Rational Software Corporation

C++ Reference

RATIONAL ROSE® REALTIME

VERSION: 2003.06.00

PART NUMBER: 800-026109-000

WINDOWS/UNIX

Rational®
the software development company

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

Contents

Preface xxiii
Audience. xxiii
Other Resources xxiii
Rational Rose RealTime Integrations With Other Rational Products xxiv
Contacting Rational Customer Support xxv
1 Overview 27
Using this Guide 27
Workflows for Your Host Workstation and Embedded Target 28
Using C++ Code in Models. 28
Model Properties 29
C++ Services Library 30
Code Generation 30
Modifying Generated Code. 31
Compilation. 31
Linking the Model with the Services Library 31
Model Executables 31
Target Observability 32
2 Using C++ Code in Models 33
Adding C++ Code to Models 33
The Syntax of Code Segments. 34
Choice Point Code Condition Segment 34
Encapsulating Target Specific Behavior 34
Code Sync 34
Making Changes Outside the Toolset. 35
Identifying Designated Code Sync Areas. 35
De-activating Code Sync 35

Macros and Arguments Available to State Machine Code 35

CALLSUPER 35

SUPER 36

RTDATA and rtdata 36

rtport 36

Limitations 37

Opening Rational Rose Models in Rose RealTime 37

Using the C++ Analyzer in Rose RealTime 37

3 Code Generation. 39

Model to Code Correspondence 39

Associations 39

Valid Code Generation Associations 40

Dependencies 41

Classes 41

Logical Packages 42

User-defined Operations 43

Standard Operations 43

Attributes 44

Capsules 45

Capsule State Diagrams 46

Protocols 47

Components 49

Relationships and Elements Ignored by C++ Code Generation 50

Code Generator Behavior 50

Incremental Generation 51

The Effect of Controlled Units 52

Generated Code Directory Layout 53

src 53

build 54

Code Generator Command-Line Arguments 54

Command-Line Arguments 54

Command-Line Build Interface 55

4 Generating and Sharing External Library Interfaces 57

Overview 57

Phase 1: Providing the Library Interface Specification. 59

Creating a Library Component 59

Setting the Target Configuration and References 60

Setting the Visibility Level of External Library Interfaces . . . 60
Redefining Visibility Settings for Unit Interfaces . . . 64
Setting Inclusion Paths and Library Paths . . . 65
Specifying the Names of Shared Packages . . . 66
Phase 2: Publishing the Interface . . . 66
Phase 3: Sharing and Using the External Library Interface . . . 68
Sharing the Interface . . . 68
Using the Interface . . . 69
Considerations and Known Issues . . . 70
5 Classes and Data Types . . . 73
Overview . . . 73
Sending Data in Messages . . . 73
Protocols . . . 74
Sending by Value . . . 74
Sending by Reference . . . 74
Memory Leaks . . . 75
Creating Data Types . . . 76
Marshallable Data Classes . . . 77
Basic Structures . . . 77
C++ Data Type Examples . . . 77
Syntax Examples of Sending Data Classes Between Capsule Instances . . . 78
Sending and Receiving Data By Value . . . 78
Sending and Receiving Data By Reference . . . 78
Class Modeling Examples . . . 79
Creating a Class Data Member From the Class Diagram . . . 79
Specifying Arrays Using Association Multiplicity . . . 81
Creating and Using Common C++ Constructs . . . 81
Creating Array and Pointer Attributes . . . 81
Creating a Constant . . . 82
Creating a Typedef . . . 83
Creating an Enumeration . . . 83
Creating a #define . . . 84
Creating a Struct . . . 86
Class Creation Examples . . . 86
Creating and Using Classes With No Pointer Attributes . . . 86
Creating and Using Classes With Attributes That Are Pointers . . . 87
Integrating an External Class . . . 88
Integration Questions . . . 88
Integration for Case #1 . . . 88
Integration for Case #2 and Case #3 . . . 89

Integration Options 89
Option 1: Describing an External Type to Rose RealTime 89
Option 2: Providing Own Marshalling Functions 90
6 C++ Services Library 93
C++ Services Library Framework. 93
The Big Advantage 93
Message Processing 94
Events and Messages. 94
Processing Overview 95
Single and Multi-Threaded Message Processing 95
Introduction to Threads. 96
Types of Concurrency. 96
Mapping Capsules to Threads 96
Single-Threaded Services Library 97
Multi-Threaded Services Library 98
Naming Considerations 99
C++ Services Library Framework 99
Capsules Become Subclasses of RTActor 101
Special Overrideable Capsule Class Operations 102
Capsule Class Information is Stored in Instances of RTActorClass 102
Capsule Roles are Attributes of Type RTActorRef 102
Protocols Become Two Classes: Base and Conjugate. 103
Ports are Protocol Type Attributes in RTActor Subclasses 103
Signals Become Operations in Protocol Classes 103
Capsule Roles are Place Holders for Zero or More Capsule Instances 104
Multiple Containment 104
Capsule Instances Have Access to a RTMessage Object 105
Capsule Instances Have Access to their Controller 105
Framework Sample Model 105
Log Service 106
Communication Services 106
Asynchronous and Synchronous Communication 107
Order-Preserving 108
Lossy 108
Request-Reply 108
The Semantics of Usage of Message Priorities 109
Support for Unwired Ports 110
Published Versus Unpublished Unwired Ports 110

Registration by Name 111
Registration String 111
Automatic Versus Application Registration 111
Deferring and Recalling Messages 112

Timing Service 112

Relative Versus Absolute Time 113
Timing Precision and Accuracy 114

Frame Service 114

Optional Capsules 115
Plug-in Capsule Roles: Multiple Containment 115
Multiple Containment 115
Replicated Capsule Roles 116

Exception Service 117

RTController Error Codes 117

Error Enumeration 118
alreadyDeferred 119
badActor 119
badClass 119
badId 120
badIndex 120
badRef 120
badSignal 120
badValue 120
deferInvoke 120
dereg 120
imported 120
noConnect 121
noMem 121
noReply 121
notImported 121
notOptional 121
prio 121
recursiveInvoke 121
refFull 122
reg 122
replRef 122
xRtsInvoke 122

External Port Service 122

7 Running Models on Target Boards 123
Step 1: Verify Tool Chain Functionality. 123
How to Test. 124
Step 2: Kernel Configuration 124
Step 3: Verify main.cpp 125
Step 4: Try Manual Loading 125
Step 5: Running with Observability 126
8 Command Line Model Debugger 129
Starting the Run Time System Debugger 129
URTS_DEBUG Parameter 129
Differences Between Single-Threaded and Multi-Threaded Run Time System Debugger. 130
Application-Specific Command-Line Arguments 130
Accessing 131
Providing Arguments on Targets That Do Not Support Command-Line Arguments . 131
Run Time System Debugger Command Summary 131
Help 131
taskId, capsuleId, portId 131
Running a Model 132
Thread Commands. 132
tasks 132
attach <taskId> 133
detach <taskId> 133
Informational Commands 133
saps 134
system [<capsuleId> [<depth>]] 134
info 135
printstats <taskId> 136
Tracing Commands. 137
log <category> <detail-level> 137
Control Commands. 139
exit. 139
go [<n>]. 139
step [<n>] 139
quit 139
continue. 139

9 Inside the C++ Services Library . . . 141

Organization of the Services Library Source . . . 141

$RTS_HOME . . . 141

Configuration Naming Convention . . . 141

Platform Name (or Configuration) . . . 142

Target Base Name . . . 142

Libset Name . . . 142

Summary . . . 143

Directory Structure . . . 143

codegen . . . 143

include . . . 143

config . . . 143

target . . . 144

lib . . . 144

libset . . . 144

src . . . 144

tools . . . 144

Configuration Preprocessor Definitions . . . 144

USE_THREADS . . . 145

RTS_COUNT . . . 145

DEFER_IN_ACTOR . . . 145

INTEGER_POSTFIX . . . 145

LOG_MESSAGE . . . 146

OBJECT_DECODE . . . 146

OBJECT_ENCODE . . . 146

OTRTSDEBUG . . . 146

RTREAL_INCLUDED . . . 147

PURIFY . . . 147

RTS_INLINES . . . 147

RTS_COMPATIBLE . . . 147

HAVE_INET . . . 147

INLINE_CHAINS . . . 148

INLINE_METHODS . . . 148

RTFRAME_CHECKING . . . 148

RTFRAME_THREAD_SAFE . . . 149

RTFRAME_USE_FREELIST . . . 150

RTMESSAGE_PAYLOAD_SIZE . . . 150

OBSERVABLE . . . 150

Creating the Minimum Services Library Configuration 151
Integrating External IPC Into a Model 151
Build Versus Buy. 152
IPC Basics 152
Single-Threaded IPC 153
Using Signal Handlers 153
Polling a Flag 153
Multi-Threaded IPC. 153
Dedicated Blocking Capsule 154
Processing Overhead 154
Custom Peer Controller 154
Enhancement to the RTPeerController Class 155
Adding Support For New IPC Mechanisms. 155
Design Components 156
Concurrency Note 157
Controller Usage 158
Usability Note 158
IPC Options Summary 159
Optimizing Designs 160
Capsule Instances and Capsule Behavior 160
Incarnation (Frame::Base::incarnate()) 160
Guards 161
State Machines 161
Capsules versus Data 161
Capsule Functions 162
RTDataObjects 162
Unnecessary Sends 162
Sending Data by Value in Messages 162
Cross Thread Message Sending 163
General C++ Performance Notes 163
Additional Design Considerations. 164
Hardware Differences 164
Availability of External Library on Different Platforms 164
Toolchains. 165
10 Configuring and Customizing the Services Library 167
Configuration and Customization. 167
Changing Pre-Processor Macros 168
Changing Build Options 170
Overriding Virtual RTActor Operations From the Toolset 171

Overriding or Adding Operations and Classes 172
Overriding Operations Within a Model 173
Overriding Operations by Creating a New Target. 174
Building the Services Library 175
Updating a Component to Use a Different Services Library 176
11 Model Properties Reference. 177
Generalization and Properties 177
Expanded Property Symbols 178
Environment Variables and Pathmap Symbols 178
C++ Model Element Properties 179
GenerateClass (Class, C++) 181
ClassKind (Class, C++) 181
ImplementationType (Class, C++) 182
HeaderPreface (Class, C++) 182
HeaderEnding (Class, C++) 182
ImplementationPreface (Class, C++) 183
ImplementationEnding (Class, C++) 183
PublicDeclarations (Class, C++) 183
ProtectedDeclarations (Class, C++). 183
PrivateDeclarations (Class, C++). 183
GenerateDefaultConstructor (Class, C++) 184
DefaultConstructorVisibility (Class, C++) 184
DefaultConstructorExplicit (Class, C++). 184
DefaultConstructorInline (Class, C++) 184
GenerateCopyConstructor (Class, C++) 184
CopyConstructorVisibility (Class, C++) 185
CopyConstructorExplicit (Class, C++) 185
CopyConstructorInline (Class, C++) 185
GenerateDestructor (Class, C++) 185
DestructorVisibility (Class, C++) 185
DestructorVirtual (Class, C++). 186
DestructorInline (Class, C++). 186
GenerateAssignmentOperator (Class, C++) 186
AssignmentOperatorVisibility (Class, C++) 186
AssignmentOperatorInline (Class, C++) 186
GenerateEqualityOperator (Class, C++) 187
EqualityOperatorsVisibility (Class, C++) 187

EqualityOperatorInline (Class, C++) 187
GenerateInequalityOperator (Class, C++) 187
AttributeKind (Attribute, C++) 187
InitializerKind (Attribute, C++). 188
OperationKind (Operation, C++). 188
Inline (Operation, C++) 188
ConstructorInitializer (Operation, C++). 189
CallFromDestructor (Operation, C++). 189
AssociationEndKind (Role, C++) 189
InitializerKind (Role, C++). 189
InitialValue (Role, C++). 190
HeaderPreface (Capsule, C++) 190
HeaderEnding (Capsule, C++) 190
ImplementationPreface (Capsule, C++) 190
ImplementationEnding (Capsule, C++). 191
PublicDeclarations (Capsule, C++). 191
ProtectedDeclarations (Capsule, C++) 191
PrivateDeclarations (Capsule, C++) 191
KindInHeader (Uses, C++) 192
KindInImplementation (Uses, C++). 192
Declaring a Private Copy Constructor or Assignment Operator in C++ Classes . . 192

C++ TargetRTS Properties. 195

GenerateDescriptor (Class, C++ TargetRTS) 196
Version (Class, C++ TargetRTS). 196
InitFunctionBody (Class, C++ TargetRTS) 196
CopyFunctionBody (Class, C++ TargetRTS) 197
DestroyFunctionBody (Class, C++ TargetRTS) 197
DecodeFunctionBody (Class, C++ TargetRTS) 197
EncodeFunctionBody (Class, C++ TargetRTS) 199
GenerateDescriptor (Attribute, C++ TargetRTS). 201
TypeDescriptor (Attribute, C++ TargetRTS) 201
NumElementsFunctionBody (Attribute, C++ TargetRTS) 201
GenerateDescriptor (Role, C++ TargetRTS). 202
TypeDescriptor (Role, C++ TargetRTS) 202
NumElementsFunctionBody (Role, C++ TargetRTS) 202

Version (Protocol, C++ TargetRTS) 202
BackwardsCompatible (Protocol, C++ TargetRTS) 203
TypeSafeSignals (Protocol, C++ TargetRTS) 203
Type Descriptors 203
What are Type Descriptors? 203
When are Type Descriptors Used? 204
Example Usage Patterns and Associated Type Descriptors 206
C++ Generation Properties 215
OutputDirectory (Component, C++ Generation) 215
CodeGenDirName (Component, C++ Generation) 215
ComponentUnitName (Component, C++ Generation) 216
CommonPreface (Component, C++ Generation) 216
CodeGenMakeType (Component, C++ Generation) 217
CodeGenMakeCommand (Component, C++ Generation) 217
CodeGenMakeArguments (Component, C++ Generation) 217
CodeGenMakeInsert (Component, C++ Generation) 217
CodeSyncEnabled (Component, C++ Generation) 218
Generate Model Tags (Component, C++ Generation) 218
C++ Compilation Properties 218
CompilationMakeType (Component, C++ Compilation) 219
CompilationMakeCommand (Component, C++ Compilation) 219
CompilationMakeArguments (Component, C++ Compilation) 220
CompilationMakeInsert (Component, C++ Compilation) 220
CompileCommand (Component, C++ Compilation) 220
CompileArguments (Component, C++ Compilation) 221
InclusionPaths (Component, C++ Compilation) 221
TargetServicesLibrary (Component, C++ Compilation) 222
TargetConfiguration (Component, C++ Compilation) 222
C++ Executable Properties 223
TopCapsule (Component, C++ Executable) 223
PhysicalThreads (Component, C++ Executable) 223
Physical Thread Properties 224
Logical Threads 225
Physical Threads 225
ExecutableName (Component, C++ Executable) 225
DefaultArguments (Component, C++ Executable) 226
LinkCommand (Component, C++ Executable) 226
LinkArguments (Component, C++ Executable) 226

UserLibraries (Component, C++ Executable) . . . 226
UserObjectFiles (Component, C++ Executable) . . . 227
C++ Library Properties . . . 227
LibraryName (Component, C++ Library) . . . 227
BuildLibraryCommand (Component, C++ Library) . . . 228
BuildLibraryArguments (Component, C++ Library) . . . 228
LogicalThreads (Component, C++ Library) . . . 228
C++ External Library Properties . . . 229
GenerateClassInclusions (Component, C++ External Library) . . . 229
CodeGenDirName (Component, C++ External Library) . . . 229
InclusionPaths (Component, C++ External Library) . . . 230
Libraries (Component, C++ External Library) . . . 230
LogicalThreads (Component, C++ External Library) . . . 231
12 Services Library Class Reference . . . 233
RTDataObject Subclasses . . . 234
RTActor . . . 234
RTActor::msg and RTActor::getMsg . . . 235
RTActor::logMsg . . . 236
RTActor::unexpectedMessage . . . 236
RTActor::context . . . 237
RTActor::getError . . . 237
RTActor::getIndex . . . 238
RTActor::getName . . . 238
RTActor::getTypeName . . . 238
RTActor::isType . . . 239
RTActor::getCurrentStateString . . . 239
RTActorClass . . . 239
RTActorRef . . . 240
RTActorRef::size . . . 241
RTActorId . . . 241
RTActorId::isValid . . . 242
RTController . . . 242
RTController::getError . . . 243
RTController::strerror . . . 244

RTController::perror 244
RTController::name 244
RTController::abort 245
Exception 245
Exception Signals. 246
RTExceptionSignal. 247
Frame 247
Frame::classIsKindOf 248
Frame::className 248
Frame::classOf 249
Frame::deport. 249
Frame::destroy 250
Frame::import. 251
Frame::incarnate 253
Frame::incarnationAt 257
RTInSignal 257
RTInSignal::purge 258
RTInSignal::purgeAt. 258
RTInSignal::recall. 259
RTInSignal::recallAt 259
RTInSignal::recallAll. 260
RTInSignal::recallAllAt 261
Log 261
Log::show and Log::log 262
Log Miscellaneous Operations. 263
RTMessage 264
RTMessage::getPriority 265
RTMessage::getSignalName 265
RTMessage::getData 266
RTMessage::getType 266
RTMessage::sapIndex0 266
RTMessage::sap 267
RTMessage::isValid 267
RTMessage::defer 268
RTObject_class 268

RTOutSignal. 270
RTOutSignal::send 270
RTOutSignal::sendAt 271
RTOutSignal::invoke 272
RTOutSignal::invokeAt 274
RTOutSignal::reply 275
RTProtocol 275
RTProtocol::size 277
RTProtocol::purge. 277
RTProtocol::purgeAt 278
RTProtocol::recall 278
RTProtocol::recallAt 279
RTProtocol::recallFront. 279
RTProtocol::recallAll 280
RTProtocol::recallAllAt 280
RTProtocol::recallAllFront. 281
RTProtocol::bindingNotification 281
RTProtocol::bindingNotificationRequested 281
RTProtocol::registerSAP. 282
RTProtocol::deregisterSAP. 282
RTProtocol::registerSPP. 283
RTProtocol::deregisterSPP. 284
RTProtocol::isIndexTo. 284
RTProtocol::indexTo 284
RTProtocol::isBoundAt 285
RTProtocol::isRegistered 286
RTProtocol::getRegisteredName 286
RTSymmetricSignal 286
Example 286
RTTimerId 287
RTTimerId::isValid 287
RTTimespec. 287
tv_sec and tv_nsec 288
RTTimespec::RTTimespec 289
RTTimespec::getclock 289

RTTimespec Basic Comparison Operators 290
RTTimespec Assignment Operators 291
RTTimespec Basic Arithmetic Operators 291
Timing 291
Timing::informAt 292
Timing::informIn 293
Passing Data in a Timer Message 294
Timing::informEvery 295
Timing::currentTime 296
Timing::cancelTimer 297
Timing System Clock Operations 298
RTTypedValue 299
Port Services 299
External Port Service 299
Index 301

Figures

Figure 1 Target Observability 32
Figure 2 Protocol Definition for NewProtocol1 47
Figure 3 Generated Protocol Class for NewProtocol1 48
Figure 4 Simplified RTOutSignal Definition 49
Figure 5 Simplified RTInSignal Definition 49
Figure 6 Compilation Paradigm for Producing C++ Executable 51
Figure 7 Component Diagram 58
Figure 8 Association Between Two Capsules 79
Figure 9 Association Between a Capsule and a Class 79
Figure 10 An Array Data Member 81
Figure 11 Macro Usages 85
Figure 12 Single-Threaded Services Library 97
Figure 13 Multi-Threaded Services Library 98
Figure 14 C++ Services Library Framework 100
Figure 15 PingPong Protocol and Talk Port 104
Figure 16 Ping Pong Model Class Diagram 105
Figure 17 Container Capsule Structure Diagram 106
Figure 18 Information Flow for the Custom Controller Based IPC Mechanism . . 155

Tables

Table 1 Visibility Level Descriptions 62
Table 2 Summary of Rules for Unit Visibility 63
Table 3 Definitions for Minimum Services Library Configuration 151
Table 4 IPC Options 159
Table 5 Operations 300

Preface

This manual provides an introduction to Rational Rose RealTime C++. The C++ module joins the current C and Java modules to add the ability to design, generate, build, and debug applications in the C++ language to the Rose RealTime product.

This manual is organized as follows:

- *Overview* on page 27
- *Using C++ Code in Models* on page 33
- *Code Generation* on page 39
- *Generating and Sharing External Library Interfaces* on page 57
- *Classes and Data Types* on page 73
- *C++ Services Library* on page 93
- *Running Models on Target Boards* on page 123
- *Command Line Model Debugger* on page 129
- *Inside the C++ Services Library* on page 141
- *Configuring and Customizing the Services Library* on page 167
- *Model Properties Reference* on page 177
- *Services Library Class Reference* on page 233

Audience

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

Other Resources

- Online Help is available for Rational Rose RealTime.

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

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

- To send feedback about documentation for Rational products, please send e-mail to `techpubs@rational.com`.

- For more information about Rational Software technical publications, see: `http://www.rational.com/documentation`.
- For more information on training opportunities, see the Rational University Web site: `http://www.rational.com/university`.
- For articles, discussion forums, and Web-based training courses on developing software with Rational Suite products, join the Rational Developer Network by selecting **Start > Programs > Rational Suite > Logon to the Rational Developer Network.**

Rational Rose RealTime Integrations With Other Rational Products

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

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

Contacting Rational Customer Support

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

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

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

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

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

Overview 1

Contents

This chapter is organized as follows:

- *Using this Guide* on page 27
- *Workflows for Your Host Workstation and Embedded Target* on page 28
- *Using C++ Code in Models* on page 28
- *Model Properties* on page 29
- *C++ Services Library* on page 30
- *Code Generation* on page 30
- *Compilation* on page 31
- *Model Executables* on page 31
- *Target Observability* on page 32

Using this Guide

Use this guide to learn how to use the C++ Language Add-in to build, compile and debug C++ based Rational Rose RealTime models. Information is provided on how to deploy the model executables to a target system, and how to optimize and configure your target to fit your project's needs.

Using the C++ Language Add-in, you can produce C++ source code, compile it, then build an executable from the information contained in a Rational Rose RealTime model. The code generated for each selected model element is a function of the model specification, model properties, and the model's design properties. Model properties provide the language-specific information required to map your model onto C++.

To understand how the C++ language add-in works, first you should understand the main aspects of the language add-in. In addition, there are a number of C++ example models that demonstrate features of the toolset, the model properties, and the C++ Services Library.

Note: You can find example models in the Examples directory located in the root Rational Rose RealTime installation directory.

Workflows for Your Host Workstation and Embedded Target

There is an expected sequence of work activities for taking a model from early prototyping to final production.

During the initial phases of model development, it is advisable to run your models primarily on the host workstation to keep the modify-compile-debug cycle as short as possible. You can then take advantage of workstation-based debug tools, such as C++ source-level debuggers and C++ analysis tools (such as PurifyTM) that may not be available on your target platform. For many projects, this is the final step, if you are using a workstation-based target.

The final step for projects using some form of RTOS-based embedded target platform is to compile the model for that target platform, and download and run it on the target. These tasks are explained in *Running Models on Target Boards* on page 123.

The workflow of Rational Rose RealTime is intended to provide as much up-front verification and debugging as possible in the the host workstation environment. This environment is typically provided by a combination of Rose RealTime host-based tools and workstation-based C++ tools. This leaves a minimal amount of debugging to do on the target, where debugging is typically more difficult. The use of target observability to monitor and control models at the model level greatly enhances the ability to debug target applications.

Using C++ Code in Models

C++ is used as a detail-level coding language in Rational Rose RealTime. At a higher level of abstraction, the program is described both structurally and behaviorally as a graphical model using the Unified Modeling Language (UML). C++ code can be added to a variety of behavioral elements in a UML model. A graphical state diagram shows the allowable sequence of events that a capsule can process. Detailed code must be added to the states, transitions, and operations in the model. There are no restrictions on the code that you enter into your model. You can also make use of external C++ classes (that is, classes defined outside of Rational Rose RealTime) and libraries in your model.

Rational Rose RealTime is designed to be the central interface point for developing C++ based models, and provides support for all activities in the development process, including requirements capture, high-level design, coding, versioning, loadbuilding, and testing. Rational Rose RealTime depends on your existing C++ tools to handle language-specific work - it coordinates and controls these activities in the context of your model. For example, the toolset does not include a C++ compiler or linker. Rational Rose RealTime requires that you already have a C++ compiler or linker installed and accessible in your environment prior to compiling a C++ model.

Model Properties

The notations supported in Rose RealTime are more abstract than the C++ programming language. Model properties enable you to provide language-specific information that is not expressed in the notation, but that is necessary for generating and building source code. Each model property can be assigned a model property value. When a model element is created, each model property is assigned a default value, which you can optionally modify.

See the property set mechanism in the *Toolset Guide*.

In order to build source code, the code generator also generates makefiles that specify how to build the generated source code. Certain properties affect how these makefiles are to be generated and their contents.

You can use model properties to:

- Add an #include directive automatically to more than one file.
- Suppress the generation of default or copy constructors.
- Specify the format of a constructor or relational operator.
- Suppress the generation of a class.
- Add compilation flags, include paths, and other build related settings.

Controlling a particular aspect of code generation may require several model properties.

Not all model components for which code is generated require model properties. For example, there are no model properties for inheritance relationships, yet the C++ generator produces base lists and #include directives from inheritance relationships. In such cases, information obtained from specifications is sufficient to control code generation.

C++ Services Library

The C++ Services Library is at the heart of the C++ Language Add-in. It is essential that you understand its architecture if you are to start optimizing and configuring it for your project's needs.

The behavior of a model is specified using a combination of capsule state diagrams and operations defined on classes and capsules. The relationships in the model are specified with a combination of capsule structure and class diagrams. When a model is built, these abstractions are automatically converted to implementation. The Rational Rose RealTime Services Library provides a set of built-in services commonly required in real-time systems. These services include: state machine handling, message passing, timing, concurrency control, thread management, and debugging facilities. The Rational Rose RealTime Services Library provides a standard set of services across all supported platforms, so that your model can be readily ported to different target platforms.

The Rational Rose RealTime Services Library provides the following facilities:

- The mechanisms that support the implementation of concurrent communicating state machines.
- Thread management and concurrency control.
- Dynamic structure.
- Timing.
- Inter-thread communication.
- Observation and debugging of a running model.

Code Generation

This section discusses some aspects of how a model is converted to C++ code and compiled. This should clarify the output you will see in the Build Log window and help you browse the generated code.

The C++ generator uses the specifications and model properties of elements in the current model to produce C++ source code. You generate code for a component which in turn references a set of elements from the logical view. The location of the source files that are generated for elements referenced by (or assigned to) a component is determined by the name of the component, the location of your model file (.rtmdl), and the OutputDirectory (Component, C++ Generation) property.

For more information on code generation, see *Code Generation* on page 30.

Modifying Generated Code

Rational Rose Real Time with Code Sync provides a means to modify certain identified sections of the generated code from outside the toolset. You can make changes to specific portions of the generated code using an external editor and, using Code Sync, have these changes propagated back into the model. Do not make changes to the generated code outside of the identified sections because you may lose these changes. For more information, see *Using C++ Code in Models* on page 28.

Compilation

The C++ Language Add-in will convert a model to C++ code but does not include the compiler which will build from the generated source code. Before trying to build a generated model ensure that your compiler tools are correctly installed. For example, try building a simple C++ program from the command line. If that works, then the C++ Language Add-in will be able to properly invoke the configured compiler.

Linking the Model with the Services Library

Rose RealTime models are created by linking the user-compiled model files with the pre-compiled C++ Services Library into a single executable file. All the versions of the pre-compiled Services Libraries are available for all supported hosts. In addition the Services Library can be ported and built for new hosts as required.

Model Executables

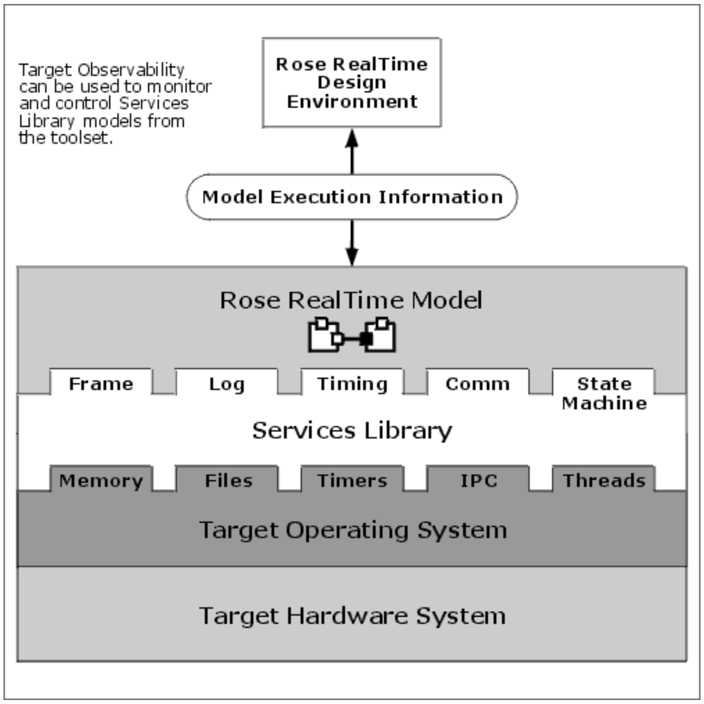
Compiling a Rose RealTime model results in a stand-alone executable. The generated executable is not connected to the Rose RealTime session unless specified. If targeted for a workstation platform, the model can be run simply by typing the name of the generated executable on the command line. If targeted for a real-time operating system, the resulting executable must be downloaded to the target and executed using the tools particular to that target operating system.

For more information,see *Running Models on Target Boards* on page 123.

Target Observability

Rational Rose RealTime's graphical observation tools are a sophisticated, yet intuitive debugging environment allowing you to use the toolset to execute, monitor and control a model running on the Services Library, even on a remote target platform. The Services Library is a high-performance implementation intended for use in a wide-range of real-time products.

Figure 1 Target Observability


Target Observability can be used to monitor and control Services Library models from the toolset.
Rose RealTime Design Environment
Model Execution Information
Rose RealTime Model
Frame
Log
Timing
Comm
State Machine
Services Library
Memory
Files
Timers
IPC
Threads
Target Operating System
Target Hardware System

Using C++ Code in Models 2

Contents

This chapter is organized as follows:

- *Adding C++ Code to Models* on page 33
- *The Syntax of Code Segments* on page 34
- *Encapsulating Target Specific Behavior* on page 34
- *Code Sync* on page 34
- *Macros and Arguments Available to State Machine Code* on page 35

Adding C++ Code to Models

You can use C++ in your Rational Rose RealTime model to:

- Perform detailed actions that occur on transitions.
- Perform detailed actions that occur on state entry or exit.
- Code capsule operations that can then be invoked from any other code segment (the common name for the C++ code contained inside any one model element, such as a transition code segment). Capsule functions can be used to capture common operations, which may be performed as part of several different transitions, state entry actions, and so forth, or to simplify the transition code.
- Perform condition tests as part of choice points or event guard conditions.
- Write operations on classes.

You can also define C++ classes and functions outside of your model and make use of them within your model, or make calls to other existing C/C++ libraries from your model. As long as the external C++ code is visible to the compiler and linker you can use them in a model.

The Syntax of Code Segments

C++ code is added to your model by filling-in the body portion of operations, transitions, etc. You do not need to add curly braces to the beginning and end of any action code segments. These will be added automatically by the code generator.

Choice Point Code Condition Segment

The choice point segments are created as operations which return an **int**. The condition C++ code that is entered in a choice point must have a return statement that returns true 1 or false 0. You can have any number of other C++ statements in the choice point segment as long as it returns an **int**.

Encapsulating Target Specific Behavior

The workflow of Rational Rose RealTime is intended to provide as much up-front verification and debugging as possible in the tool-rich environment of the host workstation. This environment is typically provided by a combination of Rose RealTime host-based tools and workstation-based C++ tools. This leaves a minimal amount of debugging to do on the target, where debugging is typically more difficult.

Isolate any platform-specific behavior in a few well-encapsulated places. If direct calls to native OS functions or target-specific libraries are spread throughout your model, you are restricted to compiling and testing on target. This can cause serious bottlenecks for testing and bug-fixing at the most crucial times in the project as developers line up for lab time, or unstable hardware makes target testing difficult. By encapsulating target-specific calls to a few key parts, the rest of the model can readily be tested on the workstation.

Code Sync

Code Sync lets you make changes to the generated code from outside the toolset within an IDE (Integrated Development Environment) or text editor of your choice, and update your model with your changes.

For more information, see Using Code Sync to Change Generated Code in the Toolset Guide.

Making Changes Outside the Toolset

In order for the changes to be recaptured into the model, Code Sync must be enabled, and the changes must be made to designated Code Sync areas.

Identifying Designated Code Sync Areas

Designated Code Sync areas are always delimited by the Code Sync identification tags. These areas may be modified from the generated code and captured into the model using the Code Sync feature.

User modifiable code for C++ is identified as follows:

```
// {{{USR
<insert or modify code here>
// }}}USR
```

In some cases where a field is omitted or left as its default, the code generator may generate an optimized code pattern that does not provide the empty Code Sync areas or its identification tags. If you use Code Sync area for an area which has been optimized out, you must provide a non-default value for the field (such as a comment) within the model, then re-generate before you can modify that Code Sync area.

De-activating Code Sync

Each component, by default, has Code Sync activated. To de-activate Code Sync, change the **CodeSyncEnabled** property of the **Generation** tab for the component.

Macros and Arguments Available to State Machine Code

The following macros and variables can be used in transition C++ code segments.

CALLSUPER

You can make a call to the superclass version of the same transition that has the same parameters from within any C++ transition code segment. **CALLSUPER** is redefined for each transition to point to the correct code segment name in the superclass. **CALLSUPER** cannot be used to invoke the superclass version of a different function. Use the `SUPER::`**`func`**`()` syntax to call the superclass version of a different function explicitly.

Note: You can only use **CALLSUPER** when the transition triggering event is the same for the base and derived classes.

SUPER

SUPER can be used to reference the generated superclass name from a capsule transition code segment, rather than having to type in the full capsule name.

Use the `SUPER::`**`func()`** syntax to call the superclass version of a function.

The **SUPER** macro is only defined for capsules. You cannot use this macro with data classes.

RTDATA and rtdata

RTDATA is used for backwards compatibility, and is defined as **rtdata**.

The **rtdata** parameter available to all transition code is a cast version of the data in a message. The **rtdata** parameter is cast to the highest common superclass of the possible data classes for the given code segment.

If the data class of a signal that triggered a transition was **int**, the **RTDATA** macro or **rtdata** parameter would be used as follows:

```
// both these statements are equivalent
const int * i = rtdata;
const int * j = RTDATA;
```

Note: **RTDATA** should only be used for backwards compatibility in models developed with previous versions of the C++ Services Library. In new models, use **rtdata**.

rtport

In each transition code block, you have access to a variable called **rtport** which is a pointer to the (common base class) **RTProtocol** for the port on which the message which triggered this transition was received. The primary purpose of **rtport** is for replying to messages:

rtport->ack().reply();

Limitations

Opening Rational Rose Models in Rose RealTime

When opening a Rational Rose model in Rose RealTime, C++ properties are ignored. This means that if you defined classes as **structs**, **enums**, **typedefs**, and unions in Rational Rose using the **Implementation Type** or **ClassKey C++** properties, when you bring these classes into Rose RealTime, these property types are lost. You must manually modify these classes to change these properties.

Using the C++ Analyzer in Rose RealTime

The C++ Analyzer cannot properly model structs, enums, typedefs, unions, and #defines in Rose RealTime. An alternative is to import the code into Rose RealTime.

Code Generation 3

Contents

This chapter is organized as follows:

- *Model to Code Correspondence* on page 39
- *Code Generator Behavior* on page 50

Model to Code Correspondence

This chapter discusses some relevant aspects of the Rational Rose RealTime code generation interface to clarify the output that users will see in the compiler output and for browsing the generated code. Developers who need to start debugging their C++ designs through external debugging tools also need to understand the generated code structure.

The C++ generator uses the specifications and model properties of elements in the current model to produce C++ source code. You generate code for a component which in turn references a set of elements from the logical view. The location of the source files that are generated for elements referenced by (or assigned to) a component is determined by the name of the component, the location of your model file (.rtmdl), and the **OutputDirectory** (**Component**, **C++ Generation**) property.

If logical view elements have not been assigned to components, either directly or by means of a dependency to other elements that are, the C++ code generator will not see those elements and they will never be generated to source.

Associations

An association is a relationship among two or more elements. The ends of each association are called association ends. Ends may be labeled with an identifier that describes the role that an associate element plays in the association. An end has both generic and language specific properties that affect the generated code which traverses to that end. For example, marking an end navigable means that traversal from the opposite role's class to this role's class is to be implemented.

By default if an end is named, association, aggregation, and composition relationships are represented in code as an attribute in the client class. The code generation does not generate attributes for ends which are not named.

Valid Code Generation Associations

Only the association relationships described below are considered by the C++ code generator.

Capsule to protocol (port)

For these associations, the code generator generates a port on the capsule. Associations between capsules and protocols are only navigable from the capsule to the protocol. The port specification page controls the specific characteristics of the port: public, protected, wired, etc.

Class to class (data member)

By default, the code generator generates a data member (attribute) for navigable and named ends of associations. Several factors affect the code that is actually generated:

- the **AssociationEndKind** (**Role**, **C++**) property affects if a member or global data member is generated
- the cardinality affects whether an array of attributes should be created
- the containment affects whether the attribute should be a reference (pointer) or an object

Capsule to class (data member)

For these associations, the code generator by default generates a data member (attribute) on the capsule. A class cannot navigate to a capsule. The same factors affecting class to class associations affect capsule to class.

Capsule to capsule (capsule role)

For these associations, the code generator generates a capsule role on the client capsule. Associations between capsules are always unidirectional. The capsule role specification page controls the specific characteristics of the capsule role: optional, fixed, plug-in, cardinality, etc.

Dependencies

When the C++ generator produces code for an element (the client) that uses another element (the supplier), the C++ generator can produce either an include directive referencing the file that contains the supplier class or a forward reference to the supplier.

You can configure which directive (include statement, forward reference, or nothing) is generated in the header file (.h) and in the implementation file (.cpp) with the **KindInHeader** (**Uses**, **C++**) and **KindInImplementation** (**Uses**, **C++**) properties.

Classes

The different kinds of classes that can exist in a model are:

- parameterized
- utility
- parameterized utility
- instantiated
- instantiated utility
- normal meta

Currently, the C++ code generator only supports classes and utility classes. There is no support for code generation of the other kinds of classes. They are ignored by the code generator, and a warning is issued.

Each class is generated in its own .h and .cpp file.

Header file (.h)

The following code is generated in the header file:

- Inclusions, forward references, value of the **HeaderPreface** (**Class**, **C++**) property.
- Class definition and base list (taken from any generalization relationships).
- Attributes generated from class associations or explicitly defined as attributes.
- Standard operations.
- User-defined operations.
- If **GenerateDescriptor** (**Class**, **C++ TargetRTS**) property, **extern** statement for a class type descriptor of type **RTObject_class**.
- A **RTTypedValue struct** for type descriptor.
- Value of the **HeaderEnding** (**Class**, **C++**) property.

Implementation file (.cpp)

The following code is generated in the implementation file:

- Inclusions, forward references, value of the **ImplementationPreface** (**Class**, **C++**) property.
- Operation bodies for Standard operations and User-defined operations.
- If **GenerateDescriptor** (**Class**, **C++ TargetRTS**) property set, default and user-defined type descriptor function bodies are generated.
- If **GenerateDescriptor** (**Class**, **C++ TargetRTS**) property set, the type descriptor structure is initialized.
- Value of the **ImplementationEnding** (**Class**, **C++**) property.

Properties that affect the way classes are generated

- The **GenerateClass** (**Class**, **C++**) property is used to turn off generation of a class.
- The **ClassKind** (**Class**, **C++**) property can be used to generate **typedefs**, **structs**, and unions instead of a class.
- The **GenerateDescriptor** (**Class**, **C++ TargetRTS**) property controls the generation of the classes type descriptor.

Logical Packages

No code is actually generated for logical packages. They provide a good way of assigning a set of elements to a component.

In the logical design of a system, related classes are grouped into packages. In a Rose RealTime model you define the mapping from logical design to a physical design via components. You can explicitly assign a logical package to a component. This assignment is contained in the logical package’s specification. Assigning a package to a component is a shorthand method of assigning every element contained within the package to the component.

User-defined Operations

When generating code for a class, the C++ generator produces a member function for each operation that is listed in the class or capsule specification. For each such operation, the C++ generator produces:

- A member function declaration in the header file for the class.
- A function body in the implementation file containing the C++ code added to the Code region. You should never modify generated code, unless Generate (C++) is "declaration only".

The C++ generator uses the information in the operation's specification as well as operation Model properties, to generate the member function. For example, the **OperationKind** (**Operation**, **C++**), **Inline** (**Operation**, **C++**), **ConstructorInitializer** (**Operation**, **C++**) properties can affect the way in which operations are generated.

Note: The C++ Generator Constructor Explicit automatically generates code for Standard operations that are generated based on the values of class properties. You do not need to list these operations in the class specification unless you want to override them.

Overriding Virtual Operations

To override an operation defined in a parent class from within a subclass, do the following:

1. Ensure that the operation on the parent has the **Polymorphic** option checked.
2. Create a new operation on the subclass with the same signature as the operation in the parent.

The term signature designates the combination of operation name, the types and order of its parameters, and, if the operation is instance scoped, its query and/or qualifiers.

Standard Operations

When generating code for a class, the C++ generator may also generate an implementation for one or more standard operations. The C++ generator determines whether and how to generate member functions for standard operations from class property values and operations already defined on the class.

For example, the class properties **GenerateAssignmentOperator** (**Class**, **C++**), **AssignmentOperatorVisibility** (**Class**, **C++**), and **AssignmentOperatorInline** (**Class**, **C++**) determine whether or not the C++ generator produces an assignment operation for a class and, if so, its definition and visibility.

For each standard operation that is enabled, the C++ generator produces:

- A member function declaration in the header file for the class.
- A function body in the implementation file containing the implementation of the standard operation.

Overloading a standard operation

You create an operation on the class with the same name and signature as a standard operation. In this case, the C++ generator will determine that you have defined your own standard operation and will not generate a default.

Attributes

By default, an attribute is represented in code as an attribute in the client class. It is a private implementation data member whose type is specified in the model, and whose name is based on the value of the attribute name property.

The derived property controls whether or not a data member is generated.

The type of the data member is affected by:

- The type specified.
- Class scope causes the static keyword to be generated for the data member.
- An initial value may be specified for the attribute.
- Visibility adornments and model properties affect access of the data member.

If the type of the attribute references declarations of other classes, you must draw a dependency relationship from the class containing the attribute to the referenced class.

Capsules

The C++ generator converts capsule structures and state diagrams into C++ code that will integrate into the C++ Services Library Framework. A metaclass **RTActorClass**, which represents a capsule's properties that belong to a class as a whole (rather than to any of the instances), is generated for each capsule.

Some of the code segments can be modified from the generated code and captured into the model using the Code Sync feature. For more information, see Using Code Sync to Change Generated Code in the Toolset Guide.

Each capsule is generated in its own .h and .cpp file.

Header file (.h)

The following code is generated in the header file:

- Inclusions, forward references, value of the **HeaderPreface** (**Capsule**, **C++**) property.
- Capsule definition (capsule class name generated as **CapsuleName_Actor**) and base list (taken from any generalization relationships). All capsules are subclasses of the **RTActor** class.
- Attributes generated from class associations or explicitly defined as attributes.
- Ports generated as attributes of a protocol role type.

```
protected:
   // {{{RME protocolClass 'Timing' port 'NewPort1'
   Timing::Base NewPort1;
```

- User-defined operations.
- Support operations for state behavior implementation.
- Value of the **HeaderEnding** (**Capsule**, **C++**) property.

Implementation file (.cpp)

The following code is generated in the implementation file:

- Inclusions, forward references, value of the **ImplementationPreface** (**Capsule**, **C++**) property.
- Operation bodies for standard and user-defined operations.
- Transition code, choice point code.
- State behavior implementation.
- Value of the **ImplementationEnding** (**Capsule**, **C++**) property.

Capsule State Diagrams

Capsule state diagrams are parsed by the C++ generator and included in the generated code for the owning capsule C++ class. All C++ code added to a state diagram is added to operations defined on the capsule class.

Note: Protocol and class state diagrams are ignored by the C++ generator.

The following is an example definition for an operation generated for a transition named **ReceiveInt**.

```
INLINE_METHODS void NewCapsule1_Actor::transition2_ReceiveInt( const
void * rtdata, Timing::Base * rtport )
{
   int i = *(const int *)rtdata;
   cout << "Received an int: " << i;
}
```

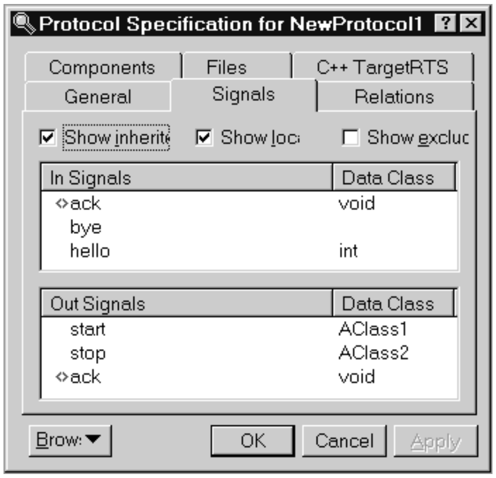
You should never modify code directly in the generated source files. It may however be useful to understand that transitions are generated as operations when debugging code using source code debuggers.

Protocols

Each protocol is generated in its own .h and .cpp file. The protocol is generated as Base and Conjugate protocol roles. Since ports will be a instance of a protocol role, it is important that you understand how a protocol is generated so that you can use the Communication services to send and receive messages via the port objects.

Figure 2 is a sample protocol definition, and Figure 3 shows a simplified version of the classes that get generated. These examples should help you understand how to use the communication service within the detail code of your models.

