Rational Software Corporation

C Reference

RATIONAL ROSE® REALTIME

VERSION: 2002.05.20

PART NUMBER: 800-025100-000

WINDOWS/UNIX

the software development company

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

Contents

Overview of the C Guide 1

Contents

This chapter is organized as follows:

Introduction

Use this guide to learn how to use the C Language Add-in to build, compile, and debug C-based Rational Rose RealTime models. You can also learn how to deploy the model executables to a target system, and how to optimize and configure your target to meet 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 that element's specification, model properties, and the models design properties. The Model Properties Reference provides the language-specific information required to map your model to C.

To understand how the C language add-in works, you need to become familiar with the main parts of the language add-in:

- *Workflows for your host workstation and embedded target* on page 2
- The ability to configure and minimize footprint.
- Static structure with the ability to map fixed capsule instances to any logical thread.
- The means of integrating a user-designed timing service.

- The ability to configure memory policy (should memory be allocated after startup).
- 8.3 file naming compliance.
- *Using C code in Models* on page 3
- *Model Properties* on page 3
- *C Services Library* on page 4
- *Code Generation* on page 5
- *Compilation* on page 6
- *Model Executables* on page 6

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 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, you probably want to run your models primarily on the host workstation. This keeps the modify-compile-debug cycle as short as possible. Also, you can take advantage of workstation-based debug tools, such as C source-level debuggers and C analysis tools (such as Purify^{TM}) which 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 75.

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 Rational 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. The abstract behavior of a capsule is described as a graphical state diagram, which shows the allowable sequence of events that the capsule can process. To carry out useful activity, 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. It does not, however, replace your existing C tools. Rather, it depends on the existence of other 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. 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 Rational 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.

To build source code, the code generator also generates **makefiles** which specify how to build the generated source code. Therefore, certain properties affect how these **makefiles** are generated and their contents.

You can use model properties to:

- Add an **#include** directive automatically to more than one file.
- Add a global prefix to functions generated for a class.
- Specify the kind of C data type generated for a class (for example, **struct**, **union**, **enum**, and **typedef**).
- 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. For example, several model properties applying to components are used to control of the aspects of building and linking a model. See *Model Properties Reference* on page 119 for detailed reference to the model properties.

Note: Not all model components for which code is generated require model properties. For example, there are no model properties for generalization relationships, yet the code generator adds the attributes from the parent into the child's **struct** as fields, then adds **#include** directives; in such cases, information obtained from specifications is sufficient to control code generation.

C Services Library

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
- 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.

In summary, the facilities provided by the RealTime Services Library are:

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

This document includes the following basic topics:

- *C Services Library* on page 51
- *Running Models on Target Boards* on page 75

This document includes the following advanced topics:

- *Organization of the Services Library Source* on page 93
- *Configuration Preprocessor Definitions* on page 97
- *Optimizing Designs* on page 106
- *Configuring and Customizing the Services Library* on page 111

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 5.

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.

Note: Do not make changes to the generated code outside of the identified sections; you may lose these changes. For more information, see *Code Sync* ***on page 11.***

Compilation

The C Language Add-in will convert a model to C code but does not include the compiler which will build 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 and make utilities.

Linking the Model with the Services Library

Rational Rose RealTime models are created by linking the two components, the user-compiled model files, and 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

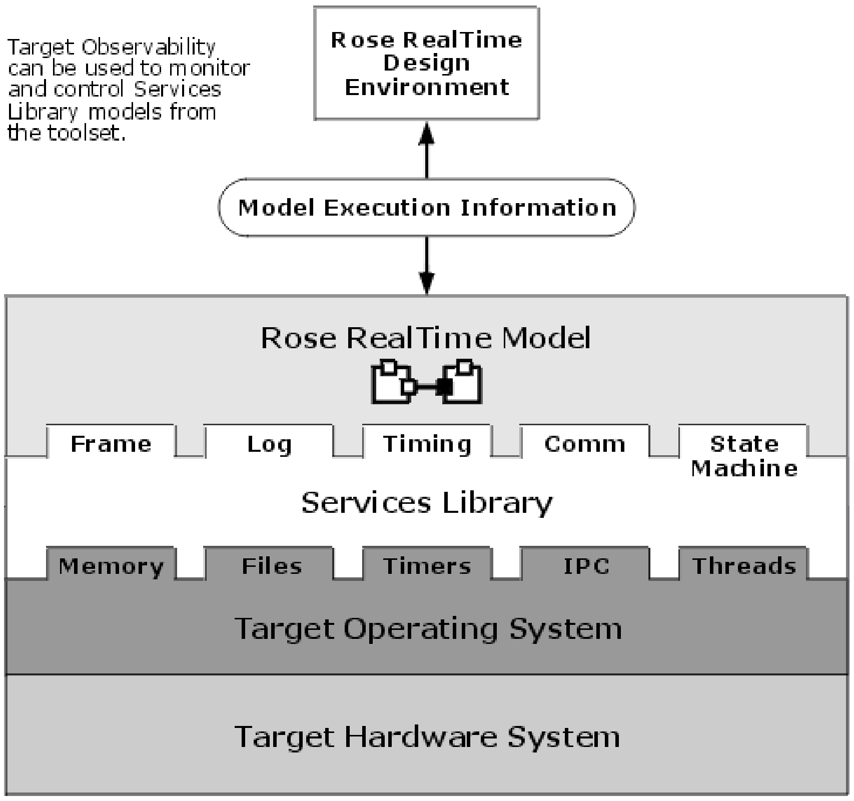
Compiling a Rational Rose RealTime model results in a stand-alone executable. The generated executable is not connected to the Rational Rose RealTime session unless desired. If targeted for a workstation platform, the model can be run by typing the name of the generated executable on the command line. 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 75.

