, although this will severely 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. For more information, see “Customizing the Target Services Library” on page 47 of the *C++ Target Guide*.

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. This requires 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. ObjecTime 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.

What to do before calling ObjecTime support

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

1 Get to know your compiler/linker/debugger tool chain. Be sure it is installed correctly and 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 multithreaded program that exercises the various features of the RTOS is compiled, linked and downloaded to the target hardware and runs successfully. Do not use ObjecTime Developer for this example program. This should be produced independently to verify toolchain and RTOS functionality.

3 Read this guide and, if you are porting to a C++ target, the *C++ Target Guide* that is included with ObjecTime Developer 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()` and `gethostbyaddr()` must be functioning.

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 ObjecTime Developer, the RTOS, and tool chain as possible.

Chapter 4

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 tool chain. 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 ObjecTime Ltd. 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 a subdirectory of `$OBJECTIME_HOME` (`$OBJECTIME_HOME/C++/TargetRTS` or `$OBJECTIME_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 as outlined in section “Before starting the port” on page 25.
- Naming the platform (see “Choose a platform name” on page 31).
- Defining the setup script (see “Create a setup script” on page 33).
- Defining the platform-specific makefiles (see “TargetRTS makefiles” on page 34).
- Defining the platform-specific header files (see “Porting the TargetRTS for C++” on page 45).
- Defining the platform-specific implementation of TargetRTS features (see Table 5, “TargetRTS constants/macros and their default values,” on page 46).
- Building the new TargetRTS and fix compile and link problems (see “Building the new TargetRTS” on page 111).
- Testing the new TargetRTS using test model updates (see “Testing the TargetRTS” on page 95).
- Tuning the performance of the TargetRTS, if required (see “Tuning the TargetRTS” on page 97).

Choose a platform name

The first step in implementing a port is picking the name for the platform. This name and parts of it are used by the various loadbuild tools to find the files needed to build the TargetRTS for that platform. It is also used during compilation of the ObjecTime models. There are two parts to the name: **target** and **lib-**

set. The resulting names for TargetRTS configurations are defined as combinations of the target and libset names in the following pattern:

<platform> ::= <target>.<libset>

Examples are given in Table 3.

Table 3 Example platform names used by the TargetRTS

Name	Description
`SUN4S.sparc-gnu-2.7.1`	SunOS 4.x SingleThreaded on a Sparc processor using Free Software Foundation gnu version 2.7.1
`SUN5T.sparc-gnu-2.7.1`	Solaris 2.x MultiThreaded on a Sparc processor using Free Software Foundation gnu version 2.7.1
`SUN5S.sparc-SunC++-4.2`	Solaris 2.x SingleThreaded on a Sparc processor using Sun Microsystems SPARCUtils C++ version 4.2
`HPUX09S.hppa-HPC++-3.76`	HPUX 9.x SingleThreaded on an HPPA processor using Hewlett Packard HPC++ version 3.76
`PSOS2T.m68040-Green-1.8.7B`	pSOS 2.x MultiThreaded on a Motorola 68040 processor using GreenHills C++ version 1.8.7B

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 identify the configuration. Currently only ‘S’ or ‘T’ are supported, which denote single-threaded and multi-threaded configurations, respectively. (for example, `T`)

Libset name

Although the actual libset names can be chosen arbitrarily, by convention those defined by ObjecTime are defined as follows:

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

For example: `sparc-gnu-2.7.1`. 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 appropriate 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 to the platform. This file is contained in the `$(RTS_HOME)/config/<target>.<libset>` directory. If the target tool chain 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 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. The preprocessor command is used to automatically generate source code dependencies for the TargetRTS. The `supported` variable defines whether this target is supported by ObjecTime Limited. Valid values for `supported` are '`Yes`' and '`No`'. 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 VRTX4T target, the `target_base` is set to '`VRTX3`'. Therefore, VRTX4 specific implementations of TargetRTS classes are found in the same source directory as those of the VRTX3 target, that is, `$(RTS_HOME)/src/target/VRTX3`.

The example file, `$(RTS_HOME)/config/VRTX4T.ppc603-Microtec-1.4/setup.pl`, includes

```
$os = $ENV{'OS'};
$os = 'default' unless defined( $os );

if( $os eq 'Windows_NT' )
{
    $usr_mri            = $ENV{'USR_MRI'};
```

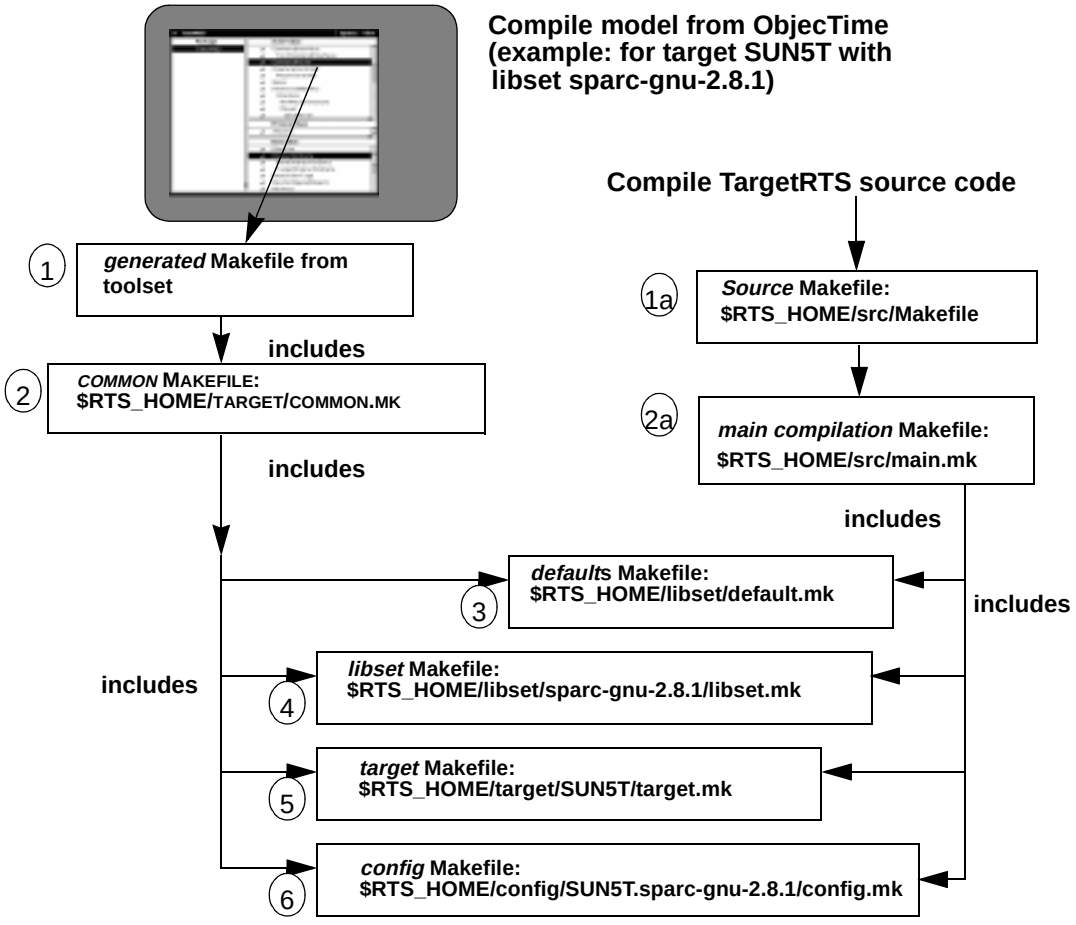
```
    $ENV{'PATH'}        = "$usr_mri/bin;$ENV{'PATH'}";
}
else
{
    $usr_mri            =
      "$ENV{'OS_HOME'}/spectra/solaris-ppc603-4.AB";
    $ENV{'USR_MRI'}     = "$usr_mri";
    $ENV{'SPECTRA'}     = "$usr_mri/spappc";
    $ENV{'MRI_PPC_BIN'} = "$usr_mri/bin";
    $ENV{'MRI_PPC_LIB'} = "$usr_mri/lib";
    $ENV{'MRI_PPC_INC'} = "$usr_mri/include/mccppc";
    $ENV{'PATH'}        = "$usr_mri/bin:$ENV{'PATH'}";
}

$preprocessor = "cccppc -E >MANIFEST.i";
$include_opt  = '-J';
$target_base  = 'VRTX3';
$supported    = 'Yes';
```

Note: The setup file is not used when compiling generated updates. The environment variables defined in the setup file must instead be defined in the user's environment before executing the ObjecTime toolset. In the given example, the setup file assumes that the user's environment has the variables USR_MRI and OS_HOME already defined. This is platform-specific.

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 35.

Figure 2 Sequencing of Makefiles



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

- The generated makefile for the update being compiled. See the *C++ Language Guide* or the *C Language Guide* for more details on how this makefile is generated.
- `$RTS_HOME/target/common.mk` is the main definition for compiling an update. This makefile cannot be customized and is not discussed further in this document.
- `$RTS_HOME/src/Makefile` is the root makefile for the TargetRTS libraries that selects the host to compile the libraries. See "TargetRTS makefiles" on page 36.
- `$RTS_HOME/src/main.mk` is the main definition for compiling the TargetRTS libraries. This makefile cannot be customized and is not discussed further in this document.
- `$RTS_HOME/libset/default.mk`, the default macro definitions that may be overridden by the platform specific makefiles. See "Default makefile" on page 36.
- `$RTS_HOME/libset/<libset>/libset.mk` is the definition specific to the compiler. See "Libset makefile" on page 39.

- `$RTS_HOME/target/<target>/target.mk` is the definition specific to the target operating system and TargetRTS configuration. See "Target makefile" on page 38.
- `$RTS_HOME/config/<target>.<libset>/config.mk` is the definition specific to the combination of the operating system, TargetRTS configuration, and compiler. See "Config makefile" on page 39.

  Note: The `target`, `libset`, and `config` makefiles are used to compile both the update and TargetRTS libraries.

Compilation of the update is usually performed by compiling an Actor in the ObjecTime Toolset; however, an update can also be compiled from the `$UPDATE_DIR` by issuing the `make` command.

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=NT40S.x86-VisualC++-5.0
```

TargetRTS makefiles

The `$RTS_HOME/src/Makefile` contains a default target that invokes a Perl script called `Build.pl`. This script checks the dependencies for the TargetRTS source code and generates a makefile called `depend.mk` in the `$(RTS_HOME)/build-<target>.<libset>` directory. It then builds the TargetRTS from this directory.

Default makefile

The `target`, `libset`, and `config` makefiles are expected to override defaults defined in `$RTS_HOME/libset/default.mk`. The defaults are as follows:

```
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

FEEDBACK      = $(PERL) "$(RTS_HOME)/tools/feedback.pl"
NOP           = $(PERL) "$(RTS_HOME)/tools/nop.pl"
PERL          = perl
PRELINK       = ld -r -o
RM            = $(PERL) "$(RTS_HOME)/tools/rm.pl"
RMF           = $(RM) -f
```

```
TOUCH         = $(PERL) "$(RTS_HOME)/tools/touch.pl"

# default pre-compile command, can be modified by libset.mk
PRECC         = $(NOP)

# 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
DEFINE_TAG    = -D
INCLUDE_TAG   = -I
LIBSETCCEXTRA =
LIBSETCCFLAGS =
OBJECT_OPT    = -c
OBJOUT_OPT    = -o
OBJOUT_TAG    =
SHLIBCCFLAGS  = -PIC
SOURCE_TAG    =
TARGETCCFLAGS =

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

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

# Macros used when creating an 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, li-
braries

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

#

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

EXTERNAL_LIBS = $(DIR_TAG)"$(RTS_LIBRARY)" \
         $(OT_LIB_TAG)ObjecTimeTransport \
         $(OT_LIB_TAG)ObjecTimeTypes

SYSTEM_LIBS   = $(DIR_TAG)"$(RTS_LIBRARY)" \
         $(OT_LIB_TAG)ObjecTime \
         $(OT_LIB_TAG)ObjecTimeTransport \
         $(OT_LIB_TAG)ObjecTimeTypes

SYSTEM_DIRECTORY  = $(LOCAL_DIRECTORY)/$(UPDATE)/C++/system
SYSTEM_DEPENDENCY = $(SYSTEM_DIRECTORY)/RTSystem.h \
                    $(SYSTEM_DIRECTORY)/initData.h

ALL_ACTORS_LIST = $(ALL_ACTORS)
```

Target makefile

The `$RTS_HOME/target/<target>/target.mk` makefile provides definitions specific to the operating system and TargetRTS configuration. The definitions in this makefile override the defaults in `$(RTS_HOME)/target/common.mk`. An example target makefile template file, `RTS_HOME/target/SUN5T/target.mk`, is as follows:

```
# Define the _REENTRANT macro to enforce thread safety
TARGETCCFLAGS = $(DEFINE_TAG)_REENTRANT
# Add in the nsl and socket libraries and to pass on the RTS
# library directory to the run-time linker
TARGETLDFLAGS = $(LIB_TAG)nsl $(LIB_TAG)socket -R$(RTS_LIBRARY)
# Add in the posix4 and thread libraries
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)/target/default.mk. An example libset makefile template file, $RTS_HOME/libset/sparc-gnu-2.7.1/libset.mk, is as follows:

```
# C++ compiler name and common arguments
CC            = g++ -V2.7.1

# Command to make shared libraries
SHLIB_CMD     = $(CC) -shared -z text -o

# Use pragmas to specify interface and implementation
LIBSETCCFLAGS = -DPRAGMA

# More c++ flags to turn on optimization, specify processor version
# and tune warnings
LIBSETCCEXTRA = -O4 -finline -finline-functions \
   -mv8 -Wall -Winline -Wwrite-strings

# C compiler flags for building shared libraries
SHLIBCCFLAGS  = -fPIC

# If SHLIBS is set to nothing,shared libraries will not be built
SHLIBS        =
```

Config makefile

The $RTS_HOME/config/<target>.<libset>/config.mk makefile provides definitions specific to the combination of the operating system, TargetRTS configuration, and the compiler. This makefile is empty for most target/libset combinations. Usually this file will only be needed to work around problems that may not appear in either the target or libset alone. In the C++ TargetRTS, an example use of this file is as follows:

$RTS_HOME/config/VRTX4T.ppc603-Microtec-1.3C/config.mk:

```
EXEC_EXT   = .x

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

Table 4 defines which make macros can be redefined and where they are set.

Table 4 Make macro definitions

TARGET	Defined in RTUpdate.mk as "$(PLATFORM)$(THREADED)"	Redefinition not recommended.
CONFIG	Defined in default.mk as "$(TARGET).$(LIBRARY)"	Redefinition not recommended.
OTCODEGEN_HOME	Defined in default.mk.	Redefinition not recommended.
VENDOR	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.
OTCOMPILE	Defined in default.mk.	Redefinition not recommended.
OTLINK	Defined in default.mk.	Redefinition not recommended.
FEEDBACK	Defined in default.mk.	Redefinition not recommended.
MERGE	Defined in default.mk.	Redefinition not recommended.
NOP	Defined in default.mk.	Redefinition from Perl scripts to (faster) OS-dependent commands is possible.
PERL	Defined in default.mk as "perl"	Some compilation hosts may require an explicit path; if necessary, redefine in libset.mk or config.mk.
RM	Defined in default.mk.	Redefinition from Perl scripts to (faster) OS-dependent commands is possible.
RMF	Defined in default.mk.	Redefinition from Perl scripts to (faster) OS-dependent commands is possible.
TOUCH	Defined in default.mk.	Redefinition from Perl scripts to (faster) OS-dependent commands is possible.
PRECC	Defined in default.mk.	
MAKEFILE	Defined in default.mk.	
CC	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.
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.
SHLIBCCFLAGS	Default defined in default.mk.	
SOURCE_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.	
LIBOUT_OPT	Default defined in default.mk.	
LIBOUT_TAG	Default defined in default.mk.	
RANLIB	Default defined in default.mk.	
SHLIB_CMD	Default defined in default.mk.	
SHLIBOUT_OPT	Default defined in default.mk.	
SHLIBOUT_TAG	Default defined in default.mk.	
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.	
TARGETLIBS	Default defined in default.mk.	
EXEOUT_OPT	Default defined in default.mk.	Redefine in libset.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 target.mk or libset.mk if necessary.
LIB_PFX	Default defined in default.mk.	
LIB_EXT	Default defined in default.mk.	
CPP_EXT	Default defined in default.mk.	
OBJ_EXT	Default defined in default.mk.	Redefine in libset.mk if necessary for a compiler/linker.
SHLIB_PFX	Default defined in default.mk.	
SHLIB_EXT	Default defined in default.mk.	
RTSYSTEM_INCPATHS	Defined in default.mk.	Redefinition not recommended.
RTS_LIBRARY	Defined in default.mk.	Redefinition not recommended.
EXTERNAL_LIBS	Defined in default.mk.	Redefinition not recommended.
SYSTEM_LIBS	Defined in default.mk.	Redefinition not recommended.
OTLINK_CMD	Defined in default.mk.	Redefine to “” while Perl is not available on the compilation host.
LD_HEAD	Default defined in default.mk.	May be used to redefine link command if necessary.
ALL_OBJS_LIST	Default defined in default.mk as the concatenation of all object files in the update.	Redefine 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.	May be used to redefine link command if necessary.
OTCOMPILE_OPTS	Defined in default.mk.	Redefinition not recommended.
OTCOMPILE_CMD	Defined in default.mk.	Redefine to “” while Perl is not available on the compilation host.
CC_HEAD	Default defined in default.mk.	May be used to redefine compile command if necessary.

CC_TAIL	Default defined in default.mk.	May be used to redefine compile command if necessary.
MAKEDEPEND_CMD	Defined in default.mk.	Redefine to “echo makedepend” while Perl is not available on the compilation host.
MAKEDEPEND_HEAD	Defined in default.mk.	May be redefined to add RTSYSTEM_INCPATHS to updates’ dependency discovery. It likely only makes sense to do this when porting requires changes to RTS include files.
USER_CC	From toolset, defined in RTUpdate_Compile.mk	Redefinition not recommended.

Chapter 5

Porting the TargetRTS for C++

TargetRTS configuration definitions

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 C preprocessor definitions. The configuration is set in two C++ header files:

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

These files override macros whose defaults appear in `$RTS_HOME/include/RTConfig.h`. The macros and their default values are listed in Table 5.

Table 4, “Make macro definitions,” on page 40 defines which make macros can be redefined and where they are set.

Note: In Table 5, in general, defining a symbol with the value 1 enables the feature the symbol represents and defining it with the value 0 disables the feature.

Table 5 TargetRTS constants/macros and their default values

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.
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 actors incarnated, and other statistics. Naturally, keeping track of statistics adds overhead.
DEFER_IN_ACTOR	0	0 or 1	If this flag is defined, the defer queues will be kept in each actor. If not, all deferred messages will be kept in one queue. This is a size/speed trade-off. Separate queues for each actor uses more memory but results in better performance.
EXTERNAL_LAYER	1	0 or 1	If this flag is 1, the TargetRTS has the capability to support layer connections over sockets with other processes. Required for target observability.
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 will log the contents of messages.

Table 5 TargetRTS constants/macros and their default values

Symbol	Default Value	Possible Values	Description
OBJECT_DECODE	1	0 or 1	Enables the conversion of strings to objects, needed for the external layer. This will always be enabled if the EXTERNAL_LAYER == 1.
OBJECT_ENCODE	1	0 or 1	Enables the conversion of objects to strings. Needed for the external layer, and log service. This will always be enabled if the EXTERNAL_LAYER == 1.
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 actors, and so on. This is necessary for the target debug 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.
RTREAL_INCLUDED	1	0 or 1	If 1, this flag allows the use of the RTReal class.
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_INLINES	1	0 or 1	Controls whether TargetRTS header files define any inline functions.

Table 5 TargetRTS constants/macros and their default values

Symbol	Default Value	Possible Values	Description
RTS_TYPES	1	0 or 1	Indicates whether the TargetRTS should use the ObjecTime type structures. This flag must be set to 1 for the current release of the TargetRTS.
RTStateId	short	any scalar type	Allows for the type definition of the RTActor state identifier. A smaller type may decrease the memory footprint of an executable. (e.g. unsigned char)
INLINE_CHAINS	<blank>	inline or <blank>	Inlines state machine chains for better performance at the expense of potentially larger executable memory size.
INLINE_METHODS	inline	inline or <blank>	Inlines actor methods for better performance at the expense of potentially largest executable memory size.

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 `$RTS_HOME/src/target/<target_base>`, where `<target_base>` is the target name without the 'S' or 'T'. For the remainder of this section, the target directory is referred to as **`$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 `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 or creating soft links for

the contents of these subdirectories into the $TARGET_SRC directory. You may also want to search the other target subdirectories to verify that the implementation of various TargetRTS classes resembles your target RTOS. You can copy any required code to the new $TARGET_SRC directory.

Table 6 on page 49 shows the classes and functions that must be provided in any port of the TargetRTS. These are the minimum requirements for a new port, as most ports will include changes to more classes than those listed.

Table 6 Required TargetRTS Classes and Functions

Required TargetRTS Classes and Functions
RTTimespec::getclock()
RTThread::RTThread()
RTMutex
RTSyncObject

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

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`. See the description of the `RTMain` class in the *C++ Target Guide* for more information on `entryPoint`.

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 VxWorks, 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, the arguments for `entryPoint` can be defined in the toolset. These arguments are made available by the code generator in the global variables `default_argv` and `default_argc`. The code in `main.cc` must explicitly pass these values to `entryPoint`. For more information, see "Application-specific command line arguments" on page 115 of the *C++ Language Guide*.

Class RTMain

`RTMain::entryPoint()` calls a number of methods for target-specific initialization and shutdown. For a more detailed discussion of class `RTMain`, see the description of the `RTMain` class in the *C++ Target Guide*. These methods are as follows:

- `targetStartup()` — provided in file `$TARGET_SRC/RTMain/targetStartup.cc`, it initializes the target in preparation for execution of the model. This includes things such

as setting the priority of the main thread, calling static constructors, and initializing devices, for example, timers and consoles.

- `targetShutdown()` — provided in file `$TARGET_SRC/RTMain/targetShutdown.cc`, it generally undoes the initialization that was performed in `targetStartup()`, for example, calling static destructor and cleaning up operating resources such as file descriptors.
- `installOneHandler()` — provided in file `$TARGET_SRC/RTMain/installOneHandler.cc`, it may also need to be overridden. In addition to target start-up and shutdown, `entryPoint` also installs Unix style signal handlers, where available. These signal handlers are used by the single threaded TargetRTS for timer and I/O interrupts. If a platform does not implement signals, the `RTMain::installOneHandler()` method must be overridden.

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 `struct timespec` which contains two fields: the number of seconds and the number of nanoseconds from some fixed point of time. This fixed point is usually the Universal Time reference point of January 1, 1970. This does not need to be the case. However, to support absolute time-outs, the TargetRTS assumes that the reference time is midnight of some day.

Class RTThread constructor

To support multi-threading, the TargetRTS provides the class `RTThread`. See the description of the `RTThread` class in the *C++ Target Guide* for more information. 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`. See the description of the `RTMutex` class in the *C++ Target Guide* for more information. There is no default declaration or implementation of `RTMutex` that must be supplied by the target. The header file for the `RTMutex` class should be placed in the file `$TARGET_SRC/RTMutex.h`. There are four methods to `RTMutex`:

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

Class RTSyncObject

An additional synchronization mechanism used by the TargetRTS is implemented by class `RTSyncObject`. See the description of the `RTSyncObject` class in the *C++ Target Guide* for more information. 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 `$(RTS_HOME)/src/target/sample/RTSyncObject` directory. There is no default declaration or implementation. The header file for the `RTSyncObject` should be in the file `$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 condvar is no longer required.
- `signal()` — in `$TARGET_SRC/RTSyncObject/signal.cc`. See the description of the `RTSyncObject` class in the *C++ Target Guide*.
- `wait()` — in `$TARGET_SRC/RTSyncObject/wait.cc`. See the description of the `RTSyncObject` class in the *C++ Target Guide*.
- `timedwait()` — in `$TARGET_SRC/RTSyncObject/timedwait.cc`. See the description of the `RTSyncObject` class in the *C++ Target Guide*.

Note: The `RTMutex` and `RTSyncObject` classes do not implement any type of priority inheritance protocol. If priority inheritance is supported by the target operating system then this may be added by modifying the implementation of `RTMutex` and `RTSyncObject`.

Class RTDiagStream

The `RTDiagStream` class handles output of diagnostic messages to the standard output. If your target does not support the `fputs()` function then you must supply a replacement for the `RTDiagStream::write()` function. This function outputs a string to the standard output device.

Class RTDebuggerInput

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()` function. This function 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.

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.

Class RTIOController

The `RTIOController` is the class used by the ioController thread in the external layer. It makes use of several TCP sockets calls. Problems encountered here will be similar to those described in “Class RTTcpSocket” on page 51.

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 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 for the TargetRTS. 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. Each entry has three or more fields separated by whitespace. The first names a make variable, which will include the name of the object file for that entry. The second field is a directory name. The third 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 fourth 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 file to the TargetRTS, you must have an entry in the `MANIFEST.cpp` file for the 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 class then place the entries next to the super class if it exists or next to similar classes in the manifest file.

Regenerating make dependencies

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

Note that include statements should not normally be put in areas considered conditional by the pre-processor (that is, between #if/#endif pairs). The dependency discovery script does not evaluate expressions used in preprocessor #if statements, and assumes these expressions to be true. Consequently, the dependency discovery script may capture more include statements than the preprocessor. However, although it may calculate more dependencies than the optimal amount, the dependency discovery script does detect and avoid endless loops of #include statements.

Chapter 6

Porting the TargetRTS for C

This chapter has been split into several major sections, including

- C TargetRTS configuration definitions
- Platform-specific implementation
- Adding new files to the C TargetRTS
- C TargetRTS run-time semantics
- Implementing timer services in the C TargetRTS

C TargetRTS configuration definitions

Much of the configurability of the C TargetRTS is done with compilation dependencies by having target-specific source files override common source files. This is illustrated in “Platform-specific implementation” on page 61. However, configurability is also available within source files using C 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 compiler specific definitions; this does not exist by default

These files override macros whose defaults appear in `$RTS_HOME/include/RTConfig.h`. The macros and their default values are listed in Table 7.

Table 7 C TargetRTS constants/macros and their default values

Symbol	Default Value	Possible Values	Description
RSLMULTI-THREADED	RSLFALSE	RSLFALSE or RSLTRUE	Determines whether the single-threaded or multi-threaded version of the C TargetRTS is used.
RSLDEBUG	RSLFALSE	RSLFALSE or RSLTRUE	If this flag is set to RSLTRUE, the C TargetRTS will be compiled with the debugger option (this option is also required for target observability).

Table 7 C TargetRTS constants/macros and their default values

Symbol	Default Value	Possible Values	Description
RSLDEBUG_LEVEL	0	0, 1, 2, or 3	This flag is for internal ObjecTime use only (it defines to what level internal progress messages are displayed).
RSLTO	RSLFALSE	RSLFALSE or RSLTRUE	Indicates whether the C TargetRTS will support target observability. If RSLTO is set to RSLTRUE, RSLDEBUG will forced to RSLTRUE.
RSLMEMORYALLO-CATION	RSLTRUE	RSLFALSE or RSLTRUE	If this flag is set to RSLTRUE, the C TargetRTS will perform all dynamic memory allocations by allocating smaller segments from larger segments. This uses memory more efficiently than other techniques.
RSLMEMORY-BLOCKSIZE	1024	64+	If RSLMEMORYALLOCATION is set to RSLTRUE, this is the size of the memory block that the C TargetRTS will sub-allocate smaller segments from.
RSLMEMORY-WORDBOUNDARY	4	1,2,4,8, etc.	If RSLMEMORYALLOCATION is set to RSLTRUE, this is the size, in number of bytes, which all new memory allocation requests must be aligned to.
RSLDEBUGGER-STACK	20480	(Positive Integer)	If RSLMULTITHREADED and RSLDE-BUG are set to RSLTRUE, this is the stack size of the debugger thread.
RSLTOSTACK	20480	(Positive Integer)	If RSLMULTITHREADED and RSLTO are set to RSLTRUE, this is the stack size of the target observability thread.
RSLNUM_FREE_GL OBAL_MSGS	50	(Positive Integer)	Specifies, in a multi-threaded TargetRTS, the number of free messages shared amongst all threads that are used for all thread inter-communication.
RSLTHRESHOLD_EX TERNAL_MSGS	10	(Positive Integer)	Specifies at which point a multi-threaded C TargetRTS will automatically move free external messages from the thread back to the global pool of free global messages.
RSLMESSAGE_ DEFERRAL	RSLTRUE	RSLFALSE or RSLTRUE	Indicates whether message deferral routines and queues are to be supported by the C TargetRTS.

Table 7 C TargetRTS constants/macros and their default values

Symbol	Default Value	Possible Values	Description
RSLNUM_FREE_INTERNAL_MESSAGES_FOR_PRIMARY_THREAD	50	(Positive Integer)	Indicates the number of messages allocated to the primary/top thread for a multi-threaded C TargetRTS. Requires RSLMULTITHREADED to be set to RSLTRUE.
RSLNUM_FREE_INTERNAL_TCBS_FOR_PRIMARY_THREAD	5	(Positive Integer)	Indicates the number of timer control blocks allocated to the primary/top thread for a multi-threaded C TargetRTS. Requires RSLMULTITHREADED to be set to RSLTRUE.
RSLNUM_TCBS	5	(Positive Integer)	Indicates the number of timer control blocks allocated for the C TargetRTS. Only used if RSLMULTITHREADED to be set to RSLFALSE.
RSLTIMERS	RSLTRUE	RSLFALSE or RSLTRUE	Indicates whether timers are required in the C TargetRTS.
RSLACTOR_TIMERS	RSLTRUE	RSLFALSE or RSLTRUE	Indicates whether timer services are implemented via an application-level actor. If set to RSLTRUE then RSLTIMERS must also be set to RSLTRUE.
RSLINTERNAL_LAYER_SERVICE	RSLTRUE	RSLFALSE or RSLTRUE	Indicates whether the internal layer service is supported in the C TargetRTS.
RSLMAX_SPPS	10	(Positive Integer)	If RSLINTERNAL_LAYER_SERVICE is set to RSLTRUE, indicates the maximum number of SAP/SPP registrations supported by the C TargetRTS.
RSLERROR(x)	x	x or nil	Indicates C TargetRTS errors to be printed.
RSLDEBUG2(x)	x	x or nil	Indicates Level 2 C TargetRTS progress messages are to be printed (internal ObjecTime use only).
RSLDEBUG1(x)	x	x or nil	Indicates Level 1 C TargetRTS progress messages are to be printed (internal ObjecTime use only).
RSLDEBUG0(x)	x	x or nil	Indicates Level 0 C TargetRTS progress messages are to be printed (internal ObjecTime use only).

Table 7 C TargetRTS constants/macros and their default values

Symbol	Default Value	Possible Values	Description
RSLMemorySize	unsigned long	(any scalar type)	Indicates the storage type used to store the maximum memory storage size of actor instance data.
RSLTimeoutSize	unsigned long	(any scalar type)	Indicates the storage type of an InformIn timeout request.
RSLPortIndex	unsigned short	(any unsigned scalar type)	Indicates the storage type of a port index in the C TargetRTS.
RSLMaxPort	65535	maximum value of RSLPortIndex	Indicates the maximum value for a port index in the C TargetRTS.
RSLActorIndex	unsigned short	(any unsigned scalar type)	Indicates the storage type of an ActorIndex in the C TargetRTS.
RSLMaxActor	65535	maximum value of RSLActorIndex	Indicates the maximum value for an actor index in the C TargetRTS.
RSLThreadIndex	unsigned short	(any unsigned scalar type)	Indicates the storage type of a ThreadIndex in the C TargetRTS.
RSLMaxThreads	65535	maximum value of RSLThreadIndex	Indicates the maximum value for a thread index in the C TargetRTS.
RSLMessageIndex	unsigned short	(any unsigned scalar type)	Indicates the storage type of a MessageIndex in the C TargetRTS.
RSLMaxMessages	65535	maximum value of RSLMessageIndex	Indicates the maximum value for a message index in the C TargetRTS.
RSLTCBIndex	unsigned short	(any unsigned scalar type)	Indicates the storage type of a TCBIndex in the C TargetRTS.
RSLMaxTCBs	65535	maximum value of RSLTCBIndex	Indicates the maximum value for a TCB index in the C TargetRTS.
RSLBool	unsigned short	(any unsigned scalar type)	Indicates the storage type of a Boolean in the C TargetRTS.
RSLFlags	unsigned short	(any unsigned scalar type of at least 4 bits)	Indicates the storage type of a flags field in the C TargetRTS.
RSLMessagePriority	unsigned short	(any unsigned scalar type)	Indicates the storage type for a message priority in the C TargetRTS.
RSLMaxMessagePriority	65535	maximum value of RSLMessageIndex	Indicates the maximum number of priority levels in the C TargetRTS.
RSLSignalIndex	unsigned short	(any unsigned scalar type)	Indicates the storage type for a signal number in the C TargetRTS.

Table 7 C TargetRTS constants/macros and their default values

Symbol	Default Value	Possible Values	Description
RSLSignalEvent	65535	maximum value of RSLSignalIndex	Indicates the maximum number of distinct signals supported in the C TargetRTS.
RSLActorType	unsigned short	(any unsigned scalar type)	Indicates the storage type for an actor class index in the C TargetRTS.
RSLMaxActorClasses	65535	maximum value of RSLActorType	Indicates the maximum number of actor classes supported in the C TargetRTS.
RSLStateIndex	unsigned short	(any unsigned scalar type)	Indicates the storage type for a state index in the C TargetRTS.
RSLMaxStates	65535	maximum value of RSLStateIndex	Indicates the maximum number of state indices in the C TargetRTS.
RSLPortType	unsigned short	(any unsigned scalar type)	Indicates the storage type for a port class in the C TargetRTS.
RSLMaxPortClasses	65535	maximum value of RSLPortType	Indicates the maximum number of port classes in the C TargetRTS.
RSLFieldType	unsigned short	(any unsigned scalar type)	Indicates the storage type for a field description in the C TargetRTS.
RSLMaxFieldTypes	65535	maximum value of RSLFieldType	Indicates the maximum number of field types supported in the C TargetRTS.
RSLObjectName	char *	char *	Indicates the storage type for all object names in the C TargetRTS.

Makefile fragments

Some of the TargetRTS configuration is done at the makefile level. When a developer compiles an update, the generated compilation makefile requires two files:

- `$(RTS_HOME)/target/common.mk`[1]
- `$(OVERRIDESFILE)`

The `OVERRIDESFILE` can be specified by the developer within the update's active configuration and is typically intended for temporary overrides of make macros. The definition for `OVERRIDESFILE` is provided in RTUpdate.mk and defaults to `$(RTS_HOME)/target/empty.mk`, which is itself an empty file.

1. `common.nmk` is required when using Microsoft's `nmake` utility. Slight formatting differences exist between this utility and most other make utilities. Specifically, when a make macro must be evaluated in order to process a makefile include statement, `nmake` requires the included file's filename to be enclosed in angle brackets. Hence, `include $(FILE_MK)` becomes `include <$(FILE_MK)>`. Structurally, `common.nmk` and `common.mk` are intended to be equivalent, and this document refers to the latter for brevity. No other makefile fragments are dependent upon which make utility you are using.

The makefile fragment `common.mk` is really intended as a common entry point independent of the value of `$(RTS_HOME)`. It typically includes four other makefile fragments:

```
$(RTS_HOME)/libset/default.mk
$(RTS_HOME)/libset/$(LIBRARY_SET) /libset.mk
$(RTS_HOME)/target/$(TARGET)/target.mk
$(RTS_HOME)/config/$(CONFIG)/config.mk
```

The make macro `TARGET` is defined by:

```
TARGET=$(PLATFORM)$(THREADED)
```

The make macro `CONFIG` is defined by:

```
CONFIG=$(LIBRARY_SET).$(TARGET)
```

`LIBRARY_SET`, `PLATFORM` and `THREADED` are all defined from within the update's active configuration.

Default.mk

This makefile fragment defines some common make macros. These macros are interpreted first and may be overridden in any of the three RTS-specific makefile fragments or even the update-specific Make Overrides file that will be included later. Consequently, these macros may serve one of several purposes:

- common definitions that should always work (for example, PERL=perl, CONFIG=$(TARGET).$(LIBRARY))
- common definitions that work in a majority of cases, but can be overridden depending on the compiler/library-set, target or configuration
- definitions that can suffice temporarily but are intended to be overridden depending on compiler/library-set, target or configuration (for example, VENDOR=generic)
- definitions that are intended to be overridden and will fail otherwise

In many cases, default definitions are provided that may only be appropriate for a subset of compiler/library-sets, targets or configurations.

For more information, see Table 4, "Make macro definitions," on page 40.

Libset.mk

This file is intended to list items specific to correctly linking with a particular library, and/or typically configuring the compiler. These configuration items include

- the name of (and possibly the path to) your compiler/linker
- compilation flags specific to this compiler/linker
- the vendor's name
- linker flags

Target.mk

Some targets will require further configuration without respect to which compiler is being used. For example Wind River System's Tornado targets require their linkers to be invoked with the –r option (to facilitate run-time linking).

Config.mk

In cases where a configuration item should only affect a particular library-set/target pairing, configurations can be put in the `config.mk` makefile fragment.

Availability of Perl on compilation host

If the compilation host does not have Perl (version 5.002 or greater), it is highly recommended to obtain and compile the source from http://www.perl.com.

Since obtaining Perl may take some time, some limited functionality will still be available on the compilation host:

- Dependency discovery (see `MAKEDEPEND_CMD`) must be disabled.
- Error parsing (see `OTCOMPILE_CMD` and `OTLINK_CMD`) must be disabled.
- The definition for NOP may need to be redefined to use an OS-dependent alternative instead of Perl scripts (RM, RMF and TOUCH are not typically required during compilation).

Platform-specific implementation

The implementation of the C TargetRTS is contained in the `$RTS_HOME/src` directory. In this directory, there is a subdirectory for each major run-time area. For example, the sub-directory C TargetRTS contains all of the standard run-time library functions. Other sub-directories, for example, `TCP`, `MONITOR`, `INITSTOP`, and `TRANSPRT` contain target observability functionality.

To port the C TargetRTS to a new platform, it may be necessary to update or replace some of these files. The target specific source is placed in a subdirectory of `$RTS_HOME/src/target/<target_base>`, where `<target_base>` is the target name without the 'S' or 'T'. For the remainder of this section, the target directory is referred to as **`$TARGET_SRC`**. For example, the target source directory for an example <**target**> `PSOS2T` is `$RTS_HOME/src/target/PSOS2`. This directory provides an overlay to the `$RTS_HOME/src` directory. When the C TargetRTS loadbuild tools search for a source file, 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 C TargetRTS for a given platform. C preprocessor macros, such as `RSLMULTITHREADED`, may be used to differentiate code for specific configurations.

For a single-threaded port without target observability, it is entirely conceivable that no source code modifications would be required, as the intent was to make the C TargetRTS completely portable to any ANSI C compiler, which should compile single-threaded C code without modification.

However, if the target is multi-threaded, modifications will be required to specify how certain RTOS services (for example, semaphore creation) are performed in that target environment. The remainder of this section discusses the most common required implementation code required for a new multi-threaded target.

Main function

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

The default code is likely to be suitable for a new target platform.

Target observability startup and shutdown

If any special initialization and/or shut-down code is required for target observability, the files `TGTinit.c`, `TGTstop.c`, `TOinit.c`, and `TOstop.c` which are located in the directory `$TARGET_SRC/INITSTOP` contain code to perform these functions, in conjunction with the C TargetRTS initialization and shutdown.

The default code is likely to be suitable for a new target platform.

Memory management

All memory allocation performed by the C TargetRTS (this does not apply to any dynamic memory allocation performed by the application) is performed through a single routine:

```
void *RSLAllocateMemory(RSLMemorySize size)
```

The C TargetRTS only allocates dynamic memory during its initialization phase. Thus, once the system has initialized, the C TargetRTS will not perform any dynamic memory allocations. Since the C TargetRTS does not support any implicit recovery capabilities, the C TargetRTS does not support the freeing of any dynamically allocated memory, which was allocated by it. Thus, users must exercise caution when using this memory management routine at an application level.

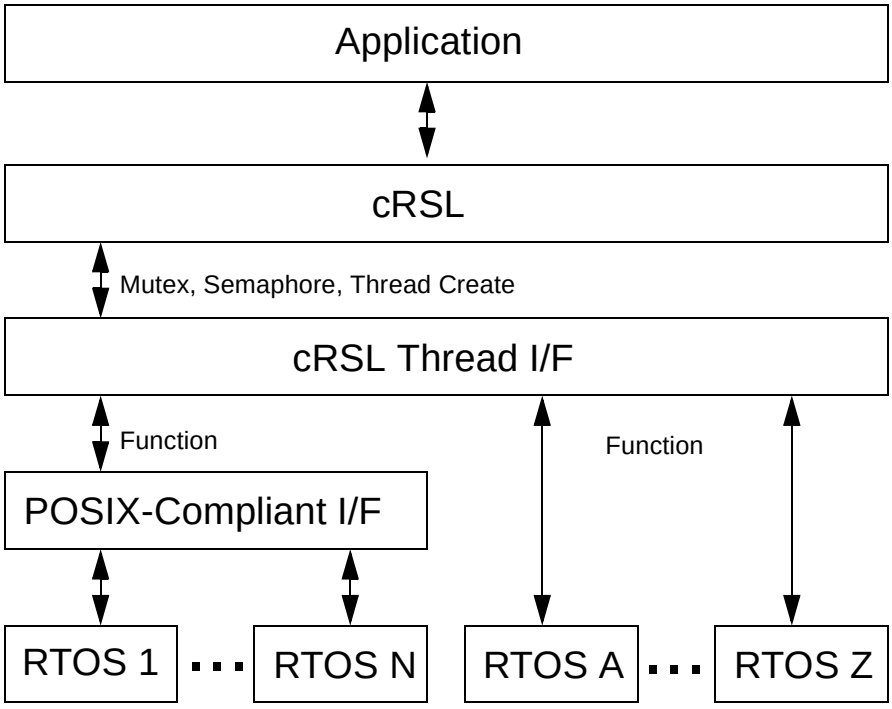
The default implementation of this routine is to invoke the standard C “malloc” run-time routine, to allocate a large chunk of memory, whereupon requests are serviced efficiently from within this block (this works particularly efficiently in certain RTOS environments where there are minimal sizes, for example, 64 bytes for *any* malloc request). If standard C run-time malloc support is not available—or alternatively, you want to have the C TargetRTS allocate memory from a specific memory region—you should modify this routine (in file `memory.c`) as appropriate for your target environment.

Multi-threaded RTOS interface

All of the RTOS interface mapping routines used by the C TargetRTS should be contained in the `RTThread.c` file located in the directory `$TARGET_SRC/target/<target>/THREAD`.

Figure 3 shows the relationships between an application, the C TargetRTS, and underlying RTOS primitives required to support a multi-threaded C TargetRTS.

Figure 3 Required relationships



RTThread.c contents

This file is split into three general sections:

- target header file include statements
- type definitions for base RTOS types
- defines for C TargetRTS macro mappings to TargetRTS

Target header files include statements

This section should specify the list of header files that are required for compilation for the TargetRTS (specifically, those header files that are required for the multi-threaded RTOS interface). An example follows:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <semaphore.h>
```

Type definitions for base RTOS types

This section specifies RTOS-equivalent types for three C TargetRTS base types:

- `RSL_mutex`

- `RSL_semaphore`
- `RSL_thread_id`

If the RTOS does not support mutexes, a mutex can be constructed via a semaphore.

```
typedef SEM_ID RSL_mutex;
typedef SEM_ID RSL_semaphore;
typedef int RSL_thread_id;
```

C TargetRTS function mappings to TargetRTS

The final section in the `RTThread.c` file is used to define a set of C TargetRTS functions providing a direct interface from the C TargetRTS to the underlying RTOS capability.

These functions are organized as:

- 3 functions related to mutex creation, locking, and unlocking:
  - `RSLmutex_init(lock)`. Given a pointer to the address "X" of a lock, create a lock, and set the address "X" to be the address of the newly created lock.
  - `RSLmutex_lock(lock)`. Given the address of a lock, lock the lock.
  - `RSLmutex_unlock(lock)`. Given the address of a lock, unlock the lock.
- 3 functions related to semaphore creation, waiting and posting:
  - `RSLsemaphore_init(semaphore)`. Given a pointer to the address "X" of a semaphore, create a semaphore as initially signalled (or with a count of 1), and set the address "X" to be the address of the newly created semaphore.
  - `RSLsemaphore_wait(semaphore)`. Given the address of a semaphore, wait for the semaphore to be available (signalled, or count >= 1).
  - `RSLsemaphore_post(semaphore)`. Given the address of a semaphore, post the semaphore as signalled, or increment the semaphore count.
- 1 function related to thread creation:
  - `RSLthr_create` (index,priority,stack,stackSize,entryPoint,args). Given the task/thread instance integer number (0 through n-1 threads), task/thread priority, task/thread stack address, task/thread stack size, task/thread entry point, and task/thread arguments, create a thread and assign the returned thread id to the C TargetRTS thread description array `RSLRTThreadInstances[index].threadId`.

A suitable definition of a complete RTThread.c file for a Tornado 1.0.1 target might appear as shown in the extract below:

```
#include <RTThread.h>
#include <crsl.h>

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <semaphore.h>
```

```
typedef SEM_ID RSL_mutex;
typedef SEM_ID RSL_semaphore;
typedef int RSL_thread_id;

extern RSLRTThreadInstanceDescription   *RSLRTThreadInstances;

int
RSLmutex_init( void **lockptr_ptr ) {
    *lockptr_ptr = semCCreate( SEM_Q_FIFO , 1 );
    return 1;
}

int
RSLmutex_lock( void *lockptr ) {
    return semTake( lockptr , WAIT_FOREVER );
}

int
RSLmutex_unlock( void *lockptr ) {
    return semGive( lockptr );
}

int
RSLsemaphore_init( void **semaphoreptr_ptr ) {
    *semaphoreptr_ptr = semBCreate( SEM_Q_FIFO , SEM_FULL );
    return 1;
}

int
RSLsemaphore_wait( void *semaphoreptr ) {
    return semTake( semaphoreptr , WAIT_FOREVER );
}

int
RSLsemaphore_post( void *semaphoreptr ) {
    return semGive( semaphoreptr );
}

int
RSLthr_create( int index , int priority , void *stack , int stackSize ,
               void *entryPoint , void *args ) {

    RSLRTThreadInstances[index].threadId =
      taskSpawn( NULL , priority , 0 , (stackSize) , entryPoint , args ,
                  0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 );
}
```

Adding new files to the C TargetRTS

If you create a new file for the C TargetRTS, then you must add the new file name to a manifest file for the C TargetRTS. This must be done in order for the dependency calculations to include the new file and thus include it into the C TargetRTS.

The MANIFEST.c file

This file lists all the elements of the run-time system. There is one file entry per line. Each entry has three or more fields separated by whitespace. The first field names a make variable that specifies which library to associate the file with. The second field is a directory name. The third field is the base name of a file. The fourth 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.

Regenerating make dependencies

If a file has been overridden in `$(RTS_HOME)/target/src/<target_name>` directory or a new file has been added to the `MANIFEST.c` you must regenerate the make dependencies in order for the modification to be included in the new TargetRTS. This is done by removing the `depend.mk` file in the build directory (`$(RTS_HOME)/build-<platform_name>`). This causes the dependencies to be recalculated and a new `depend.mk` file to be created. Note that include files must not be enclosed in `#ifdef/#endif` or similar macros because the preprocessor may not have the needed definitions to satisfy these macros. Consequently, the include file may be ignored during dependency checking.

C TargetRTS run-time semantics

Tasks, processes, and threads

The C TargetRTS does not distinguish between a task, process, or thread. In fact, some RTOSs—which support multiple tasks—represent a thread *as* a task. Instead, the C TargetRTS requires the simple notion of a “thread of execution”, which (depending upon the customer RTOS capability) might be a thread, a task, or a process. However, the C TargetRTS does require that if multiple threads of execution are required, they share a common “shared” memory space.

If multiple threads are supported in the target environment, the C TargetRTS, in conjunction with the ObjecTime toolset, provides the customer with the ability to split up an application into multiple threads. The decision to split the application into multiple threads of execution is an important architectural decision. The C TargetRTS provides simple capabilities to define which top-most ObjecTime actors are to be assigned to which threads. Thus it is possible to “code-generate” different thread topologies *easily*, without necessarily modifying any user code in the ObjecTime model.

The mainloop

Currently, most systems are comprised of one or more separate threads of execution. Each of these threads typically has a main routine that is responsible for servicing the IPC (Inter-Process Communication) messages or signals that are received by that system. These IPC routines are usually blocking RTOS IPC calls; that is, they do not return execution to the main routine until a message/signal has been

received. When a message/signal is received, an appropriate subroutine is invoked to handle the actual processing associated with the arrival of that particular message/signal. This message processing occurs for the duration of the execution of the system.

Similarly, each thread of execution in the C TargetRTS also has a "mainloop". This is in recognition of the fact that the main C TargetRTS routine processes many IPC messages, in a serial fashion, via a looping mechanism. After receiving each external message, it processes it, that is, it "walks' the state machines of one or more actors.

The following pseudo-code describes this notion of a main-loop:

```
<Initialize IPC>
<Do Forever>
{
   <Process Actor Messages>
   <Block for IPC Message>
}
```

To support user-defined messaging interfaces, each of these threads of execution requires a application-specified routine, which in conjunction with the C TargetRTS mainloop, implements how that particular task is to interface to the user's IPC. It is possible to have many different interface routines—a different one for every thread. It is also possible for different threads to use *different* IPC mechanisms. Note that a thread blocked waiting for external messages from the IPC will *not* unblock to process actor messages.

RSLThreadMap

In an ObjecTime design, the top-most actor is assigned its own thread; in fact, by default, *every* program that is run has at least one thread of execution. Unless directed otherwise by that top-most actor, all contained actors will also "run" on that same thread.