Figure 2 Protocol Definition for NewProtocol1

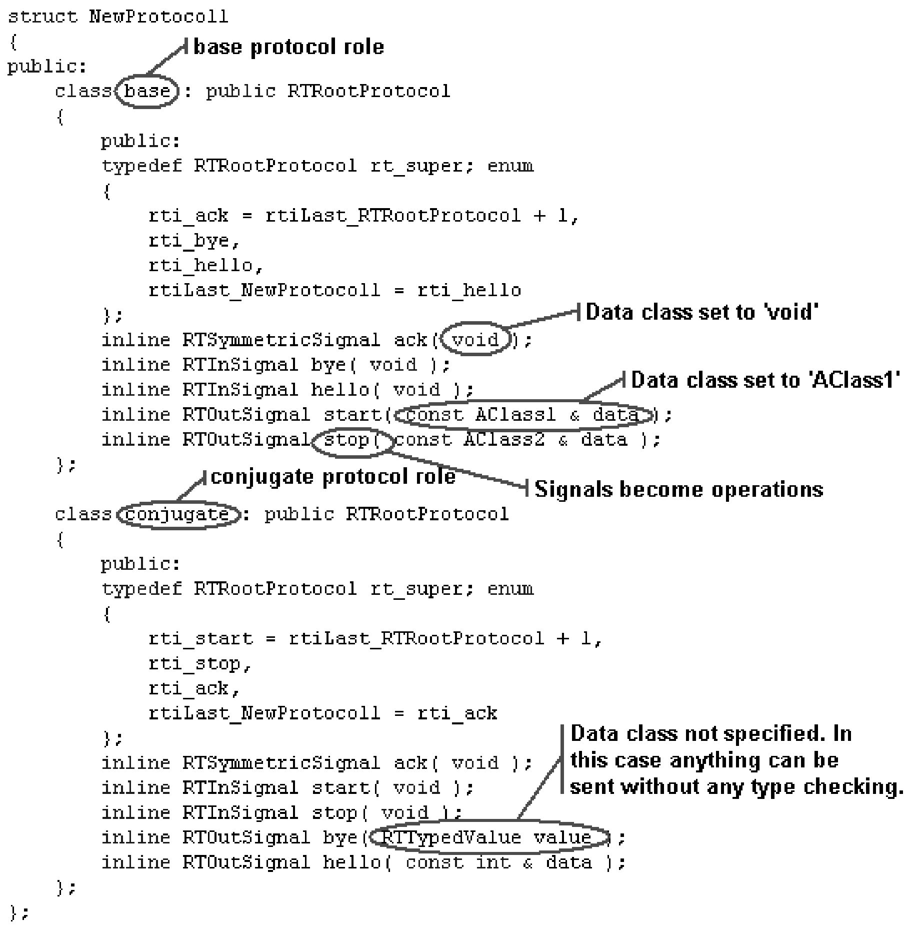


☑ Show inherite ☑ Show loc: ☐ Show excluc

In Signals	Data Class
<>ack	void
bye	
hello	int

Out Signals	Data Class
start	AClass1
stop	AClass2
<>ack	void

Figure 3 Generated Protocol Class for NewProtocol1



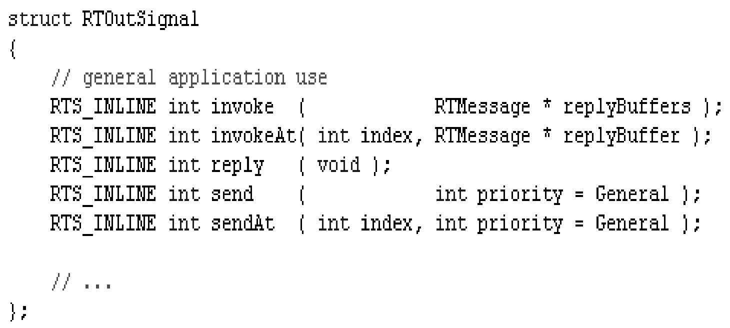
```
struct NewProtocol1
{
public:
    class base : public RTRootProtocol
    {
        public:
        typedef RTRootProtocol rt_super; enum
        {
            rti_ack = rtiLast_RTRootProtocol + 1,
            rti_bye,
            rti_hello,
            rtiLast_NewProtocol1 = rti_hello
        };
        inline RTSymmetricSignal ack( void );
        inline RTInSignal bye( void );
        inline RTInSignal hello( void );
        inline RTOutSignal start( const AClass1 & data );
        inline RTOutSignal stop( const AClass2 & data );
    };
    class conjugate : public RTRootProtocol
    {
        public:
        typedef RTRootProtocol rt_super; enum
        {
            rti_start = rtiLast_RTRootProtocol + 1,
            rti_stop,
            rti_ack,
            rtiLast_NewProtocol1 = rti_ack
        };
        inline RTSymmetricSignal ack( void );
        inline RTInSignal start( void );
        inline RTInSignal stop( void );
        inline RTOutSignal bye( RTTypedValue value );
        inline RTOutSignal hello( const int & data );
    };
};
```

base protocol role
Data class set to 'void'
Data class set to 'AClass1'
conjugate protocol role
Signals become operations
Data class not specified. In this case anything can be sent without any type checking.

For each signal in the protocol, an operation returning an **RTOutSignal** (for outgoing signals), a **RTInSignal** (for incoming signals), or a **RTSymmetricSignal** (for both incoming and outgoing) is generated in either the base or conjugate protocols. The data class (as specified in the **Protocol Specification** dialog box) becomes an argument to the operations.

The **RTOutSignal** and **RTInSignal** structures contain definitions for the actions that you can perform with the signals. For example, outgoing signals can be sent, and incoming signals can be deferred and recalled.

Figure 4 Simplified RTOutSignal Definition



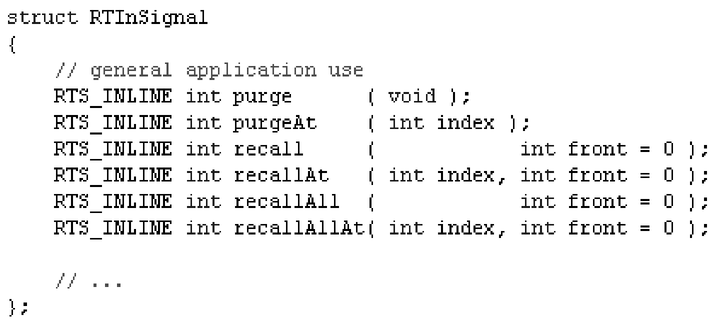
```
struct RTOutSignal
{
    // general application use
    RTS_INLINE int invoke  (            RTMessage * replyBuffers );
    RTS_INLINE int invokeAt( int index, RTMessage * replyBuffer );
    RTS_INLINE int reply   ( void );
    RTS_INLINE int send    (            int priority = General );
    RTS_INLINE int sendAt  ( int index, int priority = General );

    // ...
};
```

If you have a port assigned to the NewProtocol1 base role, you would send a `start` signal out of the `aPort` in the following way:

```
AClass1 mdata(43, 1.34);
aPort.start(mdata).send();
```

Figure 5 Simplified RTInSignal Definition



```
struct RTInSignal
{
    // general application use
    RTS_INLINE int purge      ( void );
    RTS_INLINE int purgeAt    ( int index );
    RTS_INLINE int recall     (            int front = 0 );
    RTS_INLINE int recallAt   ( int index, int front = 0 );
    RTS_INLINE int recallAll  (            int front = 0 );
    RTS_INLINE int recallAllAt( int index, int front = 0 );

    // ...
};
```

If you have a port assigned to the NewProtocol1 base role, you could recall all previously deferred bye signals on all port instances in the following way:

```
aPort.bye().recallAll();
```

Components

When generating a component, the C++ code generator creates a set of makefiles that contain rules for generating and building all elements referenced by the component. In addition, a component-wide .cpp and .h file may be created for certain types of components. These source files contain initialization, thread creation, and other classes and operations required by the C++ Services Library.

When the C++ Generator produces code for the elements referenced by a component, the resulting files are stored in a directory structure. The location and name of the root of this directory structure can be configured using the **OutputDirectory** (**Component**, **C++ Generation**) property.

By default, the directory is created in the same directory containing the model file (.rtmdl), and the name is derived from the name of the corresponding component.

Relationships and Elements Ignored by C++ Code Generation

The following elements in a model are ignored by the C++ code generator:

- Realizes relationships
- Package dependencies
- State diagrams on protocols and classes
- Collaboration diagrams
- Sequence diagrams
- Actors
- Use-cases
- Deployment diagrams

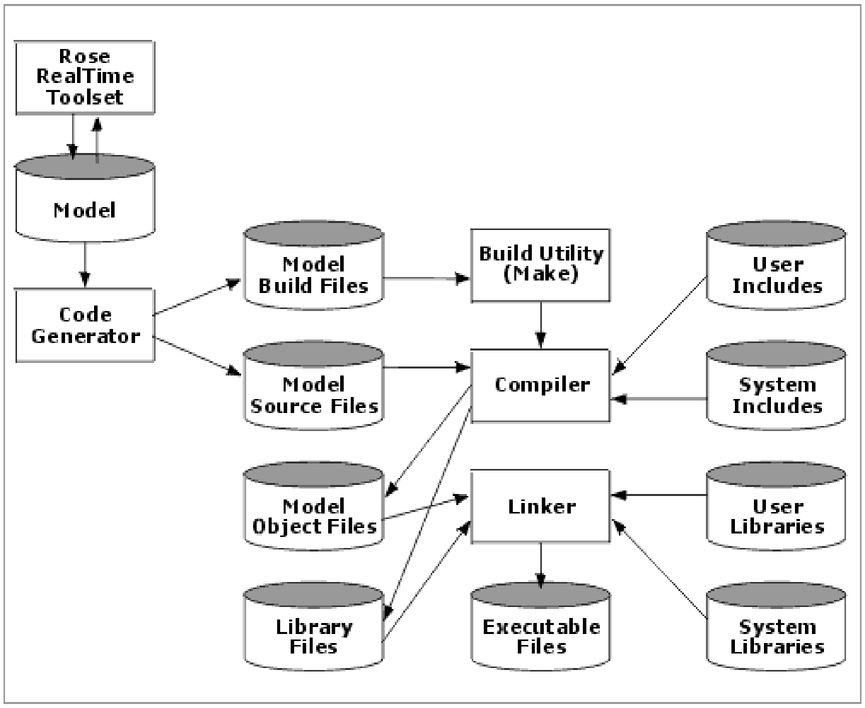
Code Generator Behavior

Code generation produces source files and makefiles for the items referenced by the component. When the source files are compiled, object code files are produced. Finally in the link stage, the object files from the top level component and all the components contained by aggregation (the whole component hierarchy) are then linked together to form an executable. The source code, object files, and executable are all build results.

Note: The source code generation, compilation, and linking is managed by the make utility, and is external from the Rose RealTime toolset. These build makefiles are called from within Rose RealTime to build a component.

The compilation paradigm for producing a working C++ executable is shown in Figure 6.

Figure 6 Compilation Paradigm for Producing C++ Executable



Incremental Generation

The code generation and compilation processes are driven by a third-party Make utility, whose behavior is dependent on **makefile** dependencies and file timestamps. Without makefile dependencies, incremental builds would produce incorrect builds. The code generator takes steps to reduce development churn and produce incremental builds quickly and reliably.

The code-generator reduces incremental compilation time by preserving previously generated files that do not need to change. When you build a component that has been previously built (or even partially built), the code generator attempts to preserve the previously built results. If the generated C++ files (header files and implementation files) do not need changing, they are not updated. This improves compilation performance because:

- if an implementation file does not need to be updated, its corresponding object file does not need to be recompiled, and
- if a header file does not need to be updated, all object files which depend on that header file do not need to be recompiled.

Consequently, the incremental generation behavior of the code generator greatly improves compilation performance.

The code generation also allows incremental code-generation by tracking its own dependencies for each invocation. Some Make utilities (such as ClearCase's clearmake and omake) can automatically track dependencies of build scripts. For other Make utilities, the code generator tracks all of the controlled units (CUs) that were read during each invocation. All of these model elements become dependencies (in a makefile sense) of the files generated by each invocation of the code generator. This dependency information is then available for the next incremental build, and the Make utility will only invoke the code generator to re-examine, and (if necessary) regenerate source code that depends on a CU that has changed. Consequently, the incremental behavior of the code generator safely reduces the time to generate subsequent builds.

The Effect of Controlled Units

Any single invocation of the code-generator will generate:

- a single specific classifier stored in its own controlled unit (CU), or
- all classifiers (that are referenced by the component) in a specific package, except for classifiers that are stored in their own CU, or
- all classifiers (that are referenced by the component) in the model, except for classifiers that are stored in their own CU or in a package CU

If a model is saved into one monolithic .rtmdl file, then every time you change anything in the model, every model element has to be re-examined during generation. To improve code generation performance it is recommended that you save your model as controlled units.

See Working with Controlled Units in the Guide to Team Development, Rational Rose RealTime for instructions on how to save models as controlled units.

The choice of controlled units does not affect compilation performance because the compiler reads generated source files (not controlled units), and the incremental generation behavior is independent of controlled units. The incremental behavior of the compiler is independent of the choice of controlled units.

Generated Code Directory Layout

The build output is contained in a separate directory from the model file. Each Component in a model is built in its own directory structure. There is an option in the **Component Specification** dialog box that allows the user to specify a different directory for this purpose.

Note: It is recommended that each component has a different output path. This is to avoid overwriting files for other components.

In the Component directory, there is a directory tree that separates the model files, generated source files, and build results, including the executable.

After building a Component, named "`Component1`", the default directory structure below the output directory would look like:

```
Component1\
   src\
   build\
```

src

This directory contains all C++ source files that have been generated for the component. Depending on the value of the component **C++ Generation** property called **CodeGenDirName**, source files may either appear directly in src or in a sub-directory of src as specified by the **CodeGenDirName** property. The generated code consists primarily of C++ representations of the classes from the users model. The code segments that contain the C++ code entered in various portions of the model are included in the generated source, including the transition actions, choice points, state entry and exit actions, operations, and so on.

There will be a header and source file generated for each model element referenced by the component. The files will have the same name as the elements from the model. In most cases, generated classes and other constructs will be named as defined in the model.

For each capsule, a class is generated with the name:

```
<capsule name>_Actor
```

The best way to understand the generated source code is to build one of the example models, or tutorials, and then browse the generated source code.

build

The build directory contains the result of the compilation. The object files as well as the linked executable are included in these results. By default the executable name will be the name of the top-level capsule for the Component. You can change this by specifying a different name in the **General** tab of the **Component Specification** dialog box.

Code Generator Command-Line Arguments

There are three methods of passing command-line parameters to the external code generator:

1 Adding the command-line options to the `ROSERT_RTGENOPTS` environment variable.

OR

2 Modifying the $RTS_HOME/codegen/rtgen.mk file by adding the command-line parameters to the `RTGEN` macro. The macro defined in this file will be included by all generated makefiles and used to generate source and build files. For example, to add command-line parameters, include to the macro definition:

```
RTGEN = rtcppgen -crlf
```

This will pass the **-crlf** command to the code generator.

3 Define RTGEN in **CodeGenMakeInsert**.

Command-Line Arguments

The **rtcppgen** program accepts the following arguments:

-crlf

-forcewrite

-spacedeps bs | dq | fail | none

-version

There are other options for internal use only.

-crlf

The **-crlf** flag forces files to be written Windows style, with lines terminated with a carriage return and line feed. By default, files are written with UNIX style end-of lines conventions.

-forcewrite

The **-forcewrite** flag disables the code-generator's incremental file output and is useful for producing incremental load-builds. It is typically only used within the environment variable `ROSERT_RTGENOPTS`, when integrating a new set of changes on top of a previously-built load-build.

-spacedeps

The **-spacedeps** flag tells the code generator how to write code generation dependencies for file paths that contain spaces, such that the Code Generation Make Type can read it. This would typically be overridden by users of a generic Unix Make utility who have experimented with space-handling in their Make variant. For the **Compilation Make Type**, there is a corresponding option to the `rtcomp.pl` script (except that "**-spacedeps none**" is replaced by "**-nodeps**").

- **bs**: precede space with backslash (for **Gnu_make**).
- **dq**: surround filename with double-quotes (for **MS_nmake**).
- **fail**: cause a fatal error (for **Unix_make**).
- **none**: no escape sequence (intended for **ClearCase_omake** and **ClearCase_clearmake** whose **dep** files need not be Clearmake-readable).

-version

The **-version** flag prints the version identifier of the code-generator to **stdout**.

Command-Line Build Interface

Rose RealTime uses an external build engine for code generation, compilation, and linking. To mimic the toolset's build mode, you can run the build from the command line. This might be useful if the build host is different then the toolset host. Before generating and building an existing model, it is important that the model has been validated by the toolset. If a model is valid (that is, there are no unresolved references), then you can generate and build a component from the command line.

The main steps that must be performed from outside the toolset are:

1. Create the makefiles
2. Generate the source code
3. Build the generated source files

Refer to the Guide to Team Development for extensive syntax examples on how to build a model from outside the toolset.

Generating and Sharing External Library Interfaces

4

Contents

This chapter is organized as follows:

- *Overview* on page 57
- *Phase 1: Providing the Library Interface Specification* on page 59
- *Phase 2: Publishing the Interface* on page 66
- *Phase 3: Sharing and Using the External Library Interface* on page 68
- *Considerations and Known Issues* on page 70

This chapter explains the Rational Rose RealTime Library Interface Generation feature.

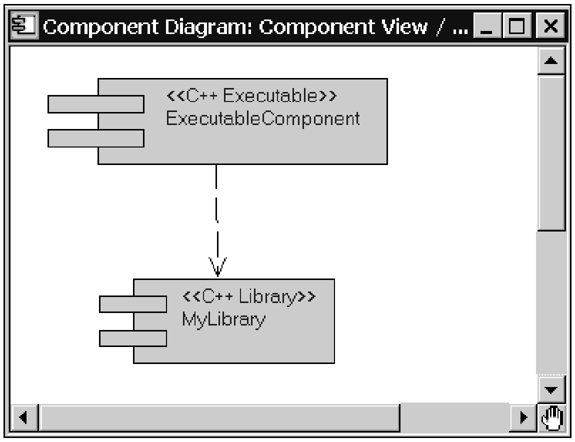
Overview

Before using this functionality, we recommend that you review the chapter "Storage of Model Data" in the *Guide to Team Development, Rational Rose RealTime* for information on control units and shared packages.

Purpose

The purpose of generating and sharing external library interfaces is to build components of a system in separate models while maintaining the interfaces required between them. Figure 7 shows a single model which has a **<C++ Executable>>** component with a dependency on a **<<C++ Library>>**. These components can be modeled and built in separate Rational Rose RealTime models. The ClientModel would build the **<<C++ Executable>>** component, and the SourceModel would build the **<<C++ Library>>** component. To do this, the dependency that the **<<C++ Executable>>** component has on modeling elements within the **<<C++ Library>>** component needs to be represented in the ClientModel. This is made possible by publishing an external library interface.

Figure 7 Component Diagram



The following terms are used to discuss the types of models involved in this process.

SourceModel

Contains a model that generates a library for use in a client model.

ClientModel

Contains a model that uses the interface generated by the source model.

InterfaceModel

The model which owns the interface modeling elements shared into a ClientModel.

Phases of Generating an External Library Interface

There are three phases involved in generating an external library interface.

Phase 1 involves creating the specification for the interface within the SourceModel.

Phase 2 publishes the external library interface.

Phase 3 involves setting up a **ClientModel** to use the interface published in Phase 2.

Note: The Generate External Library Add-In depends on the C++ TargetRTS Add-in. By default, they are enabled. You can ensure that you have both Add-ins enabled in your Add-in Manager. For further information, see the *Rational Rose RealTime Toolset Guide*.

Phase 1: Providing the Library Interface Specification

To provide the library interface specification, complete the following sections:

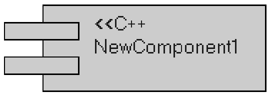
- *Creating a Library Component* on page 59
- *Setting the Target Configuration and References* on page 60
- *Setting the Visibility Level of External Library Interfaces* on page 60
- *Redefining Visibility Settings for Unit Interfaces* on page 64
- *Setting Inclusion Paths and Library Paths* on page 65
- *Specifying the Names of Shared Packages* on page 66

Creating a Library Component

If a candidate library component does not already exist, create the library component in the SourceModel.

To create a library component:

1. Open your existing SourceModel in Rational Rose RealTime.
2. In the Toolbar, select the **Browse Component Diagram** button.
3. For the **Component View** package, select **Component Diagram: Component View / Main** in the **Component diagrams** list, and click **OK**.
4. From the Toolbox, select the **Component** tool, then click in the diagram.

   A new component appears in your diagram. By default this component is a **<<C++ Executable>>** component.
5. Right-click on this executable component and select **Open Specification**.
6. Click the **General** tab.
7. In the **Type** box, select **C++ Library**.
8. Click **Apply**.

   The tabs on the Specification dialog change to reflect the change in the component type.

   Note: You must click apply to update the **Component Specification** dialog with the **C++ Library** tab.

Setting the Target Configuration and References

To set the target configuration and the references:

1 If the **Component Specification** dialog is not open, right-click on the library component and select **Open Specification**.

2 Click the **References** tab.

3 From the **Model View** tab in the browser, drag all of the modeling elements used to build the library to the **References** tab.

Note: Referencing a package includes all elements within that package.

4 Click the **C++ Compilation** tab.

For additional information, see C++ Compilation Properties in the *Rational Rose RealTime C++ Reference*.

5 Select a target configuration and click **OK**.

6 Click **Apply**.

Next, you will modify fields at the bottom of the **C++ Library** tab on the **Component Specification** dialog that contain information specific to the interface that will be generated by the library interface generator.

Setting the Visibility Level of External Library Interfaces

Before you generate the external library interface for a library, set the visibility level of elements within the library component in the SourceModel. The visibility level determines what, if any, representation **Logical View** elements from the SourceModel will have in the ClientModel. This specifies the **Logical View** interface to the **<<C++ External Library>>** from the perspective of the ClientModel. It also defines the contract of services (interface) that is realized by the **<<C++ Library>>** from the perspective of the SourceModel.

To set the visibility:

1 On the **Component Specification** dialog for the library component.

2 Click the **C++ Library** tab.

Note: You can either expand this dialog, or use the scrolls bars to move to the bottom of this tab.

3 Click the **Edit** button opposite **SetInterface**.

By default, **SetInterface** uses the value set in the **DefaultInterfaceVisibility** box, unless you previously specified units in the **Set External Library Interface** dialog. If most of your library is implementation details, and only a small portion of it needs to be represented in the ClientModel, we recommended that you change the **DefaultInterfaceVisibility** box to **Private**.

The **Set External Library Interface** diagram shows all the interface elements included in the reference section for the library, including those interface elements indirectly referenced by the package name.

The **Set External Library Interface** dialog also includes all the elements available in libraries that this library depends on. Elements from subcomponents should never be published as interface elements; set the visibility of these elements to **Private**. Because there is no indication in the graphical user interface of where the element is being listed from, it is important to note which elements should be published as interface elements before using this dialog.

You can set the default visibility level for all interface units by selecting a visibility level from the **DefaultInterfaceVisibility** box on the **C++ Library** tab in the **Component Specification** dialog.

When being used in ClientModel, there is no indication as to which elements were generated with **Limited Public** visibility and those with **Public** visibility. A problem can occur when a developer in the ClientModel attempts to use an inheritance relationship with an interface element generated using **Limited Public** visibility.To use the items in an external library, some library elements will need to be visible, and Rational Rose RealTime must be aware of them. Table 1 shows the levels of visibility.

Table 1 Visibility Level Descriptions

Level	Description
Public	Indicates that the unit (a capsule, class, or protocol referenced by the development library) is visible in any user model, and can be used as the parent in an inheritance relationship.
Limited Public	Indicates that the unit is visible in other user models, and can be used in any relationship other than an inheritance relationship. **Note:** In client models, there is no method to determine which interface elements were generated using Limited Public or Public visibility. Do not use an inheritance relationship with an interface element generated using Limited Public visibility.
Private	Indicates that the unit is not visible in any user model, and cannot be seen by a model as part of the external library interface.

Note: The external library interface consists of the **Public** and the **Limited Public** units. **Private** units are not part of the generated external library interface.

The generated external library interface does not include the internal details of **Limited Public** interface units that do not need to be included with the external library.

The following internal details are deleted from **Limited Public** interface units:

- State machines for capsules, protocols and classes (if one exists).
- Protected and private operations for capsules and classes.
- Public operations code.
- Protected and private attributes for capsules and classes.
- Protected ports for capsules.

4 Set the appropriate level of visibility for each unit in the library.

When setting visibility, there are rules regarding the visibility of related elements. For example, if a Capsule that you would like to publish as part of an interface with **Limited Public** visibility has a **Public** Port, then the Protocol on which that port is based must also be set to **Limited Public**.

Reading from the eighth row (after the heading) of Table 2 from right to left, the suggested minimum visibility is **Limited Public** for a Protocol that is involved in an Association relationship with a **Limited Public** Capsule as in the example above.

Note: If the event a Dependent Unit is defined in a subcomponent, the visibility rule should be used as outlined in Table 2 instead of setting the element to **Private**. Use Table 2 as a guide to set the levels of visibility for your unit.

Table 2 Summary of Rules for Unit Visibility

Unit	Visibility	Relationship to Dependent Unit	Dependent Unit	Suggested Minimum Visibility for Dependent Unit
Capsule	Public	Generalization	Capsule	Public
		Association	Protocol (public ports)	Public
		Association	Protocol (protected ports)	Public
		Capsule Role	Capsule	Public
		Association (public)	Class	Public
		Dependency (public)	Class	Public
	Limited Public	Generalization	Capsule	Limited Public
		Association	Protocol (public ports)	Limited Public
		Association	Protocol (protected ports)	Private
		Capsule Role	Capsule	Private
		Association (public)	Class	Private
		Dependency (public)	Class	Private
Class	Public	Generalization	Class	Public
		Association (public)	Class	Public
		Dependency (public)	Class	Public
	Limited Public	Generalization	Class	Limited Public
		Association (public)	Class	Private
		Dependency (public)	Class	Private

Table 2 Summary of Rules for Unit Visibility (continued)

Unit	Visibility	Relationship to Dependent Unit	Dependent Unit	Suggested Minimum Visibility for Dependent Unit
Protocol	Public	Generalization	Protocol	Public
		Data Type	Class	Limited Public
	Limited Public	Generalization	Protocol	Limited Public
		Data Type	Class	Limited Public

5 Click **OK** to implement your changes.

Note: If the visibility settings that you select do not conform to the rules outlined in Table 2, the **Set External Library Interface** dialog box appears. Clicking **OK** again will override the rule checking algorithm and accept the visibility values as you set them.

Redefining Visibility Settings for Unit Interfaces

If the visibility settings of your unit dependencies are not set according to the rules specified in Table 2 on page 63, the **Set External Library Interface** dialog box appears. This dialog box allows you to set the visibility level of your unit according to the unit visibility requirements, and provides suggestions that can help you correctly define the rules for the selected unit.

If you do not want to reset the visibility level of the interface unit, click **OK** to override the suggestions listed in the **Set External Library Interface** dialog box.

To reset the visibility of the unit:

1 In the **Set External Library Interface** dialog box, select the interface unit that you want to change.

2 Read the description that appears to the bottom-left of the **Set External Library Interface** dialog box.

3 If you agree with the suggested change to the visibility level of the unit, click the visibility level listed for the unit and select the new level.

4 Repeat steps 1 to 3 until all units are set according to the visibility rules.

5 Click **Recalculate**.

Setting Inclusion Paths and Library Paths

InclusionPaths and **Libraries** are fields used to generate the **<<C++ External Library>>** component in the shared **Component View** package. These fields are located at the bottom of the **C++ Library** tab for a component specification, and provide information used to locate the actual library and included header files that are necessary for the build process.

The information in **InclusionPaths** and **Libraries** populates the fields in the **<<C++ External Library>>** component. When shared into a client model, these elements are **read-only** and the buttons are grayed out; the information contained within the dialogs for these buttons is not available in client models.

The **InclusionPaths** property specifies the location of the definitions for the external library. Components which reference this external library will automatically include the definitions header file.

The **Libraries** property specifies the location and names of the libraries that this external component represents. The libraries listed in this field will be added to the link line for any executable component that references this external library. You have to specify the complete path and filename.

As part of the publishing phase, you are prompted for a location to copy the definitions header files, and a location to copy the generated library file.

To set the inclusion path and libraries properties for the interface:

1 In the **Component Specification** dialog box, click the **C++ Library** tab and scroll to the bottom of the tab.

2 Click the **Edit button** opposite **InclusionPaths**.

The **Inclusion Paths** dialog box appears.

3 Click **Insert Path**.

4 Type a path in the highlighted area.

Note: Enter the location of where you will be copying the header files generated by the build process for use by ClientModels. We recommend that you use **pathmap** symbols or environment variables for path names in this property. For additional information, see Environment variables and pathmap symbols. For Example:

$(MYINTERFACES)/ALibrary/include

5 Click **OK**.

6 Click the **Edit** button opposite **Libraries**.

   The **External Libraries** dialog box appears.

7 Click **Insert Library**.

8 Type the location and the name of the library in the highlighted area.

   Note: Enter the fully-qualified location of where you will be copying the library for use in ClientModels. For example:

   $(MYINTERFACES)/lib/$(LIB_PFX)ALibrary$(LIB_EXT).

9 Click **OK** to add the path for the library.

10 Click **OK** to close the **Component Specification** dialog.

Specifying the Names of Shared Packages

The interface of the generated library is packaged in two shared packages: one in the **Logical View**, and the other in the **Component View**. If you can specify a name for these packages in the **ShareLogcialViewPackageName** and the **ShareComponentViewPackageName** fields, the shared package will be created in the InterfaceModel with the new name. If you do not specify a name, the packages are created in the InterfaceModel with the default names, <*LibraryComponentName*>SharedLVPkg and <*LibraryComponentName*>SharedCVPkg.

To specify the name of shared packages:

1 In the **Component Specification** dialog box, type the name of the **ShareLogicalViewPackageName** and the **ShareComponentViewPackageName** in the specified text area.

2 Click **Apply** to implement your changes.

Phase 2: Publishing the Interface

The interface is published as a new model, the **InterfaceModel**.

In a **Logical View** package, the interface elements are generated representing the elements which are realized in the library component, based on the visibility set during Phase 1: Providing the Library Interface Specification. The hierarchy of enclosing packages remains the same under this new package as they were under the **Logical View** in the SourceModel.

In a **Component View** package, a **<<C++ External Library>>** component is created. The external library component references the elements in the interface.

To create an external library component:

These two new packages are set as controlled units.

1 Right-click the library from which you are generating the external library, or from the **Build** menu, and click **Generate External Library Interface**.

  Note: If you use the **Build** menu to generate the external library interface, the library from which you generate the external library must be **Set As Active**.

  You will receive a message requesting that you build the library before you generate the external interface.

2 Click **Yes** if you need to build the library component.

  Check the build results in the **Build Log** tab. If the first line does not read "Build Successful", click the **Errors** tab and correct the errors in your model before proceeding.

  If the build was successful the **Save External Library Model To** dialog appears.

3 Specify the location for the InterfaceModel. This location must be accessible to ClientModels. For example:

  ```
  w:\interfaces\aLibrary\InterfaceModel
  ```

4 Click **Save**.

  Rational Rose RealTime will now generate your external library interface. Depending on the size of your model, this can be a lengthy operation.

If the InterfaceModel was generated previously, you are notified that the contents of the existing InterfaceModel will be deleted. This will also remove any subdirectories associated with the InterfaceModel.

After the InterfaceModel is successfully created, you are prompted to select a destination for the header and binary files.

Selecting the Destination for Header and Binary Files

Only the minimum set of header files required by the external library model are copied. The visibility level of the interface element determines which files are copied.

1 Use the **Select Folder to Store Binary File** dialog box to specify the desired location to copy the library where it is accessible to ClientModels. For example:

   w:\interfaces\lib

2 Click **Open**.

   The **Select Folder to Store Header Files** dialog box appears.

3 Use the **Select Folder to Store Header File** dialog box to specify the desired location to copy header files where they are accessible to ClientModels. For example:

   w:\interfaces\ALibrary\include

4 Click **Open**.

Phase 3: Sharing and Using the External Library Interface

To provide the library interface specification, complete the following sections:

- *Sharing the Interface* on page 68
- *Using the Interface* on page 69

Sharing the Interface

To share the external library: See Known Issue (below) before proceeding.

1 In the **ClientModel**, right-click a component package, and click **Share External Library Interface**, or on the **Build** menu, click **Share External Library**.

   The **Share External Library** dialog box appears.

2 Select the name of the external library model, and click **Open**.

   Note: The **SharedLVPkg** and **SharedCVPkg** packages appear as part of the User Model classes. **Do not** re-share the same library more than once. Instead of re-sharing the library, re-sync the packages that are shared.

Known Issue

Packages shared into a model using **Share Library Interface** or **Share External Library Interface** do not correctly store the path to the shared packages in the properties for the element. This causes problems for code generation and for subsequent loads of the model in other Rational Rose RealTime sessions.

The current workaround is to:

1. Use the context menu item **File > Share External Package** on the **LogicalView** package.
2. Browse under the **LogicalView** directory of the InterfaceModel and select the generated **Logical View** package **SharedLVPkg**.
3. Use the context menu item **File > Share External Package** on the **Component View** package.
4. Browse under the **ComponentView** directory of the InterfaceModel and select the generated **Component View** package **SharedCVPkg**.

To re-sync shared packages:

1. Right-click the shared package.
2. On the **File** menu, click **Reload From File**.

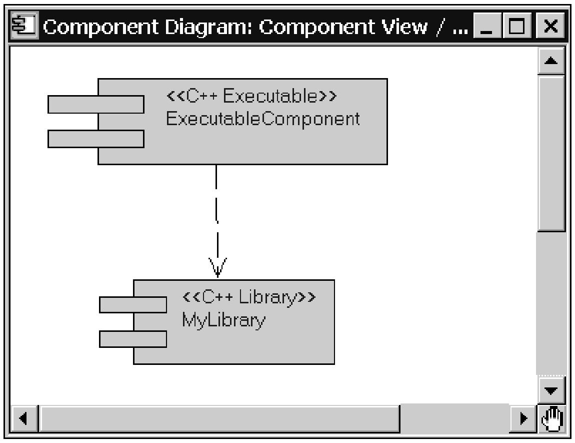
   Note: If you are re-syncing an already shared InterfaceModel, the **Logical View Shared Package** should be re-loaded before the **Component View Shared Package**.

Using the Interface

Logical view elements from the interface can be referenced by other elements in your ClientModel. You must put an explicit dependency from a component that uses these elements on the **<<C++ External Library>>** component shared into the **Component View**.

1. Drag the **<<C++ External Library>>** component onto a **Component** diagram.
2. Drag the component being built in the ClientModel that relies on the **<<C++ External Library>>** component onto the same diagram.
3. Click the dependency tool and draw a dependency from the component to the **<<C++ External Library>>** component.

For example:



Note: When building the component in the ClientModel, if no dependency is drawn on the **<<C++ External Library>>** component, the toolset prompts you to add elements from the **Logical View** shared package to the references section of the component. This is an indication that the dependency is missing. **Do not add** the elements to the reference list to the component. Abort the build process and add the dependency before trying again.

Considerations and Known Issues

- When sharing an external library interface in a client model, the path relative to the library interface model is stored in the Specification for the **Logical View** and **Component View** packages. Unless the path is also relative to the client model, neither the code generator or subsequent loading of the toolset will be able to find the control units.
- When generating multiple library interfaces from the same model, you may encounter an error message that indicates that there are multiple objects with the same unique ids.

  The elements generated in the InterfaceModel have the same **quids** (unique identifiers) as elements in the SourceModel. Two or more interfaces created from the same model with common elements such as packages cannot be shared into the same model.

- **Component names**

  Component names are used by default for the name of the binary and the name of the parent directory for the build process.

  To name the **<<C++ External Library>>** component the same as the library, the name of the InterfaceModel must be the same as the Library because the **<<C++ External Library>>** component - when generated in an InterfaceModel - is named after the InterfaceModel.

  Since the InterfaceModel is used to share packages, the model is broken into control units. The parent directory for control units is named after the model.

  In this scenario, if the InterfaceModel is stored in the same directory as the SourceModel, there will be a conflict between the directory used to store the build output from building the **<<C++ Library>>** component and the directory used to store the control units for the InterfaceModel.

- **Code generation from the ClientModel**

  The code generator needs to be able to find all subcomponents that the target component depends on when it is trying to build. The header files and the Library need to be available where the code generator expects them. This is defined in the **Libraries** and **InclusionPaths** properties for the **<<C++ External Library>>** component, information hidden to the end-user. To determine if these properties are set properly, it may be necessary to open the InterfaceModel.

Classes and Data Types 5

Contents

This chapter is organized as follows:

- *Sending Data in Messages* on page 73
- *C++ Data Type Examples* on page 77

Overview

In most models, capsules require the use of lower-level data types (or classes) to create and maintain internal data structures and variables, to send and receive data values in messages, and to interact with legacy code or third-party code libraries. With Rational Rose RealTime, you can use any C++ data classes or types within your model whether it is defined within the toolset or not as long as the type is visible to the compiler. Although the toolset can generate classes and type descriptors, you are responsible for ensuring that any classes that are created are well formed. For example, classes that do not leak memory must have appropriate constructors and destructors defined.

Note: The terminology for data type and class may cause some confusion. Throughout this book, we will use the term *data type* for the generic concept of a named definition that encompasses a notion of storage of values, and of operations that may be performed on those values.

Sending Data in Messages

In order to implement the behavior of a system, capsules send messages to either request a service or provide a service to other interconnected capsules. The messages that are sent between capsules contain:

- A signal name that identifies the message.
- An priority (relative importance of this message compared to other unprocessed messages on the same thread: default to General).
- Optional application data.

Messages do not have to be sent with application data. This is similar to operations that do not always require parameters. When operations require parameters, the developer must decide whether to pass the parameters by value or by reference. The same applies when sending application data in messages.

Protocols

The protocol definition is where you specify the type of data that is to be sent with a specific signal:

- To send data by value, specify the data type in the **Data Class** field of the signal.
- To send data by reference, clear the **Data Class** field of the signal.

Sending by Value

Sending data by value means that a deep copy of the data is sent with a signal. This option is less efficient than sending by reference but it simplifies concurrency issues.

Note: The fact that a data type is sent by deep or shallow copy depends on the constructor, copy constructor, and destructor defined on the data class.

To send data by value, the C++ Services Library must know how to copy, initialize, and destroy the objects that it sends. Type descriptors describe data types to the C++ Services Library to allow it to manipulate the objects that it sends. For further information, see *RTObject_class* on page 268.

Sending by Reference

Sending data by reference is primarily used for efficiency. Instead of copying a block of memory, a pointer to the memory is passed.

When sending pointers in messages:

- Do not send pointers across thread boundaries without considering concurrency access issues.
- Do not send pointers across process or processor boundaries unless you have shared memory. You must also consider concurrency issues.
- Do not send pointers to stack objects. The stack object gets deleted when the transition code segment completes, and most sends are asynchronous. When the receiving capsule instance de-references the pointer, the data it is pointing to has been deleted.

Memory Leaks

If you decide to send pointers in messages, you can easily introduce memory leaks into your model if you are not careful. If you plan on sending pointers in messages, carefully review the following dangers:

- When a message send operation fails, and you had expected the receiver to free the memory, this will cause a memory leak. The proper action is to verify all message sends and take appropriate action if the send fails.
- If a capsule is destroyed before it has processed all its buffered messages and the receiver was expected to free the memory, then memory will be leaked for any unprocessed message containing pointers. The most robust method for preventing these memory leaks is to create a smart pointer wrapper that manages the memory for the pointer being sent. Smart pointers can manage the pointer and automatically free any memory once the pointer is no longer referenced.
- You can also easily leak memory if you pass a pointer to a timing request operation, for example **Timing::informIn** or **Timing::informEvery**, and then cancel the timer before it fires.

If you do not address these dangers when sending pointers, your model may run for a while, but might crash unexpectedly. Most real-time systems are designed to run for long periods, which causes small memory leaks to accumulate over time and causes disasters that are hard to reproduce.

See *Sending and Receiving Data By Reference* on page 78 for an example of the send syntax.

Creating Data Types

When using data in Rose RealTime, you should provide well-formed data types so that the memory that is allocated is deleted, and copy constructors work as intended.

The sending, receiving, and integrating data model examples provides examples of how to integrate different kinds of data types within Rose RealTime.

You can create data types that are:

- Sendable by Value: A data type can safely be sent between capsules within the same process using copy semantics for objects.
- Observable: A data type can be safely output via the observability in the Rose RealTime (watches and message traces), and via the log service.
- Marshallable: A data type can be safely decoded (injected from the toolset).

Data class rule #1

Simple data types that do not contain pointers (any indirect attributes) are by default sendable, observable, and marshallable. External data types are not necessarily sendable, observable, or marshallable by default. For details, see *Integrating an External Class* on page 88.

Data class rule #2

Classes that contain pointers can be made sendable by value, observable, and marshallable. Add details to your class to make it well formed by creating or modifying the following functions:

C++ data class methods: ▪ constructor ▪ copy constructor ▪ destructor	The toolset can automatically generate these operations, however, they will not safely handle classes that contain pointers. If a class contains pointers you must provide your own constructor, copy constructor, and destructor methods.
Type descriptor functions, defined on each class in the **C++ TargetRTS** tab.	The functions define how a data class is initialized, copied, destroyed, decoded, and encoded. By default, the init, copy, and destroy functions call the class' constructor, copy constructor, and destructor. Generally, you only need to modify the data class' methods for these functions.
NumElementsFunctions, defined on an attribute in the **C++ TargetRTS** tab.	At runtime, this function determines the size of an indirect field (the number of things a pointer references). If unspecified, this will be set to 1. Used by the encode/decode functions.

Marshallable Data Classes

In addition to making data classes sendable by value, they can be made observable. *Marshallable* means that the object can be encoded and decoded into a string of bytes. This functionality allows the toolset to display the contents of objects at runtime, and can also be useful for interprocess communication. Objects can be inspected at runtime from within the Rose RealTime execution environment.

When you are debugging a running model and request that an attribute or data within a message is shown in the toolset (similar to the watch facility available in most source debuggers), the toolset sends a request to the running model. The Services Library calls the `encode` function (defined within the type descriptor) on the object instance. The result of the `encode` function is passed to the toolset, and is shown in either a watch window, or a message trace.

Basic Structures

Simple data classes are by default encoded using an ASCII encoder meaning that they are observable. For data classes containing attributes of types that are not known by the toolset, these functions must be written by the user and cannot be automatically generated by the toolset.

This flexibility allows for almost every kind of class or data type to be used within Rose RealTime.

C++ Data Type Examples

The following examples demonstrate the different methods of creating and using data types within Rose RealTime:

- *Syntax Examples of Sending Data Classes Between Capsule Instances* on page 78.
- *Class Modeling Examples* on page 79.
- *Creating and Using Common C++ Constructs* on page 81.
- *Class Creation Examples* on page 86.

Note: Before reviewing the examples, you should be familiar with the basic information described in the section, *Sending Data in Messages* on page 73

Syntax Examples of Sending Data Classes Between Capsule Instances

Sending and Receiving Data By Value

When sending data by value, you send a copy of the data instead of a pointer to the data. This is the preferred method of sending data between capsules, and simplifies concurrency issues.

The examples below demonstrate how to send and receive data by value. We assume that the detail code is part of transitions on both the sender and receiver capsules.

When not using the `rtdata` parameter, the onus is on the programmer to supply the correct type cast. With `rtdata`, the generated code handles the type cast.

Sender

```
SomeClass data("hello");

// Given a port called 'port' based on a protocol with a
// signal 'start' with data class 'SomeClass'.
port.start(data).send();
```

Receiver

```
// The following statements are equivalent
const SomeClass & data2 = *rtdata;
const SomeClass & data3 = *(const SomeClass *)getMsg()->getData();

// Accessing the pointer directly (a reference does not involve a copy)
const SomeClass * data3 = rtdata;
```

Sending and Receiving Data By Reference

For performance reasons, sending data by reference can be an effective way of sending data. However, you need to be aware of all the issues pertaining to sending data by reference. For more information, see *Sending by Reference* on page 74.

The following examples demonstrate how to send and receive data by reference. We assume that the detail code is part of transitions on both the sender and receiver capsules. You should never pass a pointer to an object allocated on the stack (local variable). You will also have to coordinate who is responsible for freeing the allocated memory. (In the example below, the receiver does).

Sender

```
SomeClass * pdata = new SomeClass("hello");

// Given a port called 'port' based on a protocol with a
// signal 'stop' with data class left empty.
port.stop(pdata).send();
```

Receiver

```
SomeClass * data2 = (SomeClass *) getMsg()->getData();
//use data
delete data2;
```

Class Modeling Examples

Creating a Class Data Member From the Class Diagram

Given an association between two classes (Figure 8) or between a capsule and a class (Figure 9), an attribute is created in the generated source code for the classes participating in the relationship.

Figure 8 Association Between Two Capsules

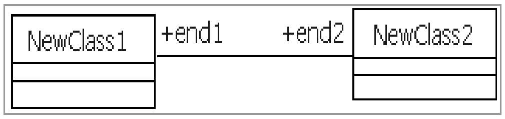
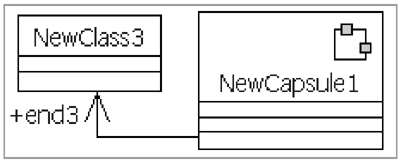


Figure 9 Association Between a Capsule and a Class



The aggregation kind determines if the attribute is contained by reference (aggregate) or by value (composite).

The above relationships result in the creation of data members named: `end2` in `NewClass1`, `end1` in `NewClass2`, and `end3` in `NewCapsule1`. The properties for the end (association end) control how the code will be generated for the data member.

Note: The end affects the class at the other end of the association.

Assuming that `end1` and `end2` are contained by reference, a simplified version of the code that would be generated is:

```
class NewClass1
{
public:
   //{{{RME associationEnd 'end2'
   NewClass2 * end2;
   //}}}RME
};

class NewClass2
{
public:
   //{{{RME associationEnd 'end1'
   NewClass1 * end1;
   //}}}RME
};
```

You can specify the aggregation kind (that is, aggregate or composite), visibility, and other attribute features that control how attributes are generated to source code. These features are found in the **Association Specification** dialog box.

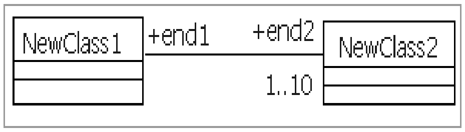
A data member is not generated if:

- The association end name is not specified.
- The end is not navigable.
- Both ends are aggregated by composition.
- The **Derived** option is selected.

Specifying Arrays Using Association Multiplicity

The association end multiplicity specifies the number of instances of this end that will appear in the related class. The data member that is created is an array with its size being the largest possible value in the specified multiplicity range. If the multiplicity is unspecified (such as, `1..*`) the association is forced to be an aggregate (by reference).