Target Observability

The graphical observation tools for Rational Rose RealTime 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 shows a graphical representation of Target Observability in Rational Rose RealTime.

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 Your Model 2

Contents

This chapter is organized as follows:

Adding C Code to a Model

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

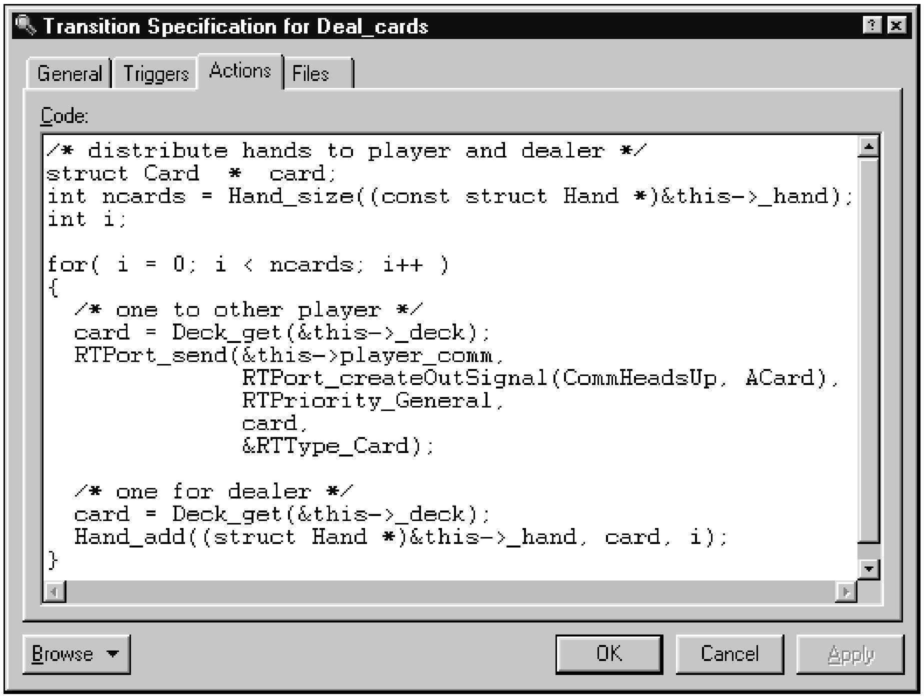
- Perform detailed actions that occur on transitions.
- Perform detailed actions that occur on state entry or exit.
- Code capsule operations that can then be invoked from any other code segment (the common name for the C code contained inside any one model element, such as a transition code segment); 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.

In addition to these various mechanisms for adding C details to your model, 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 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.

Syntax of Code Segments

C code is added to your model by adding code to the body portion of operations, transitions, and so on. For this reason, you do not have to add curly braces to the beginning and end of any action code segments. These are added automatically by the code generator.

Figure 2 Sample Transition Action Written in C.

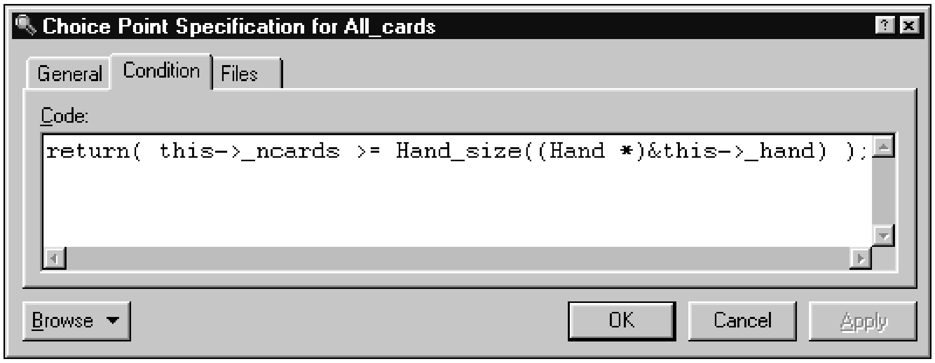


```
/* distribute hands to player and dealer */
struct Card  *  card;
int ncards = Hand_size((const struct Hand *)&this->_hand);
int i;

for( i = 0; i < ncards; i++ )
{
  /* one to other player */
  card = Deck_get(&this->_deck);
  RTPort_send(&this->player_comm,
              RTPort_createOutSignal(CommHeadsUp, ACard),
              RTPriority_General,
              card,
              &RTType_Card);

  /* one for dealer */
  card = Deck_get(&this->_deck);
  Hand_add((struct Hand *)&this->_hand, card, i);
}
```

Choice Point Code Condition Segment

The choice point segments are created as functions which return an **int**. Hence, the condition C code entered in a choice point must have a return statement that returns either **false** (0) or **true** (non-0). You can have any number of other C statements in the choice point segment as long as it returns an **int**.

Figure 3 Example C Code in a Choice Point Condition



```
return( this->_ncards >= Hand_size((Hand *)&this->_hand) );
```

Encapsulating Target-Specific Behavior

The Rational Rose RealTime workflow provides 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 Rational 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. To accomplish this, 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 code from outside the toolset within an IDE (Integrated Development Environment), or using a text editor of your choice, and then propagates changes back into the model.