However, if the user application is multi-threaded (that is, at least two or more threads of execution), then these additional threads are *contained* by this top-level actor. These contained actors are assigned a different thread at run-time through the top-level actor's function RSLThreadMap, which is a "special" function, with a non-C specification syntax; that is, it cannot be compiled by a C compiler. It is called a "special" function, since there is special code-generation handling the particular function name of "RSLThreadMap".

This RSLThreadMap function specifies the run-time characteristics of each of the threads, except the top-thread. The top-thread's characteristics are *not* defined by the C TargetRTS, as the user's RTOS specifies how the first thread of the C TargetRTS is to be started (and with which characteristics, for example, priority, stack size, and so forth).

The RSLThreadMap, in conjunction with the logical/physical thread configuration stipulated in the ObjecTime toolset (see the *C Language Guide* for additional information), specify which logical threads are mapped to which physical threads. In those instances where multiple logical threads are mapped to a single physical thread, the resource requirements of the physical thread are determined by summing all

of the resource requirements of each of the logical threads together, that is, the number of internal messages and the number of timer control blocks. Each logical thread name specified in the RSLThreadMap must have a corresponding logical thread definition in the thread configuration browser in the ObjecTime toolset.

The characteristics that can be specified, on a per thread basis (for more information see “C TargetRTS constants/macros and their default values” on page 55), include

- Number of Internal Messages (Optional)

  The maximum requirements for inter-actor messages used *within* the thread. If not specified, it defaults to 0, which implies that the thread intends to do no internal messaging between actors in the same thread (probably unlikely).

- Number of Timer Control Blocks (Optional)

  The maximum number of active timers that can be active at one time *within* the thread. If not specified, it defaults to 0, which implies that no actors contained in that thread intend to start any timers.

Finally, the RSLThreadMap serves to provide an association between an actor reference in the top actor (that is, the name of an actor in the structure diagram of the top actor in the system), and the logical thread name, as specified in the ObjecTime toolset in the Threads Browser.

For example, in the example RSLThreadMap below, two threads are defined. ‘sender’ and ‘replyer’ are the names of two actor references contained in the top-most actor. Each of these actor references are associated with their respective logical threads (which, in turn, are mapped to physical threads within the toolset). In addition, each thread has been defined to be created with five internal messages and no TCBs (Timer Control Blocks).

```
/*
thread SenderLogicalThread {
      NumberOfInternalMessages 5;
      NumberOfTCBs             0;
} sender;
thread ReplyerLogicalThread {
      NumberOfInternalMessages 5;
      NumberOfTCBs             0;
} replyer;
*/
```

Message priority

The C TargetRTS message priority, defines six RSL levels of priority (from highest to lowest):

- RSLDEBUGGER (internal C TargetRTS use only)
- RSLPANIC
- RSLHIGH
- RSLGENERAL
- RSLLOW

- RSLBACKGROUND

These message priorities are used by the internal C TargetRTS scheduler to dispatch the highest priority messages (for example, Panic) before lower priority messages (for example, General) are processed. Without the use of message priority, all messages are treated equally, in that all messages are processed FIFO (First-In, First-Out).

Often, indiscriminate use of message priority unnecessarily increases the complexity of actor behavior, since the sequence of events now also depends on message priority.

Message queues

Each thread of execution has six incoming internal message queues (five to be used by customers), one for each message priority, as defined in the previous section.

At run-time, based on the specified SAP/Endport in a message send request, the C TargetRTS determines whether the recipient of a message is within the same thread as the sender, or alternatively, is in a different thread. Once this is known, the C TargetRTS automatically selects an appropriate message enqueueing technique for the message send.

Single-threaded C TargetRTS

In the single-threaded C TargetRTS, there is no distinguishing run-time characteristic made between “internal” and “external” C TargetRTS messages, since there is only one thread of execution.

Thus, in a single-threaded C TargetRTS, there is

- one set of internal message queues
- one free message queue/free-list

The C TargetRTS mainloop will process the messages that are on the internal message queues (more about this in a subsequent section).

When an actor transition is fired that contains one or more requests to send messages to another actor, as indicated in Figure 4, the following steps are performed for each such message send:

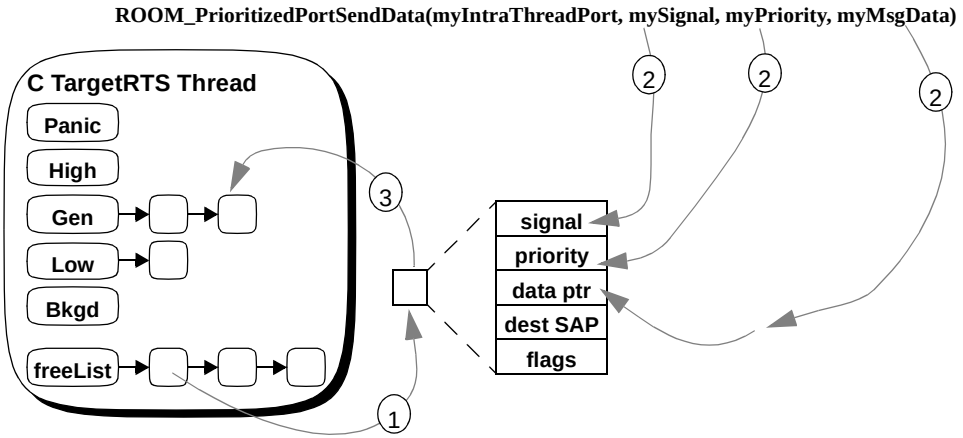
1 A “free” message is allocated from the free message queue (also referred to as the ‘free list’).

If no message is available, a false result (a value of 0) is returned by the send function. Under the assumption that the port has been bound, this is the only error that the C TargetRTS can detect on a send.

2 The message’s structure fields are populated with the parameters passed in the send request (the ‘flags’ and ‘dest SAP’ fields are set automatically by the C TargetRTS send routine).

3 The message is enqueued for internal processing at the appropriate priority.

Figure 4 Intra-thread inter-actor send



Multi-threaded C TargetRTS

In the multi-threaded C TargetRTS, the C TargetRTS distinguishes between “internal” and “external” C TargetRTS messages. Internal messages are inter-actor intra-thread messages, where each actor runs in the *same* thread of execution. External messages are inter-actor inter-thread messages, where the sending and receiving actors run in a *different* thread of execution.

In the multi-threaded C TargetRTS, each thread has

- one external incoming message queue (for receiving inter-actor, inter-thread messages)
- one set of internal message queues
- one internal free message queue (for inter-actor, intra-thread)
- one external free message queue (for inter-actor, inter-thread—one actor in one thread)

In addition, in the multi-threaded C TargetRTS, there is a global free message queue.

As discussed in “Message processing” on page 72, the multi-threaded C TargetRTS:

- moves messages from the external incoming queue to the internal queue
- invokes the behaviors associated with the arrival of those messages (which may involve the processing of both external and internally generated messages)
- eventually returns external messages to the external message queue (global) and internal message queue (local)

The previous section described the C TargetRTS implementation of an intra-thread inter-actor send. In the multi-threaded C TargetRTS, all intra-thread sends are performed in the same manner.

Inter-thread sends, however, are substantially different, in that they require processing as indicated in Figure 5, with the following numbered steps:

1 A free message is popped from the intended receiver’s external free queue/list, if available (1A).

If none are available, a free message is popped from the sender's external free queue (1B).

Once again, if not available, a free message is popped from the global free queue (1C).

If that also fails, the C TargetRTS will then move all free external messages from every thread and put them on the global free queue (then it tries to pop from the global queue again).

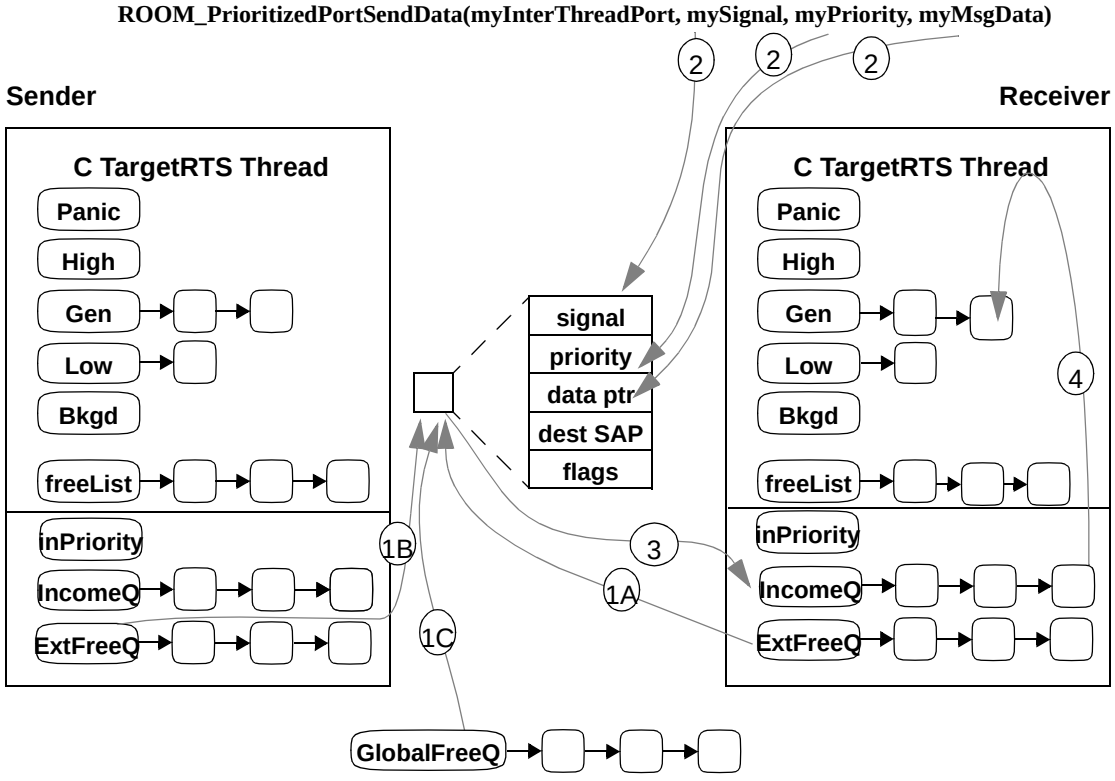
Finally, if no message is available, a false result (a value of 0) is returned by the send function. Under the assumption that the port has been bound, this is the only error that the C TargetRTS can detect on an inter-thread send.

2 The message data fields are filled in with the passed parameters in the send request.

3 The message is enqueued on the receiving thread's external incoming message queue, the receiving thread's 'InPriority' is checked and updated to the current message priority (if used, and if the current message priority is *higher* than the current posted 'InPriority'), and the thread's semaphore is posted 'ready to run'.

4 TargetRTS then processes a message in its internal queue, and dispatches the appropriate actor behavior.

Figure 5 Inter-thread sends



The C TargetRTS ensures system integrity by locking a global mutex prior to accessing any global shared data (shared amongst multiple threads), and releasing the lock once the shared data updates have been completed.

The 'InPriority' field is used by a thread to determine at what point an external incoming message is of *higher* priority than that which is currently being processed internally. Thus, this allows the C TargetRTS to "interrupt" normal actor message processing (on a per-message basis) and service a higher priority actor message. Note, however, that this does not mean that the processing of one or more user code transitions is interrupted, as new messages are selected for processing only *after* the processing of the behavior of the current message has been completed.

Message processing

Single-threaded C TargetRTS message processing

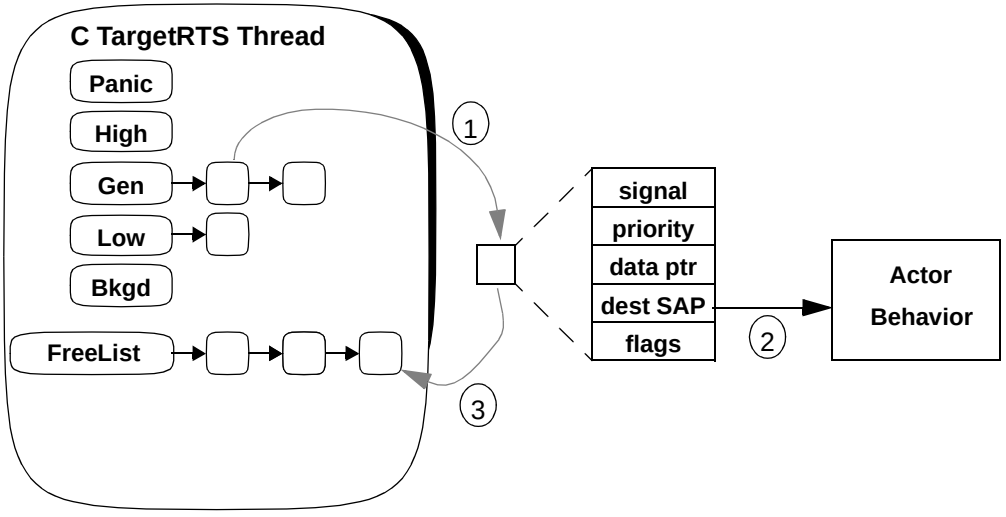
As indicated below, in Figure 6, single-threaded C TargetRTS actor message processing is implemented by:

1 Selecting the highest priority queued message.

2 Invoking the actor behavior associated with the arrival of that signal and data.

3 "Returning" the used message frame to the free message Queue.

Figure 6 Single-threaded C TargetRTS message processing



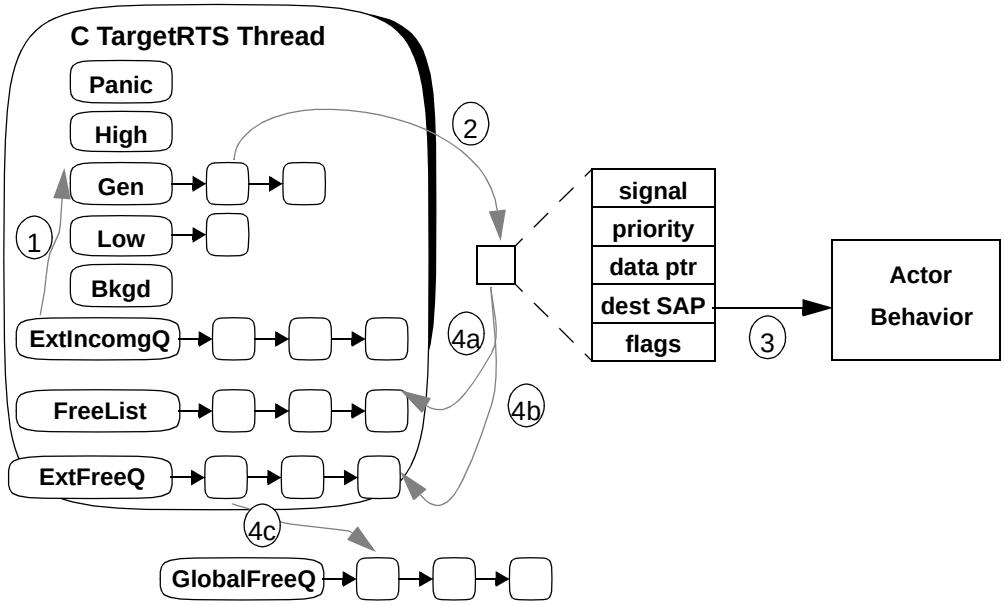
If the invocation of actor behavior causes one or more messages to be sent to other actors, the C TargetRTS mainloop may continue to invoke actor behaviors until there are no further messages to be processed. By default, all of these messages are processed prior to the C TargetRTS directing the invocation of the user-registered external interface (IPC) routine.

Multi-threaded C TargetRTS message processing

As indicated in Figure 7, multi-threaded C TargetRTS actor message processing is implemented by:

1. Checking if a higher-priority (as compared to internal messages) externally queued message exists, all external messages are dequeued from the external message queue and enqueued on the internal message queue.
2. Selecting the highest priority queued internal message.
3. Invoking the actor behavior associated with the arrival of that signal and data.
4. "Returning" the used message frame to the free message Queue. If the message was internal, it is put back onto the internal free queue (4a). If the message was external, it is put back on its external free queue (4b), and if a certain message count threshold is exceeded, it (along with other freed messages) may be put back (4c) on the global external free message queue. (See "C TargetRTS constants/macros and their default values" on page 55 for the definition of RSLTHRESHHOLD_EXTERNAL_MSGS.)

Figure 7 Multi-threaded C TargetRTS message processing



If the initial invocation of actor behavior causes one or more messages to be sent to other intra-thread actors, the C TargetRTS mainloop may continue to invoke actor behaviors until there are no further messages to be processed. By default, if both intra-thread and inter-thread messages are enqueued, all of these messages are processed prior to the C TargetRTS invoking the user-registered external interface/IPC routine.