Figure 10 An Array Data Member



Assume aggregation for `end1` is aggregate and `end2` is composite. The following code will be generated for the association:

```
class NewClass1
{
public:
NewClass2 end2[10];
};

class NewClass2
{
public:
NewClass1 * end1;
};
```

Creating and Using Common C++ Constructs

Creating Array and Pointer Attributes

Attributes can be created either as arrays or as pointers.

Tasks

Create an attribute and set the type to any valid C++ type. If it is an array, specify the array identifier and size with the type.

Creating a Constant

You can create C++ constants that are either scoped globally, or scoped to a class.

Example

```
const int num_retries(4);

class Constants
{
public:
const int max_connnections(10);
};
```

The above source code fragments show both a global and instance scoped constants.

Note: The names for these constants are examples only. You can create a constant with different names.

Tasks

To create a global constant:

1. Create a class and name it appropriately.

   Note: You can also create a constant on any class that is already defined. It is recommended that you create a constant with the class where it is used.

2. Create an attribute in this new class. This will be the constant.
3. In the **New Attributes C++ Properties** tab, change the **AttributeKind** field to **Global**.
4. In the **Attributes Detail** tab, set the **Type** and **Initial Value** for the constant. Set the **Changeability** option to **Frozen**.
5. Add a dependency between the class where the constant is defined, and the capsules or classes that use the constant. If the constant is **Global**, ensure that the **Dependency C++ Properties** are: **KindInHeader** = `inclusion`, and **KindInImplementation** = `none`.

Note: To also create a class scoped constant, follow the above steps with the exception of step 3.

Usage

You can use constants to specify the cardinality of replicated capsule roles, ports, and bindings by adding the class scoped name (even if the constant is global) of the constant to the **Cardinality** field in the **Capsule Role Specification** dialog box.

Note: Constant values have to be specified using the class name of the class in which they have been created so that Rose RealTime can resolve and verify cardinalities before generating the source code. In the generated source code the actual value of the constant is used and not the expression `class::constant`.

Apart from specifying cardinalities, constants can be used in any C++ program.

Creating a Typedef

Example

```
typedef unsigned int u_int;
```

The above source code fragment shows an example C++ `typedef`. The name of the `typedef` and the type used are examples only. You can create a `typedef` of any name and type.

Tasks

1 Create a class with the name of the `typedef`.

2 In the **Class C++ Properties** tab, change the **ClassKind** property to **typedef**, and add the desired type to the **ImplementationType** field.

Usage

You can create attributes of this type by setting the **Type** of the attribute to this new **typedef** (the `typedef` appears in the type drop-down list for **Attributes**).

Note: Add a dependency between the `typedef` class element and the capsules or classes that use the type as attribute types or in detail level code.

Creating an Enumeration

Example

```
enum e { a = 1, b };
```

Tasks

1 Create a class named **e**.

2 In the **General** tab of the **Class Specification** dialog box, set the **Stereotype** of the class to **enumeration**.

3 Create an attribute named **a** in the class.

4 In the **Detail Properties** sheet of this new attribute, change the **Initial Value** field to **1**.

5 Create an attribute named **b** in the class.

Creating a #define

Example

```
#define MAX_CONNECTIONS 24
```

The above source code fragment shows an example of a C++ macro, or #define statement. The syntax of the macro definition is:

`#define` *<macro name> <expression>*

Tasks

1 Create a class.

Note: You can also create a constant on any class that is already defined. It is recommended that you create a constant with the class where it is used.

2 Add an attribute to the class.

The name of this attribute will become the macro name.

3 Set the initial value of the attribute.

This will be generated as the macro expression.

4 In the **C++** tab of the **Attribute Specification** dialog box, set the **AttributeKind** field to **constant**.

5 Add a dependency between the class where the constant is defined and the capsules or classes that use the constant. If the constant is **Global**, ensure that the **Dependency C++** properties are: **KindInHeader** = `inclusion`, and **KindInImplementation** = `none`.

Use the above steps to define constants and not complex macros. This method does not allow macros to have names with brackets '(' or ')', or to have complex expressions. To define more complex macros, add them to the **Class C++** tab **HeaderPreface** property.

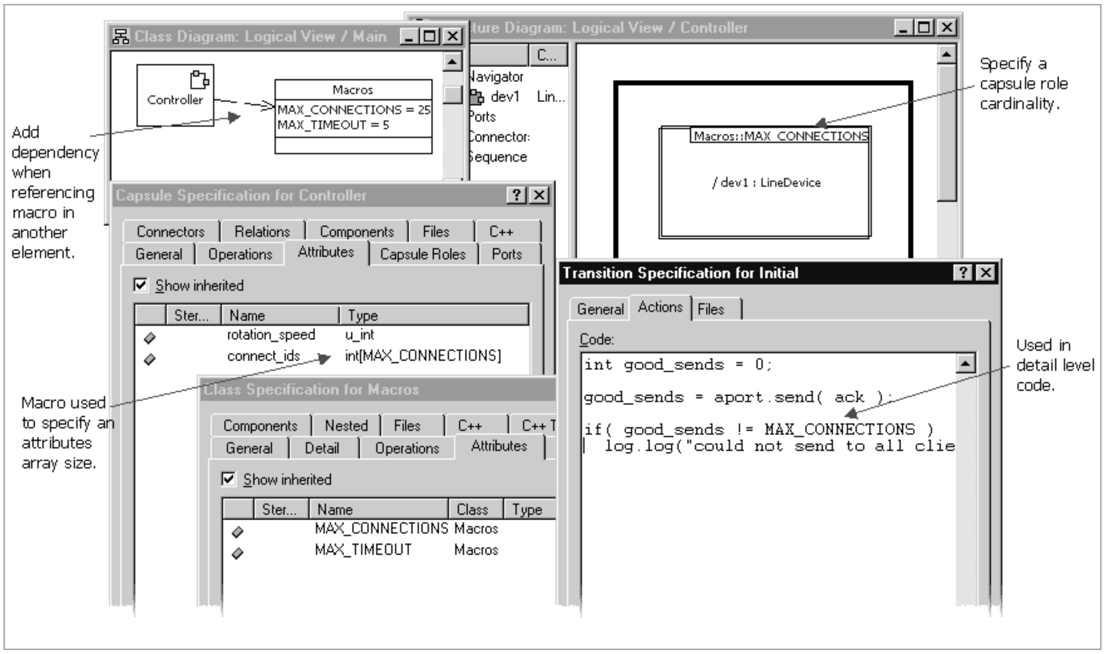
Usage

You can use macros to specify the cardinality of replicated capsule roles, ports, and bindings by adding the class scoped name of the macro to the **Cardinality** field. Macros have to be specified using the class name of the class in which they have been created so that Rose RealTime can resolve and verify cardinalities before generating the source code. In the generated source code, the actual value of the macro is used and not the name.

Note: You can specify any valid C++ expression in the **Initial Value** field for the macro. However, if the macro is used to specify a cardinality, the initial value must be a literal integer (such as, `1,50,100`). If the cardinality cannot be understood by the toolset, a warning will be generated when the model is compiled.

If the macros are to be used in detail level code, attribute array sizes, or other common C++ usages, ensure that there is a dependency added between the class containing the macros and the elements that reference the macros.

Figure 11 Macro Usages


Class Diagram: Logical View / Main
Controller
Macros
MAX_CONNECTIONS = 25
MAX_TIMEOUT = 5
Add dependency when referencing macro in another element.
ture Diagram: Logical View / Controller
Navigator
dev1
Ports
Connector
Sequence
Macros::MAX_CONNECTIONS
/ dev1 : LineDevice
Specify a capsule role cardinality.
Capsule Specification for Controller
Connectors
Relations
Components
Files
C++
General
Operations
Attributes
Capsule Roles
Ports
Show inherited
Macro used to specify an attributes array size.
Class Specification for Macros
Components
Nested
Files
C++
General
Detail
Operations
Attributes
Show inherited
Transition Specification for Initial
General
Actions
Files
Code:
Used in detail level code.

Ster...	Name	Type
	rotation_speed	u_int
	connect_ids	int[MAX_CONNECTIONS]

Ster...	Name	Class	Type
	MAX_CONNECTIONS	Macros	
	MAX_TIMEOUT	Macros	

```
int good_sends = 0;

good_sends = aport.send( ack );

if( good_sends != MAX_CONNECTIONS )
  log.log("could not send to all clie
```

Creating a Struct

You can create a C++ `struct` instead of a class.

Example

```
struct ConnectionParameters
{
int port;
unsigned long address;
short id;
};
```

Tasks

1. Create a class.
2. In the **Class C++ Properties** tab, change the **ClassKind** property to **struct**.
3. Fill in the attributes.

Class Creation Examples

Creating and Using Classes With No Pointer Attributes

These classes are:

- Sendable by value.
- Marshallable (can be observed and injected).

Classes without pointers have the above properties if all of their attributes are of types that do not have pointers, or are well-formed data classes.

Usage

The `ConnectParams` and `Nodes` classes are composed of predefined types. The C++ Services Library understands how to copy, initialize, destroy, encode, and decode because of generated type descriptors. The type descriptor generated by the toolset will be called `RTType_`<*class name*> and can be referenced directly in detail level code where an `RTObject_class` is required by a Services Library operation.

```
// Here the class is sent by value to another capsule instance
// Given a port called ’port’ based on a protocol with a
// signal 'connect' with data class 'ConnectParams'.
ConnectParams conn_p;
port.connect(conn_p).send();

// The encode function is called when the log service is used
log.log(&conn_p, &RTType_ConnectParams);
```

Creating and Using Classes With Attributes That Are Pointers

If you provide a copy constructor, destructor, and **NumElementsFunctionBody** (**Attribute**, **C++ TargetRTS**), you can make these classes:

- Sendable by value.
- Marshallable (can be observed and injected).

If you only provide the above C++ operations, and not the **NumElementsFunctionBody** (**Attribute**, **C++ TargetRTS**), the class will be Sendable by value (deep).

Note: Observing and injecting will cause memory leaks.

If you do not provide any of the operations, the class should never be sent by value because it will cause incorrect behavior.

Note: If a class has attributes that are pointers, ensure that the memory is managed properly by the class. Rose RealTime will not create a destructor that knows how to delete allocated memory. You will have to write your own destructor/constructor.

When attributes are pointers, extra steps are required. You will have to help the Services Library.

Example

The integrating data C++ model example contains an example of a class that contains a pointer. This class has well-defined default, copy, and destructor operations, as well as, the **NumElementsFunctionBody** (**Attribute**, **C++ TargetRTS**) property defined for the pointer attribute.

Integrating an External Class

You may have classes that are not defined in the toolset, either in third-party libraries or in code that will be reused for a new project. These externally defined classes can be integrated within Rose RealTime, and can be used for class modeling. They can be available in the drop-down type lists, or used within detail level code.

Any class or type defined outside the toolset can be used in your model. Depending on how the class or type is needed in your model, the class or type can be integrated within Rose RealTime in several ways.

Note: All integration examples are contained in the C++ model example, *Integrating Data*.

Integration Questions

Before integrating classes into Rose RealTime, consider how the class or data type will be used within the model.

1. Will objects of this type only be used to store information within a single capsule instance, or will they only be sent by reference and will never be observed, injected, or sent between processes?
2. Will objects of this type need to be sent by value between capsule instances?
3. Will objects of this type need to be observed during debugging, or encoded/decoded because they are injected or sent to other processes?

Integration for Case #1

In the first case, the only step required for using this class in your model is to make the external class definitions visible to the compiler by adding the `include` files to the **HeaderPreface** field in the **Class Properties** or to the **Component Compiler Inclusions** page.

After the definition is visible to the compiler, you can use the class or type within any detail level code.

Integration for Case #2 and Case #3

If you answered `Yes` to questions 2 and 3, a type descriptor will have to be created for the external types in order to describe the types to the Services Library.

There are two possibilities for handling an externally defined class or data type:

- You create a class within Rose RealTime with the same attributes as the external class, and then Rose RealTime can generate the type descriptor.

OR

- You add the code for describing how to copy, initialize, destroy, encode, and decode an instance of this type.

Integration Options

An external class can be made sendable by value without being observable, and vice versa. You have two integration options:

1. Describing an external type to Rose RealTime.
2. Providing own marshalling functions.

Option 1: Describing an External Type to Rose RealTime

If the class is described to Rose RealTime, the class can be made marshallable.

If your external class has a well-defined default constructor, copy constructor, and destructor, then the class can be sendable by value, and the default type descriptor will use the operations already defined on the class.

Example

The following class is defined in a header file outside of the toolset.

```
// This is an example definition of a class in a user-defined
external library
class Ext_Simple
{
public:
   int a;
   char b[80];
   float c[8];
};
```

Tasks

A class is sendable by value and observable if all attributes are also sendable by value and observable. In the above example, `Ext_simple` attributes are all types that are sendable by value and observable. The toolset can generate a complete type descriptor for this class. After the class is integrated within Rose RealTime, it can be used to create other more complex classes.

1 Create a class with the same name as the external class.

2 In the **Class C++** tab, clear the **GenerateClass** option.

   Note: You do not want to generate another class because the class is already defined outside the toolset. You are only describing the type to Rose RealTime.

3 In the **Class C++** tab, make the header file containing the actual class definition visible to this class by adding an `#include` statement to include the definition of the external class or type to the **HeaderPreface** property.

4 In the **Class C++ TargetRTS** tab, set the **GenerateDescriptor** property to **True**.

Note: Step 5 allows the C++ code generator to create marshalling functions for the external class, and is only required to encode/decode the class.

5 Add all the attributes that are defined in the external class to the class you have just created in Rose RealTime. The attributes must have the same names as in the external class but do not have to be declared in the same order.

If the external class contains pointers, follow the steps for creating attributes as arrays and pointers to correctly define the attribute, and to ensure that the external class has a well-formed (that is, no memory leaks) constructor and destructor.

Rose RealTime cannot describe non-public fields in classes that are externally defined. To integrate classes with non-public fields, you have to integrate the class using the method described in the following integration Option 2 example.

Option 2: Providing Own Marshalling Functions

A data type can be integrated for marshalling with Rose RealTime if it already contains operations to encode and decode to and from a string of bytes. This also applies when you want to describe an external class which has non-public fields (for encode/decode).

Note: Use this option instead of redefining all the attributes defined in an external class to allow an external data type to be marshalled (as described in the Option 1 above).

To integrate classes in this manner, you must understand the usage of the two functions defined in the **Class C++ TargetRTS** tab:

- **`DecodeFunctionBody`** (**Class, C++ TargetRTS**).
- **`EncodeFunctionBody`** (**Class, C++ TargetRTS**).

When writing the type descriptor functions, you have access to a pointer to an instance of the class (target), and in some cases, both a target and a source object instances (the source can not be modified).

Note: Ensure that the external class has well-defined default constructor, copy constructor, and destructor functions.

Tasks

1. Create a class with the same name as the external class.
2. In the **Class C++** tab, clear the **GenerateClass** property.
3. In the same tab, add **#include <An_External.h>** to the **HeaderPreface** property to make the header file (containing the actual class definition) visible to this class.
4. In the **Class C++ TargetRTS** tab, set the **GenerateDescriptor** property to **True**.
5. In the **Class C++ TargetRTS** tab, edit the **EncodeFunctionBody** (**Class, C++ TargetRTS**) property. Add code to encode the data class.
6. In the **Class C++ TargetRTS** tab, edit the **DecodeFunctionBody** (**Class, C++ TargetRTS**) property. Add code to decode the data class.

Note: Because the **GenerateClass** property was cleared`,` only a type descriptor will be generated for this new type. The class definition in the external header file should be visible to the compiler.

C++ Services Library 6

Contents

This chapter is organized as follows:

- *C++ Services Library Framework* on page 93
- *Log Service* on page 106
- *Communication Services* on page 106
- *Timing Service* on page 112
- *Frame Service* on page 114
- *Exception Service* on page 117
- *RTController Error Codes* on page 117
- *External Port Service* on page 122

C++ Services Library Framework

Together, the classes and data types defined in the C++ Services Library provide an application framework - the framework in which your Rational Rose RealTime application runs.

In general, the framework defines the skeleton of a real-time application: **messaging**, **timing**, **dynamic structure**, **concurrency**, **event based processing**, **platform independence**. You provide the classes, capsules, and protocols which are specific to your system.

The Big Advantage

Rational Rose RealTime lets you develope using state diagrams and structure diagrams which are automatically converted to C++ and placed in a framework that provides critical real-time system services.

Before you start developing, the key to using the services provided by the framework, is to understand how your application will integrate into the C++ Services Library skeleton. The framework provides:

- Communication services are the basic mechanism for using message-based communication via ports.
- Timing service provides general purpose timing facilities.
- Frame service is used to gain control over the dynamic structure in a model.
- Log service is a general purpose logging service.
- Exception service provides the ability to define custom policies to recover from exceptions.

Services are explained by introducing the general concepts related to the service followed by the classes that are used to implement that service. You should become familiar with the C++ syntax and notational conventions used in these sections as well as the *Services Library Class Reference* on page 233.

Message Processing

Events and Messages

An event is a message arriving on a capsule's port. Message-based communication is the basic mechanism for communication between capsules. Both synchronous and asynchronous communication are supported allowing a variety of different interaction semantics to be represented. Messages are also used by the Services Library to communicate with the capsules in the model.

The pre-defined capsule instance variable msg contains a pointer to the current message just received by the behavior. It is defined at the highest scope in the behavior. A message has three attributes:

- A signal that succinctly conveys the application-specific “meaning” of the message.
- A priority that indicates the “urgency” of the message. The priority of a message is determined by the sender.
- An optional data attribute, which contains additional information. This attribute can consist of an arbitrarily complex composite data object.

Processing Overview

The Services Library does not preempt capsule processing. The heart of the Services Library is a controller object that dispatches messages to capsules. Its basic mode of operation is to take the next message from the outstanding message queue and deliver it to the destination capsule for processing. When it delivers the message, it invokes the destination capsule's state machine to process the message.

Control is not returned to the Services Library until the capsule's transition has completed processing the message. Each capsule processes only one message at a time. It processes the current message to the completion of the transition chain (for example, guard, exit, transition, Choice Point, exit, and entry) and then returns control to the Services Library and waits for the next message. This is referred to as run-to-completion semantics. Typically, transition code segments are short, and result in rapid handling of messages.

Single and Multi-Threaded Message Processing

The Services Library runs in a loop executed by a system controller object. This loop waits for messages and delivers them, one at a time, to capsules for processing. Each physical thread in a Rose RealTime model has its own controller object and its own set of message queues. Messages that cross threads are placed in a special queue and picked up by the receiving thread in its processing.

The model is first initialized by queueing a special system-level message (the initialization message) for the top-level capsule. This causes initialization messages to be queued for all fixed capsules contained inside the top-level capsule. This continues recursively for all contained fixed capsules, so that all the fixed capsules in the model (those that aren't contained in optional capsules) are initialized.

After the initialization message is queued, the controller object enters its main processing loop (the **mainLoop** function). In **mainLoop**, it takes the next highest priority message from the message queues and delivers it to the receiver capsule and invokes that capsule's behavior to process the message. During start-up, the highest priority message on the queue of the main thread will be the initialization message. When a capsule processes the initialization message, the capsule's initial transition segment is executed.

When the capsule has completed processing a message, it returns control to the controller. The controller continues this loop until there are no more messages to be processed. At that point, it waits for a message from a timer or another physical thread in the model.

Introduction to Threads

A capsule can be thought of as having its own logical thread of control, and operating independently of other capsules, as if each capsule had its own dedicated processor. These independent capsules synchronize to perform higher-level scenarios through message-passing. One capsule sends a message to another capsule allowing the other capsule to update its state based on this outside stimulus. In practice, most Rose RealTime models run on a machine with a single processor, or possibly in a distributed environment, with a few processors. Capsules must share the single processor in some manner.

Types of Concurrency

The underlying operating system provides preemption to allow concurrent programs to share the processor so that each program is guaranteed to get some processing time depending on the prioritization of the programs, and any program that blocks does not stop processing of other programs. Many operating systems support one or both of the following forms of concurrency:

1. A heavy-weight unit of concurrency (usually referred to as a process), which has its own memory space, is completely separate from other processes (for integrity), and which communicates with other processes through special mechanisms (shared memory, sockets, signals, and so forth). Processes usually have a significant amount of protection such that if one process crashes it does not affect any other processes.
2. A light-weight unit of concurrency, referred to as a thread (or task on most RTOSs), shares a common memory space with other threads, and is not as robust (can be corrupted by other threads). Threads do not have as much protection as processes. Depending on the type of failure, an error in one thread may affect other threads.

Mapping Capsules to Threads

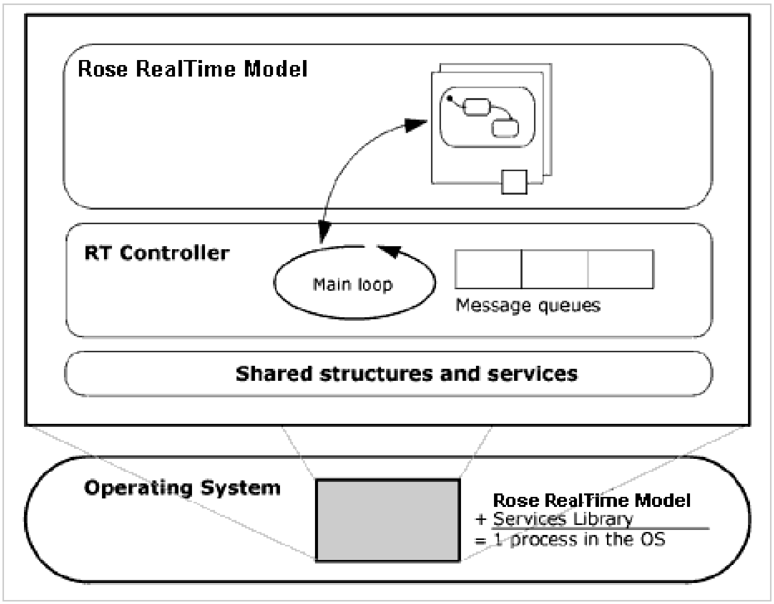
Rational Rose RealTime allows designers to make use of the underlying multi-tasking operating system so that the processing of a capsule on one thread does not block the processing of capsules on other threads. Designers can specify the physical operating system threads onto which the capsules will be mapped at run-time. In a system with only one thread, there are situations where a single capsule transition can block other capsules from running, such as if the capsule invokes a blocking system call. By placing some capsules in different threads, the designer can avoid the problems that arise from these situations, and make better use of the underlying processor. Not every capsule should run on a separate thread. For most capsules, it is sufficient to leave them in one thread and allow the Services Library controller to invoke their behavior as messages arrive.

Capsules with transitions that may block, or that have excessively long processing times, should be placed on separate threads. Deciding which capsules need to execute in different threads is a matter for design consideration.

Single-Threaded Services Library

The use of threads is not supported for certain targets, and may not be desirable for some applications. There is a single-threaded version of the Services Library, which is used for these situations. In the single-threaded model there is a single controller object that is responsible for queueing and delivering messages among capsules. The main processing loop runs inside this object. The single-threaded Services Library has the basic structure shown in Figure 12.

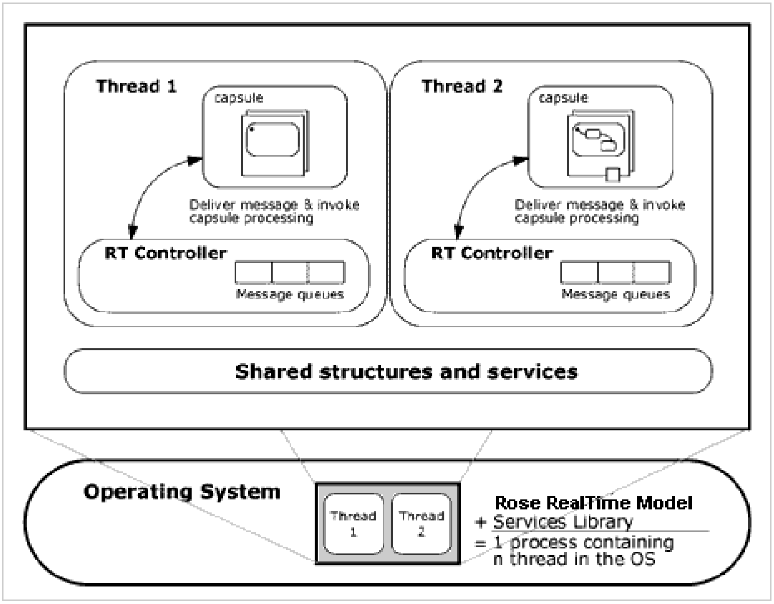
Figure 12 Single-Threaded Services Library



Multi-Threaded Services Library

Capsules can belong to different logical threads. Logical threads are mapped to a set of concurrent physical threads defined by the developer. No other capsules in a thread can execute until the currently executing capsule returns control to the main loop of that thread (except for the case of invoke). However, other capsules on other physical threads may be executing simultaneously (at least, from the designer's perspective). The operating system is responsible for switching control among active physical threads. The operating system may preempt one physical thread in the middle of execution to switch to another physical thread. Each thread can be assigned a separate priority, so that the designer has some control over the scheduling. In the multi-threaded model there is a separate controller object for each physical thread. This controller object contains the basic message delivery and processing loop. The basic structure of the multi-threaded Services Library is shown in Figure 13.

Figure 13 Multi-Threaded Services Library



Thread 1
capsule
Deliver message & invoke capsule processing
RT Controller
Message queues
Thread 2
capsule
Deliver message & invoke capsule processing
RT Controller
Message queues
Shared structures and services
Operating System
Thread 1
Thread 2
Rose RealTime Model
+ Services Library
= 1 process containing n thread in the OS

Naming Considerations

The C++ Services Library contains class names and operation names that may not exactly line up with the terminology used in the rest of the product. This is a holdover from previous versions of the Services Library, which was based on terminology used in the Real-time Object-Oriented Modeling (ROOM) method. Briefly, the changes in terminology are:

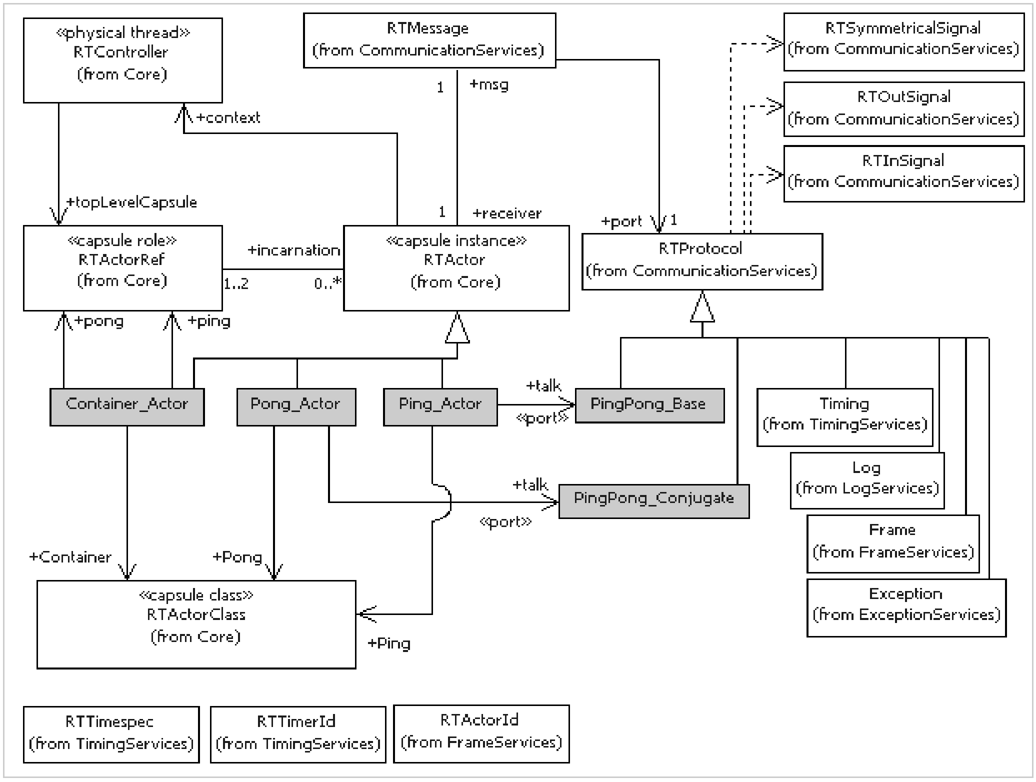
- actor = capsule instance
- actor reference = capsule role
- actor class = capsule
- SAP = unwired port for accessing a service
- SPP = unwired port for providing a service

C++ Services Library Framework

The capsules, capsule roles, protocols, ports and classes in a Rational Rose RealTime model will be generated to C++ code and integrated into the C++ Services Library framework. The class diagram below shows how a set of generated model elements integrate within the framework.

The white boxes are predefined classes in the C++ Services Library and the grey boxes are classes generated from a *Framework Sample Model* on page 105.

Figure 14 C++ Services Library Framework


«physical thread» RTController (from Core)
RTMessage (from CommunicationServices)
RTSymmetricalSignal (from CommunicationServices)
RTOutSignal (from CommunicationServices)
RTInSignal (from CommunicationServices)
+context
1
+msg
+topLevelCapsule
1
+receiver
+port
1
«capsule role» RTActorRef (from Core)
+incarnation
1..2
0..*
«capsule instance» RTActor (from Core)
RTProtocol (from CommunicationServices)
+pong
+ping
Container_Actor
Pong_Actor
Ping_Actor
+talk
«port»
PingPong_Base
Timing (from TimingServices)
Log (from LogServices)
+talk
PingPong_Conjugate
«port»
Frame (from FrameServices)
+Container
+Pong
«capsule class» RTActorClass (from Core)
+Ping
Exception (from ExceptionServices)
RTTimespec (from TimingServices)
RTTimerId (from TimingServices)
RTActorId (from FrameServices)

From this simplified class diagram, observe the following:

- The high level view of the C++ Services Library classes and their relationships.
- How your application level modeling elements integrate into this framework (grey boxes).
- The relationships between your modeling elements and the framework.

Because most modeling elements will become subclasses of framework base classes, elements will have access to operations and attributes that are defined in the base classes. Here are the main relationships that you should understand:

- Capsules become subclasses of RTActor
- Special Overrideable capsule class operations
- Capsule class information is stored in instances of RTActorClass
- Capsule roles are attributes of type RTActorRef
- Protocols become two classes - Base and Conjugate
- Ports are Protocol type attributes in RTActor subclasses

- Signals become operations in protocol classes
- Capsule roles are place holders for zero or more capsule instances
- Capsule instances have access to a RTMessage object
- Capsule instances have access to their controller

Capsules Become Subclasses of RTActor

All capsules are generated as subclasses of RTActor. This common base class contains state machine processing and messaging behavior that is used by all capsules.

Capsule state machines, operations, and attributes are included in the generated RTActor subclass. Since there can possibly be many instances of the same capsule in an application, capsule class information is kept separate from the RTActor instances, in a RTActorClass metaclass.

You can access public operations of RTActor within a capsule's behavior. For example it is common to have the following C++ code in a capsule state transition where the operations RTActor::getError and RTActor::context can be used because they are defined on the RTActor class:

```
switch( getError() )
{
   case RTController::noConnect:
   log.log("Unable to send message");
      break;

   default:
      log.show( "Unexpected error sending to peer: ");
      log.show( context()->strerror() );
      log.cr();
      break;
}
```

Special Overrideable Capsule Class Operations

There are two special operations that are defined as virtual functions on the C++ Services Library root class, RTActor, of all capsules. These functions can be used to customize the capsule's response to certain conditions.

- RTActor::unexpectedMessage
- RTActor::logMsg

Note: See the RTActor class reference for details on how to use these operations.

Capsule Class Information is Stored in Instances of RTActorClass

Characteristics common to all capsules, for example a name and external interfaces, are kept in a RTActorClass structure. All operations in the C++ Services Library which require you to specify a capsule class will take a parameter of type RTActorClass. There is only one instance of a RTActorClass for each capsule, whereas there are usually many RTActor capsule instances. The obvious advantage is that all capsule instances of the same capsule can share the capsule information stored in the RTActorClass.

The RTActorClass structures are named exactly as the capsules in your model. So if you have a capsule called **Device** in your model you can directly refer to this class in your model as **Device**. For example a common usage of RTActorClass is in the Frame::incarnate method where you can specify which type of capsule to incarnate into an optional capsule role. You specify the type by using the name of the capsule directly in the operation call. The first parameter specifies the capsule role and the second the capsule class:

```
frame.incarnate( device, Printer );
```

Capsule Roles are Attributes of Type RTActorRef

A capsule’s structure is defined by a number of communicating capsule roles. In the framework the capsule roles become attributes of type RTActorRef in the containing capsule.

The attribute name is kept the same as the role name, which means that you can reference a capsule role by name in detail C++ code of the containing capsule.

Protocols Become Two Classes: Base and Conjugate

For each protocol two classes are generated to represent the base and conjugate protocol roles. The protocol role classes are generated as subclasses of the root protocol class and are filled in with operations specific to the in and out signals defined in the protocol roles.

In the C++ Services Library Framework diagram you can see that the **Ping_Actor** and **Pong_Actor** each have a port of the same type but assigned to different protocol roles.

Ports are Protocol Type Attributes in RTActor Subclasses

A port on a capsule becomes an attribute in the generated RTActor subclass. The port attribute has the same name as the port in the model. The port attribute will be a subclass of the common base class, **RTProtocol**. It be directly referenced by name in a capsule's C++ detail code. Here a port named talk is defined on a capsule.

The port definition in the capsule class:

```
public:
   // {{{RME protocolClass 'Commands' port 'talk'
   Commands::Base talk;
```

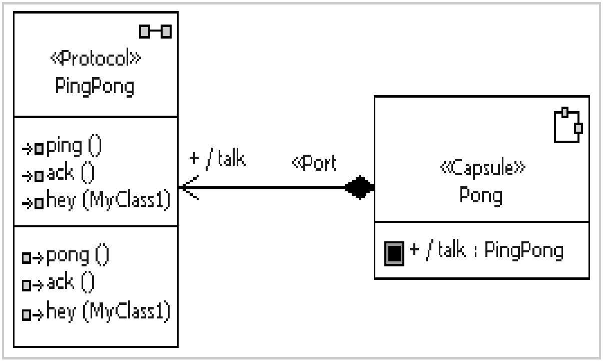
The port being referenced in detail code on the capsule:

```
talk.ping().send();
```

Signals Become Operations in Protocol Classes

Each signal defined in a protocol is generated as an operation with the same name as the signal in the generated protocol class. The return value of the operation dictates the actions that can be performed with the signal. Incoming signals **RTInSignal** and outgoing signals **RTOutSignal** will obviously differ in allowed actions.

Figure 15 PingPong Protocol and Talk Port



From this class diagram you see that the **Pong** capsule has a port named talk of type **PingPong**. The port is not conjugated. The unconjugated generated protocol class PingPong::Base will have operations for each signal and will allow you to reference them from the talk attribute generate on the **Pong_Actor** class.

```
talk.pong().send();
```

This line of code calls the generated signal operation on the port, which returns a RTOutSignal object. Then the common action on an out signal is to send it.

Capsule Roles are Place Holders for Zero or More Capsule Instances

Capsule roles, or RTActorRef classes, are basically place holders for capsule instances. Replicated capsule roles are place holders for multiple instances of compatible RTActor subclasses.

See *RTActorRef* on page 240 for more information on the main uses of capsule roles in your model.

Multiple Containment

Often a capsule instance will only run in the context of a single capsule role, but with multiple containment, a single instance can exist in two or more capsule roles simultaneously.

Capsule Instances Have Access to a RTMessage Object

An RTActor class has access to the current message, RTMessage, that it received. You will often want to access this message in your capsule C++ detail code.

Capsule Instances Have Access to their Controller

Each capsule instance has access to the controller for the thread on which it is running. The RTController class provides several operations that can be useful in a capsule's implementation.

Framework Sample Model

This is the model which was used as an example of how elements from a model will integrate into the C++ Services Library Framework. The grey boxes in the diagram on the C++ Services Library Framework page show classes generated from this model:

Figure 16 Ping Pong Model Class Diagram

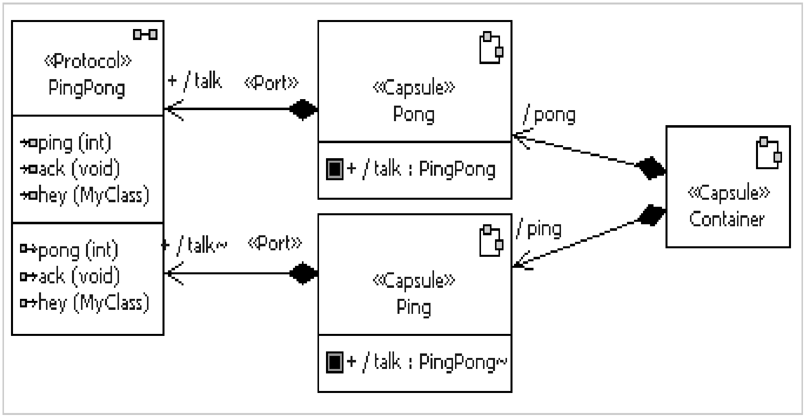
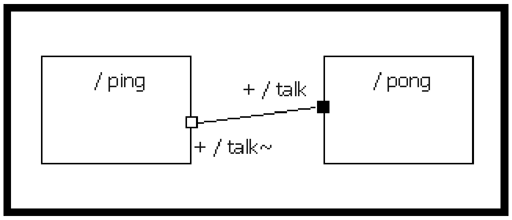


Figure 17 Container Capsule Structure Diagram



Log Service

Implementation classes

Log

Concepts

The System Log is a stream of ASCII text in which system or application events can be recorded. Currently all log output is directed to **stderr**.

Execution speed is affected since each write to the log involves an output system call, which is a relatively expensive operation.

Communication Services

Implementation classes

RTProtocol, RTOutSignal, RTInSignal, RTSymmetricSignal

Concepts

This fundamental service provides most of the standard communication models prevalent in concurrent software system design including asynchronous messaging and rendez-vous like synchronous inter-capsule communication.

The Communication Service is accessed by referencing, by name, a port (which will be a subclass of the RTProtocol class) with the appropriate operation. The port name is the user defined name of the port declared in the model. The named port is generated as a member of the capsule containing the port.

Every named port may actually have a number of port instances associated with it (depending on the multiplicity of the port). Each port instance is capable of sending and receiving messages. The port instances are encapsulated within each RTProtocol object.

A service request results in the creation of instances of RTMessage. These messages are delivered by the Services Library to the ports at the other ends of the connections. They are eventually processed by the behavior of the capsules containing these ports.

Primitives

This service is used for message passing between capsules in real time. Messages sent via this service are processed whenever the necessary CPU cycles become available.

A capsule instance accesses the message that was just received by accessing the **msg** variable.

Upon processing a message received at a particular end port, the RTMessage::sapIndex0 operation returns an index to the particular port instance that received the message. A RTOutSignal::sendAt to the port instance returned by RTMessage::sapIndex0 results in a send to only that particular port instance. The communications services also provide a number of functions for dealing with replicated ports.

Asynchronous and Synchronous Communication

If an asynchronous send is used, the sending capsule will not block while the message is in transit. This mode is well-suited for high-throughput and fault-tolerant systems.

Conversely, if synchronous communication is desired, a blocking send can be used. The semantics of this send are such that during the invocation of the method, the sender (invoker) is blocked until a reply is received even if higher-priority messages arrive in the meantime. At the other end, the receiver does not distinguish between synchronous and asynchronous communications but replies to either in the same way. This has the advantage that it de-couples the receiver from the implementation decisions of its clients regarding which communication mode to use (blocking or non-blocking). In practice, though, the receiver must know something about the expectations of the sender. There are two restrictions that must be observed:

- The receiver must reply with rtport->signal(data).reply() (within the same transition) to messages that are synchronous.
- Circular invokes are not allowed (if capsule A invokes capsule B, and capsule B tries to invoke capsule A, the invoke operation on B will fail with a return code).

Order-Preserving

Messages of equal priority sent along the same binding are delivered in the same order both for messages sent to capsules executing within the same thread and for messages going to another thread.

Note: Such guarantees may not be available when capsules are in different processes.

Lossy

Messages have a high probability of delivery to the receiving object, but it is not guaranteed. For example, messages may be lost if they are sent through unbound ports or if the destination capsule is destroyed dynamically. In distributed versions of this service, loss of messages may also be due to temporary resource depletion (no buffer space) or actual loss in the physical communications medium.

Minimal overhead in message handling

This is due to the relative simplicity of the service and its lack of any automatic form of acknowledgment or flow-control protocols.

Request-Reply

A special feature of the communications services is support for a "request-reply" communication model. These are message exchanges between a sender capsule and a receiver in which the specified reply is expected, handles the request, and responds within the scope of a single transition. See the description of send() and invoke() functions for detailed limitations.

The communications services also support synchronous messaging (similar to a rendez-vous). During a synchronous send, or invoke, the sender is blocked until the receiver has processed the message and sent back a reply. Run-to-completion semantics are enforced, such that a synchronous invoke has the same semantics as a procedure call.

The receiver of an invoked message, or an asynchronous message with expected reply, must respond to it prior to the completion of message processing.

The Semantics of Usage of Message Priorities

A message priority is interpreted as the relative importance of an event with respect to all other unprocessed messages on a thread. This is reflected in a bias towards higher-priority messages over lower-priority messages when scheduling CPU time. If two or more messages of different priority are queued and waiting to be processed, messages with a higher priority are usually processed before messages of lower priority. The slight ambiguity of this definition reflects the variability of scheduling policies due to the inherent non-determinism of distributed systems, as well as changing implementations. In general, good designs should not be critically sensitive to a particular scheduling policy. (The current Services Library scheduler, in fact, uses simple priority scheduling so that messages at a particular priority level are not processed until all higher-priority messages on that controller have been processed.)

Within a given priority level, the Services Library guarantees that messages will be processed in the order of arrival. (Keep in mind, however, that in a distributed system, the order of arrival is not necessarily the same as the order in which the messages were sent.)

Message priorities do not imply interruption of the processing of the current event even if a newly-arrived message is of a higher priority. This is due to the “run-to-completion” semantics of transitions as described in the previous section.

A user-defined message has one of five priority levels associated with it. The following predefined symbols allow the user to specify the priority of a message by name:

- Panic - highest priority available to users; to be used only for emergencies
- High - for high-priority processing
- General - for most processing; also the default
- Low - for low-priority
- Background - lowest priority used for background-type activities

Message priorities disrupt the temporal order of events, which, in practice, often leads to implementation problems. For this reason, it is recommended that, as much as possible, applications limit themselves to a single priority level. However, if priorities are used, then it is good programming practice to avoid the high and low extremes of the range in order to leave room for subsequent design changes. In addition to these user-defined message priorities, there are some system-level priorities. System-level priorities are higher than the highest user-level priority in order to guarantee the correct operation of Service Library routines.

Support for Unwired Ports

Ports can be either wired or unwired. Wired ports are explicitly connected to other wired ports with connectors. But unwired ports are not connected during design. Instead they are dynamically connected at run-time. Unwired ports are bound to other unwired ports by a registered name.

Layer communication therefore involves support for managing connections between unwired ports.

Published Versus Unpublished Unwired Ports

In the layered communication paradigm, unwired published ports (SPP) can only connect with unwired unpublished ports (SAP), or vice versa. A SAP cannot connect to another SAP, and a SPP cannot connect to another SPP. You can think of an SPP as being the server side of a connection and the SAP as being the client. The client always initiates the communication with the server. The terms SAP and SPP are used to abbreviate 'unwired [unpublished|published] port'. You will see that some of the communication service operations are named with these abbreviations to differentiate SAP and SPP operations.

The basic model is that for any given service, there is one server (the SPP), and there may be many clients (the SAPs). The notion of a "service" here is a loose one—a service is some functionality provided by the server capsule to the client capsules. The service is uniquely identified by name. There may exist many different server capsules, each providing a different service. Any given service (name) may have only one server (SPP) registered for it at any given time. Any other providers that attempt to register an SPP of the same name will be declined (the registration will fail).

SPPs are often replicated, with their multiplicity specifying the maximum number of clients that can be bound to the server at run-time. By default, a SAP or SPP is automatically registered under its reference name when the capsule containing that SAP/SPP is initialized.

Note: Multiplicity may be changed dynamically at run-time with the RTProtocol::resize() operation. This may destroy bindings if multiplicity is reduced and allow new bindings if it is increased.

Registration by Name

The basic element of layer communication is a generic name server. SAPs register to the layer service for binding to a SPP under a unique name. SPPs need also register to the layer service in order to publish its unique name for binding with SAPs.

All SAPs are bound to the first SPP that registered for binding under that name. If no SPP exists, the SAP registrations are queued (usually in order) waiting for the SPP to register. SAPs will be bound with the SPP up to the maximum multiplicity of that SPP. SAPs not bound will continue to be queued until an instance of the SPP becomes available due to either a SAP de-registering, an SPP with a larger multiplicity registering, or the SPP is resized.

Registration String

A registration string is used to identify a unique name and service under which SAPs and SPPs will connect. The string has the following format:

```
[<service_name>:]<registration_name>
```

The first part of the registration string is case sensitive. The interpretation of the remaining registration string depends on the specified communication service. Here are some examples:

```
name

service1:name

service2:name

service3://address/name
```

Automatic Versus Application Registration

SAPs and SPPs can be configured to be automatically registered with the layer service, or to be registered by the application using a name to be determined by the application at run-time. If automatic registration is chosen, the registration name must be supplied in the port specification dialog and the Services Library will register the name at startup. In the case of application registration, the SAP or SPP is registered at run-time by calling a communication service operation, such as RTProtocol::registerSAP and RTProtocol::deregisterSAP, in the detail level code of a capsule. The same port may, in fact, be registered under different names at different points in the model execution.

Deferring and Recalling Messages

The Services Library enforces the reactive model of behavior by automatically putting a capsule into a receive mode between successive transitions. This means that there is no need for an explicit user-specified receive method. When a message is selected for processing, the Services Library wakes up the capsule and starts execution of the appropriate transition.

In some cases, a message may be received and the capsule may decide that it would be more convenient to postpone the handling of this event for some later time. For example, the behavior may be in the middle of a complex sequence of state transitions when it receives an asynchronous request to handle a new sequence. Instead of trying to execute two sequences in parallel, it is often simpler to serialize them. To do this, the newly-received message must be held somehow until the current event-handling sequence is complete and then resubmitted. The Services Library allows messages to be deferred and then recalled at a more convenient time.