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

Making Changes Outside the Toolset

To re-capture changes 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. For more information, see *Using Code Sync to Change Generated Code* in the Toolset Guide.

User modifiable code for C is identified as follows:

```
/* {{{USR <location_tag> */

<insert or modify code here>

/* }}}USR <location_tag> */
```

For example,

```
/* {{{USR capsuleClass 'NewCapsule1' tool 'OT::C' property
'HeaderPreface' */

<insert or modify code here>

/* }}}USR capsuleClass 'NewCapsule1' tool 'OT::C' property
'HeaderPreface' */
```

When a field is omitted or the default is used, the code generator may generate an optimized code pattern that does not provide the empty Code Sync areas or its identification tags. If you wish to use a 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 modifying 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(s).

Code Generation 3

Contents

This chapter discusses some relevant aspects of the 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.

This chapter is organized as follows:

Model to Code Correspondence

From a modeling perspective, designing capsules, data classes, and their interactions is relatively independent from the programming language. However, with respect to code generation, certain generic parts of a model element's specification are interpreted by the code generator and translated to C code, while other elements are ignored. This section outlines how the UML model is translated into C 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.

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

For specific information about code generated for a model element, see the following topics:

- *Capsules* on page 14
- *Capsule State Diagrams* on page 16
- *Classes* on page 17
- *Attributes* on page 18
- Associations
- *Standard Operations* on page 22
- *Generalizations* on page 21
- *Dependencies* on page 22
- *Protocols* on page 23
- *Logical Packages* on page 22
- *Standard Operations* on page 22
- *Components* on page 23
- *Relationships and Elements Ignored by C Code Generation* on page 23

Capsules

Each capsule is generated in it's own .h and .c file. The code generator converts a capsule's structure and state diagrams into C code.

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.

By default, for each capsule the following files are generated:

- *Header File (.h)* on page 14
- *Implementation File (.c)* on page 15
- *'this' Pointer* on page 15

Header File (.h)

The following code is generated in a header file:

- Inclusions, forward references, value of the **HeaderPreface (Capsule, C)** property.
- Capsule definition with any attributes, and associations as fields of the generated **struct**.
- Ports generated as attributes of the capsule **struct**.
- Standard Operations prototypes prefixed with the value of the **GlobalPrefix (Capsule, C)**
- Value of the **HeaderEnding (Capsule, C)** property.

Implementation File (.c)

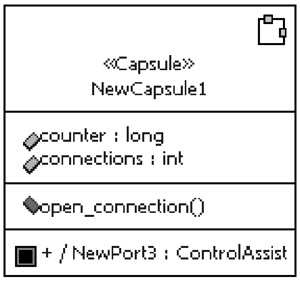
The following can be found in an implementation file:

- Inclusions, forward references, value of the **ImplementationPreface (Capsule, C)** property.
- Implementation scope operation prototypes and implementations.
- Operations implementation.
- Transition code, choice point code.
- State behavior implementation.
- Value of the **ImplementationEnding (Capsule, C)** property.

'this' Pointer

All user code in the context of a capsule (for example, state transition detail code, instance operations, choice points, entry and exit code, and so on) has a reference to its own capsule instance data through a **this** pointer passed as an argument to each generated function. The **this** pointer points to an instance of a capsule on which the function is being called, effectively allowing access to its fields (for example, attributes, ports, and so on).

If the following capsule is defined:



From within any of the state machine detail code and functions, you can access the capsule instance data via the this pointer as follows:

```
/*
   Here we assume that this is transition
   code. Via the 'this' pointer you can access
   the capsule's instance data.
*/
this->counter = 34;
this->connections++;
```

```
/*
   Here we are calling a capsule function
   which requires access to the instance
   data as well.
*/
NewCapsule1_open_connection(this);

/*
   Sending a message via the NewPort3 port, you
   are required to access the port instance
   via the ‘this’ pointer.
*/
RTPort_send(&this->NewPort3,
              RTPort_createOutSignal(NewPort3, go),
              RTPriority_General,
              &this->counter,
              &RTType_long );
```

Capsule State Diagrams

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

Note: Protocol and Class State diagrams are ignored by the C generator.

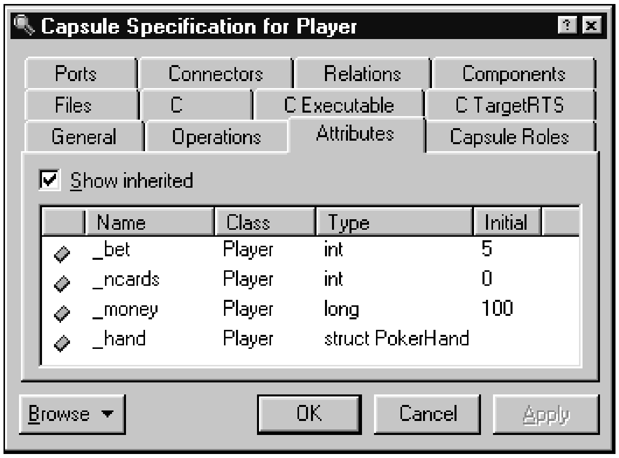
Note: You should never modify code directly in the generated source files. It may however be useful to understand that transitions are generated as operations when debugging code using source level debuggers.

Classes

Classes are emulated in C through structures. Depending on their defined scope, attributes and associations are generated as fields (instance scoped) in the structure, or global (class scoped) variables.

When creating attributes as a type of existing C classes (which are really **structs**) you are required to conform to C programming rules and prefix the class name with the class key (for example, **union**, **struct**, **enum**).