Run-to-completion

This default "run-to-completion" C TargetRTS message scheduling policy is implemented in the C TargetRTS mainloop as

```
while(RSLDispatch(RSLBACKGROUND));
```

where "RSLBACKGROUND" represents the *lowest* priority message to be processed. The RSLDispatch routine is responsible for invoking the behavior associated with the single highest-priority message on the input queue. It returns true if a message was processed, or false if no internal messages were available for processing at the specified, or higher priority.

Thus, the "while" clause causes the C TargetRTS to continue processing these "internally" queued messages, until no further messages are enqueued, at which point, the C TargetRTS will call the user-defined IPC interface routine to block for external stimuli (for example, IPC messages, timeouts, and so forth). If the C TargetRTS is multi-threaded, the C TargetRTS will also check the incoming external queue for higher priority messages than those in the internal message queue. If there are higher priority external messages, these messages are moved to the internal queue, and C TargetRTS actor message processing continues.

If no user-defined IPC routine was specified, the multi-threaded C TargetRTS will wait until its 'ready-to-run' semaphore has been signalled by another thread (for example, an inter-thread inter-actor message send). In the single-threaded C TargetRTS, if there is no other source of potential inputs, the thread will exit.

Also, in the multi-threaded C TargetRTS, an external C TargetRTS message is moved from the thread's external incoming message queue to its internal message queue (based on priority). At that point, message processing within the thread proceeds in a similar fashion in both the single and multi-threaded C TargetRTS.

RSLRegisterExternalInterface

The C TargetRTS was designed so that it requires no inherent notion of how customers' IPC mechanisms work. Instead, it provides the "internal" C TargetRTS actor message processing portion of the mainloop, and allows customers to register their own IPC external interface routines, which are invoked by the C TargetRTS once no further internal message processing is possible. Note that a blocked thread waiting for external messages from the IPC will *not* unblock to process any new actor messages.

It is possible to have many different interface routines—a different one for every thread—and it is possible for different threads to use *different* IPC mechanisms.

The C TargetRTS provides support for each thread to register its own interface function by using the following routine to dynamically perform this registration, an example of which might appear as

```
RSLRegisterExternalInterface(_actor,MyActorIPCInterface);
```

The first parameter '_actor' of type `RSLActorIndex` should be set to the internal C TargetRTS variable automatically defined for all actor transitions. 'MyActorIPCInterface' would be the name of the user-defined actor function, which implements how the thread interfaces to IPC and/or timers.

Alternative implementations can be contemplated whereby an actor implements the IPC interface directly in the actor behavior, but this technique would likely require that this IPC actor run with the lowest priority messaging to ensure that all internal actor messages have been processed before the IPC actor blocks on an IPC call.

`RSLRegisterExternalInterface` is used for all IPC and timer implementations. For example, the code fragment shown below was taken from an actor's initialization transition. Once this initialization transition is executed, the thread that the actor is executing in will have a registered external interface routine called 'ExternalInterfaceRoutine'.

```
RSLRegisterExternalInterface(_actor,ExternalInterfaceRoutine);
printf("Example thread: Alive!\n");
```

RSLRegisterMessageSignallingInterface

The C TargetRTS was designed to support different types of IPC and timing mechanisms, suitable for a particular target environment. In some instances, it may be necessary to have the C TargetRTS signal another C TargetRTS thread that it is in the process of sending that thread a message (for example, to interrupt a blocking RTOS function called from within a registered external interface function).

The C TargetRTS provides support for each thread to register a signalling function via the following example:

```
RSLRegisterMessageSignallingInterface(_actor,
                    &MyCV,MySignallingInterface);
```

The first parameter '_actor' of type RSLActorIndex should be set to the internal C TargetRTS variable automatically defined for all actor transitions. The second parameter is the address of the RTOS object to be signalled. "MySignallingInterface" is the name of a user-defined actor function, which would be responsible for signalling the passed RTOS object MyCV.

At run-time, when an inter-thread message is delivered to a thread that had previously registered a signalling interface routine, the C TargetRTS calls the specified procedure, along with the address of the specified RTOS object, via the pre-registered procedure variable.

Enqueueing external events

To allow customer-registered external interface/IPC routines to enqueue messages that represent the arrival of an external IPC message or a timeout event (discussed later), the C TargetRTS provides the following routine:

```
RSLMessage *RSLPortEnqueue(RSLActorIndex _actor,
                    RSLPortIndex portOffset,
                    RSLSignalIndex signal,
                    RSLMessagePriority priority,
                    void *data);
```

- '_actor' is automatically defined and passed for every application code transition, and this value should be passed to this routine without modification.
- 'portOffset' is the name of the actor end-port or SAP that you want to receive the message on.
- 'signal' is the desired signal name from the protocol SAP for the end-port or SAP. If the signal is a timeout, 'RSLTIMEOUT' should be used (this signal name is automatically defined). An input signal is required since it is an un-bound port/SAP, and the message is being received (as input), as opposed to being sent (as output) from what typically would be the other end of a port binding.
- 'priority' (if used) defines the message priority (for example, RSLGENERAL).
- 'data' is a 32- or 16-bit value (depending on the size of an int for the processor), which is used to identify the accompanying data (if any) with that signal.

Once a message is enqueued in this manner, control should be returned from the registered external interface routine to the C TargetRTS, which is responsible for dispatching the message. In fact, if there is no message enqueued after the registered external interface function returns, it is likely that the thread will suspend indefinitely (or if single-threaded, possibly exit).

RSLPortEnqueue is used for all IPC and timer implementations.

RSLGetFirstTimeout

This C TargetRTS function is used to get the RTTimerId (same as RSLTimerReference) of the first timer control block on the sorted list of timers for the thread associated with the specified actor. An example follows:

```
RSLTimerReference RSLGetFirstTimeout(_actor)
```

RSLGetFirstTimeout is used for all timer implementations (RSLTIMERS set to RSLTRUE).

RSLCancelTimer

This C TargetRTS function is used to cancel a pending timer request by marking the timer control block associated with that timer reference as cancelled. Thus, at a later time, the timer control block could be released and reused.

```
RSLBool RSLCancelTimer(RSLActorIndex,RSLTimerReference);
```

RSLCancelTimer is used for all timer implementations (RSLTIMERS set to RSLTRUE).

RSLDequeueTimer

Since the TCBs are managed by one or more actors or registered actor functions (as opposed to automatic control via the C TargetRTS), the C TargetRTS also provides a routine to dequeue a timer, and possibly free the timer control block:

```
RSLDequeueTimer(_actor,&TimerControlBlock,Free)
```

The first parameter '_actor' indicates which context we are executing. The second parameter supplies the address of the timer control block. The final parameter is a Boolean value that instructs the C TargetRTS to return the timer control block to the original user thread.

RSLRegisterTimerServices

The C TargetRTS supports actor-based implementations of timing services via the following routine:

```
RSLRegisterTimerServices(_actor,sap,
                    &MyCV,MySignallingInterface);
```

The first parameter '_actor' of type RSLActorIndex should be set to the internal C TargetRTS variable automatically defined for all actor transitions. The second parameter is the SAP upon which service requests are to be received. The third parameter is the address of the RTOS object to be signalled. "MySignallingInterface" is the name of a user-defined actor function, which would be responsible for signalling the passed RTOS object MyCV.

RSLRegisterTimerServices is only used for actor timer implementations (RSLTIMERS set to RSL-TRUE; RSLACTOR_TIMERS set to RSLTRUE).

RSLGetTimerServiceActor

This C TargetRTS function is used to get the run-time actor Id of the timer actor that has registered as the provider of all timing services for the C TargetRTS. An example invocation follows:

```
RSLActorIndex RSLGetTimerServiceActor(_actor)
```

RSLGetTimerServiceActor is used for timer actor implementations (RSLTIMERS set to RSLTRUE; RSLACTOR_TIMERS set to RSLTRUE).

Implementing timer services in the C TargetRTS

Often, timing events are required by system tasks to determine that some function was not completed in a timely manner, or to schedule activities. Depending upon the customer's application requirements and RTOS timing capabilities, two different timer implementation approaches can be considered:

- Local timers, where each thread implements their timing functions *locally* within that thread, in conjunction with an application registered external interface routine. This is the simplest and most efficient implementation of timers, and should be considered for all RTOS implementations. No special timing actors/packages are required.
- Actor timers, where one actor registers that it is the provider of timing services for all other C TargetRTS actors. Note that you may not have multiple "timer" actors. In a multi-threaded C TargetRTS, this "timer" actor must be assigned its own thread of execution. This is the approach used for C TargetRTS implementations in all of the toolset development environments. A special timing actor/package is required.

These two approaches are independent implementations, and the techniques used in one approach are **not** compatible with the other. Hybrid solutions should be avoided. Once again, primarily for efficiency and size considerations, it is recommended that all C TargetRTS users use integrated timers in an RTOS environment.

Local timers

Local timers implement timing services on a per-thread basis by using an RTOS capability to block for some sort of a signal/message *for a specified period of time*. Thus, once the RTOS primitive returns exe-

cution, a return code can be checked to see if either a signal/message was received or the RTOS call timed out.

Typically, the blocking RTOS call that forms the basis for implementation for local timers is either a message read request (for example, on a message queue), or a signal wait request (for example, on a semaphore). By specifying an appropriate timeout parameter each time this RTOS function is invoked, a simple and efficient timing mechanism can be implemented. Since this same approach can be used in multiple threads, each thread is responsible for managing its external interface (message queue, semaphore, and so forth) and timer requests.

By using the RSLRegisterExternalInterface C TargetRTS API function, it is possible to define an RTOS-specific actor interface function, which is registered by an actor executing in each thread. These registered functions would be responsible for implementing timer services for all actors in that thread, and possibly interfacing to a proprietary RTOS IPC (Inter-Process Communication) mechanism.

Since the C TargetRTS provides an efficient inter-thread inter-actor messaging mechanism via port bindings, if an actor provides a proprietary RTOS IPC interface, it is likely that it will interfere with normal C TargetRTS inter-thread communication. This is because it is not possible to service multiple communication mechanisms simultaneously (proprietary C TargetRTS communication and RTOS IPC), unless you consider the use of a signalling function. For more information, see “RSLRegisterMessageSignallingInterface” on page 75.

With the local timer approach, there are two inter-thread messaging alternatives:

- In those instances where it is desirable that the C TargetRTS handle all inter-thread message communication via actor port bindings, the external interface routine must *only* handle timing requests, unless you consider the use of a signalling function. This approach is called the “Integrated Timers” implementation, and it is described in “Integrated timers” on page 79. For more information, see “RSLRegisterMessageSignallingInterface” on page 75.
- In those instances where it is desirable that inter-thread native RTOS messaging be used (not using actor port bindings), the external interface function must handle both the proprietary RTOS IPC *and* timing requests. This approach is called the “Integrated IPC and Timers”, and it is described in “Integrated IPC and timers” on page 83.

Since there is a tremendous advantage in having the C TargetRTS manage all inter-thread inter-actor messaging—that is, you can change thread topologies easily and the software is likely to be more portable—the first alternative “Integrated Timers” is recommended. However, if there is a requirement to service different types of IPC (for example, in a legacy system), it is still possible to provide actor interface functions to these threads, which are responsible for providing the proprietary RTOS IPC interface and forwarding requests (via port bindings) on to other C TargetRTS threads. Thus, in some system environments, it is possible that a *mix* of approaches may be needed to satisfy all requirements.

An example of both of these types of implementations can be found under `$(OBJECTIME_HOME)/ModelExamples/C`, in an update called “C_TornadoQueuesWithTimers”. A detailed description of this update can be found in “Compliance Suite & Examples” on page 219 of the *C Language Guide*. This update should be used as a basis for implementing either Integrated Timers or Integrated IPC and Timers.

Integrated timers

`RTTarget.h` should specify that RSLTIMERS is set to RSLTRUE; RSLACTOR_TIMERS is set to RSLFALSE.

No special timing actor/packages are required.

Two actor functions must be implemented.

- A signalling function, which is passed a pointer to an RTOS object (for example, a semaphore), and will signal it in a suitable manner (for example, posting a semaphore).
- An external interface function, which is passed a pointer to the object instance data and the current executing actor Id.

At initialization time, an actor in each thread that requires the use of integrated timers must:

- Register the external interface (timing routine) function via the RSLRegisterExternalInterface C TargetRTS API function (an example pseudo-code for this function appears below).
- Create and initialize a suitable RTOS object (for example, semaphore, or possibly an unused message queue/mailbox), storing the handle to this RTOS object as an actor ESV.
- Register the signalling function with the C TargetRTS via the RSLRegisterMessageSignalling-Interface C TargetRTS API function, which the C TargetRTS will invoke whenever it is about to deliver an inter-thread inter-actor message to that thread.

An example of this type of timer implementation can be found under `$(OBJECTIME_HOME)/ModelExamples/C`, in an update called “C_TornadoQueuesWithTimers”. Specifically, the actors “Sender” and “Replyer” in this update implement this form of timer. A detailed description of this update can be found in “Compliance Suite & Examples” on page 219 of the *C Language Guide.*

The relevant transitions and functions which adhere to the above guidelines are shown below for the sender actor:

```
/* Sender -- Initialize transition */

this->MySemaphore=semCCreate(0,0);
RSLRegisterExternalInterface(_actor,SenderMainloop);
RSLRegisterMessageSignallingInterface(_actor,
         (void*)this->MySemaphore,SenderSignalRoutine);
printf("Sender thread: Alive!\n");
ROOM_InformIn(MyTimer,RSLTARGET_TIME(100));
ROOM_PortSend(inOut,InOutSignal);

/* Sender Functions */

   /* Signalling Function */
void SenderSignalRoutine(SEM_ID semaphore) {
   semGive(semaphore);
}
```

```
   /* External Interface Routine */
void SenderMainloop(Sender_InstanceData *this,
         RSLActorIndex _actor) {
   unsigned longcurrent;
   int MsgQReceiveResult;
   RSLTimerControlBlock*tcb;
   RSLTimerReferencetr;

   /* Get the current time and check to see if the lowest timer
         value has already expired */
   current=tickGet();
   tr=RSLGetFirstTimeout(_actor);
   tcb=tr.tcb;
   if((tcb)&&(current>=tcb->timeoutNSec)) {
         tcb->timeoutMessage=RSLPortEnqueue(_actor,
               MyTimer,RSLTIMEOUT,RSLGENERAL,(void *)0);
         RSLCancelTimer(_actor,tr);
         return;
   }

   /* Block for any message on the queue with a timeout
         (# clock ticks from now) */
   if(tcb)
         MsgQReceiveResult=semTake(this->MySemaphore,
               tcb->timeoutNSec-current);
   else
         MsgQReceiveResult=msgQReceive(
               this->MySemaphore,WAIT_FOREVER);
   if((MsgQReceiveResult==ERROR)&&
         (errno=S_objLib_OBJ_TIMEOUT))  { /* Timeout */
         RSLPortEnqueue(_actor,MyTimer,
               RSLTIMEOUT,RSLGENERAL,(void *)0);
         RSLCancelTimer(_actor,tr);
   }
}
```

The relevant transitions and functions that adhere to the above guidelines are shown below for the replyer actor:

```
/* Replyer -- Initialize Transition */

this->MySemaphore=semCCreate(0,0);
RSLRegisterExternalInterface(_actor,ReplyerMainloop);
RSLRegisterMessageSignallingInterface(_actor,
         (void *)this->MySemaphore,ReplyerSignalRoutine);
```

```
printf("Replyer thread: Alive!\n");
ROOM_InformIn(MyTimer,RSLTARGET_TIME(100));

/* Replyer -- Functions */

   /* Signalling Function */
void ReplyerSignalRoutine(SEM_ID semaphore) {
   semGive(semaphore);
}

   /* External Interface Function */
void ReplyerMainloop(Replyer_InstanceData *this,
         RSLActorIndex _actor) {
   unsigned longcurrent;
   int MsgQReceiveResult;
   RSLTimerControlBlock*tcb;
   RSLTimerReferencetr;

   /* Get the current time and check to see if the lowest timer
         value has already expired */
   current=tickGet();
   tr=RSLGetFirstTimeout(_actor);
   tcb=tr.tcb;
   if((tcb)&&(current>=tcb->timeoutNSec)) {
         tcb->timeoutMessage=RSLPortEnqueue(_actor,
               MyTimer,RSLTIMEOUT,RSLGENERAL,(void *)0);
         RSLCancelTimer(_actor,tr);
         return;
   }

   /* Block for any message on the queue with a timeout
         (# clock ticks from now) */
   if(tcb)
         MsgQReceiveResult=semTake(this->MySemaphore,
               tcb->timeoutNSec-current);
   else
         MsgQReceiveResult=msgQReceive(this->MySemaphore,
               WAIT_FOREVER);
   if((MsgQReceiveResult==ERROR)&&
         (errno=S_objLib_OBJ_TIMEOUT))  { /* Timeout */
         RSLPortEnqueue(_actor,MyTimer,
               RSLTIMEOUT,RSLGENERAL,(void *)0);
         RSLCancelTimer(_actor,tr);
   }
}
```

The external interface function pseudo-code for the two actors' code fragments (sender and replyer) would be defined as:

```
<GetFirstTimeout>
<GetCurrentTime>
<If (ValidTimer && TimerHasTimedOut)> {
   <EnqueueTimeoutEvent>
   <RemoveTimerRequest>
   return;
}
<If ValidTimer>
   <BlockForSignalWithTimeout>
else
   <BlockForSignalWithInfiniteTimeout>
<IfTimedOut> {
   <EnqueueTimeoutEvent>
   <RemoveTimerRequest>
}
```

The first block of pseudo code determines if (during the intervening period of time while the system was processing) any timeouts would have occurred. If timeouts have occurred, timeout events are generated, and the time requests are removed from the timer list.