Timing Service

Implementation classes

RTTimespec, Timing, RTTimerId, RTTime

Concepts

The timing services provide users with general-purpose timing facilities based on both absolute and relative time. To access the timing services, you reference, by name, a timing port that has been defined on that capsule, that is, by creating a port with the pre-defined Timing protocol. Service requests are made by operation calls to this port, while replies from the service are sent as messages that arrive through the same port. If a **timeout** occurs, the capsule instance that requested the timeout receives a message with the pre-defined (and reserved) message signal timeout. Of course, a transition with a trigger event for the 'timeout' signal must be defined in the behavior in order to receive the timeout message.

One shot timer

This kind of timer expires only once, after the specified time duration (relative time), or at the specified time (absolute time). If subsequent timeouts are required, the timer must be reset after each expiration. If repeated timeouts are required, the extra time added for processing each timeout and requesting a new timer may cause some amount of drift in the timing (for example, requesting a timeout every 10 seconds will result in a timeout occurring every 10 seconds + the amount of time required to process the timeout and reset the timer). Round-off of clock ticks may reduce or exaggerate this drift.

Periodic timer

This kind of timer is set to timeout repeatedly after the specified duration until the timer is explicitly cancelled. It does not need to be reset after each expiration. Using the periodic timing service will provide more accurate timing than repeatedly resetting a one-shot timer.

Relative Versus Absolute Time

Two forms can be used to specify a timer request: absolute time and relative time. This service defines absolute time as elapsed time since some fixed point in the past. Relative time is expressed as a number of time units from the current time instant.

In the example below for RTTimespec used with InformAt, a timer is set to expire at a given time represented by `alarm`. Timeout takes place on the hour after the occurance of `alarm`.

```
RTTimespec alarm;
RTTimespec::getclock( alarm );

alarm.tv_sec  += 3600L - ( alarm.tv_sec % 3600L );
alarm.tv_nsec  = 0L;

timer.informAt( alarm ); // at the top of the next hour
```

Relative time is represented in seconds and nanoseconds. The RTTimespec class is used to hold relative times.

```
// relative timer example: can specified with informIn() or
// informEvery().

timer.informIn( RTTimespec( 10, 0 ) );
// one-shot timer in 10 seconds
```

```
timer.informEvery( RTTimespec( 10, 0 ) );
// periodic timer every 10 seconds
```

The examples above assume that a non-wired port named **timer** using the timing protocol has been defined on the capsule invoking the timing service.

Timing Precision and Accuracy

The precision of the timing service depends on the granularity of timing supported by the underlying operating system. Although you can request timeouts with a granularity down to the nanosecond, this does not mean you will get nanosecond precision. Most operating system timing facilities only have a granularity in the millisecond range. Further, the granularity of timing supported on most real-time operating systems is much finer than that of general-purpose workstation operating systems, such as UNIX and WindowsNT.

The service does not guarantee absolute accuracy. This means that intervals can take slightly longer than specified, and events scheduled for a particular time may in fact happen slightly after the actual time has occurred. The magnitude of the delay depends on many factors. However, unless the system is under severe overload, the discrepancy is usually not significant.

Frame Service

Implementation classes

RTActorId, Frame, RTActorClass, RTActorRef

Concepts

Capsule roles can be classified into three categories: fixed, optional, and plug-in.

The latter two types of capsule references are used for dynamically changing structures.

The Frame Service provides the ability to instantiate and destroy optional capsules, to import and deport capsule instances to and from plug-in roles, plus a number of other functions. The rules and definitions governing the Frame Service can become quite involved in some cases. See the *Modeling Language Reference Guide* for more details on the concepts behind dynamic structure.

Optional Capsules

Optional capsules differ from fixed capsules in that the current number of existing instances of an optional capsule reference at any given time may be less than the cardinality specified for that capsule role. The rules governing the instantiation and destruction of optional capsules are as follows:

- An optional capsule can be instantiated as an instance of a particular class only if it is a compatible subclass of the class specified as the capsule role classifier.
- An optional capsule that is explicitly destroyed by the invocation of a method by the immediate container ceases to exist and does not appear anywhere.

Plug-in Capsule Roles: Multiple Containment

The following rules must be satisfied at run time in order for a capsule instance to appear as a plug-in capsule role:

- The capsule instance cannot already be an aspect in the imported capsule reference.
- The class of the capsule instance must be one of the compatible subclasses of the plug-in capsule.

Note: To be compatible with its superclass, a subclass must have a matching compatible port for every connected interface of the superclass reference.

- Capsules may not be imported across model boundaries. That is, a capsule cannot be imported across a process boundary, though it can be imported across a thread boundary within the same model.

None of the ports of the capsule instance to be bound in the destination plug-in role can currently be fully bound in another aspect.

Multiple Containment

Multiple containment allows you to represent capsule roles that are simultaneously part of two or more capsule collaborations. Specifying that two different capsule roles are actually bound to the same run-time instance can simplify the structure of the system by allowing it to be decomposed into different views.

When to use multiple containment

When two or more capsule roles are placed together in a common capsule, the intent is to capture some user-defined relationship between these components. The simplest example of a relationship between objects is pure physical containment; for example, a shelf contains a particular card. When we move into the domain of software, however, the types of relationships that exist can be quite diverse. In communications, for instance, when two terminals are connected to each other in order to exchange information, they are involved in a call relationship. The object-oriented approach encourages us to capture such identifiable relationships as distinct objects. Note that, in physical terms, there is no real entity corresponding to a call; however, it is quite useful to think of it in that way.

Once relationships such as these are captured in unique addressable objects, then it is possible to conceive of operations over such objects, such as terminating a particular call or adding another party to it. To the entities invoking the operations, the structure and implementation within such objects are typically of no concern. Following this line of thought leads us to conclude that these objects are in fact like any other software objects: entities with a set of externally accessible operations and an encapsulation shell that hides their internals. Therefore, capsules can be used to represent arbitrary user-defined relationships between their component actors.

Multiple containment is required to capture situations where a capsule role is involved in multiple simultaneous relationships with capsule roles in another containment.

Replicated Capsule Roles

Replication semantics are a function of the type of role that is replicated:

- All instances of a fixed capsule role with cardinality > 1 are created automatically when the containing capsule is incarnated. The number of instances is equal to the cardinality.
- Instances of an optional replicated capsule are created dynamically, as needed, by the user at run time using the Frame Service. The number of instances can vary from zero up to the number specified by the cardinality. Any attempt to increase the number of instances beyond the cardinality will fail.
- Plug-in capsule roles are filled dynamically at run time as required. However, the maximum number of instances that can be imported into the plug-in capsule role is limited by the cardinality of the role. As is the case for optional capsules, trying to import beyond this limit will fail.

Exception Service

Implementation classes

Exception

Concepts

The Exception Service gives the user the ability to define custom policies to recover from exceptions. Exceptions are program errors that normally make it impossible for a piece of code to proceed. In Rose RealTime exceptions are always raised by the application. The Services Library does not automatically raise exceptions for errors that are detected in the Services Library.

RTController Error Codes

Many of the Services Library operations can set an error code. If any operation in a controller fails, an internal variable is set with an error code. The error values are defined with an enumeration in the `RTController` class. For a detailed description of the defined error values and their meaning, see *Error Enumeration* on page 118.

Accessing the Error Value

The error **enum** identifier for the current error can be obtained via `RTController::getError()` or `RTActor::getError()` which are available directly from capsule code blocks. A description of the current error code can be accessed by calling the operation `RTController::strerror()` or printed via `RTController::perror()` on the current controller object. The controller object for any instance, port or capsule, can be retrieved by calling the RTActor::context() operation on the instance.

Example

The initialization phase of an application might include a transition with code like that shown below where a capsule instance must establish contact with a peer before beginning a more involved exchange. The relevant portions include testing the return value from the send primitive and choosing the appropriate reaction by examining the reason for failure.

The following is an example of how to obtain an error and how to recover with a send on a unconnected port:

```
if( ! peer.Hello().send() )
{
   switch( context()->getError() )
   {
   case RTController::noConnect:
      timer.informIn( RTTimespec( 1, 0 ) ); // try again later
      break;

   default:
      log.show( "Unexpected error sending to peer: ");
      log.show( context()->strerror() );
      log.cr();
      break;
   }
}
```

Error Enumeration

The error values are defined with an **enum**, `Error`, which is defined in the RTController class as follows:

```
enum Error
{
ok,
// code            triggered by
// ====            ============
alreadyDeferred,   // CommDefer
badActor,          // FrameDeport FrameDestroy FrameImport
badClass,          // FrameImport FrameIncarnate
badId,             // TimerCancel
badIndex,          // FrameIncarnate and Frame Destroy
badRef,            // FrameImport FrameIncarnate
badSignal,         // CommInvoke CommReply CommSend
badValue,          // LayerResize
deferInvoke,         // CommDefer CommInvoke
dereg,               // LayerDeregister
imported,            // FrameDestroy FrameIncarnate
noConnect,           // CommInvoke CommRelpy TimerInform
noMem,               // all
noReply,             // CommInvoke
notImported,         // FrameDeport FrameImport
notOptional,         // FrameDestroy
prio,                // CommReply CommSend TimerInform
recursiveInvoke,     // CommInvoke
refFull,             // FrameImport FrameIncarnate
reg,                 // LayerRegister
replRef,             // FrameImport
xRtsInvoke,          // CommInvoke
};
```

alreadyDeferred

A message can only be deferred once within the chain of transitions it triggers. Subsequent deferrals will fail and will set the error code to this value.

badActor

An invalid RTActorId was used with the frame service import, deport or destroy primitives. The Services Library currently only recognizes nil pointers as invalid.

badClass

The frame service incarnate and import primitives validate the request to ensure that references contain only compatible classes. In the case of import, the operation must not introduce a cycle in the dynamic structure of the system and the imported capsule must have sufficient unbound replications of the necessary interface ports. This error results when any of these conditions are not met.

badId

The cancelTimer primitive of the timing service requires a valid timer identifier returned by one of the informAt, informEvery, or informIn primitives. These identifiers are invalidated by the cancelTimer primitive and, except for the case of informEvery, during the delivery of the time-out message. This error is recorded if `cancelTimer` is applied to an expired or cancelled timer identifier.

badIndex

The frame service incarnate and import primitives accept an optional index argument that specifies where, within the reference, the incarnated or imported capsule instance should go. If a capsule instance already occupies the specified index, this error is signalled.

badRef

This signals an inconsistency in the generated code in the capsule class that owns the reference into which a component is being incarnated or imported. If you encounter this error, contact Rational Rose RealTime support.

badSignal

The signal is not valid for the protocol.

badValue

The specified replication factor is invalid. It must be greater than zero.

deferInvoke

Invoked messages may not be deferred. Attempting to use the defer primitive on a message in behavior that was invoked leads to this error.

dereg

Attempting to deregister an unwired port that is not currently registered results in this error.

imported

Only optional components may be dynamically incarnated or destroyed. Applying the incarnate or destroy primitives to an imported reference triggers this error.

noConnect

Successful use of the send, invoke, and reply primitives requires an established binding involving the port instance referenced in the primitive. This error results when that binding does not exist. Remember that send and invoke, applied to a replicated port, are equivalent to the use of the same primitive on each instance within the reference. If any port is unbound, this error will result.

noMem

RTController instances each maintain a local list of unused RTMessage objects. When this list is exhausted, and a request for more messages from the associated RTResourceMgr object is not satisfied, the result is this error. This usually indicates that available free memory on the target is exhausted. RTMessage objects are required in many Services Library primitives.

noReply

When a message is invoked, a reply is expected. If the receiver does not produce a reply, this error is recorded. As with noConnect, this error can occur if any incarnation fails to reply.

notImported

Only plug-in capsules may be the targets of the import and deport frame service primitives. Attempts to import into, or deport from, other types of references are disallowed and result in this error.

notOptional

Only optional capsules may be destroyed by the frame service.

prio

Each of the send, informAt, informIn, and informEvery primitives accept an argument that is interpreted as a message priority. Applications are restricted to the use of the five priorities: Panic, High, General, Low, and Background. Other values are disallowed and trigger this error.

recursiveInvoke

In Rose RealTime, behavior semantics are run-to-completion. While a capsule instance is reacting to one event, it is unavailable for handling other events. Normally, this does not present any problems. In the case of synchronous interactions, through the use of the invoke primitive, a chain of invocations might lead back to a capsule which is an earlier part of that chain. That capsule is not in a well-defined (application) state and thus cannot have specified an appropriate reaction to this event. The runtime system detects this situation, records this error, and causes the offending invoke call to fail.

refFull

The incarnate and import frame service primitives require a reference that has room for the new link to be created. This error results when the target reference is already full.

reg

A name must be given in the application of the register primitive of unwired ports. A nil pointer is illegal, and is the source of this error.

replRef

One variant of the frame service import primitive accepts a capsule reference. If the replication factor of that reference is not one (1), the usage is ambiguous and disallowed. This error is used to signal this condition.

xRtsInvoke

The current implementation of the Services Library does not support the use of the invoke primitive with bindings that span physical thread boundaries. This error is the result of an attempt to use invoke in such a context.

External Port Service

The External Port service example provides an API that lets non-Rational Rose RealTime threads call a function to raise an event on a port of a Capsule in a Rational Rose RealTime C++ application.

For additional information, see *Port Services* on page 299, and the C++ Examples chapter in the book **Model Examples, Rational Rose RealTime.**

Running Models on Target Boards 7

Contents

This chapter is organized as follows:

- *Step 1: Verify Tool Chain Functionality* on page 123
- *Step 2: Kernel Configuration* on page 124
- *Step 3: Verify main.cpp* on page 125
- *Step 4: Try Manual Loading* on page 125
- *Step 5: Running with Observability* on page 126

Overview

This chapter describes what you need to know to successfully compile, build, and run models with the C++ Services Library on target boards. Because of the different brands of embedded operating systems, and varying configurations found on each, it is critical that you understand your target operating system and what services the C++ Services Library will expect of the target operating system before you try to run a model on your target RTOS.

The C++ Services Library ships with supported configurations for a set of target processors, operating systems, and compilers. See the Installation Guide, Rational Rose RealTime for a listing of the supported targets. You may however have to configure and customize the shipped libraries to work with your specific configuration.

Before trying to compile and download a complex model from Rational Rose RealTime, run through the following steps to validate that your environment, operating system, kernel, and C++ Services Library is setup correctly.

Step 1: Verify Tool Chain Functionality

A functioning development environment must be in place prior to building and running models with Rose RealTime. You should be able to compile, load, and execute non Rose RealTime programs from the command line. This includes the correct installation of tools such as compilers, linkers, assemblers, debuggers included with your RTOS installation. In addition, it is important to ensure that all environment variables are defined to provide access to the header files and library files shipped with your compiler.

Often you will need to setup environment variables that point to the root of the RTOS tools installation directory and also to the include and library directories.

Rational Rose RealTime expects all tools to be available from the command line.

How to Test

An easy way to test that your tool chain is setup properly is to create, build, and run a simple "Hello World" program which prints something to the console. This program should not use (be linked with) the C++ Services Library.

Write, compile, link, download, and run the "Hello World" program on the target. If it executes successfully then your tool chain is setup properly. Your RTOS usually comes with a set of example programs that you can also use to validate your environment.

Step 2: Kernel Configuration

The standard configuration of the Services Library anticipates that the target operating system will support a set of services, for example: mutual exclusion mechanisms, multi-thread support, timing, standard input/output, memory management, and TCP/IP. In general, most commercial real-time operating systems (RTOS) have these services.

Ensure that the RTOS has the following minimum services built into the kernel:

- A service which provides infinite and timed blocking.
- A function that returns the current time.
- Task/thread creation with a specified stack size and priority.
- Standard input/output.
- For observability, TCP/IP support is required.
- Some support for memory management is required.
- Main function, some RTOS have their own defined. If so then the main function in the Services Library must be redefined. See the next step for more information.

If your RTOS kernel does not support these services then read your RTOS documentation on how to rebuild your kernel to include them.

Step 3: Verify main.cpp

In order for the execution of the model to begin, code must be provided to call `RTMain::entryPoint(int argc, const char * const * argv)` passing in arguments to the program. This code is placed in the file $RTS_HOME/src/target/<target name>/MAIN/main.cpp.

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, thus an array of strings must be explicitly created before calling `RTMain::entryPoint`. Look at the implementation added to the $RTS_HOME/src/target/TORNADO1/MAIN/main.cpp file.

A C++ Services Library model assumes that it is the root task in the system. The model will define the root task, initialize the C++ run-time, the system timer and other things. For some targets you may have to modify this behavior in main.cpp.

If your platform does not provide a mechanism for passing arguments to an executable, the arguments for `RTMain::entryPoint` can be defined from within the toolset in the **DefaultArguments** (**Component, C++ Executable**) property.

Step 4: Try Manual Loading

At this point you should be able to build a simple “Hello World” model in Rose RealTime. Build it for your target board. Then load, and run it manually.

Note: With some target operating systems when a Rose RealTime model is built you still aren’t finished. In some cases, as with pSOS+, the Rose RealTime model is built as a library and you have to compile and link the board support package with the Rose RealTime model library to create an executable. The simplest way to do all of this is to see your target board documentation, sample makefiles and programs.

To compile for a specific platform, ensure that a C++ Executable component is created in Rose RealTime with the correct `TargetConfiguration` set to the library for your platform. This will tell the code generator which build scripts and libraries to use.

Once the simple model is built, download to the target board and run it. See your target documentation for steps required to download and run an executable.

On some target boards the root process or the main function is spawned automatically, but on others, for example with Tornado, you have to specify the entry point function. Look in main.cpp for your target to see what function to call to start the model. For example, on Tornado it is `rtsMain`.

When the executable is run you will see the C++ Services Library banner and the debugger prompt:

```
Rational Rose RealTime C++ Target Run Time System
Release 6.40.C.00 (+c)
Copyright (c) 1993-2001 Rational Software
rosert: observability listening not enabled
RTS debug: ->
```

Type ‘quit’ to let your model run.

At this point you have successfully verified that the environment is setup properly and that your RTOS is configured correctly.

Step 5: Running with Observability

Next you can try running the model with observability and watch the execution of the model from within the toolset.

Try to connect the toolset to the running model. First, download the model and run it with the following command line parameter:

```
-obslisten=<portnumber>
```

For example:

```
-obslisten=12345
```

If your RTOS does not support command line arguments, you must add this argument to the **DefaultArguments** (**Component**, **C++ Executable**) property on the component you create to build this model.

When the model is started with `-obslisten` it won’t start actually running the model until you have connected to the model via the toolset and pressed the start button. You should see the following banner after running the model executable with the -obslisten command line parameter:

```
Rational Rose RealTime C++ Target Run Time System
Release 6.40.C.00 (+c)
Copyright (c) 1993-2001 Rational Software
rosert: observability listening at TCP port 12345

****************************************************************
* Please note: STDIN is turned off.
* To use the command line, telnet to the above mentioned port.
* The _output_ of any command will be displayed in _this_
* window.
***************************************************************
```

After the **telnet** client has connected to the target, you must press ENTER a few times to give the target a chance to recognize that this is a **telnet** connection rather than a toolset connection.

Next, within Rational Rose RealTime create a processor and component instance from the component that you used to build your model. In the component instance specification, change the **Target Observability Port** to the value you specified from the command line `<portnumber>`. Click **OK**, right-click on the component instance, and select **Attach Target**. The RTS Browser will appear. Press the **Start** button, and you can use the observability tools to watch the execution of your model.

Command Line Model Debugger 8

Contents

This chapter is organized as follows:

- *Starting the Run Time System Debugger* on page 129
- *Run Time System Debugger Command Summary* on page 131
- *Thread Commands* on page 132
- *Informational Commands* on page 133
- *Tracing Commands* on page 137
- *Control Commands* on page 139

Overview

This section describes the Run Time System debugger commands. The Run Time System debugger provides a mechanism to allow UML for Real-Time models executing on the Run Time System to be debugged at the UML for Real-Time concept level. The Run Time System debugger does not provide source-level debugging. Source code debugging requires an external source level debugger for C++, such as gdb. Some versions of the Run Time System libraries are supplied with the command line debugger disabled for optimum efficiency. You can recompile the Run Time System source code to configure the Run Time System without the debugger. This saves some space in the executable model. For further information, see *Configuring and Customizing the Services Library* on page 167. The debugger must be configured in order for Observability to be enabled.

Starting the Run Time System Debugger

URTS_DEBUG Parameter

You can use the `URTS_DEBUG` parameter to initialize the Run Time System debugger with a set of commands to run at start-up. This is used most commonly to tell the debugger to quit, causing the model to run without the Run Time System interaction. The `URTS_DEBUG` parameter can be passed on the command line to the executable. Add the `-URTS_DEBUG=` parameter on the command line. For example, to run the executable without the debugger interaction, set the debug command to “quit” before starting the executable as follows: `MyTopLevel_Capsule -URTS_DEBUG=quit`. You can also set `URTS_DEBUG` as an environment variable. This variable is used by default whenever no `-URTS_DEBUG` parameter is passed on the command line. The `URTS_DEBUG` variable should be set to a command sequence to be performed by the debugger on start-up. Multiple commands should be separated by semicolons (;).

Differences Between Single-Threaded and Multi-Threaded Run Time System Debugger

In single-threaded mode - that is, when using a Run Time System which has been configured to support only a single thread - the debugger must share the same thread of control as the user's capsules. This has two fundamental implications. Input to the debugger is accepted only when the system is in a stopped state, and blocking calls in user transitions may prevent the debugger from operating correctly. The system can be considered to be in a stopped state when one of the following occurs:

- The top capsule is about to be instantiated.
- A trace point is encountered.
- The debugger has accepted a command from the user to allow `N` messages, and `N` messages have been dispatched.

In multi-threaded mode, the debugger has its own thread of control. This may lead to the case where any model output is interleaved with the debugger output. In general, the thread related to timing should be detached when using the debugger; other threads can be attached or detached as desired.

Application-Specific Command-Line Arguments

You can supply additional command-line arguments for use by your model, as you would for any other application. The arguments are passed on the command line after the name of the executable, for example:

```
myTopCapsule -URTS_DEBUG=quit foo 99
```

Alternatively, they can be specified in the Parameters text box on the component instance specification dialog.

The first item on the command line is the name of the executable. Several arguments can be supplied for the Services Library (`-obslisten`), while another argument can be passed to the debugger (`-URTS_DEBUG`). All arguments are made accessible to the application.

Accessing

The following static functions are provided on the class `RTMain` to allow the user model to examine the argument list:

```
int RTMain::argCount()
const char * const * RTMain::argStrings()
```

Use `argCount()` to return the number of arguments passed on the command line. `RTMain::argCount()` is equivalent to `argc` in a traditional C/C++ program.

Use `argStrings()` to return an array of pointers to the actual arguments. Each argument is stored in a char *. `RTMain::argStrings()` is equivalent to `argv` in a traditional C/C++ program.

Providing Arguments on Targets That Do Not Support Command-Line Arguments

Some targets do not provide the ability to start up a program with command-line arguments. Rose RealTime provides an interface within the toolset that allows you to specify startup arguments that are made available to the program at run-time. You can specify arguments via the component property **DefaultArguments** (**Component**, **C++ Executable**).

Run Time System Debugger Command Summary

- Thread Commands
- Informational Commands
- Tracing Commands
- Control Commands

Help

help: Prints help information.

taskId, capsuleId, portId

Physical threads in the application are each identified by a `taskId`. Listing the threads in the application using the **tasks** command shows the `Id` of each task. Use this `Id` when referring to a particular thread for commands such as **attach**, **detach**, and **printstats**.

Each capsule instance has a unique `capsuleId`. The `capsuleId` indicates the capsule's position in the containment hierarchy. The top-level capsule instance always has an `Id` of `1`. The instances contained in it are called `1/1`, `1/2` and so on. Replicated references, however, are shown by a single `Id`. They can be identified individually by suffixing the `Id` number with `n`, where `n` is the particular instance number (for example, `1/5.1`). Note that the default replication factor is always `1`; for example, `1/5` is exactly the same as `1.1/5.1`. The `capsuleId` is used in conjunction with the **info** and system commands. The **system** command shows the `capsuleId` corresponding to each capsule.

Each port is identified by its `portId`. These `portIds` are relative to the capsule where they are defined and unique only within this capsule class.

The `portIds` for a capsule class can be listed using the **info** command.

Running a Model

When running a model using the command-line debugger, you will see the following setup:

```
Rational Rose RealTime C++ Target Run Time System
Release 6.40.C.00 (+c)
Copyright (c) 1993-2001 Rational Software
rosert: observability listening not enabled

RTS debug: ->
```

Thread Commands

tasks: Prints the list of tasks (threads).

detach [<taskId>]: Do not monitor a thread specified by `taskId`. Allows the thread to run freely. If taskId is not specified, it detaches all tasks.

attach <taskId>: Monitor a thread specified by `taskId`. `TaskIds` of the different physical threads in the model can be determined using the **tasks** command.

The example used in the following description has been configured to use threads. The output is slightly different for applications compiled in a non-threaded world.

tasks

Lists all threads in the model. Each thread is identified with a `taskId`. The main thread always appears in the list of threads. Any additional user-defined physical threads also appear in the list.

```
RTS debug: -> tasks
```

```
   0: stopped    main
   1: stopped    Thread1
   2: stopped    Thread2
   3: detached time
RTS debug: ->
```

attach <taskId>

Allows the debugger to interact with the specified task (thread). TaskId must be one of the taskIds listed by the tasks command. When a thread is attached, messages within that thread are only processed when the go command is given.

```
RTS debug: ->attach 1
   Attached Task 1
RTS debug: ->
```

detach <taskId>

Allows the thread (`taskId`) to run freely. The debugger does not control the specified thread any longer. The thread processes all outstanding messages and then waits for new messages.

```
RTS debug: ->detach 1
   Task 1 detached
RTS debug: ->
```

Informational Commands

saps: Shows all registered SPPs and the corresponding SAPs.

system [<capsule> [depth>]]: Lists all instantiated capsules in the system, starting with the specified capsule, to a specific depth.

info <capsuleId>: Shows information about the capsule instance specified by the capsuleId.

printstats <taskId>: Prints the run-time statistics for thread `taskId`.

saps

Lists all registered unwired ports (SAPs and SPPs).

```
RTS debug: ->saps
Service: ':'
   Name (SAPs, SPPs)
   prot2 (1,1)
RTS debug: ->
```

system [<capsuleId> [<depth>]]

The **system** command lists all the active capsules in the system, starting with `<capsuleId>` (default: 1 = the top capsule) and `<depth>` (default: 0 = all) levels down.

Both the parameters `<capsuleId>` and `<depth>` are optional; however, if you give the `<depth>` parameter, you must give the `<capsuleId>` parameter as well.

Each capsule is displayed in the following form:

```
refName : className (type = fixed) capsuleId [more]
```

Containment is indicated by indentation and one leading dot for each containment level. For example, in the following output, the top level capsule is listed first, followed by all the capsule instances in its decomposition:

```
RTS debug: ->system
Main_OnTop : Main (fixed) 1
. gen1 : Generator (fixed) 1/1
. gen2 : Generator (fixed) 1/2
. echo : Echo (fixed) 1/3
. . logger : LogBuffer (fixed) 1/3/1
. . . servus : GreetServer (fixed) 1/3/1/1
. . logger : LogBuffer (fixed) 1/3/1.2
. . . servus : GreetServer (fixed) 1/3/1.2/1

RTS Debug: ->
```

In the following example, we want to start with a different capsule:

```
RTS debug: ->system 1/3
echo : Echo (fixed) 1/3
. logger : LogBuffer (fixed) 1/3/1
. . servus : GreetServer (fixed) 1/3/1/1
. logger : LogBuffer (fixed) 1/3/1.2
. . servus : GreetServer (fixed) 1/3/1.2/1

RTS Debug: ->
```

And in this example, we start with a different capsule, and also limit the depth to 1 level:

```
RTS debug: ->system 1/3 1
echo : Echo (fixed) 1/3 [2 more]

RTS Debug: ->
```

In the last example, we can see the [2 more] message after the capsule. This means that the capsule in question has 2 contained capsules that were not displayed since the depth parameter we supplied limited the output. This [N more] message is not recursive, so it only indicates the number of hidden capsules in the next immediate level.

info

The **info** command returns information about a particular capsule instance. The **info** command displays the name of the capsule class for the identified instantiation, the role name (from the container), the current state of the capsule, the memory address of the capsule, whether any probes are attached to the capsule, and a list of ports, components and attributes. As with capsules, ports listed are identified by an Id number.

```
RTS debug: ->info 1/3/1
ClassName: LogBuffer
ReferenceName: logger
CurrentState: wait4activity
Address: (LogBuffer_InstanceData *)0x42beef
No Capsule Probe attached.

Relay ports:
  0: commandPort[10]
```

```
End ports:
   0: commandPort[10] (wired)
   1: echoAccess (SPP, prot2)

Components:
   1: servus

Attributes:
   0: attribute1 == 0

RTS debug: ->
```

printstats <taskId>

Prints information about the number of queued messages and a breakdown of these messages by priority. The alias `stats` is mapped to this command. The output about a timing task also informs about the number of unexpired timers as well as the time to the next timeout.

```
RTS debug: ->printstats 0

main
   No error.
RTS debug: ->
```

Messages queued	**incoming messages**
Synchronous 0	0
System 0	0
Panic 0	0
High 0	0
General 1	1
Low 0	0
Background0	0
Total	1

In this command, the output consists of the name of the thread, the last error encountered, and the number of outstanding messages available to be delivered for each of the distinct priorities, and the number of incoming messages at each priority.

Additional statistics can be gathered if the macro `RTS_COUNT` is set to 1 in the `RTConfig.h` and the `Target RTS` recompiled.

The additional statistics are:

- number of incarnated capsules
- number of destroyed capsules
- number of created ports
- number of destroyed ports
- number of allocated messages
- peak message allocation
- number of delivered messages and their breakdown by priority
- number of timers requested (number of calls to informIn)
- number of expired timers (number of timeouts)
- number of cancelled timers
- number of unexpired timers

Tracing Commands

log <category> <detail-level>: Logs UML for Real-Time primitives. Selects the service to log (`communication`, `layer`, `timer`, `system`, `all`) and the detail (**none**, **errors**, **all**).

log <category> <detail-level>

The **log** command turns ON the logging of system services.

The categories are `communication`, `exception`, `frame`, `layer`, `timer`, `system`, and `all`. The detail levels are `none`, `errors`, and `all`.

Each message log shows the direction of the message, the receiving capsule (the `to' capsule), the sending capsule (the `from' capsule), and the data. The form of each message log is as follows:

```
RTS debug: 0>


message
     to  capsule(Class)<state>.portName[index]:signalName
    from capsule(Class)<state>.portName[index]
    data dataValue An example of message trace is shown below:
```

An example of message trace is shown below:

```
RTS debug: ->log comm all

RTS debug: -> go 1
    go 1

message
   to client(Client)<Dozing>.cliServComm[0]:hello
   from server(Server)<S1>.cliServComm[0]
   data (void *)0

RTS debug: ->log comm none

RTS debug: ->go 1

go 1
RTS debug: 1>
```

Events that will be logged are:

- Communications: Defer, Recall, RecallAll, Send, Invoke, reply.
- Layer: Register SAP, Deregister SAP, Register SPP, Deregister SPP, resize.
- Timer: Cancel, InformIn.

Note that the detail levels are as follows:

- none - suppresses all log messages
- errors - logs only events that raise an error code
- all - logs all events as described above.

Control Commands

exit: Terminates the Run Time System process

go [<n>]: Delivers `n` messages

step [<n>]: Delivers `n` messages.

quit: Quits debug mode. Allows all tasks to run freely.

continue: Allows you to start running the target and make TO connections at a later time.

exit

Exits the process. If you have logs turned ON, you may notice a sequence of cancellation/stop messages before the process is exited.

go [<n>]

Delivers `n` messages in the model. If `<n>` is omitted, the default is 10.

step [<n>]

Delivers `<n>` messages in the model. If `<n>` is omitted, the default is 1.

quit

Detaches the debugger and lets the model run freely. The command line debugger is turned off and the program is run to completion (all messages are delivered).

continue

Allows you to start running the target and make TO connections at a later time. From the command line, continue is similar to clicking **Run** in the Toolset; it starts the execution while retaining control (unlike quit which gives up control). For example:

```
MyTopCapsule -obslisten=1234 -URTS_DEBUG=continue
```

Inside the C++ Services Library 9

Contents

This chapter is organized as follows:

- *Organization of the Services Library Source* on page 141
- *Configuration Preprocessor Definitions* on page 144
- *Integrating External IPC Into a Model* on page 151
- *Optimizing Designs* on page 160

Organization of the Services Library Source

This chapter provides extended details regarding the C++ Services Library. For those who want to configure the C++ Services Library for speed or size, see the *Configuring and Customizing the Services Library* on page 167.

Much of the configurability of the C++ Services Library is done at the source code level. Understanding the organization of the source code and build files will help you navigate the directory structures.

The Services Library is organized to be highly configurable, not only for customers but also to provide an easy way to support a large number of different platforms and configurations.

$RTS_HOME

The C++ Services Library source files are by default installed in the $ROSERT_HOME/C++/TargetRTS directory. $RTS_HOME will be used often in this document to refer to this directory.

For further information, see *Directory Structure* on page 143.

Configuration Naming Convention

When you start browsing the directories and files that make up the Services Library, you will notice directory names and file names that may seem cryptic. These names are actually based on an easy to use naming scheme to uniquely identify the many library configurations.

Platform Name (or Configuration)

A specific Services Library configuration is identified by its platform name. The platform name is made up of two parts:

- the target base name
- the libset name.

```
<platform name> ::= <target base name>.<libset name>
```

For example:

```
AIX4T.ppc-gnu-2.8.1
SUN5T.sparc-gnu-2.7.1
NT40T.x86-VisualC++-6.0
```

Target Base Name

The target base name identifies the operating system, and its configuration and version. For this reason, the target base name is made up of three parts that describe the operating system (os):

- the os name
- the os version
- the os configuration (single (S), multi-threaded (T))

```
<target base name> ::= <os name><os version><os configuration>
```

For example:

```
AIX4T -> AIX 4.X Multi-threaded
SUN5T -> Solaris 5.X Multi-threaded
NT40T -> WindowsNT 4.x Multi-threaded
```

Libset Name

The libset name identifies a processor architecture and compiler. The libset name is made up of three parts:

- the processor
- the compiler name
- the compiler version

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

For example:

```
ppc-gnu-2.8.1 ->
PowerPC processor using Free Software Foundation gnu version 2.8.1
```

```
sparc-gnu-2.7.1 ->
Sparc processor using Free Software Foundation gnu version 2.7.1


x86-VisualC++-6.0 ->
X86 processor using Microsoft Visual C++ version 6.0
```

Summary

You would therefore read the platform name introduced in the first section as:

```
AIX4T.ppc-gnu-2.8.1 ->
For the AIX 4.X Multi-threaded RTOS running on a PowerPC processor
using Free Software Foundation gnu version 2.8.1
```

This naming scheme is used throughout the C++ Services Library.

Directory Structure

The source structure contains directories that mirror the convention described in the Library *Configuration Naming Convention* on page 141. For example, the libset directory contains libset specific files (processor, compiler), the same goes for the target directory (operating system).

The best way to understand the directory structure is to browse it yourself.

codegen

This directory contains scripts for compiling models on different platforms.

include

This directory contains interface definitions for the Services Library classes and structures. These headers are used for both model and Services Library compilation.

config

This directory contains platform specific (operating system and compiler) configurations. Each platform has its own directory that contains the platform specific scripts and configuration files. For further information, see *Platform Name (or Configuration)* on page 142.

target

This directory contains target (operating system) configurations. Each target has its own directory that contain the target specific scripts and configuration files. For further information, see *Target Base Name* on page 142.

lib

This directory contains the compiled libraries.

libset

This directory contains processor and compiler specific configurations. Each libset has its own directory that contains the libset specific scripts and configuration files. For further information, see *Libset Name* on page 142.

src

This directory contains the generic (code which is platform independent) source files for the library. Each class has a directory that contains the class implementation. Within the src directory is a target directory which contains target specific (os) implementation files. Each target has its own directory that contains target specific source files. For further information, see *Target Base Name* on page 142.

tools

This directory contains scripts used for building models and building the libraries.

Configuration Preprocessor Definitions

Much of the configurability of the Services Library is done at the source code level 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 is not required for many compilers.

These files override macros whose defaults appear in $RTS_HOME/include/RTConfig.h. The macros and their default values are listed in the following pages.

Note: 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.

USE_THREADS

Default value: none, must be defined in the platform headers (usually `RTTarget.h`).

Possible value: 0 or 1.

Description: Determines whether the single-threaded or multi-threaded version of the Services Library is used. If `USE_THREADS` is 0, the Services Library is single-threaded. If `USE_THREADS` is 1, the Services Library is multi-threaded.

RTS_COUNT

Default value: 0.

Possible value: 0 or 1.

Description: If this flag is 1, the Services Library will keep track of the number of messages sent, the number of capsules incarnated, and other statistics. Naturally, keeping track of statistics adds overhead.

DEFER_IN_ACTOR

Default value: 0.

Possible value: 0 or 1.

Description: If this flag is 1, then the defer queues will be kept in each capsule. If not then all deferred messages will be kept in one queue per thread. This is a size/speed trade-off. Separate queues for each capsule use more memory but result in better performance.

INTEGER_POSTFIX

Default value: 1.

Possible value: 0 or 1.

Description: Set whether or not the compiler understands the post increment and decrement operators on classes.

That is:

```
Class x;
x++;
x--;
```

LOG_MESSAGE

Default value: 1.

Possible value: 0 or 1.

Description: Set whether or not `RTActor::logMsg()` is called before the delivery of each message. This operation is used by the debugger.

OBJECT_DECODE

Default value: 1.

Possible value: 0 or 1.

Description: Enable the conversion of strings to objects, needed for the external IPC.

OBJECT_ENCODE

Default value: 1.

Possible value: 0 or 1.

Description: Enable the conversion of objects to strings, needed for the IPC, and log services.

OTRTSDEBUG

Default value: DEBUG_VERBOSE.

- Possible value: DEBUG_VERBOSE.

  This flag is used to enable the Services Library debugger. It will make it possible to log all important internal events such as the delivery of messages, the creation and destruction of capsules, and so on. This is necessary for the target observability feature.

- Possible value: DEBUG_TERSE.

  This will reduce the size of the resulting executable at the expense of limiting the amount of debug information.

- Possible value: DEBUG_NONE.

  This will further reduce the executable size, while increasing performance. However, the Services Library debugger will not be available.

RTREAL_INCLUDED

Default value: 1.

Possible value: 0 or 1.

Description: If 1, this flag allows the use of the `RTReal` class.

PURIFY

Default value: 0.

Possible value: 0 or 1.

Description: If 1, this flag indicates that the Purify tool is being used. This tells the Services Library to disable all object caching which will degrade performance but allow Purify to monitor `RTMessage` objects.

RTS_INLINES

Default value: 1.

Possible value: 0 or 1.

Description: Controls whether Services Library header files define any inline functions.

RTS_COMPATIBLE

Default value: 520.

Possible value: 520 or 600.

Description: Used to indicate whether the ObjecTime Developer 5.2 features are to be included in the Services Library.

HAVE_INET

Default value: 1.

Possible value: 0 or 1.

Description: Used to indicate whether the TCP/IP stack is available. Required for Target Observability.

INLINE_CHAINS

Default value: <blank>.

Possible values: inline or <blank>.

Description: This macro is used to indicate whether transition code chains are to be inserted directly into the code or invoked as functions. The basic trade-off is performance against memory. Preliminary measurements indicate that with this feature disabled, the size of a capsule class definition is reduced by 0.5 Kilobytes on the average.

Note: This cost is incurred only once for each capsule class.

INLINE_METHODS

Default value: inline.

Possible values: inline or <blank>.

Description: This causes transition functions to be **inlined** for better performance at the expense of potentially larger executable memory size. Note that not all compilers will handle this option correctly. Failures will generally be in the form of link errors.

RTFRAME_CHECKING

Default value: RTFRAME_CHECK_STRICT.

- Possible value: RTFRAME_CHECK_NONE.

  This has the same behavior as pre-ObjecTime Developer 5.2 releases.

- Possible value: RTFRAME_CHECK_LOOSE.

  This ensures that the reference to the capsule being operated on is in the same thread as the Frame SAP.

- Possible value: RTFRAME_CHECK_STRICT.

  This ensures that the reference to the capsule being operated on is in the same capsule as the Frame SAP.

Description: The frame service is intended to provide operations on components of the capsule which has the frame SAP. The checking can be relaxed or removed.

Notes:

- Needs to be at least LOOSE if free list is disabled.
- ROOM semantics enforced.

Example:

```
// Possible values for RTFRAME_CHECKING:
// The frame service is intended to provide operations on components
// of the actor which has the frame SAP. The checking can be relaxed
// or removed.
#define RTFRAME_CHECK_NONE 0
// no checking (pre-5.2 compatible)
# define RTFRAME_CHECK_LOOSE 2
// references must be in the same thread
#define RTFRAME_CHECK_STRICT 4
// reference must be in the same actor
#ifndef RTFRAME_CHECKING
#define RTFRAME_CHECKING RTFRAME_CHECK_STRICT
#endif
```

RTFRAME_THREAD_SAFE

Default value: 1.

Possible value: 0 or 1.

Description: Setting this macro to 1 guarantees that the frame service is thread safe. This is an option because some applications may use the frame service in ways that don't require this level of safety.

Notes:

- If disabled then bindings may be inconsistent if there are concurrent frame service calls.
- ROOM semantics enforced. Potential for increased latency time before a higher priority frame service call can start.

Example:

```
// Setting this macro to 1 guarantees that the frame service is thread
// safe. This is an option because some applications may use the frame
// service in ways that don't require this level of safety.
#ifdef RTFRAME_THREAD_SAFE
#define RTFRAME_THREAD_SAFE 1
#endif
```

RTFRAME_USE_FREELIST

Default value: 1.

Possible value: 0 or 1.

Description: This maintains a free list in `RTActorRefs` for plug-in components. The free list costs at most two pointers per replication but avoids a linear search in incarnate and import operations. Pre-ObjecTime Developer 5.2 compatible behavior is available by making RTFRAME_USE_FREELIST set to 0.

Notes:

- Also provides mutual exclusion when dealing with Actor references. If disabled then RTFRAME_CHECKING should be enabled.
- 2*n* extra pointers for both optional and imported Actor references. Access time to a free slot is constant.

Example:

```
// Maintain a free list in RTActorRefs for optional components?
// The free list costs two pointers per replication but avoids
// a linear search in incarnate and import operations.
// Pre-5.2 compatible behavior is available by making
// RTFRAME_USE_FREELIST zero.
#ifndef RTFRAME_USE_FREELIST
#define RTFRAME_USE_FREELIST 1
#endif
```

RTMESSAGE_PAYLOAD_SIZE

Default value: 100.

Possible value: 0..N (where N is the size of the largest data object that can be copied to the payload).

Description: This defines the size of the area in RTMessage where small objects are copied for better performance.

OBSERVABLE

Default value: 1.

Possible value: 1 or 0.

Description: This controls availability of Target Observability.

Creating the Minimum Services Library Configuration

Configuring the Services Library with the minimum services allows you to most often reduce the size and/or increase the speed of the resulting Rose RealTime model using the library.

To create the minimum configuration, the values described below should be defined to the values in the Minimum Configuration column. This is not the only minimum configuration, you are free to configure the Services Library to fit your project needs.

Table 3 Definitions for Minimum Services Library Configuration

Definition	Default	Minimum Configuration
LOG_MESSAGE	1	0
OBJECT_DECODE	1	0
OTRTSDEBUG	DEBUG_VERBOSE	DEBUG_NONE
INLINE_CHAINS	<blank>	inline

Note: Disabling the LOG_MESSAGE definition will turn off the logging capability of the debugger.

Integrating External IPC Into a Model

Often applications are required to communicate with other applications. There are several communication mechanisms that exist for this purpose.

For communication between applications on different hosts, you can use:

- sockets
- remote procedure calls
- named pipes

For communication between applications on the same host, you can use:

- message queues
- shared memory
- pipes
- streams
- interrupts

This section describes the different options available for integrating external IPC mechanisms within a model using the C++ Services Library. In this module you will:

- Understand the differences between using IPC in single-threaded and multi-threaded models.
- Understand the Custom Controller capabilities and how to use the Custom Controller in a model.
- Understand the pros and cons of the different options that are available for integrating an IPC mechanism into a model.

The C++ model examples show how to use what is discussed in this chapter to integrate IPC with a model. The examples include a callback mechanism, use of sockets, and implementation of a Interrupt Service Routine. The example models are meant to be simple, so that you can easily understand them, and at the same time they provide enough details to allow you to expand them for your own use.

See the Model Examples for detail descriptions of the sockets, callbacks, and ISR C++ models examples.

Build Versus Buy

If you require more than a simple IPC mechanism, then you should consider Rational Connexis. Connexis works together with Rational Rose RealTime to let you model and build distributed Rose RealTime applications. Built-in middleware provides an off-the-shelf communication infrastructure that solves many of the challenges common to distributed applications including object-to-object connectivity, fault tolerance, name lookup service, reliability and performance. Capsules continue to communicate with each other in the same way as with Rose RealTime—by sending messages to ports. However, the receiving capsule can be in another process, or even on another processor. For more information on Rational Connexis see the User's Guide, Rational Rose RealTime Connexis.

IPC Basics

An application can wait for an IPC event to occur on a specific resource by:

- Blocking on a system call.
- Continuously polling a resource.

The blocking method is the preferred approach because it leaves the application in an idle state until the event occurs. The polling method uses more processor cycles.

Single-Threaded IPC

In a single-threaded model, the system cannot block while waiting for an external event. A flag/resource must be polled at some regular interval.

Using Signal Handlers

On a POSIX compliant system that supports registration and detection of Unix-style signals. Normally interrupt handlers would be separate functions that would notify the capsule when the interrupt occurred (through a semaphore, for example).

For Windows NT, a console control handler could be installed to intercept signals (such as <Ctrl>+<C>) from a console. Windows NT events cannot be used in a single-threaded situation as blocking is required.

Polling a Flag

The flag/resource that is set can be checked in one of two ways within a Rose RealTime model:

1. The flag status could be checked each time through the main processing loop.

   This is done by modifying the **mainLoop**() function on the RTSoleController capsule class. This mechanism involves making the following changes to the C++ Services Library source code:

   - the `mainLoop()` function must be modified to define the global flag/resource variable that will be set when an event is detected
   - to define a port for communicating the arrival of the signal to the Rose RealTime model
   - to check the flag/resource each time through the main processing loop.

2. Create a capsule that is on a timing loop that checks the flag at regular intervals. This is done by using the `informEvery()` call on the timing service. The interval at which the flag is checked is easily changed by changing the timer interval.