Figure 4 Example Attribute Specification with 'struct' Keyword



Name	Class	Type	Initial
_bet	Player	int	5
_ncards	Player	int	0
_money	Player	long	100
_hand	Player	struct PokerHand	

Each class has its own .h and .c files generated.

Header File (.h)

By default, for classes, the following code is generated in the header file:

- Inclusions, forward references, value of the **HeaderPreface (Class, C)** property.
- Declarations of global attributes and associations.
- Attributes generated from class associations or explicitly defined as fields of the structure.
- User-defined operations: these operations are generated with the prefix defined in **GlobalPrefix(Class, C)** property.
- If **GenerateDescriptor (Class, C TargetRTS)** property is true, a declaration for a class type descriptor of type **RTObject_class**.
- Value of the **HeaderEnding (Class, C)** property.

Implementation File (.c)

The following can be found in an implementation file:

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

Properties Affecting How Classes are Generated

The following properties affect how 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, enums, or unions instead of the default struct.
- The **GenerateDescriptor (Class, C TargetRTS)** property controls the generation of the classes' type descriptor.

Attributes

By default, an attribute is represented in code as an attribute in the client class.

The following properties affect how attributes will be generated:

- **AttributeKind (Attribute, C)**: use this property to toggle between generating the attribute as a field of the struct or as a **#define**.

- Scope: attributes can the scoped to the instance, or to the class. Class scoped attributes are generated as global variables, to avoid possible name clashes, the generated global variable is prefixed with the value of the **GlobalPrefix(Class, C)** property.

Associations

An association is a relationship among two or more elements. The ends of each association are called association ends. Ends may be labelled 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 a field in the generated structure for the client class. The code generation does not generate attributes for ends which are not named.

Valid Code Generation Associations

Only the following association relationships 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**, and so on.

- **Class to class (data member)**

  For these associations the code generator by default generates a data member (attribute) for navigable and named ends. Several factors affect the code that is actually generated: the scope property affects if a member or global data member is generated, the multiplicity affects whether an array of attributes should be created, the containment affects whether the attribute should be a reference (pointer) or an object.

  Association end multiplicity is specified as **x..z**, only the upper bound is used.

- **Capsule to class (data member)**

  For these associations the code generator by default generates a data member (field) on the generated capsule structure. 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: fixed, cardinality, and so on.

User-Defined Operations

When generating code for a class, a global function is generated for each operation that is listed in the class or capsule specification. The function is named based on the value of the **GlobalPrefix(Class, C)** property of its owning class.

For each such operation, the generator produces:

- A 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.

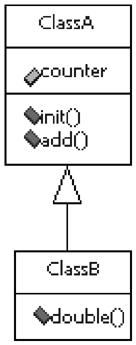
'this' Pointer

To mimic the behavior of true object-oriented languages, where operations have access to the attributes of the class instance on which they were called, the C code generator creates for each generated instance operation a parameter which is a pointer to instance data. The parameter is always named **this**. Via the **this** pointer you can access the attributes defined on the instance passed to the function.

Given the class definition shown in the diagram below, the **add()** function would be declared and access the counter attribute as follows:

```
void ClassA_add(struct ClassA * const this)

{

   this->counter++;

}
```

Note: The '**const**' modifier enforces that the user cannot change the value of this, only what it points to.



And to call the **add()** function from detail code, you would use the following syntax to pass the instance data to the function:

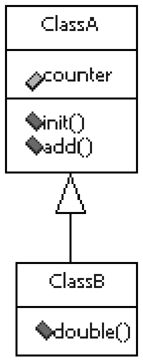
```
/*
   Here we create and initialize a temporary
   tclass variable, then call the add() function
   passing a pointer to the instance data.
*/
struct ClassA tclass;
tclass.counter = 10;
ClassA_add(&tclass);
```

Generalizations

Inheritance is emulated in C through the flattening and re-use of classes and capsules. A subclass' attributes and associations are inherited by regenerating each element in the subclass' structure. The code generator ensures that the superclass' fields are inherited in the same order as they are specified in the superclass. This means that a pointer to a subclass can be cast upwards to a superclass instance pointer.

Example

To demonstrate how you can call functions defined on a superclass, given **ClassA** and **ClassB** defined as follows:



If the **GlobalPrefix(Class, C)** prefix property for each class is defined as **${name}_** then **ClassB** could call the **add()** or **init()** functions using the following syntax:

```
ClassA_add((Class A*)this);
```

Dependencies

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

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

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 Rational 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.

Standard Operations

When generating code for a class, the C generator will also generate a **construct** function which initializes the classes' attributes with either the initial value or by calling the attribute's **construct** function.

For capsules, the **construct** function is generated automatically. Use the **GenerateConstructFunction (Capsule, C)** property to configure the generation of the construct function for capsules.

Protocols

Each protocol is generated in its own .h and .c file.

Components

When generating a component, the code generator creates a set of **makefiles** which contain rules for generating and building all elements referenced by the component. In addition, a system wide .c 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 code generator produces code for 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 mode file (.rtmdl) and the name is derived from the name of the corresponding component.

Relationships and Elements Ignored by C Code Generation

The following modeling elements are ignored by the C code generator:

- Realizes relationship
- Capsule roles specified as optional or plug-in
- Package dependencies
- State diagrams on protocols and classes
- Collaboration diagrams
- Sequence diagrams
- Actors
- Use-cases
- Deployment diagrams

For this release of the C code generator, the following aspects of a model are ignored by the code generator:

- Attribute/operation visibility: all attribute and operations are generated with public visibility, and the code generator outputs a warning to this effect if private or protected visibility is set on any of these generated elements.