If no timeout occurred, the next block of code will await for an external RTOS signal while specifying a timeout value (which is our next shortest time to a timeout). Once control is returned to the user's IPC routine, a check is then done to determine if the blocking RTOS request timed-out (at which point, a timeout event is enqueued, and the timer request is removed from the timer list).

These timers require the use of relative time (for the informIn request), and absolute time to determine when a timer is to expire (for example, clock ticks). Thus, the use of a macro (for example, RSLTarget_Time) might be used to ensure compatibility between SimulationRTS and the C TargetRTS running on a target. Specifically, the target version of `RSLTarget_Time` should be defined to add the current number of clock ticks to the relative time to create a future absolute time, which is used by the external interface routine.

```
e.g. #define RSLTarget_Time(x) (clockTicks()+x)
```

Timers are started when an actor in that thread starts a timer via `ROOM_InformIn`. In the C TargetRTS, the timer start routine will allocate a local thread TCB, populate it with appropriate data, and then insert it into the list of active timers for that thread, in sorted order. Once enqueued, the registered external interface for that routine will process the timeout requests, as described in the above pseudo-code.

If a thread is currently blocked, waiting for a timeout to occur, and if another thread sends a message to that thread, the thread will be interrupted, since the sending thread will invoke the previously registered

signalling function in the receiving thread. Once that message is processed, the external interface function will once again be invoked by the C TargetRTS.

Timers are cancelled via `ROOM_CancelTimer`. In the C TargetRTS, the timer cancel routine will mark the RTTimerID (which points to a timer control block) as cancelled, and thus will be ignored by external interface function, when it actually times out.

The number of timers that can be started is configured in the RSLThreadMap. For the first/top thread, the number of timers to be created at initialization time should be configured in `RTTarget.h`, overriding the defaults in `RTConfig.h`.

Each thread has a set of internal TCBs available for its exclusive use. TCBs are not shared amongst threads, and are to be modified by the C TargetRTS only.

Also, each thread has a linked-list of active TCBs, which are purposely stored in sorted order. That is to say, when you start a timer, the timeout is checked and inserted in order into the linked-list, where the smallest timeout values appear at the front of the queue, and the largest timeout values appear at the end of the queue. By inserting these items into the queue in sorted order, it is possible to quickly determine the next timeout value by simply reviewing the first entry on the timer queue.

Integrated IPC and timers

`RTTarget.h` should specify that RSLTIMERS is set to RSLTRUE; RSLACTOR_TIMERS is set to RSLFALSE.

No special timing actor/packages are required.

One actor function must be implemented.

- An external interface function, which is passed a pointer to the object instance data and the current executing actor Id.

At initialization time, an actor in each thread that requires the use of integrated IPC and timers must

- Register the external interface (timing routine) function via the RSLRegisterExternalInterface C TargetRTS API function (an example pseudo-code for this function appears below).
- Create/initialize (or locate) a suitable RTOS object (for example, message queue/mailbox), storing the handle to this RTOS object as an actor ESV.

An example of this type of timer implementation can be found under `$(OBJECTIME_HOME)/ModelExamples/C`, in an update called “C_TornadoQueuesWithTimers”. Specifically, the actors “Behavior” and “Responder” in this update implement this form of timer. A detailed description of this update can be found in “Compliance Suite & Examples” on page 219 of the *C Language Guide.*

The relevant transitions and functions that adhere to the above guidelines are shown below for the Behavior actor. Note that the behavior external interface function is not registered until it has received a corresponding Sync signal (with identifying Queue ID information).

```
/* Behavior -- initialize transition */

this->MyQ=msgQCreate(2,256,MSG_Q_FIFO);
ROOM_PortSendData(ids,Request,(void *)this->MyQ);
printf("Behavior: Sending QID Info (0x%x) of my Q\n",this->MyQ);

/* Behavior -- Sync transition */

this->otherQ=(MSG_Q_ID)msg->data;
printf("Behavior: Received other Q ID (0x%x)\n",this->otherQ);
RSLRegisterExternalInterface(_actor,BehaviorMainloop);
printf("Behavior: Starting Timer for 1 Second\n");
ROOM_InformIn(timer,RSLTARGET_TIME(100));

/* Behavior -- timeout transition */

printf("Behavior: Got expected timeout!\n");
msgQSend(this->otherQ,"Hello!",7,NO_WAIT,0);
printf("Behavior: Starting Timer for 1 Second\n");
ROOM_InformIn(timer,RSLTARGET_TIME(100));

/* Behavior -- Functions */

void BehaviorMainloop(Behavior_InstanceData *this,
         RSLActorIndex _actor) {
   static char copyData[256],*newCopy;
   unsigned longcurrent;
   int MsgQReceiveResult;
   RSLTimerControlBlock*tcb;
   RSLTimerReferencetr;

   /* Get the current time and check to see if the lowest timer
         value has already expired */
   current=tickGet();
   tr=RSLGetFirstTimeout(_actor);
   tcb=tr.tcb;
   if((tcb)&&(current>=tcb->timeoutNSec)) {
         tcb->timeoutMessage=RSLPortEnqueue(_actor,timer,
               RSLTIMEOUT,RSLGENERAL,(void *)0);
         RSLCancelTimer(_actor,tr);
         return;
   }

   /* Block for any message on the queue with a timeout
         (# clock ticks from now) */
   if(tcb)
```

```
            MsgQReceiveResult=msgQReceive(this->MyQ,copyData,
                  256,tcb->timeoutNSec-current);
      else
            MsgQReceiveResult=msgQReceive(this->MyQ,copyData,
                  256,WAIT_FOREVER);
      if((MsgQReceiveResult==ERROR)&&
            (errno=S_objLib_OBJ_TIMEOUT))  { /* Timeout */
            RSLPortEnqueue(_actor,timer,RSLTIMEOUT,RSLGENERAL,
                  (void *)0);
            RSLCancelTimer(_actor,tr);
      }
      else if(MsgQReceiveResult>0) { /* Received a Message! */
            newCopy=(char *)malloc(256);
            memcpy(newCopy,copyData,256);
            RSLPortEnqueue(_actor,ids,SomeEvent,RSLGENERAL,newCopy);
      }
      else {
            /* ??? Error */
      }
}
```

Since this actor supports both timers *and* proprietary IPC (Tornado message queues), the external interface function pseudo-code would be defined as:

```
<GetFirstTimeout>
<GetCurrentTime>
<If (ValidTimer && TimerHasTimedOut)> {
   <EnqueueTimeoutEvent>
   <RemoveTimerRequest>
   return;
}

<If ValidTimer>
   <BlockForIPCWithTimeout>
else
   <BlockForIPCWithInfiniteTimeout>
<IfTimedOut> {
   <EnqueueTimeoutEvent>
   <RemoveTimerRequest>
}
else
   <EnqueueIPCMessage>
```

The first block of pseudo code determines if (during the intervening period of time while the system was processing) any timeouts would have occurred. If timeouts have occurred, timeout events are generated, and the time requests are removed from the timer list.

If no timeout occurred, the next block of code will await for an external IPC message/stimulus while specifying a timeout value (which is our next shortest time to a timeout). Once control is returned to the user's IPC routine, a check is then done to determine if the IPC request timed-out (at which point, a timeout event is enqueued, and the timer request is removed from the timer list), or a real IPC message or external stimulus was received. In that instance, an appropriate message event is enqueued for subsequent execution by the actor.

These timers require the use of relative time (for the informIn request), and absolute time to determine when a timer is to expire (for example, clock ticks). Thus, the use of a macro (for example, `RSLTarget_Time`) might be used to ensure compatibility between the SimulationRTS and the C TargetRTS running on a target. Specifically, the target version of `RSLTarget_Time` should be defined to add the current number of clock ticks to the relative time to create a future absolute time, which is used by the external interface routine.

```
e.g. #define RSLTarget_Time(x) (clockTicks()+x)
```

Timers are started when an actor in that thread starts a timer via `ROOM_InformIn`. In the C TargetRTS, the timer start routine will allocate a local thread TCB, populate it with appropriate data, and then insert it into the list of active timers for that thread, in sorted order. Once enqueued, the registered external interface for that routine will process the timeout requests, as described in the above pseudo-code.

If a thread is currently blocked, awaiting for a timeout to occur, and if another thread sends a message to that thread, the thread will *not* be interrupted (assuming that the RTOS IPC is incompatible with the proprietary C TargetRTS inter-thread inter-actor messaging implementation). Thus the ObjecTime message will be deferred until after that thread's interface routine returns.

Timers are cancelled via `ROOM_CancelTimer`. In the C TargetRTS, the timer cancel routine will mark the `RTTimerID` (which points to a timer control block) as cancelled, and thus will be ignored by the external interface function, when it actually times out.

The number of timers that can be started is configured in the RSLThreadMap. For the first/top thread, the number of timers to be created at initialization time should be configured in `RTTarget.h`, overriding the defaults in `RTConfig.h`.

Each thread has a set of internal TCBs available for its exclusive use. TCBs are not shared amongst threads, and are to be modified by the C TargetRTS only.

Also, each thread has a linked-list of active TCBs, which are purposely stored in sorted order. That is to say, that when you start a timer, the timeout is checked, and inserted in order into the linked-list, where the smallest timeout values appear at the front of the queue, and the largest timeout values appear at the end of the queue. By inserting these items into the queue in sorted order, it is possible to quickly determine the next timeout value by simply reviewing the first entry on the timer queue.

The relevant transitions and functions that adhere to the above guidelines are shown below for the Responder actor. Note that the behavior external interface function is not registered until it has received a corresponding Sync signal (with identifying Queue ID information).

```
/* Responder -- initial transition */

this->stimulatorQ=msgQCreate(2,256,MSG_Q_FIFO);
printf("Responder: Sending QID Info (0x%x) of my Q\n",
   this->stimulatorQ);
ROOM_PortSendData(ids,Request,(void *)this->stimulatorQ);

/* Responder -- Sync transition */

this->otherQ=(MSG_Q_ID)msg->data;
printf("Responder: Received other Q ID (0x%x)\n",this->otherQ);
RSLRegisterExternalInterface(_actor,ResponderMainloop);

/* Responder -- ExtMsg transition */

printf("Responder: Received Q msg '%s'\n",(char *)msg->data);
msgQSend(this->otherQ,"There!",7,NO_WAIT,0);

/* Responder -- Functions */

void ResponderMainloop(Responder_InstanceData *this,
         RSLActorIndex _actor) {
   static charcopyData[256];

   /* Get a Tornado message for the queue and enqueue it
         for the cRSL */
   msgQReceive(this->stimulatorQ,copyData,256,WAIT_FOREVER);
   RSLPortEnqueue(_actor,ids,SomeEvent,RSLGENERAL,&copyData);
}
```

Since this actor supports only proprietary IPC (Tornado message queues), the external interface function pseudo-code would be defined as:

```
<BlockForExternalMessage>
<EnqueueMessage>
```

Actor timers

`RTTarget.h` should specify that RSLTIMERS is set to RSLTRUE; RSLACTOR_TIMERS is set to RSLTRUE.

A special timers package should be used as a basis for implementation. An example of this type of timer implementation can be found under `$(OBJECTIME_HOME)/ModelExamples/C`, in an update called “C_Timers”. A detailed description of this update can be found in “Compliance Suite & Examples” on page 219 of the C Language Guide.

In this approach, a timer actor registers itself as the provider of timing services for all timing requests in the C TargetRTS. If multi-threaded, this actor should be configured to run on its own thread. The C TargetRTS will handle all inter-actor inter-thread and intra-thread messaging to and from the timing actor.

At least two timing actor functions must be implemented. They are as follows:

- An external interface function, which is passed a pointer to the object instance data and the current executing actor Id.
- A signalling function, which passed a pointer to a RTOS object (for example, condition variable) will signal it in a suitable manner.

At initialization time, an actor in each thread that requires the use of integrated timers must

- Register the external interface (timing routine) function via the RSLRegisterExternalInterface C TargetRTS API function (an example pseudo-code for this function appears below).
- Create/initialize (or locate) a suitable RTOS object (for example, condition variable), storing the handle to this RTOS object as an actor ESV.
- Register itself with the C TargetRTS as the provider of *all* timing services (single- or multi-threaded), providing the address of the signalling variable and signalling function via the RSLRegisterTimerService C TargetRTS API function. The C TargetRTS will invoke the signalling function (passing the address of the signaling variable) whenever it is about to deliver a new timing request to that thread.

The external interface function pseudo-code would be defined as

```
<GetFirstTimeout>
<GetCurrentTime>
<If (ValidTimer && TimerHasTimedOut)> {
   <EnqueueTimeoutEvent>
   <RemoveTimerRequest>
   return;
}

<If ValidTimer>
   <BlockForSignalWithTimeout>
else
   <BlockForSignalWithInfiniteTimeout>
<IfTimedOut> {
   <EnqueueTimeoutEvent>
   <RemoveTimerRequest>
}
```

The first block of pseudo code determines if (during the intervening period of time while the system was processing) any timeouts would have occurred. If timeouts occurred, timeout events are generated, and the time requests are removed from the timer list.

If no timeouts occurred, the next block of code waits for an external signal while specifying a timeout value (which is our next shortest time to a timeout). Once control is returned to the timer actor's external interface routine, a check is then done to determine if the blocking RTOS request timed-out (at which point, a timeout event is enqueued, and the timer request is removed from the timer list).

These timers require the use of relative time (for the informIn request), and absolute time to determine when a timer is to expire (for example, clock ticks). Thus, the use of a macro (for example, `RSLTarget_Time`) might be used to ensure compatibility between SimulationRTS and the C TargetRTS running on a target. Specifically, the target version of `RSLTarget_Time` should be defined to add the current number of clock ticks to the relative time to create a future absolute time, which is used by the external interface routine.

```
e.g. #define RSLTarget_Time(x) (clockTicks()+x)
```

Timers are started when an actor in a thread starts a timer via `ROOM_InformIn`. In the C TargetRTS, the timer start routine will allocate a local thread TCB, populate it with appropriate data, and then enqueue it via an inter-thread message to be delivered to the registered timing actor. When the timing actor receives the new request, it will insert it into the list of active timers for the timing thread, in sorted order. Once enqueued, the registered external interface for the timing thread will process the timeout requests, as described in the above pseudo-code.

If the timing thread is currently blocked, waiting for a timeout to occur *or* for new timing requests, and if another thread sends a timing request message to the timing thread, the thread will be interrupted. Thus, once the ObjecTime timeout request message is processed, the timing threads interface routine will be re-invoked.

Timers are cancelled via `ROOM_CancelTimer`. In the C TargetRTS, the timer cancel routine will mark the RTTimerID (which points to a timer control block) as cancelled, and thus will be ignored by external interface function when it actually times out.

The number of timers that can be started is configured in the RSLThreadMap. For the first/top thread, the number of timers to be created at initialization time should be configured in `RTTarget.h`, overriding the defaults in `RTConfig.h`.

Each thread has a set of internal TCBs available for its exclusive use. TCBs are not shared amongst threads, and are to be modified only by the C TargetRTS or the timers. In the multi-threaded C TargetRTS, timer control blocks are moved back and forth between the requesting and timing threads.

Also, each thread has a linked-list of active TCBs, which are purposely stored in sorted order. That is to say, that when you start a timer, the timeout is checked, and inserted in order into the linked-list, where the smallest timeout values appear at the front of the queue, and the largest timeout values appear at the end of the queue. By inserting these items into the queue in sorted order, it is then possible to quickly determine the next timeout value by simply reviewing the first entry on the timer queue.

Chapter 7

Modifying the error parser

Setting the compiler vendor in the libset.mk file

Each libset references its associated error parser via the VENDOR make macro in the `libset.mk` file. The toolset will execute scripts in the `$OBJECTIME_HOME/codegen/compilers/$VENDOR/` directory to perform the conversion from the raw error stream to the Generic Error Stream (GES) format.

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 `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 need to pick a vendor acronym ($VENDOR) and create the directory `$OBJECTIME_HOME/codegen/compilers/$VENDOR`. You will need to create two files in this directory, `otcompile.pl` and `otlink.pl`, which you can copy from another vendor. These Perl files contain the following:

1. Inclusion of common code that performs all the error parsing functions.
2. Setting the errormode variable to indicate whether the file is parsing compiler errors or linker errors. Errormode is set to compilation by default.
3. Setting arrays of regular expressions and other information the common code will use to parse the raw error stream. There will be one set of variables per error expression handled. There will also be one set of variables for each expression that is ignored.
4. A call to the common code once array initializations are done to start parsing input.