Multi-Threaded IPC

Using IPC mechanisms in a multi-threaded environment is more flexible than in a single-threaded environment. Using a dedicated thread to isolate the blocking on a resource can be a solution to not have the entire application block. With Rose RealTime, the same solution is available. However, some additional constraints should be considered.

Dedicated Blocking Capsule

Calling a blocking function in capsule detail code is equivalent to blocking in user code of existing applications (that is, the whole thread is blocked).

This means that:

- All capsules on the thread are also blocked.
- Outstanding messages on the thread cannot be delivered.
- Messages from capsules on other threads cannot be processed.
- Includes internal messages such as the Frame Service destroy signal.

Although this behavior might be sufficient for some applications, there is a mechanism within Rose RealTime that allows a thread to block on a user resource without encountering the above issues.

Processing Overhead

The amount of processing overhead incurred by having a dedicated blocking capsule should be relatively small. This is because the signal handling threads should be idle for the most part since they are just waiting for events to occur.

Custom Peer Controller

This section describes an enhancement to the Services Library `RTPeerController` that facilitates building application-specific inter process communication (IPC) channels. These channels can be used to communicate between a number of different entities: interrupt handlers, threads, processes (separate memory space) or even processors.

The Customer Controller allows functions on capsules to be bound to the control flow of the Peer Controller so that you don’t have to override the `mainloop()` function in the Services Library but simply add mainloop behavior from within a capsule. The controller is responsible for message processing on a physical thread. Thus, there will be one controller instance for each physical thread in your model.

Note: The Peer Controller class provides all of the messaging support that a Rose RealTime thread requires.

If a capsule registers selected operations with the controller, they will be called instead of the normal `wakeup()` and `waitForEvents()` controller operations. This would allow a capsule to integrate its IPC blocking behavior into the controller.

Note: This is only applicable to the multi-threaded Service Library implementation.

Enhancement to the RTPeerController Class

The enhancement to the Peer Controller class is called the Custom Peer Controller. The Custom Peer Controller allows functions on capsules to be bound into the control flow of the Peer Controller. Controller classes, from which the Peer Controller is derived, are responsible for message delivery and processing in an executing model. During execution, there is one instance of an `RTController` subclass for each physical thread. When a new thread is spawned, the thread executes the mainloop of its associated controller. This main loop checks its queues for outstanding messages, and delivers messages to capsule instances.

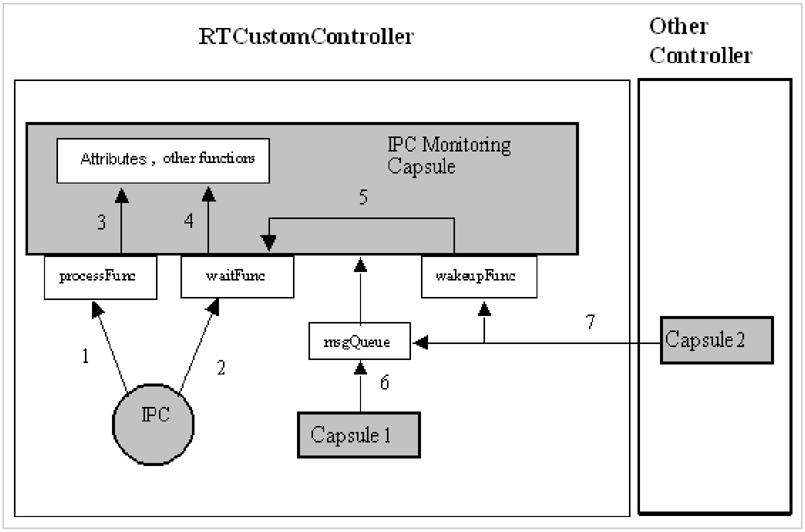
The Peer Controller encapsulates all aspects of inter-task communication in a multi-threaded environment.

- Wait for events (when there is nothing to deliver, wait until an event occurs. Usually this would be the arrival of a message from another thread.)
- Wakeup peer controller (when a message is delivered from a controller on another thread, the other controller will call this operation to wakeup the receiving controller.)
- Handle high priority message source (for example, interrupts).

These functions facilitate building/integrating various IPC mechanisms.

Adding Support For New IPC Mechanisms

Figure 18 Information Flow for the Custom Controller Based IPC Mechanism



The capabilities of the custom IPC integration approach are represented by numbers in Figure 18:

1 Ability to treat an IPC message source with a higher priority than capsule messages, retrieves IPC messages and is called before dispatching messages.

2 IPC messages can be treated with a lower priority than any capsule messages if they are retrieved inside the waitFunc (such is the case in the Socket example).

3 The Monitoring Capsule's attributes and other associated functions are accessible when IPC data is processed.

4 Same as above.

5 This represents the ability of the Monitoring Capsule to block while simultaneously waiting for IPC data and messages.

6 It is possible for the Monitoring Capsule to receive messages from capsules situated on the same thread.

7 It is possible for the Monitoring Capsule to receive messages from capsules situated on different threads.

Design Components

Each thread in a Rose RealTime model executes the `mainLoop()` function of a `RTController` subclass. The `RTPeerController` enables the integration of application-specific IPC messages and capsule messages. A `RTPeerController` subclass was added to the Services Library: the `RTCustomController`.

An application capsule designated to send/receive information over a specific IPC, can be incarnated on a thread which executes `RTCustomController::mainLoop()`. The new type of controller allows the override of `RTPeerController::wakeup()`, and `RTPeerController::waitForEvents()`. An IPC Monitoring Capsule could then register new `_wakeupFunc` and `_waitFunc` functions to service internal and external (IPC) messages at the same time.

The `RTCutomController` also provides the ability to process IPC messages at a higher priority than inter-capsule messages.

Most of the structure of the `RTCustomController::mainLoop()` is identical to that of the `RTPeerController::mainLoop()`. If present, the function `_processFunc()` is called before dispatching messages and so it is conferred a higher priority. Special attention should be given to the design of this function considering that it is frequently called.

The override mechanism is based on pointers to capsule class operations that take no arguments and return no value. There are no restrictions on the attributes of these functions (public, private, ...).

To see the definition of the custom controller, see the header file $RTS_HOME/include/RTCustomController.h.

The custom controller `registerLayer` operation allows capsules to bind in to the mainloop a custom `waitForEvents`, wakeup, and a high priority processing function. Each time `registerLayer` is called (using the macro `REGISTER_LAYER` for convenience), the previously registered pointer values are overridden. If any of the (operation pointers) macro arguments are null values (or the macro is not called at all), the following defaults apply:

```
if ( waitFunc == (RTActorFunction *)0 ) then
   RTPeerController::waitForEvents() called;
if( wakeupFunc == (RTActorFunction *)0 ) then
   RTPeerController::wakeup() called;
if( processFunc == (RTActorFunction *)0 ) then
   no special processing function called;
```

For example, in order to **register** only a high priority IPC processing function named 'processIPC', the macro call to be used is:

```
REGISTER_LAYER( 0, 0, processIPC );
```

Note: Always ensure to de-register operations when the capsule that had registered its operations is destroyed. If not, the controller will try to call operations that no longer exist, this will cause a run-time exception.

To **de-register** operations, use the following macro call:

```
REGISTER_LAYER( 0, 0, 0);
```

Concurrency Note

`RTCustomController::wakeup()` can only be called from other threads, whereas `RTCustomController::waitForEvents()` is called only from the Custom Peer Controller's thread.

Controller Usage

The main actions a designer needs to take in order to use the `RTCustomController` are as follows:

1 Create a physical thread from within the **Component Specification** dialog box. Set the **Implementation Class** in the physical thread specification dialog to **RTCustomController**.

2 Create a capsule dedicated to monitoring a custom IPC channel.

3 Incarnate this capsule on a **RTCustomController** thread.

4 Determine (create) an IPC channel.

Depending on the nature of the IPC and the desired system performance, write functions to:

- block the current thread while waiting for IPC data.
- unblock the thread waiting for IPC data when capsule messages are received.
- process IPC data (read / write, ...).

Usability Note

These functions are invoked at run-time by `RTCustomController::mainloop()` and not via `RTMessage::deliver()`. Therefore, the variable '`msg`' cannot be used. The `RTCustomController::wakeup()` function is called from another thread; therefore, the registered '`wakeupFunc`' body should not access capsule attributes and functions directly.

Register needed functions with the Custom Peer Controller using the `REGISTER_LAYER macro` (defaults are provided when any of the above functions is not registered).

Before the capsule that registered operations with the customer controller is destroyed, you must free or return any IPC resources allocated by the capsule for any communication it provided.

IPC Options Summary

It is very important to prototype a couple of IPC approaches to determine which one best meets your project requirements. Although the Custom Controller allows the best integration with the C++ Services Library, any other IPC option can be used.

Table 4 IPC Options

IPC Option	Advantage	Disadvantage
Polling in capsule using a timer.	▪ Simple. ▪ All code in capsule. ▪ Can be used for single-threaded targets.	▪ No guarantee of timely processing of IPC events. ▪ Wasted processor cycles.
Polling using mainloop modifications.	▪ Can be used for single-threaded targets. ▪ Can handle high-priority events.	▪ Starvation of event detection when controller is blocked. ▪ Design coupled to Service Library code. ▪ Need to manage code modifications outside of model. ▪ Wasted processor cycles.
Dedicated capsule blocking on a thread.	▪ Simple. ▪ All code in capsule.	▪ All capsules on the thread are also blocked. ▪ Outstanding messages on the thread cannot be delivered. ▪ Messages from capsules on other threads cannot be processed. ▪ Only one capsule can execute on the thread, it is fairly costly on resources to have a dedicated thread simply for monitoring.

Table 4 IPC Options (continued)

IPC Option	Advantage	Disadvantage
Custom Controller.	▪ Several capsules can run on the same thread. ▪ No extra context switches due to monitoring internal and external messages. ▪ Internal and external messages can be processed.	▪ Cannot be used with single-threaded targets. ▪ Requires implementation of internal/external synchronization mechanism - more source code to write.

For details of the sockets, callbacks, and ISR C++ models examples that use the Custom Controller, see the Model Examples.

Optimizing Designs

Performance is usually a significant consideration in any real-world design. This section provides some guidelines for improving the performance of your Services Library-based models in the following areas:

- *Capsule Instances and Capsule Behavior* on page 160
- *General C++ Performance Notes* on page 163
- *Additional Design Considerations* on page 164
- *Toolchains* on page 165

Capsule Instances and Capsule Behavior

Incarnation (Frame::Base::incarnate())

Problem

Instantiating capsules incurs some run-time performance overhead. The processing time required to instantiate a capsule depends on the number of capsule instances it recursively contains, the number and replication factor of bindings it contains, and the number and types of extended state variables it contains.

Recommendation

Importation is a much faster approach. If possible pre-allocate the capsule and import it into the context where it is used.

Guards

Problem

Guard conditions can incur significantly more performance overhead than choice points. A guard condition has an associated function, which is called each time the trigger event is evaluated. Because many events may be evaluated before the transitions are executed, placing guard conditions on triggers will cause the guard functions to be called for every message delivery, regardless of whether the associated transition is being fired. Event triggers are evaluated until a matching event is found. At that point, evaluation of events stops. The order in which event triggers on a given state are evaluated is arbitrary.

Recommendation

Do not use guards unless absolutely necessary.

State Machines

Problem

State machines are traversed from an innermost state to an outermost state when searching for transition triggers, which match the current event. This means that if a transition is placed on an “outer” state boundary, and that transition fires frequently while the capsule is in an “inner” state, many other transition triggers may be evaluated before the correct one is found.

Recommendation

Place frequently executed transitions on leaf states.

Capsules versus Data

Problem

Capsules and message sending have more overhead (both processing and memory) than simple data objects. You must decide at what point in your design the use of simple objects with no state machine to achieve performance becomes more important than the abstractions provided by capsules.

Recommendation

Capsules with minimal state machines and few ports may be converted to data classes.

Capsule Functions

Problem

Capsule functions can be inlined. Inline functions eliminate the overhead of a function call, but they may also increase the memory footprint of the executable.

Recommendation

Frequently called capsule functions may be declared as inline to increase speed.

RTDataObjects

Problem

Invoking constructors or assignment operators on `RTDataObjects` causes new objects to be allocated.

Recommendation

Using basic C++ data types (such as `int`) for variables will in most instances be more efficient (in both time and memory utilization) than `RTDataObjects`.

Unnecessary Sends

Problem

Broadcasting on replicated ports involves a send on every replication.

Recommendation

If you have a replicated port with only a few known connections, a send on only the connected instances may be much quicker than a broadcast.

Sending Data by Value in Messages

Problem

When an object is sent by value in a message, the object is deep copied before being sent. For large objects, this operation involves several new allocations and memory copies.

Recommendation

For best performance when sending between capsules within the same model (that is, not across process boundaries), you should consider sending pointers instead of objects. This will introduce more complexity into the design and coding (with respect to memory management), but is more efficient for performance. In particular, if a few messaging interactions are identified as happening very frequently, these interactions could be optimized to send pointers rather than objects.

Cross Thread Message Sending

Problem

Message sends across thread boundaries involve more overhead than message sends within the same thread.

Recommendation

This should be taken into consideration when determining the allocation of capsules to threads. Lower latency is achieved between two capsules on the same thread, than is obtained with two capsules on different threads.

Note: When using threads, time-ordering of messages is not preserved. That is, if you send messages to a capsule within the same thread and a capsule on another thread, subsequent messages within the same thread may be processed before the context switch occurs to allow the other thread to begin processing its messages.

General C++ Performance Notes

Problem

File input and output functions (`printf`, `scanf`, <<, >>, etc.) are quite expensive (about 100 x function call overhead)

Recommendation

In performance-critical software these IO functions should only be used in exceptional circumstances, or as part of optional debugging code (calls that can be avoided). You may also consider using a low priority logging thread to do the IO when the system is idle.

Problem

Dynamic creation and destruction of objects (new and delete), particularly of large composite objects (for example, composite capsules with fixed replicated capsule roles) is expensive (relative to a function call).

Recommendation

Do not dynamically create objects on the critical data path. Pre-allocation and application level management of objects can provide a substantial performance gain.

Additional Design Considerations

This section has probably just whetted your appetite for other ideas that will help solve your particular integration problem. As food for thought, an initial checklist of design areas to consider is provided. Many of these areas may not be critical to your application but all have been proven to be important in at least one project using Rose RealTime. Complete discussion of these topics is beyond the scope of this document.

Hardware Differences

In many cases, a key difference between the application running on a workstation-based Services Library and a RTOS-based Services Library is the presence of special hardware in the RTOS case. Before just stubbing out non-existent hardware functionality, it is important to understand its impact on the overall execution of the model in terms of the range of functionality that can be tested.

For example, real-time platforms often have integrated Non-Volatile-Store (NVS). While it is easy to stub out this behavior on the workstation (for example, use RAM) this eliminates a whole range of recovery/restart functionality. A better "stub" would be to simulate the NVS using the UNIX file system thereby allowing the full model to be tested on the workstation.

The key point here is to always consider hardware availability when using Rose RealTime so as to take full advantage of the ease of moving a model from one platform to another. It is often the case that there are more developers than there is hardware available for testing.

Availability of External Library on Different Platforms

Sometimes, for whatever technical reasons, an external library cannot be integrated with the workstation-based Services Library. In this case, the option of integrating external libraries only with the Services Library should be considered. In many cases, this allows all the capabilities of the underlying OS to be utilized and this is important when the goal is to use the library unmodified. In the cases where the library is available only as a binary (for example, CORBA ORB) this may be the only alternative.

You can use a hybrid approach to building your model, where a small portion of the model runs only on the target, and performs the interfacing to the external library, while the rest of the model executes on the workstation. Proper thought must be put into how the external library interface can be encapsulated within the target-based portion of the model, so that the rest of the model can run independently on the workstation.

Toolchains

As a project moves through its life-cycle, it is important that any conflicts that may arise from the integration of external libraries be discovered as soon as possible. It is recommended that regular builds be done for the workstation, Services Library on the workstation, and for the Services Library on RTOS so that even if the actual target board or processor is not available, the compilation and linking step can be exercised.

Configuring and Customizing the Services Library 10

Contents

This chapter is organized as follows:

- *Configuration and Customization* on page 167
- *Building the Services Library* on page 175
- *Updating a Component to Use a Different Services Library* on page 176

Configuration and Customization

This chapter discusses the different ways that are available for configuring and customizing the C++ Services Library.

The difference between configuring and customizing is that configuring modifies pre-defined parameters built-in to the Services Library to increase speed, or reduce the size of your model. Whereas with customization you are changing the behavior of the Services Library by adding source files or overriding existing operations.

There are several different ways of changing the functionality of the Services Library:

Configuration options

- *Changing Pre-Processor Macros* on page 168

  This is useful for optimizing the library for speed or size. The library must be rebuilt, as well as your model.

- *Changing Build Options* on page 170

  This is useful for rebuilding the library with different build options, for example to turn on or off compiler optimizations or add debug information to the library. The library must be rebuilt, as well as your model.

Customization options

- *Overriding Virtual RTActor Operations From the Toolset* on page 171

  This is useful to change behavior of certain capsules without the need to modify the Services Library files. The model must be rebuilt.

- *Overriding or Adding Operations and Classes* on page 172

  You can override any Services Library operation. This is most often used to change the way the library is initialized, to modify the main processing loop, or to add platform specific implementations.

Changing Pre-Processor Macros

Before you start

Ensure that you understand the following concepts:

- *Organization of the Services Library Source* on page 141
- *Configuration Naming Convention* on page 141
- *Directory Structure* on page 143

Why

Modify pre-defined parameters built-in to the Services Library. This is often useful for configuring the library for optimal speed or size.

Where

The file $RTS_HOME/include/RTConfig.h contains all Configuration preprocessor definitions, or pre-processor macros with their default values. You can override any of these macros by adding a definition in one of these files:

- To change for a specific processor and compiler: $RTS_HOME/libset/<libset name>/RTLibSet.h.

- To change for all libraries for an operating system: $RTS_HOME/target/<target name>/RTTarget.h.

It is usually preferable to perform a libset configuration (that is, to reconfigure only for a specific processor and compiler).

How

Follow these steps to reconfigure the Services Library, to build, and to update your model to use the new library.

In this example, we will create a new libset to localize the changes to a compiler. To make changes at the target level follow the same steps but create a new target instead of a new libset.

In this example, assume that the current platform is: `SUN5T.sparc-gnu-2.8.1`.

1 Select a name for the new libset.

   Usually you can just append to the existing libset name. In this example, we will name the new libset, `sparc-gnu-2.8.1-minimal`.

2 Create a new directory, $RTS_HOME/libset/sparc-gnu-2.8.1-minimal.

3 Create a new directory, $RTS_HOME/config/SUN5T.sparc-gnu-2.8.1-minimal.

4 Copy all the files from the original libset and config directories to the new directories:

   - From $RTS_HOME/libset/sparc-gnu-2.8.1 to $RTS_HOME/libset/sparc-gnu-2.8.1-minimal.
   - From $RTS_HOME/config/SUN5T.sparc-gnu-2.8.1 to $RTS_HOME/config/SUN5T.sparc-gnu-2.8.1-minimal.

5 In the new libset directory, add pre-processor statements to `RTLibSet.h`, and save the file.

   For example, to turn off logging messages you would add: `#define LOG_MESSAGE 0.`

6 Build the new Services Library.

   For details, see *Building the Services Library* on page 175.

7 Update components in the model to use the new Services Library.

   For details, see *Updating a Component to Use a Different Services Library* on page 176.

Changing Build Options

Before you start

Ensure that you understand the following concepts:

- *Organization of the Services Library Source* on page 141
- *Configuration Naming Convention* on page 141
- *Directory Structure* on page 143

Why

This is useful for rebuilding the library with different build options, for example to turn on or off compiler optimizations or add debug information to the library.

Where

The build options used to compile both the Services Library and the model can be found in these makefiles:

- To change for a specific processor and compiler: $RTS_HOME/libset/<libset name>/libset.mk.
- To change for all libraries for an operating system: $RTS_HOME/target/<target name>/target.mk.
- To change for a specific platform: $RTS_HOME/config/<platform name>/config.mk.

It is usually preferable to perform a libset configuration, that is to reconfigure only for a specific processor and compiler.

How

Follow these steps to build a Services Library with debug symbols.

In this example, we will create a new libset to localize the changes to the compiler. To make changes at the target level follow the same steps but create a new target instead of a new libset.

For this example, we will assume that our current platform is:
`SUN5T.sparc-gnu-2.8.1.`

1 Select a name for the new libset.

Usually you can just append to the existing libset name. In this example, we will name the new libset: `sparc-gnu-2.8.1-debug`.

2 Create a new directory, $RTS_HOME/libset/sparc-gnu-2.8.1-debug.

3. Create a new directory, $RTS_HOME/config/SUN5T.sparc-gnu-2.8.1-debug.
4. Copy all the files from the original libset and config directories to the new directories:
   - From $RTS_HOME/libset/sparc-gnu-2.8.1 to $RTS_HOME/libset/sparc-gnu-2.8.1-debug.
   - From $RTS_HOME/config/SUN5T.sparc-gnu-2.8.1 to $RTS_HOME/config/SUN5T.sparc-gnu-2.8.1-debug.
5. In the new libset directory, open the libset.mk file and change the **-04** flag from **LIBSETCCEXTRA** and replace with **-g**. **LIBSETCCEXTRA** should now look like:

   ```
   LIBSETCCEXTRA=-g -finline -finline-functions -fno-builtin \
                  -Wall -Winline -Wwrite-strings
   ```
6. Build the new Services Library.

   For details, see *Building the Services Library* on page 175.
7. Update components in the model to use the new Services Library.

   For details, see *Updating a Component to Use a Different Services Library* on page 176.

Now you have a Services Library with debug information. You can use your source level debugger to step through the code.

Overriding Virtual RTActor Operations From the Toolset

Why

Some operations defined on the capsule base class RTActor can be overridden from within the toolset. The advantage being that you do not have to modify Services Library files, and you can scope the changes to specific components.

Where

The following operations can be overridden:

- void RTActor::unexpectedMessage(void);

  This operation is called if a message is received that can't be handled by the capsule given its current state.
- void RTActor::logMsg(void);

  This operation is called before each message is delivered to a capsule.

How

To override one of these operations in an ANSI compliant way, the header code needs to be commented out. Use the HeaderPreface and the HeaderEnding of the "override" class to place code at the beginning and end of the header file.

By placing "`#if 0 // Declarations disabled for ANSI compliance.`" in the HeaderPreface and "`#endif // Declarations disabled for ANSI compliance.`" in the HeaderEnding, the following ANSI compliant code is generated.

```
#if 0 // Declarations disabled for ANSI compliance.

void RTActor::unexpectedMessage( void );

#endif // Declarations disabled for ANSI compliance.
```

Overriding or Adding Operations and Classes

Why

You can override any Services Library operation. This is most often used to change the way the Service Library is initialized, to modify the main processing loop, or to add platform specific implementations.

Where

There are two ways of modifying operations and classes:

1. Override at the model level:
   - Changes are local to the model.
   - No need to recompile the Services Library.
   - Changes can easily be added to source control with the model.
   - Packages which contain the overrides can be shared.
2. Override at the target level:
   - Everyone can easily access the changes.
   - You have to rebuild the Services Library.
   - Changes are added to source control outside of the toolset.

The most interesting operations that can be candidates for overriding are the following:

- `RTMain::targetStartup/RTMain::targetShutdown`

  These operations are typically overridden to initialize/cleanup drivers specific to the target environment, startup OS services (such as clock or timings etc.), initialize specific libraries or structures that are needed by the Services Library, or initialize signal handlers.

- `RTController::mainloop(),RTPeerController::mainloop()`

  This operation is typically overridden if you want a message handling strategy that is different than the default. For example, you could perform regular sanity checks or audits, or receive message from other applications.

Note: If you override the **mainloop** on the controller class all controllers incarnated from that class will have the same overrides. This may or not be desirable.

Because of the way the Services Library is organized you can override any operation.

How

Overriding Operations Within a Model on page 173

Overriding Operations by Creating a New Target on page 174

Overriding Operations Within a Model

In general you can override any operation defined within the Services Library by creating an alternate implementation and linking it into your model. For example, to override the `RTMain::targetStartup()` operation you would:

1. Create a class named as you like, deselect the **C++** property **GenerateClass**. Set the **C++ Target RTS** property **GenerateDescriptor** to **False**.
2. Create an operation, `RTMain::targetStartup`, that returns void and has no arguments.
3. Add code to the operation.
4. Set **OperationKind** to **global**.
5. Drag the class onto a component for which you use to build your model and rebuild the component.

6 The operation you created will override the one defined in the Services Library, and in this case the code you added to `targetStartup` will be called before the Services Library initializes the model.

You can repeat this same process to override any operation in the Services Library.

Overriding Operations by Creating a New Target

In general any operation in the Services Library can be overridden by placing an override version of the operation into the following subdirectory:

$RTS_HOME/src/target/<target name>/<class name>/

The target base directory mirrors the $RTS_HOME/src directory, thus must have a directory for each class. When the library is built the directories in $RTS_HOME/src/target/<target name> are searched first then $RTS_HOME/src. For more informations, see *Organization of the Services Library Source* on page 141.

Tasks

In this example, you will override the `RTActor::logMsg()` operations by creating a new target configuration. You can also override for an existing target configuration but you won’t be able to easily go back and forth between the original libraries and the customized versions.

For this example, assume that the current platform is: `SUN5T.sparc-gnu-2.8.1.`

1 Select a name for the new target.

  Usually you can just append to the existing libset name. In this example, you will name the new libset: `SUN5NEWT`.

2 Create a new directory, $RTS_HOME/target/SUN5NEWT.

3 Create a new directory, $RTS_HOME/config/SUN5NEWT.sparc-gnu-2.8.1.

4 Create a new directory, $RTS_HOME/src/target/SUN5NEW/RTActor.

5 Copy all the files and subdirectories from the original target and config directories to the new directories:

  - From $RTS_HOME/target/SUN5T to $RTS_HOME/target/SUN5NEWT.
  - From $RTS_HOME/config/SUN5T.sparc-gnu-2.8.1 to $RTS_HOME/config/SUN5NEWT.sparc-gnu-2.8.1.
  - From $RTS_HOME/src/target/SUN5 to $RTS_HOME/src/target/SUN5NEW.

6 Edit config/setup.pl

   $target - base='SUN5NEW, SUN5

7 Copy the file that contains the `logMsg()` operation from the generic source directory to the new target source directory:

   - From `$RTS_HOME/src/RTActor/logMsg.cc` to $RTS_HOME/src/target/SUN5NEW/RTActor/logMsg.cc.

8 Edit $RTS_HOME/src/target/SUN5NEW/RTActor/logMsg.cc.

9 Build the new Services Library.

   For details, see *Building the Services Library* on page 175."

10 Update components in model to use the new Services Library.

   For details, see *Updating a Component to Use a Different Services Library* on page 176.

Building the Services Library

When you create a new libset or target, you have to build the Services Library to include the modifications that you have made. The Services Library is always built from the $RTS_HOME/src directory and the target for the make utility is the Platform name (or configuration) (`<target name>.<libset name>`).

Assuming we are using a custom configured Diab C++ compiler version 4.2b for the Motorola PowerPC platform, the name of our reconfigured platform is PSOS2T.ppc603-Diab-4.2b-Debug.

To build this Services Library perform the following commands:

UNIX:

```
cd $ROSERT_HOME/C++/TargetRTS/src

make CONFIG=PSOS2T.ppc603-Diab-4.2b-Debug
```

Windows:

```
cd %ROSERT_HOME%\C++\TargetRTS\src

nmake CONFIG=PSOS2T.ppc603-Diab-4.2b-Debug
```

Note: You may encounter instances when you should not use **nmake**. For example, if you compile for Tornado on Windows NT, you should use **make** instead of **nmake**.

After the Services Library has been re-built, you must rebuild your Rose RealTime models to link against the new Services Library libraries. For details, see *Updating a Component to Use a Different Services Library* on page 176.

Note: If your new Services Library changed the debugging, logging, or target observability functionality, visibility into the model is removed. Debugging the resulting model via the toolset is no longer possible.

Updating a Component to Use a Different Services Library

After building a new Services Library, you must ensure that your components reference the new library.

1. Open the **Component Specification** dialog box.
2. On the **C++ Compilation** tab, press the **Select...** button.
3. A list of built libraries found in the current Services Library root ($RTS_HOME) directory are listed. If your library built properly it should be listed. Select it and click **OK**.
4. Rebuild your model.

Model Properties Reference

11

Contents

This chapter is organized as follows:

- *Generalization and Properties* on page 177
- *Expanded Property Symbols* on page 178
- *C++ Model Element Properties* on page 179
- *C++ TargetRTS Properties* on page 195
- *C++ Generation Properties* on page 215
- *C++ Compilation Properties* on page 218
- *C++ Executable Properties* on page 223
- *C++ Library Properties* on page 227
- *C++ External Library Properties* on page 229

Overview

Using the C++ code generator, you can produce C++ source code from the information contained in a model. The code generated for each selected model component is a function of that component's specification, and C++ Language Add-in model properties. The model properties provide the language-specific information required to map your model on to C++.

To facilitate the management of C++ code generation properties, use the property set mechanism. This mechanism establishes settings for each of the properties associated with a model element type. This allows you to create your own property sets, each new set having its own default values for any of the properties.

Generalization and Properties

Custom properties that are added to a model element, for example code generation properties, are not inherited when two model elements participate in a generalization relationship. For example, if class A is the parent and B the child, and class A has overridden the default value of the **Class::C++::ClassKind** property to **typedef**, this property in class B will remain set to the default. For this reason it is important that you use property sets to define default values that can be re-used in different model elements.

Expanded Property Symbols

When the C++ code generator parses the properties, it expands a set of pre-defined symbols. To delimit these symbols within a composite property string, use curly braces '{' and '}'. For example, the **Component::C++ Generation::OutputDirectory** property is defined as:

```
$@/${name}
```

If the component name is **Component1**, and the .rtmdl file is saved in /home/projects/, this property will be expanded by the code generator to:

```
/home/projects/Component1
```

The following symbols are recognized by the code generator and are expanded as defined below:

If you enter:	**Gets expanded to:**
${name}** or **$name	The name of the model element on which the property is defined.
$@	The full directory path to where the owning model file is saved. The model file name is not included when the symbol is expanded.
$defaultMakeCommand	On Windows expands to **nmake** and on all others to **make**.
$(MACRO)	**$(MACRO)** This may be useful in some Makefile fields so that Make can expand MACRO.
$$	**$** (a single dollar sign) This may be useful for some Makefile fields such as CodeGenMakeInsert or CompileCommand.
$VARIABLE	This is expanded to whatever the toolset's Path Map is defined for VARIABLE. If no such Path Map variable exists, this is evaluated to nothing.

Environment Variables and Pathmap Symbols

You can use environment variables and pathmap symbols in property fields. Environment variables are not interpreted by the code generator, instead they are passed as is into the generated files. Naturally, environment variables don't make sense in .cpp and .h files, however they do in makefiles. For this reason we encourage that environment variables be primarily used with components. For example, it is very common to define inclusion paths as an environment variable instead of a hard-coded value.

Pathmap symbols are expanded by the code generator into the generated source files. Use these to avoid having to hard code paths information into a component.

Note: The Rational Rose RealTime pathmap symbol is $ not $&. Pathmap symbols are only expanded when building a model from within the toolset. If you are building from the command line, you must ensure that equivalent environment variables exist for each pathmap symbol.

The following properties are usually defined using environment variables or pathmap symbols:

- *InclusionPaths (Component, C++ Compilation)* on page 221
- *TargetServicesLibrary (Component, C++ Compilation)* on page 222
- *UserLibraries (Component, C++ Executable)* on page 226
- *UserObjectFiles (Component, C++ Executable)* on page 227
- *InclusionPaths (Component, C++ External Library)* on page 230
- *Libraries (Component, C++ External Library)* on page 230

C++ Model Element Properties

This group of model properties is used to control the general aspects of the C++ language. For example, several C++ properties applying to classes are used to control the generation of operations, and class kinds. This page contains a summary of the C++ properties grouped by model element to which they are associated.

Class

- *GenerateClass (Class, C++)* on page 181
- *ClassKind (Class, C++)* on page 181
- *ImplementationType (Class, C++)* on page 182
- *HeaderPreface (Class, C++)* on page 182
- *HeaderEnding (Class, C++)* on page 182
- *ImplementationPreface (Class, C++)* on page 183
- *ImplementationEnding (Class, C++)* on page 183
- *PublicDeclarations (Class, C++)* on page 183
- *ProtectedDeclarations (Class, C++)* on page 183
- *PrivateDeclarations (Class, C++)* on page 183
- *GenerateDefaultConstructor (Class, C++)* on page 184
- *DefaultConstructorVisibility (Class, C++)* on page 184

- *DefaultConstructorExplicit (Class, C++)* on page 184
- *DefaultConstructorInline (Class, C++)* on page 184
- *GenerateCopyConstructor (Class, C++)* on page 184
- *CopyConstructorVisibility (Class, C++)* on page 185
- *CopyConstructorExplicit (Class, C++)* on page 185
- *CopyConstructorInline (Class, C++)* on page 185
- *GenerateDestructor (Class, C++)* on page 185
- *DestructorVisibility (Class, C++)* on page 185
- *DestructorVirtual (Class, C++)* on page 186
- *DestructorInline (Class, C++)* on page 186
- *GenerateAssignmentOperator (Class, C++)* on page 186
- *AssignmentOperatorVisibility (Class, C++)* on page 186
- *AssignmentOperatorInline (Class, C++)* on page 186
- *GenerateEqualityOperator (Class, C++)* on page 187
- *EqualityOperatorsVisibility (Class, C++)* on page 187
- *EqualityOperatorInline (Class, C++)* on page 187
- *GenerateInequalityOperator (Class, C++)* on page 187

Attribute

- *AttributeKind (Attribute, C++)* on page 187
- *InitializerKind (Attribute, C++)* on page 188

Operation

- *OperationKind (Operation, C++)* on page 188
- *Inline (Operation, C++)* on page 188
- *ConstructorInitializer (Operation, C++)* on page 189
- *CallFromDestructor (Operation, C++)* on page 189

Association End

- *AssociationEndKind (Role, C++)* on page 189
- *InitializerKind (Role, C++)* on page 189
- *InitialValue (Role, C++)* on page 190

Capsule

- *HeaderPreface (Capsule, C++)* on page 190
- *HeaderEnding (Capsule, C++)* on page 190
- *ImplementationPreface (Capsule, C++)* on page 190
- *ImplementationEnding (Capsule, C++)* on page 191
- *PublicDeclarations (Capsule, C++)* on page 191
- *ProtectedDeclarations (Capsule, C++)* on page 191
- *PrivateDeclarations (Capsule, C++)* on page 191

Dependency

- *KindInHeader (Uses, C++)* on page 192
- *KindInImplementation (Uses, C++)* on page 192

GenerateClass (Class, C++)

Determines if a class is generated by the code generator. If **GenerateClass** is not checked, the C++ code generator does not generate a definition for this class. This should be used when modeling code that has already been implemented external to the tool, and hence doesn't need to be generated.

For example, it is common to create a class within the toolset which is a placeholder for an external data type. This allows you to specify the data type in a protocol and use it for modeling purposes. If you leave the GenerateDescriptor (Class, C++ TargetRTS) property set, a type descriptor can still be generated even if the class won’t be.

Even if the **GenerateClass** property is not checked you should set the ClassKind (Class, C++) so that the C++ code generator can generate forward references when needed.

ClassKind (Class, C++)

Defines the kind of C++ construct generated for the class element. Possible values are: **class**, **struct**, **union**, **typedef**, **none**.

If **ClassKind = typedef**, the ImplementationType (Class, C++) property is used to specify the type

ClassKind set to **none** is only used for backwards compatibility. If you don't want a class element to be generated use the GenerateClass (Class, C++) property to turn off code generation. The code generator will issue a warning when ClassKind is set to none. When ClassKind is set to **none** the code generator won't be able to create forward references to the class.

Note: If you still have to set ClassKind to none, then you should set the class kind to ClassUtility. A ClassUtility with ClassKind of none will not cause a warning.

ImplementationType (Class, C++)

Provides the type for the typedef when the ClassKind (Class, C++) property is set to **typedef**.

Example:

```
typedef char MyString[30];
```

Would be generated by creating a class named MyString, setting the **ClassKind** to typedef, and setting the **ImplementationType** to char[30].

HeaderPreface (Class, C++)

Specifies the text that will appear immediately before the declaration of the class in the header file.

This field may also be modified from the generated code and captured into the model using the Code Sync feature. For more information, see Using Code Sync to Change Generate Code in the Toolset Guide.

HeaderEnding (Class, C++)

Specifies the text that will appear immediately after the declaration of the class in the header file.

This field may also be modified from the generated code and captured into the model using the Code Sync feature. For more information, see Using Code Sync to Change Generate Code in the Toolset Guide.

ImplementationPreface (Class, C++)

Specifies the text that will appear immediately before the class implementation.

This field may also be modified from the generated code and captured into the model using the Code Sync feature. For more information, see Using Code Sync to Change Generate Code in the Toolset Guide.

ImplementationEnding (Class, C++)

Specifies the text that will appear immediately after the class implementation.

This field may also be modified from the generated code and captured into the model using the Code Sync feature. For more information, see Using Code Sync to Change Generate Code in the Toolset Guide.

PublicDeclarations (Class, C++)

Specifies text that will appear in a public section in the class.

This field may also be modified from the generated code and captured into the model using the Code Sync feature. For more information, see Using Code Sync to Change Generate Code in the Toolset Guide.

ProtectedDeclarations (Class, C++)

Specifies text that will appear in a protected section of the class.

This field may also be modified from the generated code and captured into the model using the Code Sync feature. For more information, see Using Code Sync to Change Generate Code in the Toolset Guide.

PrivateDeclarations (Class, C++)

Specifies text that will appear in a private section of the class.

This field may also be modified from the generated code and captured into the model using the Code Sync feature. For more information, see Using Code Sync to Change Generate Code in the Toolset Guide.

GenerateDefaultConstructor (Class, C++)

Specifies whether the default constructor will be automatically generated. The generated default constructor simply initializes the attributes of the class with their initial values.

The way in which the class attributes are initialized depends on the kind of attribute (for example, array or other) combined with the value of the **InitializerKind** (Attribute, C++) property. Arrays are initialized using a memberwise assignment loop and other attributes can either be initialized via the constructor initialization list or by assignment.

You can overload this property by creating your own default constructor on the class. In these cases even if the **GenerateDefaultConstructor** is checked, the C++ code generator will notice that one already exists and won't generate another.

Note: When you overload the default constructor on a class, all attributes' initial values will no longer be used during code generation.

DefaultConstructorVisibility (Class, C++)

Specifies the visibility of the generated constructor.

DefaultConstructorExplicit (Class, C++)

Specifies whether the generated constructor will be declared as explicit.

DefaultConstructorInline (Class, C++)

Specifies whether the generated constructor will be declared as inline.

GenerateCopyConstructor (Class, C++)

Specifies whether the copy constructor is automatically generated. The generated copy constructor provides a memberwise construction of each attribute from the source object.

Note: The generated copy constructor copies only pointers and not pointed-to values. If a class has pointers, you should create your own explicit copy constructor.

You can overload this property by creating your own copy constructor on the class. If you create you own copy constructor and the **GenerateCopyConstructor** option is selected, the C++ code generator will realize that a copy constructor already exists, and will not generate another one.

Note: If you declare a private copy constructor, ensure that the **GenerateCopyConstructor** option is not selected. For information on how to declare a private copy constructor, see *Declaring a Private Copy Constructor or Assignment Operator in C++ Classes* on page 192.

CopyConstructorVisibility (Class, C++)

Specifies the visibility of the generated copy constructor.

CopyConstructorExplicit (Class, C++)

Specifies whether the generated copy constructor will be declared as explicit.

CopyConstructorInline (Class, C++)

Specifies whether the generated copy constructor will be declared as inline.

GenerateDestructor (Class, C++)

Specifies whether the destructor will be automatically generated. The generated destructor will be empty unless any operations have the CallFromDestructor (Operation, C++) property checked.

You can overload this property by creating your own destructor on the class. In these cases even if the **GenerateDestructor** is checked, the C++ code generator will notice that one already exists and won’t generate another.

DestructorVisibility (Class, C++)

Specifies the visibility of the generated destructor.

DestructorVirtual (Class, C++)

Specifies whether the generated destructor will be declared as virtual.

DestructorInline (Class, C++)

Specifies whether the generated destructor will be declared as inline.

GenerateAssignmentOperator (Class, C++)

Specifies whether the assignment operator is automatically generated.The generated assignment operator provides a memberwise assignment of each attribute from the source object.

Note: The generated assignment operator copies only pointers, and not pointed-to values. If a class has pointers, you should create your own explicit assignment operator.

You can overload this property by creating your own assignment operator on the class. If you create your own assignment operator and if the **GenerateAssignmentOperator** option is selected, the C++ code generator will realize that an assignment operator already exists and will not generate another one.

Note: If you declare a private assignment operator, ensure that the **GenerateAssignmentOperator** option is not selected. For information on how to declare a private assignment operator, see *Declaring a Private Copy Constructor or Assignment Operator in C++ Classes* on page 192.

AssignmentOperatorVisibility (Class, C++)

Specifies the visibility of the generated assignment operator.

AssignmentOperatorInline (Class, C++)

Specifies whether the generated assignment operator will be declared as inline.

GenerateEqualityOperator (Class, C++)

Specifies whether the equality operator will be automatically generated. The generated equality performs a memberwise equality check.

If you choose not to have Rational Rose RealTime generate these, you can define your own. If both == and != are generated, the **Inline** check box for != is meaningless, as Rose RealTime generates the != as an inline call to the negation of ==.

EqualityOperatorsVisibility (Class, C++)

Specifies the visibility of the generated equality operator.

EqualityOperatorInline (Class, C++)

Specifies whether the generated equality operator will be declared as inline.

GenerateInequalityOperator (Class, C++)

Specifies whether the inequality operator will be automatically generated.

If you choose not to have Rational Rose RealTime generate these, you can define your own. If both == and != are generated, the **Inline** check box for != is meaningless, as Rose RealTime generates the != as an inline call to the negation of ==.

AttributeKind (Attribute, C++)

Specifies whether the attribute is generated as a member of the class, as a global variable defined within the file generated for the class, or as a #define defined within the file generated for the class. Options are **member**, **global**, and **constant**.

If an attribute is set to **global** or **constant** and is to be used in detail level code, attribute array sizes, or other common C++ usages, ensure that there is a dependency added between the class containing the definition and the elements which use the definitions.

InitializerKind (Attribute, C++)

Specifies how the code generator should initialize the attribute. Options are **constructor** or **assignment**. Use this property to configure how the attribute generated for this association end is initialized. When the owner class generates and uses a constructor function, then the constructor will try and initialize its attributes however it can.

If **InitializerKind = assignment** then in the owner's constructor the attribute will be initialized by assignment with the attribute's initial value as defined in the Attribute::General Page::Initial value.

If **InitializerKind = constructor** then the class constructor will initialize the attribute in the initializer list calling the attribute's constructor with the parameters defined in the Attribute::General Page::Initial value.

OperationKind (Operation, C++)

Determines whether the operation is generated as:

- A **member** function of the class
- A **global** function defined in the same file as the class
- A **friend** allowing access to non-public operations.

Note: If an operation is set to **global** and used in detail level code, or other common C++ usages, ensure that there is a dependency added between the class containing the definition and the elements that use the operations. Also ensure that you set the **dependencyKindInHeader** (Uses, C++) property to **inclusion**.

To specify an operation as **member**, **global**, or **friend** using the GUI, open the **Operation Specification** dialog, select the **C++** tab, and select the desired option from the **OperationKind** list.

Inline (Operation, C++)

Specifies whether the inline function specifier is applied to the function. Options are True or False.

ConstructorInitializer (Operation, C++)

Provides the initialization parameters for this operation if it is a constructor. This is used to control the initialization of parent classes and member variables.

Example

```
: _length(9), _angle(4.556)
```

Note: Be sure to add the colon to start the initializer list.

This field may also be modified from the generated code and captured into the model using the Code Sync feature. For more information, see Using Code Sync to Change Generate Code in the Toolset Guide.

CallFromDestructor (Operation, C++)

Options are **True** or **False**. If set to True, this property specifies that the operation should be called from the automatically generated destructor for the class. In order to be called from the destructor, the operation must not have any parameters. More than one operation in a class may be invoked by the destructor. The operations will be invoked in the order in which they are listed in the Operations list.

AssociationEndKind (Role, C++)

Specifies whether the association end is generated as a member of the other end's class or as a global variable defined within the file generated for the association end class. Options are **member** or **global**.

InitializerKind (Role, C++)

Use this property to configure how the attribute generated for this association end is initialized. Possible values are **by assignment** and **call constructor**. When the owner class generates and uses a constructor function, then the constructor will try and initialize its attributes however it can.

If **InitializerKind = by assignment** then in the owners constructor the attribute will be initialized by assignment with the association ends initial value as defined in InitialValue (Role, C++).

If **InitializerKind = call constructor** then the classes constructor will initialize the attribute in the initializer list calling the attributes constructor with the parameters defined in InitialValue (Role, C++).

InitialValue (Role, C++)

Use this property to define an initial value that will be used by the target class constructor when initializing the attribute generated for this association end.

HeaderPreface (Capsule, C++)

Specifies a block of C++ code to be included in the generated code of the capsule class header, just after any generated #include's and just before the generated capsule declarations. Code can include: comments, #define's, #include's, declarations, etc.

This field may also be modified from the generated code and captured into the model using the Code Sync feature. For more information, see Using Code Sync to Change Generate Code in the Toolset Guide.

HeaderEnding (Capsule, C++)

Specifies a block of C++ code to be included at the end of the generated code for the capsule class header. The **HeaderEnding** is generated after the generated capsule declarations.

This field may also be modified from the generated code and captured into the model using the Code Sync feature. For more information, see Using Code Sync to Change Generate Code in the Toolset Guide.

ImplementationPreface (Capsule, C++)

Specifies a block of C++ code to include in the generated code of the capsule class implementation, just after any generated #include's and before the generated capsule definitions. Code can include: comments, #define's, #include's, declarations, etc.

This field may also be modified from the generated code and captured into the model using the Code Sync feature. For more information, see Using Code Sync to Change Generate Code in the Toolset Guide.

ImplementationEnding (Capsule, C++)

Specifies a block of C++ code to be included at the end of the generated code for the capsule class implementation. The ImplementationEnding is generated after the generated capsule definitions.

This field may also be modified from the generated code and captured into the model using the Code Sync feature. For more information, see Using Code Sync to Change Generate Code in the Toolset Guide.

PublicDeclarations (Capsule, C++)

Specifies text that will appear in a public section in the capsule class.