- Polymorphic operations: a **v-table** mechanism for function pointers is not provided.
- Multiple inheritance
- Nested classes

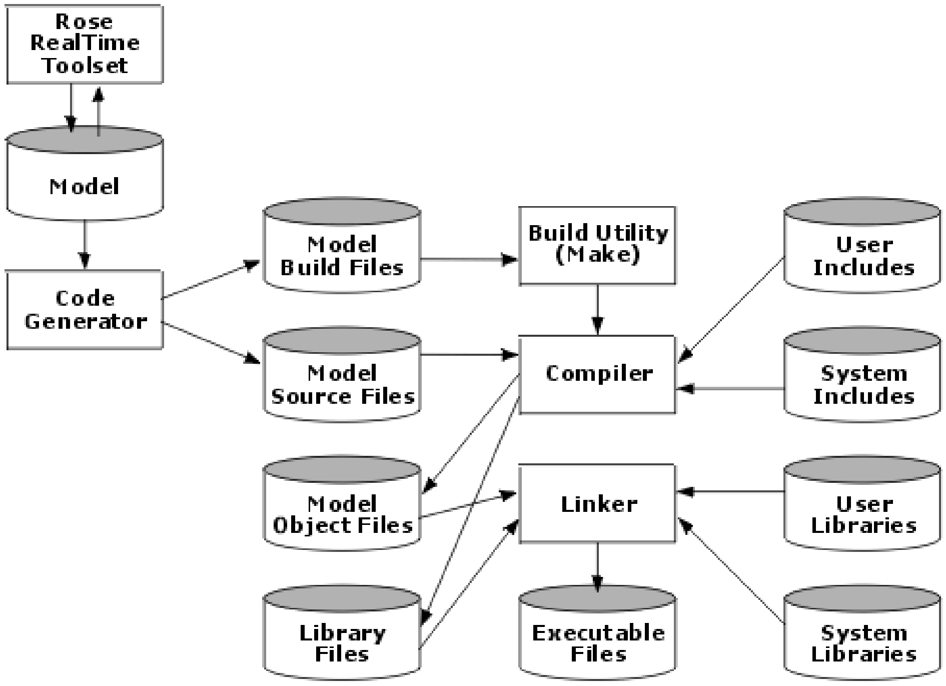
Code Generator Behavior

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 Rational Rose RealTime toolset. These build **makefiles** are called from within Rational Rose RealTime to build a component.

Figure 5 shows the compilation paradigm for producing a working C executable.

Figure 5 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, since:

- 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.

Note: If a model is saved into one monolithic .rtmdl file, 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 Team Development Guide for instructions on how to save models as controlled units.

Because the compiler reads generated source files not controlled units, and because the incremental generation behavior is independent of controlled units, the choice of controlled units does not affect compilation performance. The incremental behavior of the code generator 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(s). Each component in a model is built in its own directory structure. There is an option in the component specification dialog, that allows the user to specify a different directory for this purpose.

Note: We recommend that each component has a different output path to avoid overwriting files for other components.

Inside the component directory is a directory tree that separates the model files, generated source files, and build results, including the executable.

After building a component whose name is "**Component1**" the default directory structure below the output directory would look like the following:

```
Component1\

   src\
   build\
```

src

This directory contains all C source files 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 is 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 **struct** is generated with the name:

```
<capsule name>_InstanceData
```

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

Code Generator Command Line Arguments

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

- Adding the command line options to the **ROSERT_RTGENOPTS** environment variable.
- 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 simply add these to the macro definition:

  ```
  RTGEN = rtcgen -crlf
  ```

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

Command Line Arguments

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

```
-crlf

-forcewrite

-spacedeps bs | dq | fail | none

-version
```

There are other options for internal use only.

The descriptions for the arguments are:

- 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.
- 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.
- 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).

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

Command Line Build Interface

Rational 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 is useful if the build host is different from the toolset host. Before generating and building an existing model, it is important that the model be validated by the toolset. If a model is valid (for example, there are no unresolved references), you can then generate and build a component from the command line.

To perform outside the toolset:

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.

Classes and Data Types 4

Contents

This chapter is organized as follows:

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 types within your model, whether they are defined within the toolset or not, as long as the type is visible to the compiler.

Terminology

The terminology for data type and class may be confusing at times. Throughout this section, we will use the term **class** for the generic concept of a named definition that encompasses a notion of storage of values, and of operations which may be performed on those values. In C, there is technically speaking no such thing as a class, so the data is implemented as a **struct**, and the methods are implemented as global functions with a pointer to the struct as the first parameter. We will also use the term instance rather than object.

As with any C program, although the toolset can generate classes and type descriptors, the user is still responsible for ensuring that the classes created are well formed. For example, they should not leak memory, and they should have appropriate initialize, cheapened destroy methods defined.

This section provides a pragmatic overview of how to use data classes within Rational Rose RealTime.

Introduction to Sending Data in Messages

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 required signal name (which identifies the message), a priority (relative importance of this message compared to other unprocessed messages on the same thread - default to **General**), and optional application data. If there is application data to send, it can be sent either by value or by reference.

Similarly to operations, which do not always require parameters, messages do not always have to be sent with application data. However, 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 what type of data 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, leave the data class field empty.