To complete the ports, delete the variables that deal specifically with the error expressions that are handled. You will then need to figure out what error expressions your compiler and linker generate and populate `otcompile.pl` or `otlink.pl` appropriately with new variables. There are a couple ways to efficiently determine what errors your compiler generates.

1. There is an error parser update available at http://www.objectime.com/support/restricted-dir/documentation.html, which contains a good number of common compilation errors. You can compile it and look at the compilation details file (`$update_dir/compile.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.

Each regular expression used is a Perl regular expression. If you are not familiar with Perl regular expressions it is suggested that you get a Perl book or find an equivalent reference online. Below is an explanation of how each variable can be used for error expression number X.

The expression @oterr::start_error[X] indicates what pattern to look for to indicate that the current line being parsed is the beginning of the type of error expression X.

```
@oterr::start_error[X]  = '^.*:\d*: warning: .*';
```

The expression @oterr::end_error[X] indicates which pattern to look for to indicate that the current line being parsed is the end of the type of error expression X. If the error is a single line error, it can be set to empty string. If the end of the error is too difficult to determine via a regular expression, it can be set to UNKNOWN. For the unknown case, it will consider every line part of the current error being parsed until it finds the beginning of another error, or a line it is supposed to ignore.

```
@oterr::end_error[X]    = '';
@oterr::end_error[X]    = 'UNKNOWN';
@oterr::end_error[X]    = '.*then contact your local service provid-
er.*';
```

Once the error has been isolated, the lines that compose it are concatenated into one string and the expression @oterr::regexps[X] is used to extract useful information from it. The parentheses in the regular expression pick out the filename, line number and error text respectively. Note that the Perl script is set up so that its wildcard characters can include newlines.

```
@oterr::regexps[X]      = '^(.*):(\d*): warning: (.*)';
```

Each error type has to report to the toolset the severity of the error. Errors will cause the compilation to report a failure, while warnings will allow other steps, such as linking, to proceed.

```
@oterr::severity[X]     = 'warning';
@oterr::severity[X]     = 'error';
```

Optionally, if a weird error turns up that does not have the filename argument first, then the line number, then the error text, you can include the expression @oterr::argorder[X] to change the order in which the program extracts the arguments from the parentheses in the @oterr::regexps[X] expression. By default this variable is set for all error types to be 'FNE'.

```
@oterr::argorder[X]    = 'EFN';
```

You can tell the error parser to ignore lines that the compiler or linker generate but are actually not anything you want to display in the error browser or map back to the toolset. There is one set of variables for each ignore statement that work very much like the variables for parsing a specific error type. Below is an explanation of how each variable can be used to ignore a specific type of expression Y.

The @oterr::start_ignore[Y] statement indicates the beginning of a statement to ignore.

```
@oterr::start_ignore[Y] = 'Copyright \(C\) .* Corp.*';
```

The expression @oterr::end_ignore[Y] indicates which pattern to look for to indicate that the current line being parsed is the end of the type Y of expression that is being ignored. If the expression to be ignored is only one line, it can be set to empty string. Note that an 'UNKNOWN' end of ignore statement is not supported.

```
@oterr::end_ignore[Y] = '';
@oterr::end_ignore[Y] = '\s*Version \d*.*';
```

Chapter 8

Testing the TargetRTS

A port to a new platform requires testing the TargetRTS. There are some standard ObjecTime updates that can be used to test the functionality of the TargetRTS. These tests are not comprehensive but provide some assurance that the port was successful. The C++ models are available for download by customers only from our support webpage (start from http://www.objectime.com). These models are provided as is and ObjecTime Limited provides no warranty expressed or implied for their use.

Testing the TargetRTS for C++

HelloWorld update

The HelloWorld update is a single actor model that simply outputs “Hello World” on 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 actor functionality.

Performance update

The Performance update is a suite of tests that provides measurements of basic TargetRTS functions. This model is also useful for testing the functionality of the TargetRTS since many features are testing during the performance measurements. Test results are sent to the standard output and correct functionality can easily be determined by observing the output.

FiveStates update

The FiveStates update is a model that tests the functionality of the target observability feature of the TargetRTS. This test provides a model with a state machine consisting of five states. The transitions between these states can be monitored from the behavior monitor in the ObjecTime Developer Model Execution Browser. If target observability is functioning then the Model Execution Browser will be able to connect to the application running on the target via TCP/IP and provide debugging and execution monitoring via the Model Execution Browser.

Testing the TargetRTS for C

The CRSL Compliance Tests

The TargetRTS for C ships a set of compliance test models that can be compiled and run on a new target platform to ensure the functionality of the newly ported C TargetRTS.

These models are part of the `%OBJECTIME_HOME\ModelExamples\C` folder and are fully described in “Compliance Suite & Examples” on page 219 of the *C Language Guide*.

Chapter 9

Tuning the TargetRTS

This section briefly describes areas in the TargetRTS that can be tuned to improve performance. The Performance update described in “Performance update” on page 95 can be used to verify the success of performance enhancements.

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 “TargetRTS Customization Example” in the *C++ Target Guide* describes how to create such a version of the TargetRTS. For C Language usage, please refer to “Recommended Configurations” on page 213 in the *C Language Guide* for suggested optimizations. The so-called “minimum TargetRTS” disables the external layer (and target observability), logging service and the RTS debugger. The minimum TargetRTS should provide significant performance gains over the fully featured version.

Target compiler optimizations

Most compilers provide optimizations at the 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 the Performance update may prove useful. See “Performance update” on page 95. For C usage, please refer to “Compliance Suite & Examples” on page 219 of the *C Language Guide*.

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` class. 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 10

Common problems and pitfalls

This section contains common problems and pitfalls that we have encountered with previous ports. The TargetRTS is supported on approximately 71 platforms and has been verified on each of these platforms. In general, the problems and pitfalls encountered are due to RTOS and tool chain oddities and bugs. Other problems arise from lack of support for certain features required by the TargetRTS and thus require a custom workaround to satisfy the TargetRTS.

Problems and pitfalls with target toolchains

Compiler optimizations

Compiler optimizations, in general, help speed up the application. Some optimizations can 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. This can cause problems when a memory location is used to store a RTOS primitive, such as a semaphore. The operating system modifies the contents of the semaphore variable but the application does not. The compiler optimizes the references to the semaphore and consequently removes proper access to the semaphore.

Optimizations vary from compiler to compiler, so refer to the documentation for your specific tool chain. Review the optimizations that are available and be aware that some may cause errors in the application. Running a test suite such as the one described in section “Testing the TargetRTS” on page 95 is a good way to ensure the optimizations have not broken the TargetRTS.

Linking problems

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.

Duplicate references with global signal names

In ObjecTime the signals are defined globally. Many of these standard signal names may be duplicates of names used in RTOS libraries. This can cause duplicate reference errors at link time. One approach is to recompile the TargetRTS with these signal names redefined. This can be done by globally replacing

<signal name> during compile time using the compile option "-D<signal_name>=rt<signal_name>".

System include files

The structure and content of include files can be a challenge when moving to a new tool chain. 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 $(RTS_HOME)/src/target/<target_name> 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 $(RTS_HOME)/src/target/<target_name> directory. This file should contain all the necessary include files required for TCP/IP functions. For RTOSs that provide a POSIX interface to OS functions then a file called RTPosixDefines.h should be used to encapsulate all POSIX header files. Other, more specific, header files may be required to isolate unique interfaces for your RTOS. These may be added to the $(RTS_HOME)/src/target/<target_name> directory as needed, and are typically prefixed by "RT".

Problems and pitfalls with TargetRTS/RTOS interaction

Synchronization primitives

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.

Priority inversion

The TargetRTS provides no specific mechanism to prevent priority inversion. Some RTOSs, such as Tornado, support a priority inheritance mechanism for synchronization primitives such as mutexes and semaphores. The C++ TargetRTS for Tornado 1.0.1 enables this option for the RTMutex and RTSyncObject classes. If your RTOS supports this then enable this option when creating a mutex or semaphore.

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 the external layer, timers, debugger or even the ObjecTime application.

Real-time clock

Most RTOSs provide a function to retrieve the current system time. Typically it may return clock ticks, milliseconds or even nanoseconds. In the C++ TargetRTS, a conversion from the RTOS time to `RTTimespec` is typically required in order to satisfy the requirements of the `RTTimespec::getclock` function. Some RTOSs 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 ObjecTime 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 between the `RTTimerSAP::adjustTimeBegin()` and `RTTimerSAP::adjustTimeEnd()` functions. If, however, system clock changes do not affect the `RTTimespec::getclock()` function, do not use the `RTTimerSAP::adjust-TimeBegin` and `RTTimerSAP::adjustTimeEnd()` functions. Timers will fail in this case and cause unwanted behavior in your ObjecTime application.

For example:

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

    timer.adjustTimeBegin();//stop ObjecTime timer service

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

    delta = new_time;
    delta -= old_timer;

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

Signal handlers

Many RTOSs do not use signals that are typical of UNIX operating systems. If your RTOS does not provide signals then 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 there 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 `$(RTS_HOME)/src/target/`

<target_name>/MAIN/main.cc (/main.c for the C TargetRTS). Typically, you have to start a thread that contains the main() function for the ObjecTime application. The documentation for the RTOS will describe how to start your application in this manner.

Application 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, use two globally defined variables default_argc and default_argv, defined in the generated code from the toolset, which contain command line arguments from the ObjecTime toolset. In the C and C++ TargetRTS, you can pass these variables to the RTMain::entryPoint() function from your modified main() function. See section "RTOS supplies main() function" on page 101. Default arguments can be specified in the toolset via the Update Browser's *Configuration>Language Options>Targets* menu.

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 $(RTS_HOME)/src/target/<target_name>/RTDiag/panic.cc to use the exit function specific to your RTOS.

Problems and pitfalls with target TCP/IP interfaces

Select() statement

Some implementations of the select() statement do not correctly use the value set in the width parameter. Consequently the function thinks the file descriptor sets are larger than they really are. This can cause memory corruption and, consequently, serious failures in the running application. To overcome this problem in the C++ TargetRTS, some targets (CLASSIX, OSE3, VRTX3) override the RTIOMonitor::min_size() function in $(RTS_HOME)/src/target/<target_name>/RTIOMonitor/min_size.cc. In these cases, the minimum size is assumed to be the maximum file descriptor set size.

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 multi-thread safe. If this is the case for your target OS then replace the call RTTcpSocket::lookup() in $(RTS_HOME)/src/target/<target_name>/RTTcpSocket/lookup.cc in the C++ TargetRTS.

Part 3
Appendices

Appendix A

TargetRTS for C++ porting example

Introduction

This section 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++ Target Guide* for a customization example. This porting example should help implement the information presented in previous sections. The target platform for this example is the VRTX 4.0 real-time operating system using the Microtec Research C++ Compiler Version 4.5T for Motorola 68040 microprocessors. This is a currently supported platform, but it is assumed that no previous version of the TargetRTS for this platform exists.

Choosing the platform name

The platform 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 VRTX4. The hardware architecture is Motorola 68040 (m68040). The compiler is the Microtec Research C++ Compiler Version 4.5T. For this example we will only consider the multi-threaded version of the TargetRTS since this provides the most interesting porting challenges. The resulting platform name is as follows:

```
<OS> = VRTX4T
<LIBSET> = m68040-Microtec-4.5T
<OS>.<LIBSET> = VRTX4T.m68040-Microtec-4.5T
```

Create setup script

The setup script is in the file `$(RTS_HOME)/config/VRTX4T.m68040-Microtec-4.5T/setup.pl`. This file is a Perl script that defines environment variables for the compilation of the TargetRTS. The contents of the setup script are as follows:

```
$OS_HOME             = $ENV{‘OS_HOME‘};

$USR_MRI             = “$OS_HOME/spectra/solaris-68k-4.AAA”;

$ENV{‘USR_MRI’}      = “$USR_MRI”;

$ENV{‘SPECTRA’}      = “$USR_MRI/spa68k”;

$ENV{‘MRI_68K_BIN’} = “$USR_MRI/bin”;
```

```
$ENV{‘MRI_68K_LIB‘} = “$USR_MRI/lib”;

$ENV{‘MRI_68K_INC‘} = “$USR_MRI/include/mcc68k”;

$ENV{‘PATH‘}        = “$USR_MRI/bin:$ENV{‘PATH’}”;

$preprocessor = “ccc68k -E >MANIFEST.i”;
$include_opt  = ‘-J‘;
$target_base  = ‘VRTX3‘;
$supported    = ‘Yes‘;
```

The setup script must contain the mandatory definitions for the `preprocessor` and `supported` flags. The tool chain 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 VRTX3. This means the VRTX4T target uses the same code base for the TargetRTS classes as the VRTX3T target. In a TargetRTS port to a native compiler tool chain these definitions are probably not required.

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 ObjecTime 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 differs significantly, particularly for cross-compilers such as the Microtec 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 36 for details. In any port of the TargetRTS there are certain commands required in the tool chain in order to support the building of the TargetRTS. Table 8 illustrates these required commands, the Unix equivalent, and the Microtec variant.

Table 8 Compiler tool chain requirements

Command	Unix	Microtec
library archive	ar	microtec_ar (script)
C++ Compiler	CC	ccc68k
Linker	ld -r	lnk68k -r
Pre-linker	ld -r -o	lnk68k -r -o
Shared library builder	CC -G -z text -o	ccc68k -G -z text -o
VENDOR	n/a	Microtec

The library archive command (`ar`) for the Microtec tool chain requires the use of a script to work the way the TargetRTS build requires. The Microtec development environment does not supply an `ar` com-

mand. Instead it provides a `lib` command that behaves differently than the `ar` command. A script file, `microtec_ar` was written to provide a wrapper around the lib command. This is an exception to other supported TargetRTS platforms but illustrates a possible pitfall when moving to a new platform. The libset makefile must define the `VENDOR` variable 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 ObjecTime Developer toolset.

Another important role of the libset makefile is the definition of command line options. Table 9 illustrates the typical subset of command line options, the Unix equivalent, and the Microtec variant.

Table 9 C++ Command line options

Option	Unix	Microtec
DEBUG_TAG	-g	-g -Gf
LIBSETCCEXTRA		-p68040 -c -Qms0401 -Qs -Xp -Mca -Mda[a]
LIBSETCCFLAGS	-O	-O -Ob -Oe -Ot -Qfs -NM$(@F:.o=)[b]
INCLUDE_TAG	-I	-J
DIR_TAG	-l	-l
LIB_EXT	.a	.lib

a. The -Qms0401 option suppresses the 0401 error message “destructor for base class is not virtual”. The -Qs option suppresses the summary message. The -Xp allocates space for global variables that have not been explicitly initialized.
b. The -Qfs option suppresses the display of the source file line number for diagnostic messages. The -NM options sets the module name.