This field may also be modified from the generated code and captured into the model using the Code Sync feature. For more information, see Using Code Sync to Change Generate Code in the *Toolset Guide*.

ProtectedDeclarations (Capsule, C++)

Specifies text that will appear in a protected section of the capsule class.

This field may also be modified from the generated code and captured into the model using the Code Sync feature. For more information, see Using Code Sync to Change Generate Code in the *Toolset Guide*.

PrivateDeclarations (Capsule, C++)

Specifies text that will appear in a private section of the capsule class.

This field may also be modified from the generated code and captured into the model using the Code Sync feature. For more information, see Using Code Sync to Change Generate Code in the *Toolset Guide*.

KindInHeader (Uses, C++)

Specifies the representation of the dependency in the header file of the source class.

The options are:

- Inclusion: include the header file for the target class
- Forward reference: declare a forward reference to the target class
- None: dependency is not generated in header

KindInImplementation (Uses, C++)

Specifies the representation of the dependency in the implementation file of the source class.

The options are

- **Inclusion**: include the header file for the target class
- **Forward reference**: declare a forward reference to the target class
- **None**: dependency is not generated in implementation

Declaring a Private Copy Constructor or Assignment Operator in C++ Classes

If a copy constructor is not specified, C++ compilers provide a default copy constructor and assignment operator for a class. Typically, you will not want to have the copy constructor and assignment operator called for a specific class. To ensure that the copy constructor and assignment operator are not called, declare them as private methods of the class.

For information on declaring a copy constructor or assignment operator as private, see:

- *To declare a private copy constructor:* on page 193
- *To declare a private assignment operator:* on page 194

Note: Although C++ compilers provide a default copy constructor and assignment operator for a class, the implementation for these methods is not mandatory.

To declare a private copy constructor:

1 In a **Class Diagram**, right-click on a class object, and then click **Open Specification**.

2 In the **Class Specification** dialog box, click the **C++** tab.

3 In the **Item Properties** area, scroll down to the bottom and clear the **GenerateCopyConstructor** option.

Note: When the text for a label appears in bold font, this means that the option is overridden from the default setting.

4 Click **Apply**.

5 Click the **C++ TargetRTS** tab.

6 In the **CopyFunctionBody** box, create a new operation for the class and assign it a name and signature for the copy constructor.

For example:

```
target->myAttribute = source->myAttribute;
```

Note: You must add your own operation to override this box because the copy function uses the copy constructor, by default.

7 Click **Apply**.

Next, you want to insert a new operation for the class and assign it a name and signature of the copy constructor.

8 Click the **Operations** tab.

9 Right-click and select **Insert** and press ENTER.

10 Select the new operation, right-click and select **Open Specification**.

11 Cick the **Detail** tab.

12 Right-click in the **Parameters** box and select **Insert**.

13 In the **Name** column, type **source**.

14 Double-click in the **Type** column opposite **source**.

15 In the **Type** box, type the following:

```
const NewClass1 &
```

16 Click **OK**.

17 Click the **General** tab.

18 In the **Visibility** box, select **Private**.

19 Click **Apply**.

20 Click the **C++** tab.

21 In the **Generate** box, select **declaration only**.

22 Click **OK**.

To declare a private assignment operator:

1 In a **Class Diagram**, right-click on a class object, and then click **Open Specification**.

2 Click the **C++** tab.

3 In the **Item Properties** box, scroll down to the bottom and clear the **GenerateAssignmentOperator** option.

Note: When the text for a label appears in bold font, this means that the option is overridden from the default setting.

4 Click **Apply**.

5 Click the **Operations** tab.

Next, you want to insert a new operation for the class and assign it a name and signature for the assignment operator.

6 Right-click and select **Insert** and type the following:

```
operator=
```

7 Select the new operation, right-click and select **Open Specification**.

8 Cick the **Detail** tab.

9 Right-click in the **Parameters** box and select **Insert**.

10 In the **Name** column, type **source**.

11 Double-click in the **Type** column opposite **source**.

12 In the **Type** box, type the following:

```
const NewClass1 &
```

13 Click **OK**.

14 In the **Return Type** box, type the following:

NewClass1 &

15 Click **Apply**.

16 Click the **General** tab.

17 In the **Visibility** box, select **Private**.

18 Click the **C++** tab.

19 In the **Generate** box, select **declaration only**.

20 Click **OK**.

C++ TargetRTS Properties

This group of model properties is used to control the C++ Service Library aspects of the code generation. For example, several C++ Target RTS properties applying to classes are used to control the generation of specialized classes and structures which describe the class to the Services Library. This page contains a summary of the C++ TargetRTS properties grouped by model element to which they are associated.

Class

- *GenerateDescriptor (Class, C++ TargetRTS)* on page 196
- *Version (Class, C++ TargetRTS)* on page 196
- *InitFunctionBody (Class, C++ TargetRTS)* on page 196
- *CopyFunctionBody (Class, C++ TargetRTS)* on page 197
- *DestroyFunctionBody (Class, C++ TargetRTS)* on page 197
- *DecodeFunctionBody (Class, C++ TargetRTS)* on page 197
- *EncodeFunctionBody (Class, C++ TargetRTS)* on page 199

Attribute

- *GenerateDescriptor (Attribute, C++ TargetRTS)* on page 201
- *TypeDescriptor (Attribute, C++ TargetRTS)* on page 201
- *NumElementsFunctionBody (Attribute, C++ TargetRTS)* on page 201

Association End

- *GenerateDescriptor (Role, C++ TargetRTS)* on page 202
- *TypeDescriptor (Role, C++ TargetRTS)* on page 202
- *NumElementsFunctionBody (Role, C++ TargetRTS)* on page 202

Protocol

- *Version (Protocol, C++ TargetRTS)* on page 202
- *BackwardsCompatible (Protocol, C++ TargetRTS)* on page 203
- *TypeSafeSignals (Protocol, C++ TargetRTS)* on page 203

GenerateDescriptor (Class, C++ TargetRTS)

If checked the C++ code generator will create a type descriptor (RTObject_class) for the class. The type descriptor will allow marshalling (encode/decode) of the class.

The type descriptor contains information that the C++ Services Library requires to initialize, copy, destroy, encode, and decode data types. If the GenerateDescriptor property is False, the data type cannot be sent by value in messages and won't be observable or injected.

Version (Class, C++ TargetRTS)

Specifies the version of the data type.

InitFunctionBody (Class, C++ TargetRTS)

Specifies the body of a function to initialize a data type. By default the C++ code generator generates a function which calls the data types default constructor.

```
static void rtg_AClass1_init( const RTObject_class * type, AClass1 *
target )

{

   (void)new( target ) AClass1;

}
```

You should only have to modify this property if your data type cannot be initialized with a default constructor.

This field may also be modified from the generated code and captured into the model using the Code Sync feature. For more information, see the topic **Using Code Sync to Change Generate Code** in the *Rational Rose RealTime Toolset Guide*.

CopyFunctionBody (Class, C++ TargetRTS)

Specifies the body of a function to copy a data type. By default the C++ code generator generates a function which calls the data type's copy constructor.

```
static void rtg_AClass1_copy( const RTObject_class * type, AClass1 *
target, const AClass1 * source )
{
   (void)new( target ) AClass1( *source );
}
```

This field may also be modified from the generated code and captured into the model using the Code Sync feature. For more information, see Using Code Sync to Change Generate Code in the Toolset Guide.

DestroyFunctionBody (Class, C++ TargetRTS)

Specifies the body of a function to destroy a data type. By default the C++ code generator calls the data types default constructor.

```
static void rtg_AClass1_destroy( const RTObject_class * type, AClass1
* target )
{
   target->~AClass1();
}
```

This field may also be modified from the generated code and captured into the model using the Code Sync feature. For more information, see Using Code Sync to Change Generate Code in the Toolset Guide.

DecodeFunctionBody (Class, C++ TargetRTS)

Specifies the body of a function to decode a data type from a stream of bytes. By default the C++ code generator uses a built-in function. If the C++ Services Library does not know about a data type, because it may be externally defined, or have private fields, then you can write your own decoder. The function is passed a pointer to an object of the RTDecoding class called coding and a pointer is passed to an object of the externally defined type called target. the RTDecoding class has a number of operations that can be used to decode different data values to the target argument.

Note: The C++ code generator does not automatically generate proper decode functions for classes which have more than one base class.

Because Enumerations (type enum) are defined in C and C++ as type int, they are internally represented as integers in the application generated from the model. To display these values in a trace window an additional step, called encoding, is required to map the values from their literal to their symbolic representations. A second step, called decoding, is used to map the symbol to the value when injecting signals using a probe.

On the C++ (or C) TargetRTS Tab of the Class Specification Dialog for MotorStatus, the DecodeFunctionBody field contains the code used to decode information injected while the model is running using a Probe. If present, code in this field will trigger the generation of a static function called rtg_MotorStatus_decode. The complete signature for the function is:

```
static int rtg_MotorStatus_decode( const RTObject_class * type,
enum MotorStatus * target,
RTDecoding * coding )
```

The parameters "type" and "coding" are defined as above. The parameter "target" provides access to the data field of the signal.

In the following example, the injected data (contained in "coding") is used to generate the proper enum value for the signal. The return value of the function indicates the success or failure of the decode.

```
// BEGIN CODE EXAMPLE
char * StringValue;
int ReturnValue = 0;

coding->get_string(StringValue);

if( strcmp(StringValue,"On") == 0)
{
*target = On;
ReturnValue=1;
}
else if( strcmp(StringValue,"Off" ) == 0 )
{
*target = Off;
ReturnValue=1;
}
else if( strcmp( StringValue, "Standby" ) == 0 )
```

```
{
*target = Standby;
ReturnValue=1;
}

return ReturnValue;
// END CODE EXAMPLE
```

Refer to the following files for more information in the decode function:
$ROSERT_HOME/C++/TargetRTS/include/RTDecoding.h
$ROSERT_HOME/C/TargetRTS/include/RTPubl/Decoding.h

You will also have to provide encode/decode functions for attributes that have double indirection (for example, int **).

This field may also be modified from the generated code and captured into the model using the Code Sync feature. For more information, see Using Code Sync to Change Generate Code in the Toolset Guide.

EncodeFunctionBody (Class, C++ TargetRTS)

Specifies the body of a function to encode a data type to a stream of bytes. By default the C++ code generator uses a built-in function. If the C++ Services Library does not know about a data type, because it may be externally defined, or have private fields, then you can write your own encoder. The function is passed a pointer to an object of the RTEncoding class called coding and a pointer is passed to an object of the externally defined class called source. The RTEncoding class has a number of operations that will encode different data values from the source argument.

Note: The C++ code generator does not automatically generate proper encode functions for classes which have more than one base class.

Because Enumerations (type enum) are defined in C and C++ as type int, they are internally represented as integers in the application generated from the model. To display these values in a trace window an additional step, called encoding, is required to map the values from their literal to their symbolic representations. A second step, called decoding, is used to map the symbol to the value when injecting signals using a probe.

To demonstrate this concept, we will declare a new class called "MotorStatus" using the stereotype "enumeration" that has three public attributes: On, Off, and Standby. The class will generate the following code:

enum MotorStatus { On, Off, Standby };

On the C++ (or C) TargetRTS Tab of the Class Specification Dialog for MotorStatus, the EncodeFunctionBody field contains the code used to encode the data. If present, code in this field will trigger the generation of a static function called rtg_MotorStatus_encode. The complete signature for this function is:

```
static int rtg_MotorStatus_encode( const RTObject_class * type,
const enum MotorStatus * source,
RTEncoding * coding )
```

The parameter "type" points to a data structure that provides an internal description of MotorStatus. The parameter "source" is a pointer to the data being sent in the signal, and "coding" provides a pointer to the data structure used to hold the descriptive information that will be used in a trace window.

In the following code example, the signal data (contained in "source") is used to enter the proper string representation in "coding" using the "put_string" function:

```
// BEGIN CODE EXAMPLE
switch(*source)
{
case On :
coding->put_string("On");
break;
case Off:
coding->put_string("Off");
break;
case Standby:
coding->put_string("Standby");
break;
default:
coding->put_string("ERROR");
}
return 1;
// END CODE EXAMPLE
```

Refer to the following files for more information in the encode function:
$ROSERT_HOME/C++/TargetRTS/include/RTEncoding.h
$ROSERT_HOME/C/TargetRTS/include/RTPubl/Encoding.h

Note: You will also have to provide encode/decode functions for attributes that have double indirection (for example, int **).

This field may also be modified from the generated code and captured into the model using the Code Sync feature. For more information, see Using Code Sync to Change Generate Code in the Toolset Guide.

GenerateDescriptor (Attribute, C++ TargetRTS)

Specifies whether to generate a descriptor for the attribute. If a descriptor is not generated the C++ Services Library won't be able to encode/decode the attribute.

TypeDescriptor (Attribute, C++ TargetRTS)

Specifies an explicit descriptor for the attribute. Normally the code generator will determined which descriptor should be used for the attribute, but in some cases you may want to override this.

NumElementsFunctionBody (Attribute, C++ TargetRTS)

If the attribute is a pointer to an object, this pointer may point to one or many objects. The **NumElementsFunctionBody** property provides the body of the function which calculates the number of objects the pointer points to. If the body is empty, the pointer is assumed to point to only one object.

This function is required to make attributes which are pointers to arrays observable in the execution monitors.

In the function body you have access to the attributes containing object. In the example below the attribute is part of a PointerInts object. You will usually use information contained in the containing object to determine how many things the pointer is pointing to.

```
static int rtg_nefb_PointerInts_ints( const RTTypeModifier * modifier,
const PointerInts * source )
{
   return( source->n_ints );
}
```

This field may also be modified from the generated code and captured into the model using the Code Sync feature. For more information, see Using Code Sync to Change Generate Code in the Toolset Guide.

GenerateDescriptor (Role, C++ TargetRTS)

Specifies whether to generate a descriptor for the attribute. If a descriptor is not generated the C++ Services Library won't be able to encode/decode the attribute.

TypeDescriptor (Role, C++ TargetRTS)

Specifies an explicit descriptor for the attribute. Normally the code generator will determined which descriptor should be used for the attribute, but in some cases you may want to override this.

NumElementsFunctionBody (Role, C++ TargetRTS)

If the association end is generated as a pointer, the pointer may point to one or many objects. For more details, see *NumElementsFunctionBody (Attribute, C++ TargetRTS)* on page 201.

This field may also be modified from the generated code and captured into the model using the Code Sync feature. For more information, see Using Code Sync to Change Generate Code in the Toolset Guide.

Version (Protocol, C++ TargetRTS)

Specifies the version of the data type.

BackwardsCompatible (Protocol, C++ TargetRTS)

If checked the protocol class will be generated with code to allow use of previous release communication services syntax.

TypeSafeSignals (Protocol, C++ TargetRTS)

If this property is unchecked then all signal data classes specified for the signals in the protocol are ignored. Instead it is assumed that you can send anything (for example, void *) with the signals.

This turns off all compile and run-time type safety checks. This is meant for backwards compatibility.

Type Descriptors

This topic describes type descriptors as follows:

- *What are Type Descriptors?* on page 203
- *When are Type Descriptors Used?* on page 204
- *Example Usage Patterns and Associated Type Descriptors* on page 206

What are Type Descriptors?

Type descriptors are used to describe a class to the Services Library, so that the Services Library can manipulate the objects in order to send them or use them in the UML debugger. Specifically, the Service Library needs to be able to initialize, copy, destroy, encode, and decode objects of the corresponding type. A type descriptor is implemented in the code through **RTObject_class**. For additional information, see *RTObject_class* on page 268.

Type descriptors are generated for most classes which are defined using basic types. The code generator is not able to generate a type descriptor for classes that:

- contain pointers
- contain non-basic types (e.g. unions)*,
- are externally defined
- are typedefs

For a complete list of basic types, see the file RTObject_class.h in the $ROSERT_HOME/C++/TargetRTS/include directory. The basic types all have a type descriptor of the form **RTType_<*type*>** defined in this file.

If the type descriptor cannot be generated automatically for a class, the five functions (init, copy, destroy, encode, and decode) used by the type descriptor can be implemented by the user. For additional information, see *C++ TargetRTS Properties* on page 195.

When are Type Descriptors Used?

Type descriptors are used whenever data is passed to the Services Library. In particular, this occurs in the following situations:

- Scenario 1 - Sent by value in a message between capsules
- Scenario 2 - Observed at run-time (for example, in a Watch or Trace window)
- Scenario 3 - Output via the log service
- Scenario 4 - Injected in a message on a probe.
- Scenario 5 - Sent by value in a message between capsules in different processes using Connexis

If scenarios 2 to 5 are not required, then it is not necessary to provide encoding and decoding functionality.

Problem:

A type descriptor is not available for the implementation type of this typedef.

Symptom:

The following code generation error is output:

Error: A type descriptor is not available for the implementation type of this typedef (perhaps the implementation type is non-trivial or the RT functions are partially overridden). Either provide overrides for zero or five of the RT function bodies, or turn off descriptor generation for this typedef.

Cause:

The code generator is not able to determine how to create the type descriptor for classes which are typedefs.

In earlier versions of Rose RealTime, some assumptions were made when generating the type descriptor for classes that are **typedefs**. These assumptions were not necessarily valid. Therefore, it is now necessary for the user to provide the implementation of the five functions used by the type descriptor if a type descriptor is to be generated.

Resolution:

If it is not necessary to generate a type descriptor, then the **GenerateDescriptor** property on the C++ TargetRTS tab of the typedef class can be set to **False**.

If it is necessary to generate a type descriptor, then the five function bodies (init, copy, destroy, encode, and decode) on the C++ Target RTS tab must be implemented.

Here are some examples of how to implement the five type descriptor functions.

Basic Example 1:

The class is a **typedef** to an int (for example, typedef int MyInteger;)

InitFunctionBody:

```
RTType_int._init_func( type, target );
```

CopyFunctionBody:

```
RTType_int._copy_func( type, target, source );
```

DestroyFunctionBody:

```
RTType_int._destroy_func( type, target );
```

DecodeFunctionBody:

```
return coding->get_int( *target );
```

EncodeFunctionBody:

```
return coding->put_int( *source );
```

Basic Example 2:

The class is a **typedef** to a pointer (for example, typedef foo * fooPtr;)

There are many different ways that you can use fooPtr. It could be a pointer to a single object, or a pointer to an array of objects (with fixed length, length determined by other variables, or some form of null termination). The operations on this pointer may also be "by value" or "by reference". This makes it impossible to generate a descriptor for all cases.

Example Usage Patterns and Associated Type Descriptors

Example: typedef Person * PersonPtr;

There are many different ways to use PersonPtr. It could be:

1 A pointer with no explicit definition of what it points at.
   - a) passed by reference,
2 A pointer to a single object which passed by value,
3 A pointer to an array of objects (the only interesting cases are pass by value – by reference these cases reduce to the equivalent of case 1a).
   - a) with fixed length,
   - b) with length determined by null termination
   - c) with length determined by other variables

It is it impossible to automatically generate a descriptor for all cases since the descriptor is tied to the usage pattern.

This topic provides the usage at the call site, the usage at the **Receipt** site, and the type descriptors necessary for this behavior. It is important to note that in all cases (except case 1) the system takes care of memory management.

All of these cases are implemented in the model Example.rtmdl which should accompany this document.

Case 1: The object was a simple pointer that would be copied by reference

In this case the user must explicitly manage the allocation and de-allocation of memory. This is the most flexible semantics, since the target of the pointer can be a single object, an array of object, or any user defined interpretation of the pointer.

This example shows the case where the pointer points to a single value

```
typedef Person * PersonRefPtr;
```

Call Site

```
log.log( "Sending Tim:10 by reference" );
Person * aPerson = new Person( "Tim", 10 );
commPort.sendRef( aPerson ).send();
```

Receipt

```
   Person * aPerson = *rtdata;
   log.log( "Receiving byRef" );
   log.log( aPerson, &RTType_Person );
   delete aPerson;
```

Descriptors

```
static void rtg_PersonRefPtr_init( const RTObject_class * type,
PersonRefPtr * target )
{
   *target = (Person *)0;
}
static void rtg_PersonRefPtr_copy( const RTObject_class * type,
PersonRefPtr * target, const PersonRefPtr * source )
{
*target = *source;
}
static int rtg_PersonRefPtr_decode( const RTObject_class * type,
PersonRefPtr * target, RTDecoding * coding )
{
return 0;
}
static int rtg_PersonRefPtr_encode( const RTObject_class * type, const
PersonRefPtr * source, RTEncoding * coding )
{
   return coding->put_address( source );
}

static void rtg_PersonRefPtr_destroy( const RTObject_class * type,
PersonRefPtr * target )
{
   *target = (Person *)0;
}
```

Case 2: The object was a pointer to a single object that would be copied by value

For this case, the user does not need to manage memory. A copy of the data is made at the send site, and this copy is reclaimed at the end of the receiving transition.

```
typedef Person * PersonValPtr;
```

Call Site

```
   log.log( "Sending Tim:10 by reference" );
   Person * aPerson = new Person( "Tim", 10 );
   commPort.sendRef( aPerson ).send();
```

Receipt

```
   log.log( "Recieved Person by Value" );
   Person * aPerson = *rtdata;
   log.log( aPerson, &RTType_Person )
```

Descriptors

```
static void rtg_PersonValPtr_init( const RTObject_class * type,
PersonValPtr * target )
{
   *target = new Person;
}


static void rtg_PersonValPtr_copy( const RTObject_class * type,
PersonValPtr * target, const PersonValPtr * source )
{
   *target = new Person( **source );
}


static int rtg_PersonValPtr_decode( const RTObject_class * type,
PersonValPtr * target, RTDecoding * coding )
{
   return coding->get_indirect( target, &RTType_Person );
}


static int rtg_PersonValPtr_encode( const RTObject_class * type, const
PersonValPtr * source, RTEncoding * coding )
{
   return coding->put_indirect( source, &RTType_Person );
}

static void rtg_PersonValPtr_destroy( const RTObject_class * type,
PersonValPtr * target )
{
   delete *target;
   *target = (Person *)0;
}
```

Case 3a: The object was a pointer to an array of objects of fixed length that would be copied by value

```
typedef Person * PersonArray4Ptr;
```

Call Site

```
   log.log( "sending array of length 4" );
   Person people[ 4 ] =
   {
      Person( "Tim",     10 ),
      Person( "Mary",    20 ),
      Person( "Tom",     30 ),
      Person( "Monique", 40 )
   };
   commPort.sendArray4( people ).send();
```

Receipt

```
   log.log( "Received Array4 of names" );
   log.log( rtdata, &RTType_PersonArray4Ptr );
```

Descriptors

```
static void rtg_PersonArray4Ptr_init( const RTObject_class * type,
PersonArray4Ptr * target )
{
   *target = PersonArrayUtil::allocate( 4, true );
}

static void rtg_PersonArray4Ptr_copy( const RTObject_class * type,
PersonArray4Ptr * target, const PersonArray4Ptr * source )
{
   Person * people = PersonArrayUtil::allocate( 4, false );
```

```
    *target = people;

    for( int i = 0; i < 4; ++i, ++people )
        (void)new( people ) Person( (*source)[ i ] );
}

static int rtg_PersonArray4Ptr_decode( const RTObject_class * type,
PersonArray4Ptr * target, RTDecoding * coding )
{
    return coding->get_array( *target, 4, &RTType_Person );
}

static int rtg_PersonArray4Ptr_encode( const RTObject_class * type,
const PersonArray4Ptr * source, RTEncoding * coding )
{
    return coding->put_array( *source, 4, &RTType_Person );
}

static void rtg_PersonArray4Ptr_destroy( const RTObject_class * type,
PersonArray4Ptr * target )
{
    PersonArrayUtil::release( 4, *target );
    *target = (Person *)0;
}
```

Case 3b - the object was a pointer to an array of objects of variable length (with null termination) that would be copied by value.

```
typedef Person * PersonArrayNullPtr;
```

Call Site

```
    log.log( "Sending Null Terminated list" );
    Person people[ 3 ] =
    {
        Person( "Tim",    10 ),
        Person( "Mary",   20 ),
        Person( "nobody",  0 )
    };
```

```
   commPort.sendArrayNull( people ).send();
```

Receipt

```
   log.log( "Received ArrayNull of names" );
   log.log( rtdata, &RTType_PersonArrayNullPtr );
```

Descriptors

```
static int countPeople( const Person * people )
{
   int count = 0;
   for( ; people->getAge() != 0; ++count, ++people ) { }
   return count;
}

static void rtg_PersonArrayNullPtr_init( const RTObject_class * type,
PersonArrayNullPtr * target )
{
   *target = PersonArrayUtil::allocate( 1, true );
}

static void rtg_PersonArrayNullPtr_copy( const RTObject_class * type,
PersonArrayNullPtr * target, const PersonArrayNullPtr * source )
{
   Person * src    = *source;
   int      count  = 1 + countPeople( src );
   Person * people = PersonArrayUtil::allocate( count, false );
   Person * dst    = people;
   while( --count >= 0 )
      (void)new( dst++ ) Person( *src++ );
   *target = people;
}

static int rtg_PersonArrayNullPtr_decode( const RTObject_class * type,
PersonArrayNullPtr * target, RTDecoding * coding )
{
   int      length = 0;
   Person * people;
   if( coding->get_int( length ) == 0 )
```

```
    {
    }
    else if( length < 0 )
    {
    }
    else if( (people = PersonArrayUtil::allocate( length + 1, true )) ==
(Person *)0 )
    {
    }
    else if( coding->get_array( people, length, &RTType_Person ) == 0 )
    {
        PersonArrayUtil::release( length + 1, people );
    }
    else
    {
        Person * trash = *target;
        if( trash != (Person *)0 )
            PersonArrayUtil::release( countPeople( trash ) + 1, trash );
        *target = people;
        return 1;
    }

    return 0;
}

static int rtg_PersonArrayNullPtr_encode( const RTObject_class * type,
const PersonArrayNullPtr * source, RTEncoding * coding )
{
    int count = countPeople( *source );
    return coding->put_int( count ) != 0
        && coding->put_array( *source, count, &RTType_Person ) != 0;
}

static void rtg_PersonArrayNullPtr_destroy( const RTObject_class *
type, PersonArrayNullPtr * target )
{
    Person * people = *target;
```

```
   if( people != (Person *)0 )
   {
      *target = (Person *)0;
      PersonArrayUtil::release( countPeople( people ) + 1, people );
   }
}
```

Case 3c: The object was a pointer to an array of objects of variable length (determined by another variable) that would be copied by value

This particular case is simpler if the pointer is contained within a class. For this example the length of the array will be another field within the same class. This class is defined as follows:

```
class PersonArrayObj
{
public:
   PersonArrayObj( void );
   ~PersonArrayObj( void );
   PersonArrayObj & operator=( const PersonArrayObj & rhs );
   PersonArrayObj( int num );
   PersonArrayObj( const PersonArrayObj & other );
   int length;
   Person * people;
};
```

Call Site

```
   log.log( "Sending variable length array" );
   PersonArrayObj obj( 4 );
   log.log( &obj, &RTType_PersonArrayObj );
   commPort.sendArrayObj( obj ).send();
```

Receipt

```
log.log( "Received ArrayObj" );
log.log( rtdata, &RTType_PersonArrayObj );
```

Descriptors

```
// Use default “init”, “copy”, and “destroy”
```

```
static int rtg_PersonArrayObj_decode( const RTObject_class * type,
PersonArrayObj * target, RTDecoding * coding )
{
   int      length = 0;
   Person * people;
   if( coding->get_int( length ) == 0 || length < 0 )
   {
   }
   else if( (people = PersonArrayUtil::allocate( length, true )) ==
(Person *)0 )
   {
   }
   else if( coding->get_array( people, length, &RTType_Person ) == 0 )
   {
      PersonArrayUtil::release( length, people );
   }
   else
   {
      PersonArrayUtil::release( target->length, target->people );
      target->length = length;
      target->people = people;
      return 1;
   }
   return 0;
}

static int rtg_PersonArrayObj_encode( const RTObject_class * type,
const PersonArrayObj * source, RTEncoding * coding )
{
   return coding->put_int( source->length ) != 0
   && coding->put_array( source->people, source->length,
&RTType_Person ) != 0;
}
```

C++ Generation Properties

Code generation properties are used to configure the way in which a component is generated to C++. These properties apply equally to Executable and Library component types.

Component

- *OutputDirectory (Component, C++ Generation)* on page 215
- *CodeGenDirName (Component, C++ Generation)* on page 215
- *ComponentUnitName (Component, C++ Generation)* on page 216
- *CommonPreface (Component, C++ Generation)* on page 216
- *CodeGenMakeType (Component, C++ Generation)* on page 217
- *CodeGenMakeCommand (Component, C++ Generation)* on page 217
- *CodeGenMakeArguments (Component, C++ Generation)* on page 217
- *CodeGenMakeInsert (Component, C++ Generation)* on page 217
- *CodeSyncEnabled (Component, C++ Generation)* on page 218
- *Generate Model Tags (Component, C++ Generation)* on page 218

OutputDirectory (Component, C++ Generation)

The output path can be changed to allow you to set the directory into which the generated files resulting from a component build will be written. By default this property is set to **$@/$name** where **$@** is the model file directory, and **$name** is the name of the component.

CodeGenDirName (Component, C++ Generation)

Specifies the name of the directory that will be created to hold the generated source code for the component elements. This directory will be generated as a subdirectory of **<output directory>/src**.

By default this property is left blank, meaning that source files can be found in the src subdirectory.

Note: Output directory is specified in the OutputDirectory (Component, C++ Generation) property.

ComponentUnitName (Component, C++ Generation)

Specified the name of the source files generated for the component itself. You should only have to modify this property if the component has the same name as another model element, or conflicts with other source files. The name given here will change the name of the generated component .cpp and .h files.

CommonPreface (Component, C++ Generation)

Component level inclusion files are entered as inclusions in this list. Any number can be specified and are entered independently of any directory search list. The list of directories to search for these inclusions is entered through the InclusionPaths (Component, C++ Compilation). Inclusions items can be added and deleted as desired.

The scope of inclusions is system level. For example, if all elements being built by this component make use of a set of math routines, the math header file can be specified here instead of on each individual element. In addition, the inclusions are declared in exactly the sequence they appear in the list (top to bottom). One way this ordering can be useful is by using it to have normal system include files specified before user includes. Specifying system includes in this way can aid visibility and ensure completeness.

Note: The compiler you are using may search some paths automatically; for example, a compiler hosted on UNIX often searches /usr/include.

Generally, inclusions can be declared at both the component and class level. The former specified in this inclusion list, the latter specified through the classes specification dialog. In either case the directory search list is taken from the InclusionPaths (Component, C++ Compilation) property.

You can add class level inclusions via the ImplementationPreface (Class, C++) and HeaderPreface (Class, C++) properties on classes and capsules.

This field may also be modified from the generated code and captured into the model using the Code Sync feature. For more information, see Using Code Sync to Change Generate Code in the Toolset Guide.

Both inclusion types get dropped into the global space. However, the only semantic difference between them is the scope guarantee: the component-level inclusions are guaranteed to have all classes in their scope, while the class-level inclusions guarantee that only that class and its subclasses will have the declared inclusion in scope (that is, visible). These includes are actually in the global space regardless of type, so we recommend that you restrict usage of these inclusions to extern and type declarations; otherwise, multiple definition are reported at link time.

CodeGenMakeType (Component, C++ Generation)

Can be one of **<default>**, **Unix_make**, **MS_nmake**, **ClearCase_clearmake** or **Gnu_make**. This influences the format of the generated makefiles so they conform to differences in the make variants.

Leaving the entry as **<default>** will allow the code generator to automatically select the make type based on the platform on which the component is being generated. Either **Unix_make** (for Unix) or **MS_nmake** (for WindowsNT) will be substituted for <default>. If you require another make type, then you should explicitly specify the make type in this field.

CodeGenMakeCommand (Component, C++ Generation)

The name of the make utility being used to control the code generation. The make name must be the exact name of the make command. By default the default make command is $defaultMakeCommand which will allow the code generator to automatically select the make type based on the platform on which the component is being generated. Either **make** (for UNIX) or **nmake** (for Windows) will be substituted for <default>. If you require a different make utility, just type it in.

When a model is built, Rose RealTime generates the model files, and then invokes the make utility to generate the source code from the model files. Code generation is, therefore, external. Make handles incremental code generation by using the timestamps on the toolset generated model files.

CodeGenMakeArguments (Component, C++ Generation)

Any flags supported to be passed to the make utility.

CodeGenMakeInsert (Component, C++ Generation)

The make insert is a makefile fragment which is included in the compilation makefile that allows for the addition of user-defined dependencies, compile, and link options.

CodeSyncEnabled (Component, C++ Generation)

Specifies whether **codesync** is enabled for a component. Disabling this feature removes tags from the code making it easier to read. Codesync tags annotate the generated code with tags that show you where you can edit code, and then let you synchronize it back into the model using the **codesync** feature. Codesync tags look like the following:

```
// {{{USR


// }}}USR
```

You can make modifications between the tags, and then synchronize these changes back into the model.

This option is available in the **Item Properties** area on the **C++ Generation** tab for a component.

See Also C++ TargetRTS properties, Code Generation

Generate Model Tags (Component, C++ Generation)

Specifies whether to annotate the generated code with information used to correlate the lines of code back to specific model elements. For example, if there is a compiler error or warning in the generated code, you can find that associated model element. Model tags look like the following:

```
// {{{RME operation 'main(int,const char *)'

int main( int argc, const char * argv );

// }}}RME
```

C++ Compilation Properties

Compilation properties are used to configure the way in which the generated source files for a component are compiled. These properties apply equally to Executable and Library component types. Both executables and libraries require compilation.

Component

- *CompilationMakeType (Component, C++ Compilation)* on page 219
- *CompilationMakeCommand (Component, C++ Compilation)* on page 219
- *CompilationMakeArguments (Component, C++ Compilation)* on page 220
- *CompilationMakeInsert (Component, C++ Compilation)* on page 220
- *CompileCommand (Component, C++ Compilation)* on page 220

- *CompileArguments (Component, C++ Compilation)* on page 221
- *InclusionPaths (Component, C++ Compilation)* on page 221
- *TargetServicesLibrary (Component, C++ Compilation)* on page 222
- *TargetConfiguration (Component, C++ Compilation)* on page 222

CompilationMakeType (Component, C++ Compilation)

Can be one of **<default>**, **Unix_make**, **MS_nmake**, **ClearCase_clearmake** or **Gnu_make**. This influences the format of the generated makefiles so they conform to differences in the make variants. For example, if nmake on Windows NT is being used, then **MS_nmake** must be selected as the make type.

Leaving the entry as **<default>** will allow the code generator to automatically select the make type based on the platform on which the component is being generated. Either **Unix_make** (for UNIX) or **MS_nmake** (for WindowsNT) will be substituted for **<default>**. If you require another make type, then you should explicitly specify the make type in this field.

CompilationMakeCommand (Component, C++ Compilation)

When a model is built, Rational Rose RealTime generates the model files then invokes the **make** utility to generate the source code from the model files. Code generation is, therefore, external. **Make** handles incremental code generation by using the timestamps on the toolset generated model files.

The name of the **make** utility being used to control the code generation. The make name must be the exact name of the make command. By default the default make command is **$defaultMakeCommand** which defaults to **nmake** on Windows and **make** on UNIX. If you want to use a different make utility than what is shown just type it in.

By default, the **CompilationMakeCommand** box contains the following command:

rtperl -S rtsetup.pl $defaultMakeCommand

You use the Perl script **rtsetup.pl** to reuse the environment used to build the TargetRTS. The script examines the generated **makefile** to determine the **TargetConfiguration** specified for the current component, executes $RTS_HOME/config/<TargetConfiguration>/setup.pl to re-create the environment required to build the TargetRTS, and then, in that environment, executes the command that follows. It is useful for designers who must build components for different targets which depend on conflicting environment variable definitions.

To use vssetup.pl, add the following to the **CodeGenMakeCommand**:

```
rtperl -S vssetup.pl $defaultMakeCommand
```

CompilationMakeArguments (Component, C++ Compilation)

Any flags supported to be passed to the make utility.

CompilationMakeInsert (Component, C++ Compilation)

The make insert is a makefile fragment which is included in the compilation makefile that allows for the addition of user-defined dependencies, compile, and link options.

Refer to the generated makefiles to understand what make macros and variables are generated and can be used when writing the make insert. In addition both the Callbacks and ISR C++ example models use make inserts. You can use these as a starting point.

For additional information on the ISR C++ example, see the book **Model Examples, Rational Rose RealTime**.

CompileCommand (Component, C++ Compilation)

The compiler command property is used to replace the pre-configured compiler shell command defined in libset.mk. You would normally leave this entry (usually set to $(CC))and use the default compiler specified in the libset makefile.

When building your model, a compiler will be used to compile the generated code and a linker will be used to link the executable. By default, when you specify the Service Library, you identify the make files to be used to build the component and the tools are specified in the makefile called:

$ROSERT_HOME/RTSType/TargetLibrary/libset/Library/libset.mk

While you can override in the component, the compiler and/or the linker to be used, the new tools used should be compatible with the ones being overridden. Typically you want to override the compiler/linker to:

- perform preprocessing.

  For example, instead of invoking the compiler straight away, you can invoke a script that will perform some preprocessing, as well as compiling (such as running the source file through lint before invoking the compiler).

- qualify the path to the compiler/linker because they are not in the current path.

  If you want to choose a completely different type of compiler (gnu vs. greenhills), or even a different release of a compiler, you should be changing instead the Service Library specification. That way the make files used will pass flags understood by the compiler/linker. As well, you will be sure the precompiled Service Library that is to be linked will have been compiled with the compiler you are using.

CompileArguments (Component, C++ Compilation)

Any flags supported by your compiler utility. This is where you would specify a parallel make flag to increase compilation efficiency.

InclusionPaths (Component, C++ Compilation)

Any number of entries can appear as inclusion path items. As a group they comprise the directory search set used by the compiler to find user-specified inclusion files. They are searched in the order specified in the list. The inclusion paths property has an advanced property dialog. When the **Edit...** button is pressed, in the dialog that opens, you can directly edit existing include paths from within the list and add and remove entries using the Insert and Remove buttons.

You should avoid adding unnecessary inclusion paths to this list. The number of directories that need to be searched for a file can slow down the compilation process because of the file access that is required for searching all the directories.

It is recommended that pathmaps/shell/environment variables be used when specifying the inclusion paths. This way other team members can configure their environment without having to modify the component.

Note: Path map variables, those defined within the toolset, can be used to specify indirect inclusion paths.

If **Compute Dependencies** is set to **Yes**, then the make depends utility will be used to calculate dependencies in that directory and the object file for the model becomes dependent on the inclusion files in this directory that it needs.

Note: The only time you should set **Compute Dependencies** to **No** is if the inclusion file timestamp changes artificially and you don't want this to trigger a recompile, or if the inclusion is a system level which very rarely changes.

TargetServicesLibrary (Component, C++ Compilation)

The text field is used to specify the path to the root directory for the specific Services Library desired. This can be any legal directory name. This name must be specified as a full path to the root directory of the Services Library root.

The Target Services directory contains all the scripts and programs to generate and compile a component. Hence, if this directory is not configured correctly, you won't be able to generate or compile. You are likely to see the "name not found" or "Build Failed" error appear in the **Build Log Window** if it is incorrectly configured.

By default this field references the Services Library in your Rose RealTime home directory $ROSERT_HOME/C++/TargetRTS. This can be changed to any other directory that contains the C++ Services Library.

TargetConfiguration (Component, C++ Compilation)

This property is used to uniquely identify the configuration of the Services Library that will be used to compile and link the component. The configuration name is composed of three parts: os.processor-compiler-version.

For example, the configuration for a WindowsNT 4.0 multi-threaded platform with an x86 processor built with version 6.0 of Microsoft Visual C++ would be called:

```
NT40T.x86-VisualC++-6.0
```

If you would like to see the valid configuration names, look at the directories located in the lib subdirectory of the Services Library root. If you build different configurations of the Services Library the new configuration will appear in this list.

C++ Executable Properties

This group of model properties is used to control the aspects of generating an executable from a C++ model. C++ Executable properties apply only to components which are of type C++ Executable. This page contains a summary of the C++ Executable properties grouped by model element to which they are associated.

Component

- *TopCapsule (Component, C++ Executable)* on page 223
- *PhysicalThreads (Component, C++ Executable)* on page 223
- *ExecutableName (Component, C++ Executable)* on page 225
- *DefaultArguments (Component, C++ Executable)* on page 226
- *LinkCommand (Component, C++ Executable)* on page 226
- *LinkArguments (Component, C++ Executable)* on page 226
- *UserLibraries (Component, C++ Executable)* on page 226
- *UserObjectFiles (Component, C++ Executable)* on page 227

TopCapsule (Component, C++ Executable)

Specifies the top capsule to be compiled for this component. The top capsule defines the compilation closure for the component. All classes, including capsule and protocol classes referenced directly or indirectly by the top capsule will be compiled as part of the component. Dependencies are verified before every component build, and are added to this list before the build. The top capsule also defines the default executable name to be produced by the compilation.

This property uses an advanced property editor. When you click **Select**, a dialog lists all capsules referenced by the component. Select the desired top level capsule, and click **OK**.

PhysicalThreads (Component, C++ Executable)

On some platforms, the Services Library supports multiple threads. Optional capsule roles can be assigned to different logical threads. These logical threads can then be assigned to a physical thread configuration for the target system. The physical thread configuration can be changed without affecting the logical thread design of the model.

By default, all capsules are assigned to a pre-defined thread called MainThread. If you want a capsule to run in another user-defined thread, you must incarnate that capsule in that thread at run-time. Only optional capsules may be placed on threads other than the MainThread. Fixed capsules reside in the same thread as their container. When an optional capsule is incarnated on another thread, all fixed capsules contained inside the optional capsule are also placed on that thread. The top-level capsule is always fixed, so it, and all fixed capsules that it contains, are placed on the MainThread.

To incarnate an optional capsule in a particular thread, there is an optional parameter that should be specified for the Frame Service incarnate method.

The physical threads property is edited with an advanced property editor which opens when you press the **Edit...** button.

The physical threads list contains the list of physical threads that are defined for this component and shows which logical threads are associated with each physical thread. Depending on the implementation of threads provided by the Target Real-Time Operating System, each physical thread is a light-weight, time-sliceable process, running in a shared address space with the Services system threads and the other physical threads in the model.

By default, every configuration defines the following physical threads :

- MainThread: Where all of the capsules in your model execute by default. If you want capsules to execute in a thread other than the MainThread, you must define additional physical threads.
- TimerThread: Where the system timer service executes. TimerThread is always present, even if you do not use the timer services.

At this point you can create new physical threads, and either drag and drop logical threads to other physical threads or use the Logical threads list at the bottom of the dialog to assign the logical threads to physical threads.

Physical Thread Properties

For each physical thread you define you can also modify the following thread properties:

Stack Size	Size (in bytes) of the call stack allocated for this thread. By default is set to 20KB.
Priority	The priority at which this thread will run.

Note: Although stack size is configurable, for some target operating systems this stack size is effective at the time the main thread is created. This is because on some targets the OS creates the main thread with a default thread size, and this thread size cannot be modified at run-time. In these situations, the desired stack size for the main thread can be set by configuring the OS kernel or by the way in which the executable is spawned on the target.

Logical Threads

The logical threads list shows the different threads in the architectural design of the model. It also shows all logical threads defined in any component on which the executable component depends (e.g. any library or external library components). The name of the component in which the logical thread is defined is shown in brackets beside the logical thread name. You can’t modify these logical threads from within this dialog. Thus, if a library was build with references to logical threads, you must ensure that you list the names of the logical threads in the library LogicalThreads (Component, C++ Library) property and for external library components in LogicalThreads (Component, C++ External Library).

Each logical thread is a conceptually independent thread of execution. Logical threads may be mapped to different actual physical thread configurations for generating the executable implementation. However, the model entities are defined purely in terms of logical threads. That is, in the design, the model entities get allocated to a particular logical thread. Only at implementation time does the designer have to worry about mapping these to physical threads on the target system.

Physical Threads

The physical threads list contains the list of physical threads that are defined for this component. Depending on the implementation of threads provided by the Target Real-Time Operating System, each physical thread is a light-weight, time-sliceable process, running in a shared address space with the Services system threads and the other physical threads in the model.

ExecutableName (Component, C++ Executable)

You can specify the name, or a name with an absolute path, of the executable that will be created as a result of the component being built. If left unspecified, the executable name is set to the name of the component's top-level capsule.

If an absolute path is not used in the executable name, the executable will be located in the following component build output directory:

```
<output_dir>/build
```

DefaultArguments (Component, C++ Executable)

Some platforms do not allow command line arguments to be passed to an executable at load time (namely, on some real-time operating systems). In this case, the default arguments provides a mechanism for getting execution arguments into the executable. You can use RTMain::argStrings() to retrieve any passed command line argument within your model. Enter a comma separated list of quoted arguments into this field.

```
"134.434.344.4","barneyht","delay=98"
```

The default arguments field will only be used for targets that cannot accept command line arguments. Targets that accept command line arguments will ignore the content of this field.

LinkCommand (Component, C++ Executable)

The linker override field is used to replace the pre-configured linker shell command defined in libset.mk. You would normally leave this entry and use the default linker specified in the libset makefile.

LinkArguments (Component, C++ Executable)

Any flags supported by your linker utility.

UserLibraries (Component, C++ Executable)

Specifies libraries that are to be passed to the linker. You have to specify the library prefix, path, and extension correctly. The code generator does not modify these library names. For example, you can either add libraries on separate lines or separated by a space on the same line:

```
$@/userfiles.lib
$PROJECTX/lib/userfiles.lib
-lmath
```

Note: Enclose pathnames with spaces in double quotes ‘””.

This property is intended for backwards compatibility. We recommend that you model externally created libraries with external library components instead of adding them to this property. This will allow libraries to be visible in the toolset and more easily re-used with different executable components.

UserObjectFiles (Component, C++ Executable)

Specifies object that are to be passed to the linker. You have to specify the library prefix, path, and extension correctly. The code generator does not modify these object names. For example:

```
$@/userfiles.o
```

```
$PROJECTX/lib/userfiles.o
```

Note: Enclose pathnames with spaces in double quotes '"'.

This property is intended for backwards compatibility. It would be more flexible to create libraries for object files and then create external library components to model externally created libraries. This will allow libraries to be visible in the toolset and more easily re-used with different executable components.

C++ Library Properties

This group of model properties is used to control the aspects of generating a library from a C++ model. C++ Library properties apply only to components which are of type C++ Library. This page contains a summary of the C++ Library properties.

Component

- *LibraryName (Component, C++ Library)* on page 227
- *BuildLibraryCommand (Component, C++ Library)* on page 228
- *BuildLibraryArguments (Component, C++ Library)* on page 228
- *LogicalThreads (Component, C++ Library)* on page 228

LibraryName (Component, C++ Library)

The name of the generated library file. By default this name is **${LIB_PFX}$name${LIB_EXT}**. The library file is written to a directory called build which is located in the directory specified by the OutputDirectory (Component, C++ Generation) property.