Sending by Value

An alternative to sending data by reference is to send it by value. This means that a deep copy of the data is sent instead of a pointer to the data. This option is less efficient but simplifies concurrency issues.

To send data by value, the C Services Library must know how to initialize, copy and destroy instances that are sent. This is where type descriptors come in (see **RTObject_class** for more details). Type descriptors describe the class to the Services Library to allow it to manipulate the instances that it sends.

See Sending/Receiving data by value for an example of the **Send** syntax.

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.

The rules for sending pointers in messages are:

- 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 to other capsules because the stack object gets deleted when the transition code segment completes. Since sends are asynchronous, when the receiving capsule instance dereferences the pointer, the data it is pointing to has been deleted.

See Sending/Receiving Data by Reference for an example of the send syntax.

Considerations

When using data in Rational Rose RealTime, as in any other program that you will write, you must provide well-formed classes. Memory that is allocated on the heap should be deleted at the proper time, and initialize, copy and destroy methods should work as intended.

Classes can be created that have any combination of the following:

- **Sendable by value** - The class can safely be sent between capsules using the init, copy and destroy semantics for the class.
- **Marshallable** - The class can be safely encoded and output via the observability feature to the Rose RealTime Toolset (when tracing a message or inspecting an attribute), or via the log service to a console. And/or it can be safely decoded when received from the toolset (when injecting a message or modifying an attribute).

Note: Any piece of data is sendable by reference - it's a pointer value that's being transferred, and no data initialization, copying, destruction, encoding or decoding takes place.

Data class rule #1

Simple data types that do not contain pointers (any indirect attributes) are by default sendable by value and marshallable.

Data class rule #2

Data types which contain pointers can be made sendable by value and marshallable. You will, however, have to add details to your class to make it well-formed by creating or modifying, when needed, the following functions:

Data type constructor	A construct method will be automatically generated, and populated with each attribute's entry from the 'Initial value' field.
Type descriptor functions (defined for each class under the C TargetRTS tab)	These functions define how a class is initialized, copied, destroyed, decoded and encoded. By default the functions RTstruct_init, RTstruct_copy, RTstruct_destroy, RTstruct_decode, and RTstruct_encode are called. Generally you won't have to modify these type descriptor functions.
NumElementsFunction (defined for each attribute under the C TargetRTS tab)	This is a function which determines (at run-time) the size of an indirect field (the number of things a pointer references), which if left unspecified, will be set to 1. This is used by the encode/decode functions.

Data Classes that are Marshallable

In addition to making data classes sendable by value - see Introduction to Sending Data in Messages - they can be made marshallable. This means that the instance can be encoded and decoded into a string of bytes. This functionality allows the toolset to display the contents of instances at run-time.

When you are debugging a running model and request an attribute or data within a message to be displayed 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 then calls the encode function (defined within the type descriptor) on the class instance. The result of the encode function is passed as is to the toolset and shown in either a **Watch** window or a message **Trace**.

Basic Structures

Simple data classes - see Introduction to Sending Data in Messages - are by default encoded using an ASCII encoder meaning that they are marshallable. However, for data classes which contain attributes of types which are not known by the toolset, these functions must be written by the user. They are not automatically generated by the toolset.

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

C Data Type Examples

This section contains examples which demonstrate the different methods of creating and using data types within Rational Rose RealTime.

Syntax examples of sending data classes between capsule instances

- Sending/Receiving Data by Reference
- Sending/Receiving data by value

Class modeling examples

- Creating a Class Data Member from the Class Diagram
- Specifying Arrays using Association Multiplicity

Creating and using common C constructs

- Creating Array and Pointer Attributes
- Creating a Constant (#define) or a #define
- Creating a typedef
- Creating an enumeration
- Creating a Union

Class Creation Examples

Before starting the examples, please make sure you are familiar with the considerations described in Introduction to Sending Data in Messages.

- Creating and using classes with no pointer attributes
- Creating and Using Classes with Attributes that are Pointers
- Integrating an External Class (not defined in the toolset)

Sending/Receiving data by value

An alternative to sending data by reference, is to send it by value. Meaning that a copy of the data is sent instead of a pointer to the data. This is the preferred method of sending data between capsules. Although this option is sometimes less efficient it does simplify concurrency issues.

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

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.

Sender

```
int result;
SomeClass sendData;
SomeClass_construct( &sendData, "hello" );
/* Given a port called ‘port’ based on a protocol with a
** signal ‘start’ with data class ‘SomeClass’. */
result = RTPort_send( &this->port,
     RTPort_createOutSignal( port, start ),
     RTPriority_General,
     &sendData,
     &RTType_SomeClass );
/* If ‘result’ is > 0, send was successful. */
```

Receiver

```
int result;
SomeClass recData;
result = RTMessage_copyData( this->std.msg,
    &recData,
    sizeof( recData ) );
/* If ‘result’ is > 0, copyData was successful. */
```

Sending/Receiving Data by Reference

Users should be aware of the issues around sending data by reference. (See Introduction to Sending Data in Messages.) Nevertheless, for performance reasons, it is sometimes an effective way of sending data.

The examples below 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.

Note: The most important thing to remember is to **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 following example, the receiver will free the allocated memory.