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` for other TargetRTS platforms for guidance. A list of required features follows:

- to compile source files into object files only (that is, not 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 produce shared libraries, typically the ‘-G’ 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 instructions 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’ option
- to turn on code optimization, typically ‘-O’ option and sub-options

The contents of the libset makefile, `$(RTS_HOME)/libset/m68040-Microtec-4.5T/libset.mk` , for Microtec compiler is as follows:

```
AR_CMD        = $(RTS_HOME)/targets/microtec_ar lib68k
CC            = ccc68k
LD            = lnk68k -r
PRELINK       = cat >
SHLIB_CMD     = $(CC) -G -z text -o

#VENDOR is used in definition of OTCOMPILE and OTLINK in default.mk
VENDOR        = Microtec

#override the extension for executables
EXEC_EXT      = .x

LIBSETCCFLAGS = -p68040 -c -Qms0401 -Qs -Xp -Mca -Mda
LIBSETCCEXTRA = -O -Ob -Oe -Ot -Qfs -NM$(@F:.o=)
SHLIBS        =

LIB_EXT       = .lib

DEBUG_TAG     = -g -Gf
DIR_TAG       =
INCLUDE_TAG   = -J
```

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 VRTX targets. The contents of the target makefile, `$(RTS_HOME)/target/VRTX4T/target.mk`, is as follows:

```
TARGETCCFLAGS = $(INCLUDE_TAG)$(SPECTRA)/target/include \
                $(DEFINE_TAG)timeout=otTimeout
```

Configuration makefile

The configuration makefile is used to make definitions required by the operating system and compilation environment together. In the case of VRTX, the definitions for libraries are made here since they are specific to the compiler and operating system combination. Therefore these definitions are not appropriate in the target makefile. The content of the configuration makefile, `$(RTS_HOME)/config/VRTX4T.m68040-Microtec-4.5T/config.mk`, is as follows:

```
EXEC_EXT    = .x

SYSTEM_LIBS = $(RTS_LIBRARY)/libObjecTime$(LIB_EXT) \
```

```
        $(RTS_LIBRARY)/libObjecTimeTransport$(LIB_EXT) \
        $(RTS_LIBRARY)/libObjecTimeTypes$(LIB_EXT)

TARGETLIBS  = $(USR_MRI)/lib/ccc68kab040.lib
```

TargetRTS configuration definitions

The configuration definitions for the TargetRTS are found in the include file `$(RTS_HOME)/include/RTConfig.h`. The definitions in this file are overridden by `$(RTS_HOME)/target/VRTX4T/RTTarget.h` and possibly `$(RTS_HOME)/libset/m68040-Microtec-4.5T/RTLibSet.h`. These definitions are used to enable and disable various features in the TargetRTS. By default all of the TargetRTS features are enabled (for example, target observability). The porting effort may be made easier if these features are disabled. See section "TargetRTS Customization Example" in the *C++ Target Guide* for instructions on how to build a minimized TargetRTS. The content of the file `$(RTS_HOME)/target/VRTX4T/RTTarget.h` is as follows:

```
#ifndef __RTTarget_h__
#define __RTTarget_h__ included

#define TARGET_VRTX 4
#define TARGET platVRTX

#define USE_THREADS 1
#define EXTERNAL_LAYER 1

#define DEFAULT_MAIN_PRIORITY  75
#define DEFAULT_LAYER_PRIORITY 73
#define DEFAULT_TIMER_PRIORITY 70
#define DEFAULT_IOMON_PRIORITY 72
#define DEFAULT_DEBUG_PRIORITY 60

#define CLOCK_TICKS_PER_SEC    100
#define NSECS_PER_TICK         10000000

#define __READY_EXTENSIONS__

#ifdef _SIZE_T
#ifndef __size_t
#define __size_t
#endif
typedef _SIZE_T size_t;
#endif

#endif // __RTTarget_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. A description of TargetRTS classes that may require changes is already given in Table 5, “TargetRTS constants/macros and their default values,” on page 46.

The required changes to the TargetRTS source for VRTX 4 and the Microtec compiler are located in the `$(RTS_HOME)/src/target/VRTX3` directory. These files override the versions in `$(RTS_HOME)/src`. To override a definition from the source directory a new subdirectory is created in `$(RTS_HOME)/src/target/VRTX3` (that is, a new definition for `RTTimespec::getclock` requires a subdirectory `$(RTS_HOME)/src/target/VRTX3/RTTimespec`). The new file containing `RTTimespec::getclock` would be `$(RTS_HOME)/src/target/VRTX3/RTTimespec/getclock.cc`.

The required changes to the TargetRTS are too large to include in this document. Table 10 contains a summary of the required changes to each file.

Table 10 Required changes to TargetRTS source

Class	File	Change
main function	main.cc	This change was required due to a bug in the Microtec compiler with respect to static constructors. Normally changes are required for operating systems that already provide a `main` function.
RTDebugger-Input	nextChar.cc	
RTDiagStream	flush.cc	if `fflush` is not supported, implement a flush empty method
	ls_string	<< operator overridden with code to output string (`fputs` not supported)
RTIOMonitor	min_size.cc	`min_size` method changed to use FD_SETSIZE.
RTMain	targetStartup.cc	`targetStartup` method overridden with code to perform VRTX specific startup code.
	installOneHandler.cc	`installOneHandler` empty method
	installHandlers.cc	`installHandlers` empty method
RTMutex	ct.cc	constructor for `RTMutex` defined for VRTX mutex creation (`sc_mcreate`)
	dt.cc	destructor for `RTMutex` defined for VRTX mutex destruction (`sc_mdelete`)
	enter.cc	`enter` method created to use VRTX `sc_mpend` function
	leave.cc	`leave` method created to use VRTX `sc_mpost` function

Table 10 Required changes to TargetRTS source

Class	File	Change
RTSyncObject	ct.cc	constructor for `RTSyncObject`
	dt.cc	destructor for `RTSyncObject`
	signal.cc	`signal` method for `RTSyncObject` created to use VRTX `sc_post` function.
	wait.cc	`wait` method defined to use VRTX `sc_pend` function with no time-out.
	timedwait.cc	`timedwait` method defined to use VRTX `sc_pend` function with time-out specified. NOTE: In VRTX the mailbox feature is used — it behaves much like a binary semaphore and provides a time-out on the `sc_pend` function — this greatly simplifies the implementation of the `RTSyncObject`
RTTcpSocket	getPrimary.cc	`getPrimary` method overridden to support different method of establishing host name.
	lookup.cc	`lookup` method overridden due to differences in `gethost-byname` function
	set_nonblocking.cc	`set_nonblocking` method overridden due to differences in `ioctl` function.
RTThread	ct.cc	constructor for `RTThread` overridden to use VRTX-specific thread creation function (`sc_tecreate`)
RTTimespec	getclock.cc	`getclock` method overridden to use VRTX-specific clock function (`sc_gclock`).

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 VRTX-Microtec target, type the following in the `$(RTS_HOME)/src` directory:

```
make VRTX4T.m68040-Microtec-4.5T
```

This will create a directory `$(RTS_HOME)/build-VRTX4T.m68040-Microtec-4.5T` which will contain the dependency file and object files for the TargetRTS. If the build completes successfully the resulting ObjecTime libraries will be placed in the `$(RTS_HOME)/lib` directory.

TargetRTS for C Porting example

Introduction

This section 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. This porting example should help implement the information presented in previous sections. The target platform for this example is the VRTX 4.0 real-time operating system using the Microtec Research C Compiler Version 1.3C for Motorola PowerPC 603 microprocessors. It is assumed that no previous version of the TargetRTS for this platform exists.

Choosing the platform name

The platform 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 VRTX4. The hardware architecture is Motorola PowerPC 603 (ppc603). The compiler is the Microtec Research C Compiler Version 1.3C. For this example we will only consider the multi-threaded version of the TargetRTS since this provides the most interesting porting challenges. The resulting platform name is as follows:

```
<OS> = VRTX4T
<LIBSET> = ppc603-Microtec-1.3C
<OS>.<LIBSET> = VRTX4T.ppc603-Microtec-1.3C
```

Create setup script

The setup script is in the file `$(RTS_HOME)/config/VRTX4T.ppc603-Microtec-1.3C/setup.pl`. This file is a Perl script that defines environment variables for the compilation of the TargetRTS. The contents of the setup script are as follows:

```
$os = $ENV{'OS'};
$os = 'default' unless defined( $os );
if( $os eq 'Windows_NT' )
{
    $usr_mri       = $ENV{'USR_MRI'};
    $ENV{'PATH'}   = "$usr_mri/bin;$ENV{'PATH'}";
}
else
{
    $os_name = `uname -s`;  ## get the flavor of unix
    chomp( $os_name );
```

```
    if( $os_name eq 'SunOS' )  ## if the flavor of unix is solaris
    {
      $usr_mri       = "$ENV{'OS_HOME'}/spectra/solaris-ppc603-4.AB";
    }
    elsif( $os_name eq 'HP-UX' ) ## if the flavor of unix is HP-UX
    {
      $usr_mri       = "$ENV{'OS_HOME'}/spectra/hp-ppc603-4.AB";
    }
    else   ## this flavor of unix is not a supported flavor
    {
       printf "This'%s'is not a supported flavor of unix\n",
                   $os_name;
    }
    $ENV{'USR_MRI'}     = "$usr_mri";
    $ENV{'SPECTRA'}     = "$usr_mri/spappc";
    $ENV{'MRI_PPC_BIN'} = "$usr_mri/bin";
    $ENV{'MRI_PPC_LIB'} = "$usr_mri/lib";
    $ENV{'MRI_PPC_INC'} = "$usr_mri/include/mccppc";
    $ENV{'PATH'}        = "$usr_mri/bin:$ENV{'PATH'}";
}
    $preprocessor = "mccppc -E >MANIFEST.i";
    $include_opt  = '-J';
    $target_base  = 'VRTX4';
    $supported    = 'Yes';
```

The setup script must contain the mandatory definitions for the `preprocessor` and `supported` flags. The tool chain 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 VRTX4. The value VRTX4 for `target_base` implies that the name of the directory for the VRTX code base is `$(RTS_HOME)/src/target/`VRTX4. In a TargetRTS port to a native compiler tool chain these definitions are probably not required.

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 ObjecTime 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 differs significantly, particularly for cross-compilers such as the Microtec 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 36 for details. In any port of the TargetRTS there are certain commands required in the tool

chain in order to support the building of the TargetRTS. Table 11 illustrates these required commands, the Unix equivalent, and the Microtec variant.

Table 11 Compiler tool chain requirements

Command	Unix	Microtec
library archive	ar	ar.pl (a perl script)
C++ Compiler	CC	mccppc
Linker	ld -r	lnkppc -i
Pre-linker	ld -r -o	lnkppc -i -o
Shared library builder	CC -G -z text -o	mccppc -G -z text -o
VENDOR	n/a	Microtec

The library archive command (ar) for the Microtec tool chain requires the use of a script to work the way the TargetRTS build requires. The Microtec development environment does not supply an `ar` command. Instead it provides a `lib` command that behaves differently than the `ar` command. A Perl script file, `ar.pl` that resides in the directory `$(RTS_HOME)/libset/ppc603-Microtec-1.3C`, was written to provide a wrapper around the `lib` command. This is an exception to other supported TargetRTS platforms but illustrates a possible pitfall when moving to a new platform. Another Perl script file, `ld.pl` that resides in the directory `$(RTS_HOME)/libset/ppc603-Microtec-1.3C`, was written to provide a wrapper around the `lnkppc -i` command. This script takes `lnkppc -i` as arguments and is invoked to link an ObjecTime Developer update. The libset makefile must define the `VENDOR` variable 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 ObjecTime Developer toolset.

Another important role of the libset makefile is the definition of command line options. Table 12 illustrates the typical subset of command line options, the Unix equivalent, and the Microtec variant.

Table 12 C++ Command line options

Option	Unix	Microtec
DEBUG_TAG	-g	-g -Gd -Gf -Gm -Gs
LIBSETCCFLAGS		-p603 -DPPC
LIBSETCCEXTRA	-O	-O -Qfs[a]
INCLUDE_TAG	-I	-J
DIR_TAG	-l	-l
LIB_EXT	.a	.lib

a. The -Qfs option suppresses the display of the source file line number for diagnostic messages.

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` for other TargetRTS platforms for guidance. A list of required features follows:

- to compile source files into object files only (that is, not 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 produce shared libraries, typically the '-G' 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 instructions 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' option
- to turn on code optimization, typically '-O' option and sub-options

The contents of the libset makefile, `$(RTS_HOME)/libset/ppc603-Microtec-1.3C/libset.mk` , for Microtec compiler is as follows:

```
VENDOR        = Microtec
AR_CMD        = $(PERL) $(RTS_HOME)/libset/$(LIBRARY_SET)/ar.pl
CC            = mccppc
LD            = $(PERL)$(RTS_HOME)/libset/$(LIBRARY_SET)/ld.pl \
                lnkppc -i
SHLIB_CMD  = $(CC) -G -z text -o
LIBSETCCFLAGS = -p603 -DPPC
LIBSETCCEXTRA = -O -Qfs
SHLIBS     =
OT_LIB_TAG    = -llib
LIB_EXT    = .lib
INCLUDE_TAG   = -J
ALL_OBJS_LIST = %$(ALL_OBJS_LISTFILE)
DEBUG_TAG     = -g -Gd -Gf -Gm -Gs
```

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 VRTX targets. The contents of the target makefile, `$(RTS_HOME)/target/VRTX4T/target.mk`, is as follows:

```
TARGETCCFLAGS = $(INCLUDE_TAG)$(SPECTRA)/target/include \
                $(DEFINE_TAG)timeout=otTimeout
```

Configuration makefile

The configuration makefile is used to make definitions required by the operating system and compilation environment together. In the case of VRTX, the extension to be used at the end of the filename for the final executable is defined here. The content of the configuration makefile, `$(RTS_HOME)/config/VRTX4T.ppc603-Microtec-1.3C/config.mk`, is as follows:

```
EXEC_EXT    = .x
```

TargetRTS configuration definitions

The configuration definitions for the TargetRTS are found in the include file `$(RTS_HOME)/include/RTConfig.h`. The definitions in this file are overridden by `$(RTS_HOME)/target/VRTX4T/RTTarget.h`. These definitions are used to enable and disable various features in the TargetRTS. The content of the file `$(RTS_HOME)/target/VRTX4T/RTTarget.h` is as follows:

```
#ifndef __RTTarget_h__
#define __RTTarget_h__ included
#define USE_THREADS        1
#define RSLMULTITHREADED RSLTRUE
/* Following definition is due to  PR 7890 */
#ifdef RSLTHREAD_RETURNVAL
#undef RSLTHREAD_RETURNVAL
#endif
#define RSLTHREAD_RETURNVAL
#define RSLTO RSLTRUE
#define RSLDEBUG RSLTRUE
#define RSLTIMERS RSLTRUE
#define RSLACTOR_TIMERS RSLFALSE
#define DEFAULT_MAIN_PRIORITY  75
#define DEFAULT_LAYER_PRIORITY 73
#define DEFAULT_TIMER_PRIORITY 70
#define DEFAULT_DEBUG_PRIORITY 60
#define __READY_EXTENSIONS__
#define RSL_OVERRIDE_BASIC_SIZES
/*
 *
 * typically long
 *
*/
typedef unsigned long RSLMemorySize;
typedef unsigned long RSLTimeoutSize;
/*
 * typically short or int
 *      -- each actor may only have 64k-1 of local data
 */
```

```
typedef unsigned short RSLDataSize;
/*
typedef unsigned short RSLFieldOffset;
*/
typedef unsigned long RSLFieldOffset;   /* for VRTX-MRI C compiler */
/*
 * typically unsigned short
 *    -- there may only be 64k-2 ports
 *    -- there may only be 64k-2 actors
 *    -- there may only be 64k-2 threads
 */
/* override RSLPortIndex definition for VRTX - PR7804, PR7808 */
typedef short           RSLPortIndex;
#define RSLMaxPort      32767
typedef unsigned short  RSLActorIndex;
#define RSLMaxActor     65535
typedef unsigned short  RSLThreadIndex;
#define RSLMaxThread    65535
typedef unsigned short  RSLMessageIndex;
#define RSLMaxMessages  65535
typedef unsigned short  RSLTCBIndex;
#define RSLMaxTCBs      65535
/*
 * typically char
 *      -- there may only be 255 events per protocol
 *      -- there may only be 255 ports references per actor
 *      -- there may only be 255 Actor Classes
 *      -- there may only be 255 states in each actor
 *      -- there may only be 255 port classes
 */
typedef unsigned short  RSLBool;
typedef unsigned short  RSLFlags;
#define RSLMaxMessagePriority 65535
typedef unsigned short  RSLMessagePriority;
#define RSLSignalEvent  65535
typedef unsigned short  RSLSignalIndex;
#define RSLMaxActorClasses 65535
typedef unsigned short  RSLActorType;
typedef unsigned short  RSLStateIndex;
#define RSLMaxPortClasses 65535
typedef unsigned short  RSLPortType;
#define RSLMaxFieldTypes 65535
typedef unsigned short  RSLFieldType;
#endif    /* #ifndef __RTTarget_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.

The required changes to the TargetRTS source for VRTX 4 and the Microtec compiler are located in the `$(RTS_HOME)/src/target/VRTX4` directory. These files override the versions in various directories in `$(RTS_HOME)/src`. To override a definition from the source directory a new subdirectory is created in `$(RTS_HOME)/src/target/VRTX4.`

The required changes to the TargetRTS are too large to include in this document. Table 13 contains a summary of the required changes to each file.

Table 13 Required changes to TargetRTS source

Class	File	Change and Description
MAIN	main.c	
DEBUG	debugio.c	`RSL_nextChar` function is overridden to support getting the next character from Input stream
INITSTOP	TGTinit.c	`RSL_Target_Startup` function is overridden to support relevant initialization during startup
TCP	lookup.c	`cRSL_lookup` function is overridden due to differences in `gethostbyname` function
THREAD	RTThread.c	The functions `RSLmutex_init`, `RSLmutex_lock`, `RSLmutex_unlock`, `RSLsemaphore_init`, `RSLsemaphore_wait`, `RSLsemaphore_post`, `RSLthr_create` are overridden.

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 VRTX-Microtec target, type the following in the `$(RTS_HOME)/src` directory:

```
make VRTX4T.ppc603-Microtec-1.3C
```

This will create a directory `$(RTS_HOME)/build-VRTX4T.ppc603-Microtec-1.3C` which will contain the dependency file and object files for the TargetRTS. If the build completes successfully the resulting ObjecTime libraries will be placed in the `$(RTS_HOME)/lib/` VRTX4T.ppc603-Microtec-1.3C directory.

Part 4
Index

Index

N

O

P

R

S

T

U

ObjecTime Limited
340 March Road
Kanata, Ontario
Canada K2K 2E4