LIB_PFX is defined as “lib” and can be configured. You can change the default setting for this make macro by modifying its definition in either of the following files:

$RTS_HOME/libset/default.mk

$RTS_HOME/libset/<libset name>/libset.mk.

LIB_EXT is defined as the default library extension for your platform. You can change the default setting for this make macro by modifying the following file:

$RTS_HOME/libset/<libset name>/libset.mk.

Note: $RTS_HOME is the location of your Services Library root directory. For more details about the Services Library directory, see *TargetServicesLibrary (Component, C++ Compilation)* on page 222.

BuildLibraryCommand (Component, C++ Library)

Specifies the archiving command. You would normally leave this entry and use the pre-configured linker shell command defined in libset.mk.

BuildLibraryArguments (Component, C++ Library)

Any flags supported by your archiver utility. They are passed as is to the archiver.

LogicalThreads (Component, C++ Library)

If the capsules referenced by this library component reference logical thread names, you have to list them here, one per line. For example:

```
LogicalThread1
LogicalThread2
LogicalThread3
```

If the logical thread names are not listed here your library component will not compile. The mapping from logical to physical threads is done on executable components. If an executable component uses a library (has a dependency between executable and library component) then the logical threads defined in this property will show in the logical threads list of the executable components threads dialog. This is where you can map the logical threads defined in the library to physical threads.

C++ External Library Properties

This group of model properties is used to control the aspects of generating the makefile fragments which allow pre-built libraries to be re-used when building an executable. C++ External Library properties apply only to components of type C++ External Library.

Component

CodeGenDirName (Component, C++ External Library) on page 229

InclusionPaths (Component, C++ External Library) on page 230

Libraries (Component, C++ External Library) on page 230

LogicalThreads (Component, C++ External Library) on page 231

After creating an external library component and configuring the properties, draw a dependency relationship between an executable component and the external library component to have the executable use the library referenced by the external component.

GenerateClassInclusions (Component, C++ External Library)

Turn this property off if you don't want inclusions generated in classes and capsules that use the elements referenced by the external library. This is useful if the inclusion is actually provided somewhere. Normally this should stay on.

CodeGenDirName (Component, C++ External Library)

This property is only required if GenerateClassInclusions (Component, C++ External Library) is turned on and the external library represents a library build from the toolset. This is the prefix directory for the generated source code. This should be set to the same value as CodeGenDirName (Component, C++ Generation) for the library component that was used to create the library which this external library references.

Having this prefix will ensure that all inclusions generated for model elements that reference elements in the external library will be prefixed with this value. This will reduce the chance of having inclusion conflicts. For example if this property is set to **rtg**, then inclusions will be generated as:

```
#include <rtg/foo.h>
```

InclusionPaths (Component, C++ External Library)

Specifies the location of the definitions for the external library. Components which reference this external library will automatically include the definitions header file.

```
$@/include
$PROJECTX/include
$@/ALibraryComponent/src
```

It is recommended that you use pathmap symbols or environment variables for pathnames in this property. For details, see *Environment Variables and Pathmap Symbols* on page 178.

If Compute Dependencies is set to Yes, then the make depends utility will be used to calculate dependencies in that directory and the object file for the model becomes dependent on the inclusion files in this directory that it needs.

Libraries (Component, C++ External Library)

Specifies the location and names of the libraries that this external component represents. The libraries listed in this field will be added to the link line for any executable component that references this external library.You have to specify the complete path and filename. For example:

On UNIX:

```
/home/projectX/lib/classes.a
$@/lib/classes.a
$PROJECTX/lib/classes.a
-L@/lib
-lclasses
```

On Windows:

```
$@/lib/classes.lib
C:\local\projects\ProjectX\lib\classes.lib
```

We recommend that you use **pathmap** symbols or environment variables for pathnames in this property.

For details, see *Environment Variables and Pathmap Symbols* on page 178.

If **Generate Dependencies** is set to **Yes**, the executable for the model becomes dependent on the library files. You must set **Generate Dependencies** to **False** for any entries which are directories (-L) or prefixed libraries (-lmath).

LogicalThreads (Component, C++ External Library)

If the capsules referenced by this external library component reference logical thread names, you have to list them here, one per line. For example:

```
LogicalThread1
LogicalThread2
LogicalThread3
```

The mapping from logical to physical threads is done on executable components. If an executable component uses a external library (a dependency exists between executable and external library component) then the logical threads defined in this property will show in the logical threads list of the executable components threads dialog. This is where you can map the logical threads defined in the external library to physical threads.

Services Library Class Reference 12

Contents

This chapter is organized as follows:

- *RTDataObject Subclasses* on page 234
- *RTActor* on page 234
- *RTActorClass* on page 239
- *RTActorRef* on page 240
- *RTActorId* on page 241
- *RTController* on page 242
- *Exception* on page 245
- *Frame* on page 247
- *RTInSignal* on page 257
- *Log* on page 261
- *RTMessage* on page 264
- *RTObject_class* on page 268
- *RTOutSignal* on page 270
- *RTProtocol* on page 275
- *RTSymmetricSignal* on page 286
- *RTTimerId* on page 287
- *RTTimespec* on page 287
- *Timing* on page 291
- *RTTypedValue* on page 299
- *Port Services* on page 299

Overview

The C++ Services Library Class Reference is a reference to the structures and classes that you will need to use within the detailed code of a capsule to access the services provided by the C++ Services Library.

The *C++ Services Library Framework* on page 93 shows the classes in a class diagram. The remainder of the Class Library Reference consists of an alphabetical listing of the classes.

In the alphabetical listing section, each class description includes a member summary by category, followed by alphabetical listings of operations and attributes. This reference does not describe private or restricted operations and attributes from the Services Library. Some features and classes in the Services Library are internal to the library itself and thus are not supported as interfaces into a user's application.

RTDataObject Subclasses

Classes derived from **RTDataObject** are only required for backwards compatibility with previous releases of ObjecTime Developer. These subclasses include **RTInteger**, **RTString**, **RTByteBlock**, **RTPointer**, **RTReal**, **RTCharacter**, **RTEnumerated**, **RTSequence**, and **RTSequenceOf**. In Rose RealTime you can create your own classes or you can import third-party class libraries into the Data Modeler. Classes no longer have to be derived from **RTDataObject**.

RTActor

Every capsule when generated as C++ code is a subclass of **RTActor**. This common base class for all capsules defines attributes and operations which allow the Services Library to communicate with the running capsule instances.

Since all detail level code added to a capsule class is generated as part of a capsule class, the detail level code has direct access to some useful attributes and operations that are defined on **RTActor**. You should only be calling the operations of **RTActor** or using attributes that are defined below:

The attributes and operations on **RTActor** are private. One capsule may not manipulate another capsule's attributes.

Operations

RTActor::context	Gets the controller for the physical thread on which a capsule instance is executing.
RTActor::getCurrentStateString	Gets the current state name containing the executing segment.
RTActor::getError	Gets the last error value for this thread.
RTActor::getIndex	Gets the replication index of this capsule instance in the home capsule role.
RTActor::getName	Gets the capsule role name in which this capsule instance is running.

RTActor::getTypeName	Gets the capsule class name of this capsule instance.
RTActor::isType	Queries the capsule class of this capsule instance.
RTActor::logMsg	Called before every message is delivered to a capsule instance (if configured in Services Library).
RTActor::msg and RTActor::getMsg	Accesses the msg attribute.
RTActor::unexpectedMessage	Called when a message is delivered to a capsule instance for which there is no trigger defined.

RTActor::msg and RTActor::getMsg

const RTMessage * msg;

const RTMessage * getMsg(void);

Return value

The **msg** attribute can be accessed via the **getMsg** operation. A pointer to the message is returned.

Remarks

Every capsule class has an attribute **msg** which contains a pointer to the current message delivered to a capsule instance. This attribute can be used within transition detail level code to retrieve a message that was sent to the capsule instance.

Examples

Retrieve the void * pointer to the data portion of the message.

```
const void *data_ptr = msg->data;
```

You can also use getMsg to access the current message.

```
const void *data_ptr = getMsg()->getData();
```

For most cases, the data can be accessed directly using the **rtdata** parameter that is passed to every transition code segment:

```
// The following is commonly needed to make a copy of the data
// that was sent with a message
const ADataClass & data1 =
   *((const ADataClass *)getMsg()->getData());
```

```
// the above statement can be written using the
// rtdata parameter available in all state
// transition segments.
const ADataClass & data1 = *rtdata;
```

RTActor::logMsg

virtual void logMsg(void);

Remarks

This operation is called by the Services Library before a received message is processed by a capsule instance (if so configured).

As implemented by the **RTActor** class, this operation prints to the log every message that was delivered to the capsule, depending on the debug level. Since this operation is defined as virtual, it can be useful in some circumstances to override this operation within a capsule class in order to provide some alternative processing for each message.

To override this operation, simply add a new operation to the capsule class with the same name and prototype (it takes no parameters and returns void). It can also be overridden for the entire model by creating a new RTActor::logMsg() operation, compiling it, and including it in the model using the model link options.

RTActor::unexpectedMessage

virtual void unexpectedMessage(void);

Remarks

This operation is called by the Services Library when there is no transition event found that is triggered by the current message about to be delivered. This happens when the capsule's rtsBehavior() function is called to process a message and no corresponding trigger event is found. The default unexpectedMessage() behavior prints a message to stderr. This operation can be overridden on a capsule class basis to provide any additional functionality that may be required.

To override this function, simply add a new function to the capsule class with the same name and prototype. It can also be overridden for the entire model by creating a new RTActor::unexpectedMessage() operation, compiling it, and including it in the model using the model link options.

RTActor::context

RTController * context(void);

Return value

A pointer to the controller for the thread on which this capsule instance is running.

Remarks

There are some public operations on the RTController class that can be accessed this way. In particular, you may find it useful for printing error information, as in the example below.

Examples

```
if( ! port.ping().send() )
{
   log.show("Error on physical thread: ");
   log.log( context()->name() );
   context()->perror("send");
}
```

RTActor::getError

RTController::Error getError(void) const;

Return value

The value of the most recent error within a particular thread. The Error enumeration is defined within the RTController class.

Remarks

The error code is not reset by a subsequent successful primitive operation call. It should be called immediately following the failure of a Services Library operation call.

Examples

See the example shown in the error codes descriptions.

RTActor::getIndex

int getIndex(void) const;

Return value

The replication index of this capsule instance in its “home” role (where it was incarnated). The replication value is zero (0) based.

RTActor::getName

const char * getName(void) const;

Return value

The name of the capsule role in which this capsule instance is running (where it was incarnated).

RTActor::getTypeName

const char * getTypeName(void) const;

Return value

Returns the class name of this capsule instance.

RTActor::isType

int isType(const char * class_name) const;

Return Value

Returns 1 (true) if this capsule instance is of class **class_name**, and 0 (false) otherwise.

Parameters

class_name

The name of a capsule class.

Example

```
if( isType("PhoneManagerCapsule") )
{
   log.log(“This capsule role is of type: “);
   log.log( getTypeName() );
}
```

RTActor::getCurrentStateString

const char * getCurrentStateString(void) const;

Return value

The name of the current state containing the executing segment.

RTActorClass

This class is created to represent the common external features (interface ports and capsule name) of each capsule in your model. Only one instance of a **RTActorClass** structure exists for all capsule instances. This way common information about the capsule class can be stored only once.

You can reference this capsule information object in detail level code, by referencing it by name.

Common usage

The RTActorClass object is commonly required when using the Frame::incarnate operation. When incarnating (creating a new capsule instance) you always have to specify which capsule class should be instantiated in an optional capsule role.

Below the first parameter is the capsule role (**RTActorRef**) and the second the capsule class (**RTActorClass**):

```
frame.incarnate( aCapsuleRoleName, ACapsuleClass );
```

You should not create new instances of **RTActorClass**, but you can reference existing class objects.

Objects of type **RTActorClass** cannot be passed as message data, but it is safe to pass an address within a process.

Operations

This class does not have any operations available, and is only used in conjunction with the frame service to refer to specific capsule classes for manipulating the dynamic structure of a model.

RTActorRef

The **RTActorRef** class maintains information about each capsule role in your model. For each capsule role in the structure of a capsule an attribute of this type is added to the RTActor subclass generated C++ capsule class.

You can reference this capsule role in detail-level code of the containing capsule by referencing the capsule role by name. There are basically only two reasons why you would want to directly access capsule roles:

1 To incarnate a capsule instance into a capsule role

For example to specify which role to incarnate a capsule you would use the name of the capsule role directly in the incarnate operation:

```
frame.incarnate( devices, Device );
```

Where **devices** is the target capsule role in which you want to incarnate a capsule of type **Device**.

2 To find the replication of a capsule role

It is also commonly useful to use the replication size of a replicated capsule role:

Operations

RTActorRef::size	Returns the replication size of a capsule role.

RTActorRef::size

int size(void) const;

Return value

The replication value of the capsule role.

Remarks

The operation returns the replication size whether or not there is a capsule instance currently incarnated at a specific slot or not.

RTActorId

The Frame service operations Frame::incarnate return an object of type **RTActorId** to identify a particular capsule instance. The **RTActorId** object instance is used as a handle to import the capsule into a plug-in capsule role, and to destroy or deport a capsule instance.

In a capsule that has a Frame SAP called “frame”, the capsule gets it’s RTActorId as follows:

RTActorId id = frame.me();

Note: **RTActorId** is a pointer to the capsule. **If the capsule is destroyed, the pointer is invalid and the functions that use it will crash.** It is important to guarantee, at the application level, that the capsule will not be destroyed.

Operations

RTActorId::isValid	Used to test whether the id contains a valid capsule reference.

RTActorId::isValid

int RTActorId::isValid(void) const

Return Value

Returns 0 (false) if the id refers to an invalid capsule instance, and 1 (true) otherwise.

Remarks

This operation should not be used to test for the state of a capsule instance (regardless of whether it is still alive). It should only be used immediately after a call to the Frame::incarnate operation. Once the capsule instance has been created, **isValid** always returns 1 (true), even if the capsule instance is subsequently destroyed.

Example

The example shows how the capsule instance id is checked after the incarnate operation. If the incarnation fails an error message is printed to the log, and if the incarnation is successful the capsule instance is immediately destroyed.

```
RTActorId capsule_id = frame.incarnate( terminal,
LongDistanceTerminal, RTTypedValue(), callThread, 0 );


if( ! capsule_id.isValid() )
{
   context()->perror("Incarnation failed: ");
}
else
   frame.destroy( capsule_id );
```

RTController

The RTController is an abstract class that defines the interface to a group of executing capsule instances within a single thread of concurrency. There is one controller object for each physical thread in the system. The controller object maintains information about the state of the thread as a whole, including the most recent error. Since the majority of operations in the Services Library return either 1 (true) if successful, and 0 (false) otherwise, the controller object can provide the precise cause of failure.

Refer to the error values description for a complete listing of the Services Library run-time errors.

Note: From within a capsule instance, you can retrieve a pointer to its controller by calling the RTActor::context operation. You can also use RTActor::getError to obtain the error value maintained by the controller.

Operations

RTController::abort	Terminates the current process.
RTController::getError	Returns the value of the most recent error within a particular thread.
RTController::name	Obtains the name of the controller (physical thread name).
RTController::perror	Prints a user-supplied error message along with the string for the current error as returned by getError.
RTController::strerror	Describes the current error code.

RTController::getError

Error getError(void) const;

Return Value

The value of the most recent error within the thread.

Remarks

The error code is not reset by a subsequent successful primitive operation call. It should be called immediately following the failure of a Services Library operation call.

Examples

See the example shown in the RTController error codes descriptions.

RTController::strerror

const char * strerror(void) const;

Return Value

A description of the current error code on the current RTController, that is, the controller for a physical thread.

Examples

See the example shown in the RTController error codes descriptions.

RTController::perror

void perror(const char * error_string = "error");

Parameters

error_string [optional]

The string to be printed to stderr along with the current error string as returned by the RTController::strerror operation. By default, the string "error" will be printed.

Example

```
if( ! aPort.ack().send() )
   context()->perror("Error sending ack");
```

Output

```
Error sending ack: Port not connected.
```

RTController::name

const char * name(void) const;

Return value

Returns the name of the controller. Controllers are named based on the physical thread on which they run. The assigned physical thread names are taken from the physical thread specification dialog. This method is a way of allowing capsules to find out what thread they are running on.

RTController::abort

void abort(void);

Remarks

Calling this operation on any controller will terminate the controller on which the capsule instance is running which in turn destroys all capsule instances running on that controller. Messages that have not been processed are deleted.

If this is called on the main thread then all threads are destroyed, and the process quits.

Examples

```
context()->abort();
```

Exception

The Exception Service, like other run-time system services, is accessed through an exception port. Exceptions manifest themselves in the form of Services Library messages arriving on appropriate Exception Service ports. Any capsule class that needs to raise or handle exceptions must define an exception port in its structure. Exception Service ports are instances of the class **Exception**.

Exceptions in Rose RealTime are defined as signals in the Exception Service Protocol class. The input signals are those exceptions that an application may handle or raise.

Exception signals

These are a set of valid signals that are defined on the exception protocol class. A special RTExceptionSignal is defined with the only action that is allowed on a **Exception** signal, that is to raise an error.

Exception Signals

RTExceptionSignal arithmeticError(const RTTypedValue &);

RTExceptionSignal error(const RTTypedValue &);

RTExceptionSignal notFoundError(const RTTypedValue &);

RTExceptionSignal notUnderstoodError(const RTTypedValue &);

RTExceptionSignal serviceAccessError(const RTTypedValue &);

RTExceptionSignal streamError(const RTTypedValue &);

RTExceptionSignal subclassResponsibilityError(const RTTypedValue &);

RTExceptionSignal timeSliceError(const RTTypedValue &);

RTExceptionSignal userError(const RTTypedValue &);

Return value

Returns 1 (true) if the operation was successful, and 0 (false) otherwise

Parameters

Remarks

The exception must be raised by the application. The Services Library does not automatically raise any exceptions by itself.

With **userError()**, you can provide any relevant data that is required to send along with the exception.

Examples

```
// How to handle service errors using the exception service

if( ! myPort.start().send() )
   ex.userError( RTString("Send on ring port failed.")).raise();
```

RTExceptionSignal

The **RTExceptionSignal** contains the raise operation that can be called on the signals defined within the Exception protocol class.

raise

The only action (operation) that can be called on an exception signal is to raise it.

Example

```
exception.userError(RTString("Send on ring port failed")).raise();
```

Frame

The Frame service is accessed via Frame service ports, declared in the structure of a capsule class. Frame service ports are instances of the class **Frame**. The operations take, as their parameters, either of:

- static capsule role names RTActorRef (design-time names of capsule roles), and capsule class names RTActorClass
- dynamic capsule instance idsRTActorId (generated at run time)

The **Frame** class also provides a number of query primitives that can be used to get information about the structure of the model. These functions may be useful in some circumstances, particularly for writing generic capsules that must deal with very dynamic structures.

Operations

Frame::classIsKindOf	Tests whether a particular capsule class is a subclass of another.
Frame::className	Finds the string name of a capsule class.
Frame::classOf	Determines the class of a given capsule instance.
Frame::deport	Removes a capsule instance from a plug-in capsule role.
Frame::destroy	Destroys an instance of an optional capsule.
Frame::import	Imports a capsule instance into a plug-in capsule role.

Frame::incarnate	Creates optional capsule role instances. This operation must be used to create and run capsule instances on different logical threads.
Frame::incarnationAt	Retrieves a particular capsule instance of a capsule role.

Frame::classIsKindOf

int classIsKindOf(const RTActorClass & ***child*****,**
const RTActorClass & ***parent*** **);**

Return value

The function returns 1 if parent is equal to or an ancestor of child, and returns 0 otherwise.

Parameters

child

Child is the name of the capsule class in question

parent

Parent is an capsule class which, if it is the same as, or a superclass of, the class in question, the method returns 1.

Frame::className

const char * className(const RTActorClass & ***capsule_class*** **)**

Return value

The operation returns the name of the specified capsule class in the form of a null-terminated string.

Parameters

capsule_class

Is the name of a capsule class.

Remarks

The Services Library stores run-time information about each capsule class in the model using a separate class (often referred to as a metaclass. The information is contained within a RTActorClass object. There is one object for each capsule class, having the same name as that of the class that it represents.

Frame::classOf

const RTActorClass & classOf(const RTActorId & ***instance*** **);**

Return value

The function returns the class of the specified actor. If an error occurs the EmptyActorClass is returned.

Parameters

instance

Capsule instance id

Frame::deport

```
int deport(    const RTActorId & instance,
               RTActorRef & role );
```

Return value

These functions return false (0) if an error occurred, and true (non-0) otherwise.

The operation can fail if:

- the capsule instance being removed was not present in the role.
- the role is not uniquely identified.

Parameters

instance

Is the id of the capsule instance to be removed.

role

Is the capsule role name from which the capsule instance is to be removed.

Frame::destroy

int destroy(RTActorId & capsule_instace);

int destroy(RTActorRef & capsule_role);

Return value

These functions return false (0) if an error occurred, and true (non-0) otherwise.

Parameters

RTActorId

Using destroy with the capsule id will destroy the capsule instance and all of its component capsule roles.

RTActorRef

Instead of destroying one capsule instance, you can destroy all instances represented by a capsule role. Any and all instances of this capsule role will be destroyed. The capsule role can only be destroyed by the immediate container of that role.

Examples

```
// receive capsule instance identifier from
// the capsule instance to destroy.
RTActorId cid = rtdata;
frame.destroy( cid );

// or you can destroy all instances by specifying the
// capsule role instead of a specific instance.
frame.destroy( terminal );
```

Frame::import

int import(**const RTActorId &** *instance***,**
RTActorRef & *dest***,**
nt *index* **= -1);**

int import(**RTActorRef &** *role***,**
RTActorRef & *dest***,**
int *index* **= -1);**

Return value

These functions return false (0) if an error occurred, and true (non-0) otherwise.

The operation fails in the following cases:

- if the capsule identified no longer exists.
- the class of the capsule instance is not a compatible class.
- a port of the instance that is bound in the imported capsule role is already bound elsewhere.
- the target capsule is not uniquely identified.

Parameters

instance

Is the instance id of the capsule instance which is to be imported.

dest

Is the name of the capsule role into which the capsule instance will be imported. The capsule role must be in the immediate decomposition frame within the calling capsule instance.

index

Is the replication index within the plug-in capsule role into which the capsule instance is to be imported. If unspecified, the capsule instance is imported into the first available index.

role

Using an alternate form of the operation, you can provide a capsule role name instead of the capsule instance id. To use this form of importation, the capsule role must not be replicated, and a valid capsule instance for this role must be active. This operation will import the capsule instance represented by the role (it cannot be a replicated role) into the destination capsule role.

Frame::incarnate

RTActorId incarnate(RTActorRef & *capsule_role* **);**

RTActorId incarnate(RTActorRef & *capsule_role* **,**
const RTActorClass & *capsule* **);**

RTActorId incarnate(RTActorRef & *capsule_role***,**
const void * data,
const RTObject_class * *type***,**
RTController * *log_thread***,**
int *index***);**

RTActorId incarnate(RTActorRef & *capsule_role***,**
const RTActorClass & capsule,
const void * *data***,**
const RTObject_class * *type***,**
RTController * *log_thread***,**
int *index***);**

RTActorId incarnate(RTActorRef & *capsule_role***,**
const RTActorClass & *capsule_class***,**
const RTDataObject & *rtdata***,**
RTController * *log_thread* **= 0,**
int index = -1);

RTActorId incarnate(RTActorRef & *capsule_role***,**
const RTDataObject & *rtdata***,**
RTController * *log_thread* **= 0,**

int *index* **= -1);**

RTActorId incarnate(**RTActorRef &** *capsule_role*,
const RTActorClass & *capsule_class*,
const RTTypedValue & *info*,
RTController * *log_thread* **= 0,**
int *index* **= -1);**

RTActorId incarnate(**RTActorRef &** *capsule_role*,
const RTTypedValue & info,
RTController * *log_thread* **= 0,**
int *index* **= -1);**

Return value

The operations returns a valid RTActorId if the operation is successful. To test if the returned operation failed, use the RTActorId::isValid operation on the returned object. For example:

```
RTActorId ind = frame.incarnate( mySubcapsule );
if (ind.isValid() )
...//use index
else
...//getError() to see what's wrong, don't use index
```

If the operation fails you can use the RTActor::getError operation to find out why the operation failed.

Parameters

There are alternate forms of the incarnate which leave out the RTActorClass parameter (defaulting to the class specified for the capsule role) and for sending different types of initialization data to the new instance, either as RTDataObject classes or anything with a descriptor.

capsule_role

Is the name of the optional capsule role contained in the structure of the capsule instance making the incarnate call.

capsule_class [optional]

Is the name of the class that should be instantiated into the optional capsule role. If absent, the incarnated class defaults to the class of the capsule role. You can also use the predefined variable **EmptyActorClass** to specify that the incarnated class default to the class of the capsule role.

data, type, rtdata, info [optional]

Is the data to be sent to the created capsule instance. The data sent is accessible in the capsule instances initial transition. Be sure to specify if no data is to be sent. See the examples below.

thread

Is the name of the logical thread (given in the thread configuration dialog) where you want the incarnated capsule instance to run. If no thread is specified the capsule instance is incarnated in the thread of the caller.

index

Is the replication index into which the new capsule instance should be incarnated. This is only valid when incarnating capsule instances into replicated capsule roles. Indexing begins at 0, that is index 0 is the first capsule instance. If specified as -1, capsule will be incarnated in first free slot.

The first free slot is the last slot number that was made available after a capsule was deleted. For example, you create and then deleted capsules 0, 1, 2, 3 in this order. The list of free slots is as follows in order: 3, 2, 1, 0, 4, 5, 6, 7, 8, 9.

Remarks

To use the frame operations, create a private end port using the Frame protocol.

Examples

This will incarnate a capsule instance into the optional capsule role named 'terminal'. No initialization data is sent to the capsule instance. The incarnated class defaults to the class of the capsule role:

```
RTActorId capsule_id;
capsule_id = frame.incarnate( terminal );

if( ! capsule_id.isValid() )
   context()->perror("Incarnation failed: ");
```

If you want to incarnate the incarnated class of the capsule role and send initialization data you can specify the EmptyActorClass variable as the second argument:

```
RTActorId capsule_id;
ControlData data( 15, 8.98 );

capsule_id = frame.incarnate( terminal, EmptyActorClass, &data,
&RTType_ControlData, (RTController *)0, -1 );
```

This will incarnate a capsule instance into the terminal optional capsule role at index 0, with initialization data, on a specific logical thread.

```
RTActorId capsule_id;
PrinterData data( 14, “ott05” );

capsule_id = frame.incarnate(
             device, // capsule role name
             Printer, // capsule
             &data, // initialization data
             &RTType_PrinterData, // type descriptor
             callThread, // logical thread name
             0 // index
           );

if( ! capsule_id.isValid() )
   context()->perror("Incarnation failed: ");
```

The following could be used to incarnate a capsule instance without initialization data, but with a specific logical thread or a replication index:

```
RTActorId capsule_id;
PrinterData data( 14, “ott05” );

capsule_id = frame.incarnate(
   device,
   Printer,
   (const void *) 0, // initialization data
   (const RTObject_class *) 0, // type descriptor
   PrintThread, // logical thread
   0 // replication index
   );

if( ! capsule_id.isValid() )
   context()->perror("Incarnation failed: ");
```

Frame::incarnationAt

RTActorId incarnationAt (const RTActorRef & ***role*****, int** ***index*** **);**

Return value

The instance id of the capsule instance at the specified capsule role index.

Parameters

role

Capsule role name for which you want to find a particular instance.

index

Index of the desired capsule instance. The index is zero-based.

RTInSignal

This class is used to work with incoming signals defined within a protocol. As explained in RTProtocol, each signal defined on a protocol becomes an operation. For incoming signals the operations return an **RTInSignal** object on which you can specify what action to perform with the signal.

The only actions defined on incoming signals are to manipulate the defer queue, that is to retrieve specific messages that have been deferred.

For example if a message was deferred at some point in a capsules behavior:

```
getMsg()->defer();
```

You can recall the specific signal by calling:

```
aPort.ack().recall();
```

Operations

RTInSignal::purge	Delete all of these deferred signals for all port instances.
RTInSignal::purgeAt	Delete all of the deferred signals on a specific port instance.
RTInSignal::recall	Recall one deferred signal on all port instances.
RTInSignal::recallAll	Recall all deferred signals on all port instances.
RTInSignal::recallAllAt	Recall all deferred signals on a specific port instance.
RTInSignal::recallAt	Recall one deferred signal on a specific port instance.

RTInSignal::purge

int purge(void);

Return value

Returns the number of deleted messages from the defer queue.

Remarks

If a port is replicated then the purge operation will delete all deferred signals on all port instances.

RTInSignal::purgeAt

int purgeAt(int ***index*** **);**

Return value

Returns the number of deleted messages from the defer queue.

Parameters

index

Port instance index on which to delete deferred messages.

Remarks

If a port is replicated then this operation will delete deferred signals on only the specified port instance.

RTInSignal::recall

int recall(int front = 0);

Return value

Returns the number of recalled messages.

Parameters

front [optional]

Front is a boolean int that indicates whether the message should be recalled to the front of the system message queue. If false or left unspecified, the message is sent to the back of the message queue. By recalling to the front, it is possible to avoid overtaking of messages.

Remarks

There is no time-limit on deferral. Applications must take precautions against forgetting messages on defer queues.

This operation recalls the first deferred message of this signal type on any port instance. To recall the first message of any signal type then use the RTProtocol::recall operation.

RTInSignal::recallAt

int recall(int *index*, int *front* = 0);

Return value

Returns the number of recalled messages.

Parameters

index

Port instance index on which to recall a deferred message.

front [optional]

Front is a boolean int that indicates whether the message should be recalled to the front of the system message queue. If false or left unspecified, the message is sent to the back of the message queue. By recalling to the front, it is possible to avoid overtaking of messages.

Remarks

There is no time-limit on deferral. Applications must take precautions against forgetting messages on defer queues.

This operation recalls the first deferred message of this signal type on a specific port instance. To recall the first message of any signal type then use the RTProtocol::recallAt operation.

RTInSignal::recallAll

int recallAll(int *front* = 0);

Return value

Returns the number of recalled messages.

Parameters

front [optional]

Front is a boolean int that indicates whether the message should be recalled to the front of the system message queue. If false or left unspecified, the message is sent to the back of the message queue. By recalling to the front, it is possible to avoid overtaking of messages.

Remarks

There is no time-limit on deferral. Applications must take precautions against forgetting messages on defer queues.

This operation recalls ALL deferred messages of this signal type on ALL port instances. To recall all messages of any signal type, use the RTProtocol::recallAll operation.

RTInSignal::recallAllAt

int recallAllAt(index *index*, int *front* = 0);

Return value

Returns the number of recalled messages.

Parameters

index

Port instance index on which to recall all deferred messages.

front [optional]

Front is a boolean int that indicates whether the message should be recalled to the front of the system message queue. If false or left unspecified, the message is sent to the back of the message queue. By recalling to the front, it is possible to avoid overtaking of messages.

Remarks

There is no time-limit on deferral. Applications must take precautions against forgetting messages on defer queues.

This operation recalls ALL deferred messages of this signal type on a specific port instance. To recall all messages of any signal type, use the RTProtocol::recallAllAt operation.

Log

The System Log is accessed via ports using the **Log** protocol, which are instances of the class **Log**. The log port only takes incoming messages and does not pass any information in the reverse direction. The operations available for accessing the system log are listed below.

Note: Currently all log service output is directed to stderr. Meaning that the open(), clear(), and close() operations should not be used

Operations

Log miscellaneous operations	Various utility operations.
log Log::show and Log::log	Writes an object as an ASCII string to the log with a trailing carriage return.
show Log::show and Log::log	Writes an ASCII string to the log with no leading or trailing carriage returns.

Log::show and Log::log

- **void show(const char * data);**
- **void show(char data);**
- **void show(double data);**
- **void show(float data);**
- **void show(int data);**
- **void show(long data);**
- **void show(short data);**
- **void show(unsigned data) ;**
- **void show(ushort data) ;**
- **void show(ulong data) ;**
- **void show(const RTDataObject & data);**
- **void show(const void * data, const RTObject_class * type);**
- **void show(const RTTypedValue & data);**
- **void log(const char *);**
- **void log(const RTString &);**
- **void log(char);**
- **void log(double);**
- **void log(float);**
- **void log(int);**
- **void log(long);**
- **void log(short);**
- **void log(const RTDataObject &);**
- **void log(const void *, const RTObject_class *);**
- **void log(const RTTypedValue &);**

Parameters

data, type

Is the object, type information, or simple type that is to be displayed to the log.

Remarks

The log knows how to display simple types, but it can also display any user-defined type as well. To display a user-defined type, it must have type information defined with a function to encode the object. The log will simply call this encode function.

The only difference between the log() and show() operations is that log() outputs a carriage return after the data is output to the log.

Examples

```
// Print as an ASCII string the contents of a class
log.show( &SubscriberData, &RTType_SubscriberData );

// Print a string
log.show( "Timer has expired" );

// Print an int
log.show( 19 );
```

Log Miscellaneous Operations

void cr(void);

void crtab(int ***num_tabs*** **= 1);**

void space(void);

void tab(void);

void commit(void);

Parameters

num_tabs

Is the number of tabs to insert, the default is one (1). Tab settings are defined by the system and cannot be altered by the user.

Remarks

These are various operations that can be used to output predefined characters to the log. commit()will output all buffered characters in the log.

Examples

```
log.cr();
log.space();
log.tab();
log.commit();

// The previous commands can be supplied using show()
log.show( "\n \t" );
log.commit();
```

RTMessage

This class is the data structure used within the Services Library to represent messages that are communicated between capsule instances. The messages that are sent between capsules contain a required signal name (which identifies the message), a optional priority (relative importance of this message compared to other unprocessed messages on the same thread - default to General), and optional application data.

You will most often use the operations on the **RTMessage** class to manipulate the messages that trigger transitions.

Do not treat an **RTMessage** as an object that can be stored, instead, you should extract the relevant information from the message and store it separately.

Note: Applications should treat the **msg** field of an RTActor and all data addressed beyond that pointer as read-only.

Operations

RTMessage::defer	Defer the current message against the receiving ports defer queue.
RTMessage::getData	Returns a pointer to the data that was sent along with a message.

RTMessage::getPriority	Returns the priority of the message.
RTMessage::getSignalName	Returns the name of the message signal.
RTMessage::getType	Returns a pointer to the type information describing the data contained within the message.
RTMessage::isValid	Determines if the message contains a valid signal and data.
RTMessage::sap	Retrieves a pointer to the port which received the message.
RTMessage::sapIndex0	Finds the index of the port on which the message was received (0 based).

RTMessage::getPriority

int getPriority(void) const;

Return value

Returns the numeric value of the priority of the message.

RTMessage::getSignalName

const char * getSignalName(void) const;

Return value

Returns the name of the signal that was sent with the message. This name will be the same as the name of the signal defined in the protocol.

Example

```
log.show(“Signal named: “);
log.log(getMsg()->getSignalName());
```

RTMessage::getData

void * getData(void) const;

Return value

Returns the pointer to the data that was sent along with a message.

Remarks

It is recommended to use the predefined **rtdata** parameter to access the data of a message in a transition. The **rtdata** parameter is already casted for you

```
const ADataType & dt = *rtdata;
```

Examples

In cases where there are multiple triggers for a transition you will have to cast the received data depending on the signal that triggered the transition.

```
const ADataType & dt2 = *(ADataType *)(getMsg()->getData());
```

RTMessage::getType

const RTObject_class * getType(void) const;

Return value

Returns a pointer to an RTObject_class which contains the type information that describes the data in the message, or (RTObject_class *)0 if not type was specified.

RTMessage::sapIndex0

int sapIndex(void) const;

int sapIndex0(void) const;

Return Value

Returns the index of the port on which the message was received. The **sapIndex** function returns a one-based index (index values begin at 1) and **sapIndex0** is 0 based.

Example

Use to send a message to a particular port instance, as follows:

```
int idx = msg->sapIndex0();
port.hello().sendAt( idx );
```

RTMessage::sap

RTOutSignal * sap(void) const;

Return Value

Returns a pointer to the port instance on which this message was received, or (RTProtocol *)0 if called in the initial transition.

Examples

```
// find out where the message was received, and send a message
// back.
RTProtocol * port = msg->sap();
if( port != (RTProtocol *)0 )
((MyProtocol::Base *)port)->hello().send();
```

RTMessage::isValid

int isValid(void) const;

Return Value

Returns 1 (true) if the message has been initialized with a valid signal and potentially some data, and 0 (false) otherwise.

Remarks

This method is intended to verify that the returned message has been properly filled by the reply to a RTOutSignal::invoke() operation call.

Examples

See RTOutSignal::invoke for an example.

RTMessage::defer

int defer(void) const;

Return value

Returns true (1) if the message was successfully deferred and false (0) otherwise. An error will be returned if you try and defer an invoked message or a message which has already been deferred.

Remarks

Deferred messages can be recalled using the operations defined on the RTInSignal class.

Examples

In the transition where a message is to be deferred you would defer the message as follows:

```
getMsg()->defer();
```

RTObject_class

The **RTObject_class** is a structure that contains information describing a data type. These type descriptors may be generated automatically for any class created in the toolset. The Services Library uses the information in the descriptors to initialize, copy, destroy, encode, and decode objects of the corresponding type.

Using type descriptors has several advantages:

- Arbitrary structures can be used in models even if they cannot be expressed in the toolset or are provided by third-parties.
- Encoding and decoding can be extended to arbitrary data structures.
- More efficient handling of data is possible by avoiding memory allocation and de-allocation. By adding the size to the type descriptor, the Library Services can decide when a payload area of a message is large enough to hold the data to be sent.
- Any user-defined type can be sent (by value), using the copy, and destroy functions in the type descriptor, and inspected via the observability interface using the init, encode, and decode functions.

The important thing to remember is that the toolset will generate these descriptors for most classes which are defined using basic types (see the list defined in the `RTObject_class.h` file located in RoseRT/C++/TargetRTS/include). If classes contain more complicated structures you can write your own type descriptor functions from within the toolset. For further details, see *C++ TargetRTS Properties* on page 195.

```
// A type is described by one of these structures.
//
// Field            Meaning
// -----            -------
// _super           The base type of this type
// _name            The name of this type
// _version         The version of this type
// _size            The byte size of this type (sizeof)
// _init_func       The default constructor for this type
// _copy_func       The copy constructor for this type
// _decode_func     The decode function for this type
// _encode_func     The encode function for this type
// _destroy_func    The destructor for this type
// _num_fields      The number of fields or array elements
// _fields          The field types or array element type
```

When would you use the type descriptor?

Whenever data is passed to the Services Library, you need to provide the type descriptor, along with the data to be sent. If the type descriptor is not provided to the Services Library, data objects will not be observed with the debugger, or sent to another process.

RTType_<typename> structure

For every generated class in your model there is a type descriptor created which is called RTType_<typename>. For example, if you define a class called RobotControlData the generated type descriptor would be:

```
const RTObject_class RTType_RobotControlData;
```

You can provide the generated type descriptor for a generated class to any Service Library operation that requires it.

RTOutSignal

This class is used to work with outgoing signals defined within a protocol. As explained in RTProtocol, each signal defined on a protocol becomes an operation. For outgoing signals the operations return an **RTOutSignal** object on which you can specify what action to perform with the signal. For example to send a signal first call initialize the **RTOutSignal** by calling the operation on the port then specify an action to perform with the signal:

```
port.ack().send();
port.hello( 1089 ).sendAt(1);
```

Operations

RTOutSignal::invoke	Synchronous message broadcast to all port instances.
RTOutSignal::invokeAt	Synchronous message send to a specific port instance.
RTOutSignal::reply	Used to respond to a synchronous message.
RTOutSignal::send	Asynchronous message broadcast to all port instances.
RTOutSignal::sendAt	Asynchronous message send to a specific port instance.

RTOutSignal::send

int send(int *priority* = General) const;

Return value

The operation returns a count of the successful sends (remember that ports can be replicated in which case this operation will broadcast to all port instances). A send can fail if the port is not connected (no connection to the receiver end port).

Parameters

priority [optional]

Specify the priority at which this message should be sent. A message priority is interpreted as the relative importance of an event with respect to all other unprocessed messages on a thread. The priority evaluates to one of the defined global priority values.

Remarks

Since a port can be replicated, the send operation effectively sends a message through all instances of the port - broadcast. If you want to send to only one instance of a replicated port, use the RTOutSignal::sendAt operation.

Examples

```
// In this case the ack signal does not require
// data to be sent with the signal.
aPort.ack().send();
```

It is always good practice to check the return codes.

```
if( ! aPort.ack().send() )
   context()->perror("Error with send");
```

You can also send data with a message.

```
// Sending some data by value, that is a copy of the data is
// sent.
SomeDataClass mdata("123-4356", "Ottawa");
aPort.Info( mdata ).send();
```

RTOutSignal::sendAt

int sendAt(int *index*, int *priority* = General) const;

Return value

The operation returns 1 (true) if the operation succeeded and 0 (false) otherwise. A send can fail if the port is not connected (no connection to the receiver end port) or an invalid replication index was provided.

Parameters

index

The port replication index of the port instance on which the message should be sent.

priority [optional]

Specify the priority at which this message should be sent. Default is General. A message priority is interpreted as the relative importance of an event with respect to all other unprocessed messages on a thread. The priority evaluates to one of the defined global priority values.

Remarks

This operation is used instead of **RTOutSignal::send** to send a message to a specific instance of a replicated port.

Examples

```
// In this case the ack signal does not require
// data to be sent with the signal.
aPort.ack().sendAt(5);

// Send to a specific port instance
aPort.ack().sendAt( 1 );

// send to the port instance on which the current
// message was received
int idx = getMsg()->sapIndex0();
rtport->ack().sendAt( idx );
```

RTOutSignal::invoke

int invoke(RTMessage * *replyBuffers* **) const;**

Return value

The operation returns the number of replies received. If it returns 0 (false), the operation failed. An invoke can fail if the port is not connected (no connection to the receiver end port). An error will be returned if you try and invoke across a physical thread boundary.

Parameters

replyBuffer

A user-supplied message object that stores the reply message resulting from the invoke. The user is responsible for allocating and deleting the message when it is no longer required. Typically a local variable will be declared to hold the returned message. To verify that the returned message is valid call RTMessage::isValid once the invoke returns.

Remarks

If a port is replicated all port instances will be invoked. Use RTOutSignal::invokeAt to invoke a specific port instance.

The communications services also support synchronous messaging (similar to a rendezvous). During a synchronous send, or invoke, the sender is blocked until the receiver has processed the message and sent back a reply. Run-to-completion semantics are enforced, such that a synchronous invoke has the same semantics as a procedure call. Note that the data field is not copied on invoke.

Note: Do not use invoke in the initial transition of a capsule as the system may still be processing initialization messages. Also, because of its blocking nature, invoke cannot be used across threads or capsules connected through a network.

Examples

```
RTMessage replies[ aPort.size() ];
aPort.ack().invoke( &replies );
for( int i = 0; i < aPort.size(); i++ )
{
   if( replies[i].isValid() )
   {
      //code to handle valid reply
   }
   else
   {
      //code to handle invalid reply
   }
}
```

The receiver of the invoke must use RTOutSignal::reply to respond to the invoke. Data can be optionally sent back with the reply.

```
rtport->nack().reply();
```

RTOutSignal::invokeAt

int invokeAt(int *index*, RTMessage * *replyBuffer*) const;

Return value

The operation returns 1 (true) if the invoke is successful and 0 (false), if the operation failed. An invoke can fail if the port is not connected (no connection to the receiver end port). An error will be returned if you try and invoke across a physical thread boundary.

Parameters

index

The port replication index of the port instance on which the message should be sent.

replyBuffer

A user-supplied message object that stores the reply message resulting from the invoke. The user is responsible for allocating and deleting the message when it is no longer required. Typically a local variable will be declared to hold the returned message. To verify that the returned message is valid call RTMessage::isValid once the invoke returns.

Remarks

The communications services also support synchronous messaging (similar to a rendezvous). During a synchronous send, or invoke, the sender is blocked until the receiver has processed the message and sent back a reply. Run-to-completion semantics are enforced, such that a synchronous invoke has the same semantics as a procedure call. Note that the data field is not copied on invoke.

Note: Do not use invoke in the initial transition of a capsule as the system may still be processing initialization messages. Also, because of its blocking nature, invoke cannot be used across threads or capsules connected through a network.

Examples

```
RTMessage reply;
aPort.ack().invokeAt( 0, &reply );
if( reply.isValid() )
{
   // code to handle valid reply
}
```

```
else
{
   // code to handle invalid reply
}
```

The receiver of the invoke must use RTOutSignal::reply to respond to the invoke. Data can be optionally sent back with the reply.

```
rtport->nack().reply();
```

RTOutSignal::reply

int reply (void);

Return Value

Returns 1 (true) if the reply is successful, and 0 (false) otherwise.

Examples

The receiver of the invoke must use RTOutSignal::reply to respond to the invoke. Data can be optionally sent back with the reply.

```
rtport->nack().reply();
```

RTProtocol

For each protocol class in your model, two subclasses of the **RTProtocol** class are generated for each direction of the protocol or protocol roles. Each port defined on a capsule is generated as an attribute of the generated C++ capsule class. The port attribute has the same name as the port, with the type as either the base or conjugate protocol role.

For an example of the code generated for a protocol, and protocol roles, see *Protocols Become Two Classes: Base and Conjugate* on page 103.

A **RTProtocol** instance (port) class contains a list of all the individual instances of that port.

Operations

RTProtocol::bindingNotification	Use this operation to request notification of the creation and destruction of bindings to instances of this port.
RTProtocol::bindingNotificationRequested	Use this operation to request status of notification for this port.
RTProtocol::deregisterSAP	Deregisters an unwired end port (SAP).
RTProtocol::deregisterSPP	Deregisters an unwired end port (SPP).
RTProtocol::getRegisteredName	Get the name that an un-wired port has registered with the layer service.
RTProtocol::indexTo	Find the smallest replication index (0-based) which is connected to the given actor.
RTProtocol::isBoundAt	Is the given replication index (0-based) connected?
RTProtocol::isIndexTo	Is the given replication index (0-based) connected to the actor?
RTProtocol::isRegistered	Find out if an unwired port has been registered with the layer service.
RTProtocol::purge	Empties the defer queue of all port instances without recalling any deferred message.
RTProtocol::purgeAt	Empties the defer queue of a specified port instance recalling any deferred messages.
RTProtocol::recall	To recall a deferred message on all instances of this port for processing. Recalls from back of queue.
RTProtocol::recallAll	To recall all deferred messages on all instances of this port for processing. Recalls from back of queue.
RTProtocol::recallAllAt	To recall all deferred messages on a specified port instance. Recalls to back or front of queue.
RTProtocol::recallAllFront	To recall all deferred messages on all port instances. Recalls to front of queue.
RTProtocol::recallAt	To recall a deferred message on specific instance of this port for processing. Can recall to back or front of queue.
RTProtocol::recallFront	To recall a deferred message on all instances of this port for processing. Recalls to front of queue.