Sender

```
SomeClass * pSendData = create_SomeClass();
/* Initialize with default values */
SomeClass_construct( pSendData );
/* Given a port called 'port' based on a protocol with a
** signal 'stop' with data class left empty. */
result = RTPort_send( &this->port,
     RTPort_createOutSignal( port, stop ),
     RTPriority_General,
     pSsendData, (const RTObject_class *)0 );
/* If 'result' is > 0, send was successful. */
```

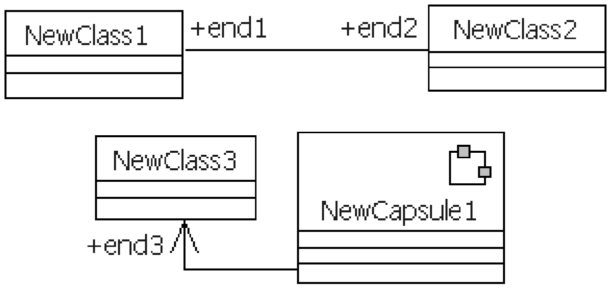
Receiver

```
const SomeClass * pRecData =
   (const SomeClass *)RTMessage_getData( this->std.msg );
/* Free memory when finished with the data */
delete_SomeClass( pRecData );
```

Note: Subtle bugs are possible if the receiver actually writes to the data at the end of the received pointer. This is why the const type modifier is used.

Creating a Class Data Member from the Class Diagram

Given an association between two classes or between a capsule and a class, a data member is created in the generated source code for the classes participating in the relationship.



The above relationships results in the creation of a data member named **end2** in **NewClass1,** as well as another named **end1** in **NewClass2**, and one named **end3** in **NewCapsule1**. The properties for the end (association end) control how the code is generated for the data member; the end affects the class at the other end of the association. For example, assume that **end1** and **end2** are contained by reference. The following code represents a simplified version of the code that would be generated:

```
struct NewClass1
{
    /* {{{RME classItem 'NewClass2' associationEnd 'end2' */
    struct NewClass2 * end2;
    /* }}}RME */
};

struct NewClass2
{
    /* {{{RME classItem 'NewClass1' associationEnd 'end1' */
    struct NewClass1 * end1;
    /* }}}RME */
};
```

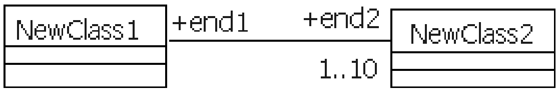
You can specify the containment, visibility, and other attribute features to control how attributes are generated. These are found in the **Association Specification** dialog.

A data member is not generated if...

- the association end name is not specified
- the **Derived** option is checked
- the end is not navigable
- both ends are defined as aggregate

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 multiplicity range specified. If the multiplicity is unspecified (for example, 1..*) the association is forced to be by reference.



For example, assume **end1** is contained by value and **end2** is contained by reference. The following code will be generated for the association:

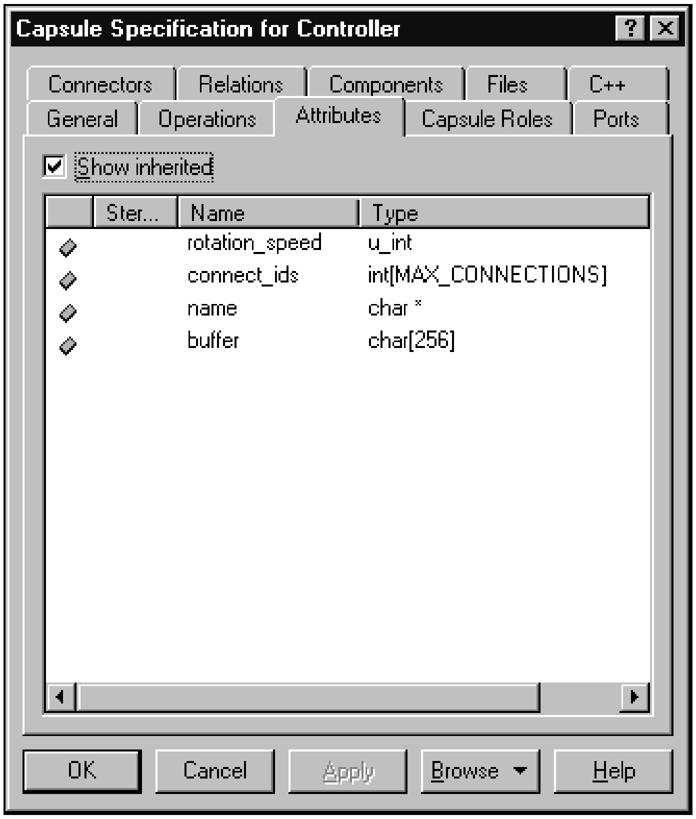
```
struct NewClass1
{
    /* {{{RME classItem 'NewClass2' associationEnd 'end2' */
    struct NewClass2 end2[10];
    /* }}}RME */
};

struct _NewClass2
{
    /* {{{RME classItem 'NewClass1' associationEnd 'end1' */
    struct NewClass1 * end1;
    /* }}}RME */
};
```

Creating Array and Pointer Attributes

Attributes can be created as arrays or as pointers.

Tasks

Create an attribute and set its type to any valid C type. If it is an array, then specify the array size within brackets after the type If it is a pointer then add a star after the type.



Ster...	Name	Type
	rotation_speed	u_int
	connect_ids	int[MAX_CONNECTIONS]
	name	char *
	buffer	char[256]

Creating a Constant (#define)

C constants are implemented as #**defines** and are scoped globally.

Note: Symbolic capsule role and port multiplicity values must be defined using constants created within the toolset.