RTProtocol::registerSAP	Registers a unwired end port (SAP) with the layer service (as a "client").
RTProtocol::registerSPP	Registers an unwired end port (SPP) with the layer service (as the "provider").
RTProtocol::size	Obtains the replication factor of a port.

RTProtocol::size

int size(void) const;

Return Value

Returns the multiplicity factor of the port.

Remarks

Remember that port instances are indexed in the Services Library as 0 based. That means that if a port has a cardinality of N, you should only reference instances using index numbers 0..N-1.

```
for( int i = 0 ; i < port.size(); i++ )
  port.ack().sendAt( i );
```

RTProtocol::purge

int purge(void);

Return Value

Returns the number of messages deleted from the defer queue.

Remarks

To delete deferred messages for one port instance use RTProtocol::purgeAt.

RTProtocol::purgeAt

int purgeAt(int *index*);

Parameters

index

The port index for which deferred messages should be purged.

Return Value

Returns the number of messages deleted from the port instance defer queue.

Remarks

To delete deferred messages for all port instance use RTProtocol::purge.

RTProtocol::recall

int recall(void);

Return Value

Returns the number of messages recalled from the defer queue (either 0 or 1).

Remarks

Calling recall on a port gets the first deferred message from one of the port instances. Messages are recalled behind other queued messages.

There is no time-limit on deferral so that applications must take precautions against forgetting messages on defer queues.

This operation recalls the first deferred message on any port instances. To recall the first message on one port instance of a replicated port, use the RTProtocol::recallAt operation.

Examples

The first deferred message on any instance of the replicated port named port1 is recalled as follows:

```
port1.recall();
```

RTProtocol::recallAt

int recallAt(int *index*, int *front* = 0);

Return Value

Returns the number of recalled messages (either 0 or 1).

Parameters

index

Port instance index for which to recall a deferred message.

front

Specifies whether recalled messages should be queued ahead (non-zero) or behind (0) other queued messages.

Examples

The first deferred message on any instance of the replicated port named port1 is recalled as follows:

```
port1.recallAt( 3 );
```

RTProtocol::recallFront

int recallFront(void);

Return Value

Returns the number of recalled messages (either 0 or 1).

Remarks

This operation recalls the first deferred message on any port instances. Calling recall on a port gets the first deferred message from one of the port instances, starting from the first (instance 0). Messages are recalled to the front main queue.

RTProtocol::recallAll

int recallAll(void);

Return Value

Returns the number of recalled messages.

Remarks

Calling recallAll on a port will get all the deferred message from each of the port instances. Messages will be recalled starting to the back of the main queue.

To recall all messages on only one port instances of a port with replication factor > 1, use the RTProtocol::recallAllAt operation.

RTProtocol::recallAllAt

int recallAllAt(int *index*, int *front* = 0);

Return Value

Returns the number of recalled messages.

Parameters

index

Port instance index for which to recall all deferred messages.

front

Specifies whether recalled messages should be queued ahead (non-zero) or behind (0) other queued messages.

Remarks

To recall all messages on only one port instances of a port with replication factor > 1, use the RTProtocol::recallAllAt operation.

RTProtocol::recallAllFront

int recallAllFront(void);

Return Value

Returns the number of recalled messages.

Remarks

To recall all messages on only one port instances of a port with replication factor > 1, use the RTProtocol::recallAllAt operation.

RTProtocol::bindingNotification

void bindingNotification(int *on_off*);

Parameters

on_off

If called with 1 (true) the port will receive messages as ports become bound. Calling the function with 0 (false) will prevent such messages from being sent, but will not purge any messages already queued.

Remarks

Use this operation to request notification of the creation and destruction of bindings to this port. The signals sent to the port by the Services Library are rtBound and rtUnbound.

Note: No messages are sent for ports which are bound prior to the call to this function.

RTProtocol::bindingNotificationRequested

int bindingNotificationRequested(void) const;

Return Value

Returns 1 (true) if notification has been enabled for this port, and 0 (false) otherwise.

RTProtocol::registerSAP

int registerSAP(const char * *service* **);**

Return Value

Returns 1 (true) if the registration of the service name was successful, and 0 (false) otherwise. The registration can fail if this operation is called on a port instance which is not an unwired end port. If this SAP is already registered with this same name, the operation returns 1.

Parameters

service

This parameter is a string that is used to identify a unique name and service under which SAPs and SPPs will connect.

Remarks

If this operation is invoked on a SAP which is already registered with a different name, then the original registered name is automatically deregistered, and the SAP is registered with the new name.

When a SAP is registered, it does not necessarily mean that the port has been connected to a SPP. The successful completion of the register operation simply indicates that the name has been registered. For example, if the SAP is registered with no corresponding SPP, the connection is only made later when a SPP is registered. The SAP registration is buffered until a SPP is registered with the same service name.

If application registration has been selected from the Port Specification dialog for a SAP (protected unwired end port) or SPP (public unwired end port) registration is handled automatically by the Services Library.

RTProtocol::deregisterSAP

int deregisterSAP(void);

Return Value

Returns 1 (true) if the deregistration of the service name was successful, and 0 (false) otherwise.

Remarks

When a SAP is deregistered if it is currently connected to a SPP, the connection is terminated.

RTProtocol::registerSPP

int registerSPP(const char * *service* **);**

Return Value

Returns 1 (true) if the registration of the service name was successful, and 0 (false) otherwise. The registration can fail if this operation is called on a port which is not an unwired end port. If this SPP is already registered with this same name, the operation returns 1.

Parameters

service

This parameter is a string that is used to identify a unique name and service under which SAPs and SPPs will connect.

Remarks

If this operation is invoked on a SPP which is already registered with a different name, then the original registered name is automatically deregistered, and the SPP is registered with the new name.

When a SPP is registered, it does **not** necessarily mean that the port has been connected to a SAP. The successful completion of the register operation simply indicates that the name has been registered. For example, if a SPP is registered with no corresponding pending SAP registrations, the connection will be made later when a SAP is registered. The SPP registration is buffered until a SAP is registered with the same service name.

If application registration has been selected from the Port Specification dialog for a SAP (protected unwired end port) or SPP (public unwired end port) registration is handled automatically by the Services Library.

RTProtocol::deregisterSPP

int deregisterSPP(void);

Return Value

Returns 1 (true) if the deregistration of the service name was successful, and 0 (false) otherwise.

Remarks

When a SPP is deregistered all connected port instances are disconnected from all connected SAPs. Although the SAPs are disconnected they remain registered, and available to be re-connected.

RTProtocol::isIndexTo

int isIndexTo(int *index*, RTActor * *capsule_instance*) const;

Return Value

Returns true (1) if the given port instance is bound to the specified capsule instance?

Parameters

index

A port instance index (0 based).

capsule_instance

A pointer to an capsule instance.

RTProtocol::indexTo

int indexTo(RTActor * *capsule_instance*) const;

Return Value

Find the smallest replication index (0-based) on the given port which is connected to the given capsule instance. The result is -1 if there is no such index or the id is invalid.

Parameters

capsule_instance

The capsule instance for which you are trying to find a port that is connected to it.

Example

This example demonstrates how to find the port index that is connected to a newly incarnated capsule role. In this example **port** is a replicated port and **role1** is a capsule role in the structure of a capsule on which this code is run:

```
RTActorId aid = frame.incarnate(role1);
int port_index;
if( aid.isValid() )
{
  port_index = port.indexTo( aid );
  if( port_index != -1)
    port.Signal().sendAt(port_index);
}
else
  context()->perror(“Error incarnating role1:”);
```

RTProtocol::isBoundAt

int isBoundAt(int *index*) const;

Return Value

Return true (1) if the given replication index (0-based) is connected to another port and false (0) if it is not connected.

Parameters

index

A port instance index (0 based).

RTProtocol::isRegistered

int isRegistered(void) const;

Return Value

Returns true (1) if an un-wired port has been registered with the layer service and false (0) otherwise.

RTProtocol::getRegisteredName

const char * getRegisteredName(void) const;

Return Value

Returns the name that an un-wired port has registered with the layer service.

RTSymmetricSignal

This class is used for symmetric signals defined within a protocol. Symmetric signals are defined where a signal is both incoming and outgoing in the same protocol and both have the same data class.

As explained in RTProtocol, each signal defined on a protocol becomes an operation.

Since symmetric signals can be both incoming and outgoing you can perform the combined actions of both RTOutSignal and RTInSignal on these classes.

Example

```
// to send the talk signal
port.talk.send();

// to recall all deferred talk signals
port.talk.recall();
```

RTTimerId

The Rose RealTime Timing services use **RTTimerId** as an identifier for timer requests. The timer identifier is returned by a request to Timing::informIn, Timing::informAt or Timing::informEvery. The timer identifier can be used subsequently to cancel the timer.

Operations

RTTimerId::isValid	Determines if a timer request was successful.

RTTimerId::isValid

int isValid(void);

Return value

Returns true (1) if the timer identifier is a valid timer id, and 0 (false) otherwise.

Remarks

This operation should be used to test the result of a timer request. This operation should not be used to test the state of a timer after the timer has been successfully started.

Examples

```
RTTimerId tid = timer.informIn( RTTimespec(4,0) );
if( ! tid.isValid() )
   context()->perror("Error requesting periodic timer");
```

RTTimespec

The RTTimespec class is used to create timer values for passing to the Timer Service. It is intended for compatibility with POSIX.

RTTimespec is a struct with two fields: **tv_sec** and **tv_nsec**, where tv_sec is the number of seconds for the timer setting, and tv_nsec is the number of nanoseconds.

Operations

RTTimespec assignment operators	Assignment operators
RTTimespec basic arithmetic operators	Arithmetic operators
RTTimespec basic comparison operators	Comparison operators
RTTimespec::getclock	Returns the current time
RTTimespec::RTTimespec	Constructs an RTTimespec object

tv_sec and tv_nsec

long tv_sec;

long tv_nsec;

Remarks

Where **tv_sec** is the number of seconds for the timer setting, and **tv_nsec** is the number of nanoseconds. There are 10e9 nanoseconds in one second.

Examples

This will initialize an RTTimespec with one second.

```
RTTimespec t1(1,0);
```

This class is used most often in conjunction with the Timing Service to specify time values. For example to set a one-shot timer to go off in 5 seconds you would use the RTTimespec constructor.

```
timer.informIn( RTTimespec(5,0) );
```

RTTimespec::RTTimespec

RTTimespec(void);

RTTimespec(long *sec***, long** *nsec***);**

RTTimespec(const RTTimespec & *ts* **);**

Parameters

sec

Is the number of seconds.

nsec

Is the number of nanoseconds.

ts

Initializes an RTTimespec with the value of another RTTimespec object.

Examples

A RTTimespec of two seconds can be created and passed to the Timing Service informEvery() as follows:

```
// 2 seconds
RTTimespec t( 2 , 0 );
// 6am coordinated universal time (UTC)
RTTime abst( 6,0,0 );

timer.informEvery( t );
timer.informAt( RTTimespec( abst ) );
```

RTTimespec::getclock

static void getclock(RTTimespec & *tspec* **);**

Parameters

tspec

The values of this RTTimespec parameter are filled in with the current time.

Remarks

This is a class-scoped operation.

Examples

```
RTTimespec t;
RTTimespec::getclock( t );
```

RTTimespec Basic Comparison Operators

int operator==(const RTTimespec & *t1***, const RTTimespec &** *t2* **);**

int operator!=(const RTTimespec & *t1***, const RTTimespec &** *t2* **);**

int operator<=(const RTTimespec & *t1***, const RTTimespec &** *t2* **);**

int operator> (const RTTimespec & *t1***, const RTTimespec &** *t2* **);**

int operator>=(const RTTimespec & *t1***, const RTTimespec &** *t2* **);**

int operator< (const RTTimespec & *t1***, const RTTimespec &** *t2* **);**

Return value

Nonzero if the objects meet the comparison condition; otherwise 0.

Parameters

t1, t2

RTTimespec objects used to compare.

Example

```
RTTimespec t1(2,0), t2(3,0);

if (t1 < t2)
   //t1 is less than t2
```

RTTimespec Assignment Operators

RTTimespec & operator=(const RTTimespec & ***t1*** **);**

Remarks

The RTTimespec assignment (=) operator re-initializes an existing RTTimespec object with new second and nanosecond values.

RTTimespec Basic Arithmetic Operators

RTTimespec & operator+=(const RTTimespec & ***t1*** **);**

RTTimespec & operator-=(const RTTimespec & ***t1*** **);**

RTTimespec operator+ (const RTTimespec & ***t1*****, const RTTimespec &** ***t2*** **);**

RTTimespec operator- (const RTTimespec & ***t1*****, const RTTimespec &** ***t2*** **);**

Parameters

t1, t2

RTTimespec objects used to add or subtract.

Examples

```
RTTimespec t1(2,0), t2;
RTTimespec::getclock( t2 );
t2 += t1;
```

Timing

All service methods through which the user requests timeout, at an absolute time or in a time interval, return a unique handle which can be used to construct an RTTimerId. The timer id is used to identify a timing request to be cancelled.

Timing ports are instances of the class Timing.

Operations

Timing system clock operations	Used to adjusts the internal real-time system clock.
Timing::cancelTimer	Cancels an outstanding timer.
Timing::currentTime	Determines the current absolute time.
Timing::informAt	Starts a timer which expires at a particular absolute time.
Timing::informEvery	Starts a periodic timer.
Timing::informIn	Starts a timer which expires in some time interval from the current time.

Timing::informAt

RTTimerNode * informAt(const RTTimespec & *when*, **const void *** *data*, **const RTObject_class *** *type*, **int** *prio* **= General);**

RTTimerNode * informAt(const RTTimespec & *when*, **int** *prio* **= General);**

RTTimerNode * informAt(const RTTimespec & *when*, **const RTDataObject &** *data*, **int** *prio* **= General);**

RTTimerNode * informAt(const RTTimespec & *when*, **const RTTypedValue &** *info*, **int** *prio* **= General);**

Return value

A timer handle is returned which can be used to construct an RTTimerId. It can be used to cancel the timer prior to expiry. A NULL pointer is returned if the timer request fails.

Parameters

when

Is the desired absolute time when the timer is to expire.

data, info, type [optional]

Is the message data that will be added to the timeout message and delivered to the capsule when the timer expires. These parameters are optional.

prio [optional]

Is the priority at which the timeout message will be delivered. The **prio** parameter is optional.

Example

```
RTTimespec now;
RTTimespec::getClock( &now );
timer.informAt( now + RTTimespec( 15, 0 ) );
```

Timing::informIn

RTTimerNode * informIn(const RTTimespec & *delta*, **const void *** *data*, **const RTObject_class *** *type*, **int** *prio* **= General);**

RTTimerNode * informIn(const RTTimespec & *delta*, **int** *prio* **= General);**

RTTimerNode * informIn(const RTTimespec & *delta*, **const RTDataObject &** *data*, **int** *prio* **= General);**

RTTimerNode * informIn(const RTTimespec & *delta*, **const RTTypedValue &** *info*, **int** *prio* **= General);**

Return value

A timer handle which can be used to construct an RTTimerId object is returned. It can be used to cancel the timer prior to expiry. A NULL pointer is returned if the timer request fails.

Parameters

delta

Represents the desired time interval (in seconds and nanoseconds as an RTTimespec) from the current time at which a timer will expire. If the timer interval is less than or equal to zero, the timer will expire immediately.

data, info, type [optional]

Is the message data that will be added to the timeout message and delivered to the capsule when the timer expires. These parameters are optional.

prio [optional]

Is the priority at which the timeout message will be delivered. The **prio** parameter is optional.

Examples

```
// request a timer to expire in 10 seconds
if( ! timer.informIn( RTTimespec( 10, 0 ) ) )
log.log("error requesting a timer");
```

If the timer is to be cancelled, then an RTTimerId must be constructed for use when canceling the timer. Ensure that if the timer is going to be cancelled in another transition, that the timer id is saved in a class attribute, and not in a transition local variable.

```
if( ! (tid = timer.informIn( RTTimespec( 10, 0 ))))
log.log("error requesting a timer");

// this could be done in an other transition
timer.cancelTimer( tid );
```

Passing Data in a Timer Message

Data can be passed in a timer message using the following informIn signatures:

RTTimerNode * informIn(const RTTimespec & delta, const void * data, const RTObject_class * type, int prio = General);

RTTimerNode * informIn(const RTTimespec & delta, const RTDataObject & data, int prio = General);

RTTimerNode * informIn(const RTTimespec & delta, const RTTypedValue & info, int prio = General);

To pass an `int` in a timer message, you can use one of two methods:

```
myint = 5;  // myint is an attribute of type int
RTTimespec t(2,0);
timer.informIn(t, &myint, &RTType_int);
```

Or

```
myint = 5;  // myint is an attribute of type int
RTTimespec t(2,0);
RTTypedValue typevalue(&myint, &RTType_int);
timer.informIn(t, typevalue);
```

This code can be easily modified to pass other data. For example, to send a class instead of an integer, replace `RTType_int` with `RTType_<`*`myClassName`*`>` where *`myClassName`* is the name of the class of the type of data to be passed.

Timing::informEvery

RTTimerNode * informEvery(const RTTimespec & *delta*, **const void *** *data*, **const RTObject_class *** *type*, **int** *prio* **= General);**

RTTimerNode * informEvery(const RTTimespec & *delta*, **const RTDataObject &** *data*, **int** *prio* **= General);**

RTTimerNode * informEvery(const RTTimespec & *delta*, **const RTTypedValue &** *info*, **int** *prio* **= General);**

RTTimerNode * informEvery(const RTTimespec & delta);

Return value

A timer handle which can be used to construct an RTTimerId is returned. It can be used to cancel the timer prior to expiry. A NULL pointer is returned if the timer request fails.

Parameters

delta

represents the desired time interval from the current time at which a periodic timer will expire. If the timer interval is less than or equal to the current time or equal to zero, the timer will expire immediately.

data, info, type [optional]

is the message data that will be added to the timeout message and delivered to the capsule when the timer expires. These parameters are optional.

prio [optional]

is the priority at which the timeout message will be delivered. The **prio** parameter is optional.

Examples

```
if( ! timer.informEvery( RTTimespec( 10, 0 ) )
log.log("error requesting a periodic timer");
```

If the timer is to be cancelled, then an RTTimerId object must be constructed for use when canceling the timer. Ensure that if the timer is going to be cancelled in another transition, that the timer id is saved in a class attribute, and not in a transition local variable.

```
if( ! (tid = timer.informEvery( RTTimespec( 10, 0 ))))
   log.log("error requesting a periodic timer");

// this could be done in an other transition
timer.cancelTimer( tid );
```

Timing::currentTime

RTTimespec currentTime(void) const;

Remarks

It is recommended, for performance reasons, that you use the RTTimespec::getclock operations instead of currentTime.

Example

```
RTTimespec ctime = timer.currentTime();
```

Timing::cancelTimer

int cancelTimer(RTTimerId &*tid*);

Return value

Returns true (1) if a pending request was cancelled, and false (0) if no outstanding request was found.

Parameters

tid

Is the identifier of the timer that was provided when the service request was made.

Remarks

Note that this operation guarantees that no timeout message will be received from the cancelled timer, even if the timer had already expired, that is, was waiting to be processed, when the command was issued.

Examples

If timer is the name of a timing port, you can create, and subsequently delete, a timing request as follows:

```
RTTimerId tid = timer.informEvery( RTTimespec( 2, 0 ) );
timer.cancelTimer( tid );
```

Timing System Clock Operations

void adjustTimeBegin(void);

void adjustTimeEnd(const RTTimespec & ***delta*** **);**

Parameters

delta

Stop all timing services in preparation for adjusting the clock time used by the timing service.

Remarks

If there is a need to adjust system time, you must stop the timing service, compute the new time, and then restart the timing service with the new time. The Services Library will make adjustments to its internal data structure so that relative timeouts are not affected by the system clock change. Use **adjustTimeBegin** to stop the timing service and **adjustTimeBegin** to restart it at the new time.

Examples

The application must coordinate time changes through a capsule with a timing port (meaning that it has access to the timing service). The example below encapsulates the clock adjustment behavior in a capsule operation. We assume that the timing port is called timer and that there are operating system primitives for reading and writing to the system clock which we term as sys_getclock() and sys_setclock(), respectively.

```
void AdjustTimeCapsule::setClock( const RTTimespec & new_time )
{
   RTTimespec old_time;
   RTTimespec delta;

   // Stop Services Library timer service
   timer.adjustTimeBegin();

   sys_getclock( old_time );
   sys_setclock( new_time );

   delta = new_time;
```

```
    delta -= old_time;

    // Resume Services Library timer service
    timer.adjustTimeEnd( delta );
}
```

RTTypedValue

RTTypedValue is a struct which is used to encapsulate a data and type pair. For each generated class a structure named RTTypedValue_<class name> is generated. The only time you will have to use this structure is when sending subclass data with a signal that was defined with a data class of the parent class. For example, given class A and a subclass B, with the signal ack defined with a data class of A, you would have to use to following syntax to send B with the ack signal:

```
B subclass;
port.ack(RTTypedValue_A(subclass,&RTType_B)).send();
```

If you do not explicitly specify the type descriptor for class B, the Services Library will use the type descriptor for class A.

Port Services

Note: For additional information on the External Service Example, see the "C++ Examples" chapter in the book *Model Examples, Rational Rose RealTime*.

External Port Service

The External Port service example provides an API that lets non-Rational Rose RealTime threads call a function to raise an event on a port of a Capsule in a Rational Rose RealTime C++ application.

External ports are instances of the class External

Table 5 Operations

External API Operations	**Used to enable/disable events external events**
External::enable	Enables the port to receive an event from the external thread. May be used only by the thread on which the owner capsule executes
External::disable	Disables the port from receiving an event from the external thread. May be used only by the thread on which the owner capsule executes
Extenal::raise	If the port is enabled, delivers one event to the port, and then disables the port. The port must be re-enabled before another event can be raised. This function may be used only on threads other than the one on which the owner capsule executes. Returns zero if the event was not successfully raised.

Example

Given an External port named external:

```
//Enable the external port to receive events
external.enable();
//Disable the external port to receive events
external.disable();
And from the external thread
if (theExternalPort->raise()==0){
   //fail
}
else {
   //pass
}
```

Index

Symbols

go 139
step 139
#define
  creating 84
#define_macro name_expression 84
#include 90
#include An_External.h 91
$RTS_HOME 141
_processFunc() 156
_waitFunc 156
_wakeupFunc 156

A

absolute time 113
adding
  operations and classes 172
aggregate 80
  attributes 80
  See also by reference
aggregations
  types 80
alreadyDeferred 119
application registration
  of SAPs and SPPs 111
application registrations 111
argCount() 131
array data member 81
arrays
  creating 81
  multiplicity 81
  specifying 81
ASCII encoder 77
assignment operator
  setting as private 192
AssignmentOperatorInline (Class, C++) 186
AssignmentOperatorVisibility (Class, C++) 186
association ends
  properties 80
Association Specification dialog box 80
AssociationEnd 195
AssociationEndKind (Role, C++) 189
associations
  between capsules and classes 79
  between two classes 79
asynchronous communication 107
attach 132
attach
  thread commands 133
Attach Target 127
Attribute Specification dialog box 84
Attribute, C++ TargetRTS 87
AttributeKind 82, 84
AttributeKind (Attribute, C++) 187
attributes
  aggregate 80
  array 81
  composite 80
  creating arrays 81
  creating pointers 81
  pointer 81
  pointers 86, 87
Attributes Detail 82
automatic registration 111
  SAPs and SPPs 111

B

BackwardsCompatible (Protocol, C++ TargetRTS) 203
badActor 119
badClass 119
badId 120
badIndex 120
badRef 120
badSignal 120
badValue 120

Binary files 68
bindings
  cardinality 83, 85
blocking functions 154
boundaries
  process 74
  threads 74
build (C++) 54
build options
  changing 170
building
  Services Library 175
BuildLibraryArguments (Component, C++ Library) 228
BuildLibraryCommand (Component, C++ Library) 228
by reference
  examples 79
  sending data 74
by value
  sending 86
  sending data 74

C

C++
  performance issues 163
C++ classes
  externally defined 88
C++ constants
  creating 82
  using 83
C++ constructs
  examples 81
C++ data types
  examples 77
C++ Executable Properties 223
C++ Generation Properties 215
C++ Library Properties 227
C++ macros 84, 85
  defining complex 85
C++ Services Library 74, 86
  configuring 167
  customizing 167
  integrating external IPC mechanisms 152
  naming considerations 99
  processing overview 95
C++ Services Library class reference 233
C++ Services Library Framework 93
  graphic overview 100
C++ Services Library framework 99
C++ struct
  creating 86
C++ tab 84
C++ Target RTS 173
C++ TargetRTS 76
C++ TargetRTS Properties 195
C++ TargetRTS properties 195
CallFromDestructor (Operation, C++) 189
CALLSUPER 35
Cancel 138
cancelTimer 120
capsule functions 162
capsule instances
  placeholders 104
Capsule Role Specification dialog box 83
capsule roles
  cardinality 83, 85
capsuleId 131
system 134
capsules
  associations with classes 79
  sending messages 73
  subclasses of RTActor 101
capsules versus data 161
Cardinality 83, 85
cardinality 83, 85
Changeability 82
class
  constant 83
Class C++ 90, 91
Class C++ Properties 83
Class C++ Properties tab 86
Class C++ tab 85
Class C++ TargetRTS 90, 91
class creation
  examples 86
class modeling
  examples 79

Class Properties 88
class scoped constants
  creating 82
Class Specification dialog box 84
Class, C++ TargetRTS 91
classes
  associations with capsules 79
  complex 90
  constant values 83
  containing pointers 76
  marshallable 87, 89
  non-public fields 90
  overriding and adding 172
  sendable by value 87
  with pointers 87
  without pointers 86
classes with no pointer attributes 86
ClassKind 83, 86, 181
ClassKind (Class, C++) 181
ClassKind = typedef 181
Code Generator
  command line arguments 54
Code Generator command line arguments
  -crlf 54
  -forcewrite 55
  -spacedeps 55
  -version 55
Code Sync
  deactivating 35
codegen directory 143
CodeGenDirName (Component, C++ Generation) 215
CodeGenMakeArguments (Component, C++ Generation) 217
CodeGenMakeCommand (Component, C++ Generation) 217
CodeGenMakeInsert 178
CodeGenMakeInsert (Component, C++ Generation) 217
CodeGenMakeType (Component, C++ Generation) 217
CodeSyncEnabled (Component, C++ Generation) 218
command line arguments 130
  code generator 54
  providing on targets that do not support command line arguments 131
command line build interface 55
command summary (RTS debugger) 131
  capsuleId 131
  control commands 139
  help 131
  informational commands 133
  portId 131
  running a model 132
  taskId 131
  thread commands 132
  tracing commands 137
CommonPreface (Component, C++ Generation) 216
Communication Services 106
communication services 106
  application registration 111
  automatic registration 111
  concepts 106
  deferring messages 112
  Lossy 108
  message priorities 109
  order-preserving 108
  ports 110
  primitives 107
  published ports 110
  recalling messages 112
  register by name 111
  registration string 111
  request reply 108
  support for unwired ports 110
  unpublished ports 110
  unwired ports 110
CompilationMakeArguments (Component, C++ Compilation) 220
CompilationMakeCommand (Component, C++ Compilation) 219
CompilationMakeInsert (Component, C++ Compilation) 220
CompilationMakeType (Component, C++ Compilation) 219

CompileArguments (Component, C++ Compilation) 221
CompileCommand 178
CompileCommand (Component, C++ Compilation) 220
compiler
  external class definitions 88
complex classes 90
Component
  C++ Generation - OutputDirectory 178
Component Compiler Inclusions 88
ComponentUnitName (Component, C++ Generation) 216
composite 80
  attributes 80
  See also by value
config directory 143
configuration naming convention (Services Library) 141
configuration processor definitions 144
configuring
  C++ Services Library 167
ConnectionParameters 86
ConnectParams classes 86
constant 84
  creating 82
constant values
  specified 83
constants
  creating 82
  scoped globally 82
  scoped to classes 82
  usage 83
  using 83
constructor
  setting as private 192
constructor operations 76
ConstructorInitializer (Operation, C++) 189
contacting Rational customer support xxv
continue 139
control commands 139
  exit 139
  go 139
  quit 139
  step 139
controlled units
  effect 52
Controller
  capsule instances 105
copy constructor 192
  setting as private 192
copy constructor operations 76
CopyConstructorExplicit (Class, C++) 185
CopyConstructorInline (Class, C++) 185
CopyConstructorVisibility (Class, C++) 185
CopyFunctionBody (Class, C++ TargetRTS) 197
creating
  array attributes 81
  class scoped constants 82
  classes with attributes that are pointers 87
  classes with no pointer attributes 86
  constants 82
  data types 76
  enumeration 83
  external library component 67
  global constants 82
  Library Component for External Library Interface 59
  pointer attributes 81
  Struct 86
  typedef 83
cross thread message sending 163
Custom Peer Controller 155
custom peer controller
  Services Library 154
customizing
  C++ Services Library 167

D

data by value
  examples 78
Data Class 74
data class
  basic structures 77
  copied 76
  decoded 76
  destroyed 76
  encoded 76

initialized 76
marshallable 77
data class rules 76
data classes
marshallable 77
syntax examples 78
data type
examples 77
data types
creating 76
decoding 88
definition 73
encoding 88
marshallable 76
observable 76
sendable by value 76
data versus capsules 161
deactivating Code Sync 35
DEBUG_NONE 147, 151
DEBUG_TERSE 146
DEBUG_VERBOSE 146, 151
debugging
objects 88
running models 77
DecodeFunctionBody 91
DecodeFunctionBody (Class, C++ TargetRTS) 197
decoding
data types 88
DefaultArguments (Component, C++ Executable) 131, 226
DefaultConstructorExplicit (Class, C++) 184
DefaultConstructorInline (Class, C++) 184
DefaultConstructorVisibility (Class, C++) 184
Defer 138
DEFER_IN_ACTOR 145
deferInvoke 120
deferring messages 112
dependencies 83, 84
Dependency C++ 84
Dependency C++ Properties 82
dereg 120
deregister operations 157
Deregister SAP 138
Deregister SPP 138
Derived 80
design considerations 164
DestroyFunctionBody (Class, C++ TargetRTS) 197
destructor operations 76
DestructorInline (Class, C++) 186
DestructorVirtual (Class, C++) 186
DestructorVisibility (Class, C++) 185
detach 132
detach
thread commands 133
Detail Properties 84
directory structure
Services Library 143

E

encode function 77
EncodeFunctionBody 91
EncodeFunctionBody (Class, C++ TargetRTS) 199
encoding
data types 88
ends
association 80
enumeration
creating 83
enumerations
creating 83
environment variables 178
EqualityOperatorInline (Class, C++) 187
EqualityOperatorsVisibility (Class, C++) 187
error codes
C++ Services Library 117
error enumeration 118
events 94
examples
of class creation 86
of class modeling 79
of common C++ constructs 81
of sending data by reference 79
of sending data by value 78
of sending data classes 78
type descriptors 206

Exception 245
exception service 117
  concepts 117
Exception Signals 246
ExecutableName (Component, C++ Executable) 225
exit 139
  control commands 139
external C++ classes
  integrating 88
external class definitions
  making visible to compiler 88
external classes 88
  creating type descriptors 89
  integration options 89
external data types
  marshalling 90
external library
  platforms 164
External Library Interface
  Binary files 68
  client model 58
  Considerations and Known Issues 70
  creating a library component 59
  creating an external library component 67
  DefaultInterfaceVisibility 61
  Header files 68
  InclusionPaths 65
  interface model 58
  Library Interface Specification 59
  Library Paths 65
  Limited Public visibility 62
  overview 57
  pathmap symbols 65
  Phase 1 59
  Phase 2 66
  Phase 3 68
  phases to generate 58
  Private visibility 62
  Public visibility 62
  Publishing the interface 66
  purpose 57
  redefining visibility settings 64
  References 60
  SetInterface 61
  setting inclusion paths and library paths 65
  setting the Target Configuration and References 60
  setting visibility level 60
  shared package names 66
  sharing 68
  sharing - known issue 69
  source model 58
  unit visibility rules 63
  using 68
  using the Interface 69
  visibility level 60
  visibility level descriptions 62
  visibility rules 63
external port service example (C++) 122, 299

F

flags
  polling 153
Frame 114, 247
  classIsKindOf 248
  className 248
  classOf 249
  deport 249
  destroy 250
  import 251
  incarnate 253
  incarnationAt 257
Frame Service 114
frame service
  114
  concepts 114
  implementation classes 114
  multiple containment 115
  optional capsules 115
  plug-in capsule roles (multiple containment) 115
  replicated capsule roles 116
Frozen 82

G

General tab 84
generate
  model tags 218
GenerateAssignmentOperator (Class, C++) 186
GenerateClass 90, 91, 173
GenerateClass (Class, C++) 181
GenerateCopyConstructor (Class, C++) 184
GenerateDefaultConstructor (Class, C++) 184
GenerateDescriptor 90, 91, 173
GenerateDescriptor (Attribute, C++ TargetRTS) 201
GenerateDescriptor (Class, C++ TargetRTS) 196
GenerateDescriptor (Role, C++ TargetRTS) 202
GenerateDestructor (Class, C++) 185
GenerateEqualityOperator (Class, C++) 187
GenerateInequalityOperator (Class, C++) 187
GenerateModelTags 218
Generating an External Library Interface 57
Global 82, 84
global constants
  creating 82
go
  control commands 139
guard conditions 161

H

hardware differences 164
HAVE_INET 147
Header files 68
HeaderEnding (Capsule, C++) 190
HeaderEnding (Class, C++) 182
HeaderPreface 85, 88, 90, 91
HeaderPreface (Capsule, C++) 190
HeaderPreface (Class, C++) 182
heavy-weight unit of concurrency 96
help 131
help commands 131

I

implementation classes for exception service 117
implementation classes for frame service 114
implementation classes for timing service 112
ImplementationEnding (Capsule, C++) 191
ImplementationEnding (Class, C++) 183
ImplementationPreface (Capsule, C++) 190
ImplementationPreface (Class, C++) 183
ImplementationType 83
ImplementationType (Class, C++) 181, 182
imported 120
Incarnation - RTFrame
  incarnate() 160
include directory 143
InclusionPaths (Component, C++ Compilation) 221
InclusionPaths property 65
info 133
  informational commands 135
informAt 120
informAt primitive 121
informational commands 133, 134
  info 135
  printstats 136
  saps 133
  system 134
informEvery 120
informEvery primitive 121
informEvery() 153
InformIn 138
informIn 120
informIn primitive 121
InitFunctionBody (Class, C++ TargetRTS) 196
Initial Value 82, 84, 85
InitializerKind (Attribute, C++) 188
InitializerKind (Role, C++) 189
InitialValue (Role, C++) 190
Inline (Operation, C++) 188
INLINE_CHAINS 148, 151
INLINE_METHODS 148
INTEGER_POSTFIX 145

integrating
  external classes 88, 89
integration
  case 1 88
  case 2 89
  options 89
  questions 88
integration options 89
IPC
  multi-threaded 153
  single-threaded 153
IPC events 152
IPC mechanisms
  adding support 155
IPC options summary 159
IPC processing 157

K

kernel configuration 124
KindInHeader 82, 84
KindInHeader (Uses, C++) 192
KindInImplementation 82, 84
KindInImplementation (Uses, C++) 192

L

lib directory 144
Libraries property 65
Library Interface Specification 59
LibraryName (Component, C++ Library) 227
libset directory 144
libset name
  Services Library 142
light-weight unit of concurrency 96
limitations
  opening Rose models in Rose RealTime 37
  using the C++ Analyzer in Rose RealTime 37
Limited Public visibility 62
LinkArguments (Component, C++ Executable) 226
LinkCommand (Component, C++ Executable) 226
literal integer 85
Log 106, 261
  log 262
  show 262
log 137
log
  tracing commands 137
Log Miscellaneous Operations 263
Log Service 106
log service 106
LOG_MESSAGE 146, 151
logical threads 225
LogicalThreads (Component, C++ Library) 228
logMsg() 175
loss of messages 108

M

macro usage 85
macros 85
  defining complex 85
  in C++ 84
main.cpp 125
mainLoop() 153, 154, 156
manual loading 125
mapping capsules to threads 96
marshallable 86
  classes 87, 89
  data classes 77
marshallable data classes 77
marshalling
  external data types 90
marshalling functions 90
memory
  managed by class 87
memory leaks 75
message
  processing 94
message pointers 74
message priorities 109

message priority 109
- background 109
- general 109
- high 109
- low 109
- panic 109

message processing 94
message trace 77
messages 94
- cross thread 163
- deferring 112
- from capsules 73
- minimal overhead in handling 108
- recalling 112
- sending data by value 162

model
- generate tags 218
- running 132

models
- debugging 77
- overriding operations 173

MS_nmake 219
msg 158
multiple containment 104, 115
multiplicity
- arrays 81

multi-threaded IPC 153
multi-threaded message processing 95
multi-threaded Services Library 98

N

naming considerations
- C++ Services Library 99

New Attributes C++ Properties 82
NewCapsule1 80
NewClass1 80
noConnect 121
Nodes classes 86
noMem 121
non-public fields
- in classes 90

Non-Volatile-Store 164
noReply 121
notImported 121
notOptional 121
NumElementsFunctionBody 87
NumElementsFunctionBody (Attribute, C++ TargetRTS) 201
NumElementsFunctionBody (Role, C++ TargetRTS) 202
NumElementsFunctions 76
NVS 164

O

object instances
- source 91
- target 91

OBJECT_DECODE 146, 151
OBJECT_ENCODE 146
objects
- debugging 88

OBSERVABLE 150
-obslisten 126
one shot timer 112
opening
- Rose models in Rose RealTime 37

OperationKind (Operation, C++) 188
operations
- overriding and adding 172

operator
- setting assignment as private 192

optional capsules 115
order-preserving
- messages 108

OTRTSDEBUG 146, 151
OutputDirectory (Component, C++ Generation) 215
overriding
- operations and classes 172
- operations with targets 174
- operations within models 173

P

pathmap (External Library Interface) 65
pathmap symbols 178
Peer Controller class 155
periodic timer 113
physical threads 225
platform name
  Services Library 142
plug-in capsule roles 115
pointers
  classes 87
  classes without 86
  creating 81
  in classes 76
  sending in messages 74
polling a flag 153
port services 122, 299
portId 131
ports 110
  cardinality 83, 85
  published 110
  unpublished 110
  unwired 110
POSIX 153
pre-processor macros
  changing 168
printf 163
printstats 133
printstats
  informational commands 136
prio 121
private
  setting assignment operator 192
  setting copy constructor 192
private assignment operator 194
Private visibility 62
PrivateDeclarations 191
PrivateDeclarations (Class, C++) 183, 191
process boundaries 74
processing overhead
  blocking capsule 154
ProtectedDeclarations 191
ProtectedDeclarations (Class, C++) 183, 191
Protocol 196
protocols
  base class 103
  conjugate class 103
  sending data by reference 74
  sending data by value 74
  signals 103
Public visibility 62
PublicDeclarations 191
PublicDeclarations (Class, C++) 183, 191
published ports 110
Publishing the interface 66
PURIFY 147

Q

quit 139
  control commands 139

R

Rational customer support
  contacting xxv
Recall 138
RecallAll 138
recalling messages 112
receiver 78
recursiveInvoke 121
Redefining Visibility Settings 64
refFull 122
reg 122
Register SAP 138
Register SPP 138
REGISTER_LAYER 157
REGISTER_LAYER macro 158
registering operations 157
registerLayer 157
registration by name 111
registration by name (communication services - C++) 111
registration string 111
relative time 113
replicated capsule roles 116
replRef 122
request-reply 108

RTActor 234
  context 237
  context() 117
  getCurrentStateString 239
  getError 237
  getError() 117
  getIndex 238
  getMsg 235
  getName 238
  getTypeName 238
  isType 239
  logMsg 236
  logMsg() 146, 174
  msg 235
  overrideable operations 102
  unexpectedMessage 236
RTActor subclasses
  ports 103
RTActorClass 114, 239
  capsule class information 102
RTActorId 114, 241
  isValid 242
RTActorRef 114, 240
  capsule roles 102
  size 241
RTController 105, 121, 155, 156, 242
  abort 245
  getError 243
  getError() 117
  mainLoop() 173
  name 244
  perror 244
  perror() 117
  strerror 244
  strerror() 117
RTController class 117
RTCustomController 156
  mainLoop() 156, 158
  waitForEvents() 157
  wakeup() 157, 158
RTDATA 36
rtdata 36, 78
RTDataObject subclasses 234
RTDataObjects 162
RTExceptionSignal 247
RTFRAME_CHECK_LOOSE 148
RTFRAME_CHECK_NONE 148
RTFRAME_CHECK_STRICT 148
RTFRAME_CHECKING 148
RTFRAME_THREAD_SAFE 149
RTFRAME_USE_FREELIST 150
RTInSignal 106, 257
  purge 258
  purgeAt 258
  recall 259
  recallAll 260
  recallAllAt 261
  recallAt 259
RTMain
  argCount() 131
  entryPoint 125
  targetShutdown 173
  targetStartup 173
  targetStartup() 173
RTMessage 121, 147, 264
  defer 268
  deliver() 158
  getData 266
  getPriority 265
  getSignalName 265
  getType 266
  isValid 267
  sap 267
  sapIndex0 266
RTMessage object
  capsule instances 105
RTMESSAGE_PAYLOAD_SIZE 150
RTObject_class 86, 268
RTOutSignal 106, 270
  invoke 272
  invokeAt 274
  reply 275
  send 270
  sendAt 271
RTPeerController 154, 156
  enhancement 155
  mainloop() 173
  waitForEvents() 156
  wakeup() 156
rtport 36

RTProtocol 106, 275
  bindingNotification 281
  bindingNotificationRequested 281
  deregisterSAP 282
  deregisterSPP 284
  getRegisteredName 286
  isBoundAt 285
  isIndexTo 284
  purge 277
  purgeAt 278
  recall 278
  recallAll 280
  recallAllAt 280
  recallAllFront 281
  recallAt 279
  recallFront 279
  registerSAP 282
  registerSPP 283
  size 277
RTReal 147
RTREAL_INCLUDED 147
RTResourceMgr 121
RTS debugger command summary 131
RTS_COMPATIBLE 147
RTS_COUNT 145
RTS_INLINES 147
RTTimerId 112
RTTimespec 112
  getclock 289
  RTTimespec 289
RTTimespec assignment operators 291
RTTimespec basic arithmetic operators 291
RTTimespec basic comparison operators 290
RTType_class name 86
Run Time system debugger
  starting 129
running
  model 132
  with observability 126
run-time system debugger
  differences between Single and Multi-threaded 130

S

SAP 110
saps 133
  informational commands 134
scanf 163
Send 138
send
  sending and receiving data by value 78
  sending data classes between capsule instances 78
send primitive 121
sendable by value 86
  classes 87
sender 78
sending data by reference 74
  examples 79
sending data by value 74
  examples 78
sending data by value in messages 162
sends
  unnecessary 162
Services Library 86, 89
  building 175
  custom peer controller 154
  directory structure 143
  error codes 117
  libset name 142
  minimum services 151
  naming conventions 141
  overriding operations 172
  overriding operations with targets 174
  overriding operations within models 173
  platform name 142
  target base name 142
Services Library Framework
  C++ 93
Services Library Source 141
  $RTS_HOME 141
  codegen 143
  config 143
  directory structure 143
  include 143
  lib 144
  libset 144

Libset Name 142
Naming Convention 141
Platform Name 142
src 144
target 144
Target Base Name 142
tools 144
setting
header and binary files 68
shared package names 66
Sharing an External Library Interface 57
sharing the External Library Interface 68
signal handlers
using 153
single-threaded IPC 153
single-threaded message processing 95
single-threaded Services Library 97
source object instances 91
SPP 110
src (C++) 53
src directory 144
starting
Run Time system debugger 129
state machines 161
step
control commands 139
Stereotype 84
Struct 86
struct
creating 86
SUPER 36
synchronous communication 107
syntax examples
of sending data classes 78
system 133

T

tags
generate model tags 218
target base name
Services Library 142
target directory 144
target object instances 91
Target Observability Port 127
TARGETRTS properties 195
targets
overriding operations 174
targetStartup 174
taskId 131
tasks 132
thread commands 132
thread boundaries 74
thread commands 132
attach 133
detach 133
tasks 132
threads 96
Timing 112, 291
cancelTimer 297
currentTime 296
informAt 292
informEvery 75, 295
informIn 75, 293
timing accuracy 114
timing precision 114
Timing Service 112
timing service 112
absolute time 113
accuracy 114
concepts 112
implementation classes 112
one shot timer 112
periodic timer 113
precision 114
relative time 113
tool chain functionality 123
tool chains 165
tools directory 144
TopCapsule (Component, C++ Executable) 223
tracing commands 137
log 137
tv_nsec 288
tv_sec 288
Type 82, 83
type descriptors
Associated Type Descriptors 206
defined 203
examples 206

for external classes 89
usage 204
Usage patterns 206
what are they? 203
when are they used? 204
typedef 83
creating 83
TypeDescriptor (Attribute, C++ TargetRTS) 201
TypeDescriptor (Role, C++ TargetRTS) 202
types of concurrency 96
TypeSafeSignals (Protocol, C++ TargetRTS) 203

U

Unit Visibility rules 63
Unix_make 219
unnecessary sends 162
unpublished ports 110
unwired ports 110
support for 110
unwired support 110
URTS_DEBUG 129
URTS_DEBUG Parameter 129
USE_THREADS 145
UserLibraries (Component, C++ Executable) 226
UserObjectFiles (Component, C++ Executable) 227
using the C++ Analyzer in Rose RealTime 37
using the External Library Interface 68

V

Version (Class, C++ TargetRTS) 196
Version (Protocol, C++ TargetRTS) 202
Visibility Level
External Library Interfaces 60
Visibility Level Descriptions 62
visibility rules (External Library Interface) 63

W

waitForEvents 157
waitForEvents() 154
wakeup() 154
wakeupFunc 158
wired ports 110

X

xRtsInvoke 122