Examples

The following source code fragment shows an example of a global constant.

```
#define num_retries 4
```

Tasks

To create a global constant:

4 Create an attribute that will be the constant, so name it appropriately.

5 In the attribute's C properties tab, change the **AttributeKind** field to **constant**.

6 In the attribute's detail tab, set the **Initial value** for the constant. The **type** field is ignored, since it's implemented as a #define, so it can be left blank.

7 Add a dependency between the class where the constant(s) are defined and the capsules or classes which use the constant. If the constant is global, ensure that the dependency C properties are: **KindInHeader** = inclusion, and **KindInImplementation** = none.

Note: This mechanism can not be used to create parametrized macros with names containing '(' **and** ')'. We also advise against creating complex macro expressions using this method. If either of these mechanisms are required, create the macro in the class **C** tab for the **HeaderPreface** property instead.

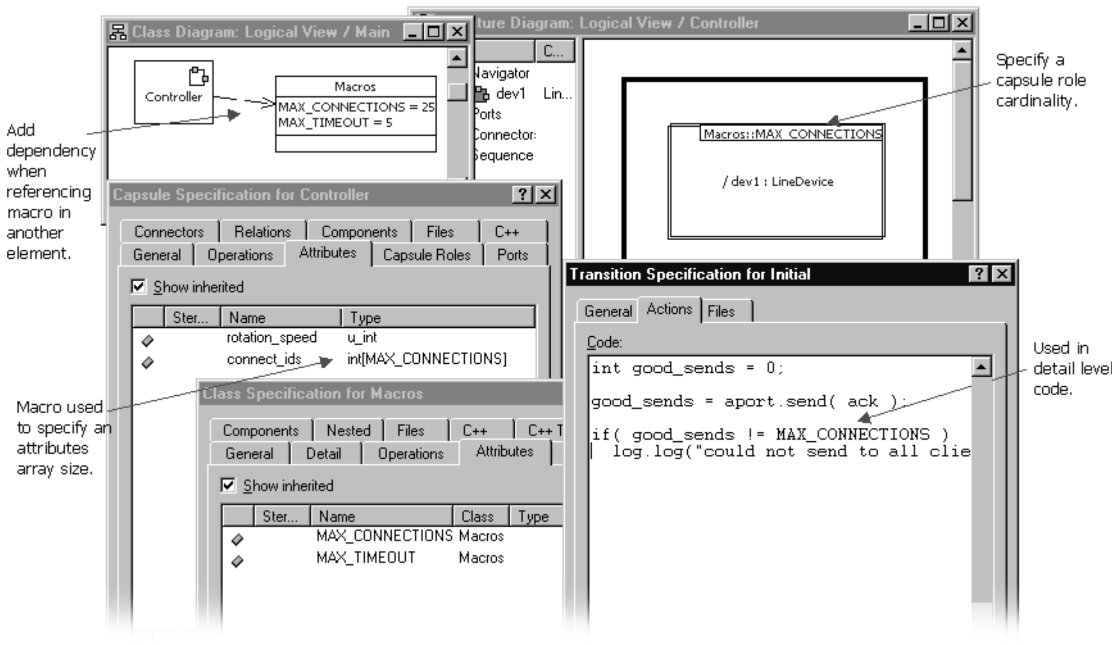
Usage

You can use constants to specify the cardinality of replicated capsule roles, ports, and bindings by adding the fully qualified name (for example, **Package1::ClassX::Constant**) of the constant to the **Cardinality** field in the **Capsule Role Specification** dialog.

Constant values must be specified using the class name of the class in which they have been created because Rational Rose RealTime must 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**.

Note: You can specify any valid C expression in the **Initial Value** field for the constant/define. However, if the constant specifies a cardinality, the constant's initial value must be a literal integer (for example, 2, 50, 100). If the cardinality cannot be understood by the toolset at generation time, a warning is issued and a default value of 0 is used.

If the constants are 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 constants and the elements which reference the constants. Apart from specifying cardinalities, constants can be used as in any C program.


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

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

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).

Creating an enumeration

Example generated code

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

Tasks

1 Create a class named **e.**

2 In the **General** tab of the **Class Specification** dialog, 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 Union

You can create a C union instead of a **struct** or **typedef**.

Example

```
union NewClass3
{
   int theInt;
   float theFloat;
   unsigned long int theUnsignedLongInt;
};
```

Tasks

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

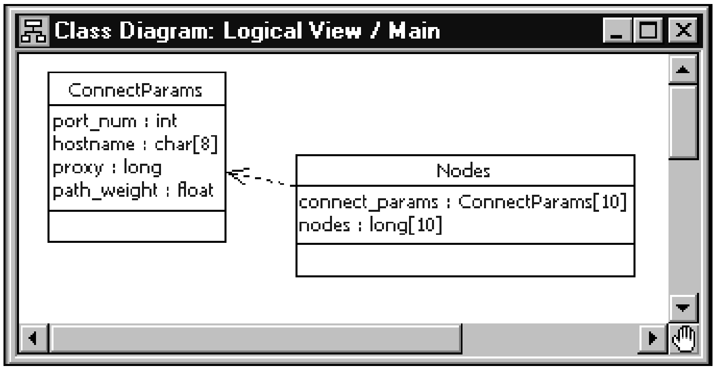
Creating and using classes with no pointer attributes

These classes are:

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

Classes without pointers have the above properties if all of its attributes are of types which do not have pointers or are also well-formed data classes.

Figure 6 Classes Composed of Predefined Types



Usage

In Figure 6, the classes **ConnectParams** and **Nodes** are composed of predefined types (the Services Library knows how to **init**, **copy**, **destroy**, **encode**, and **decode** because of generated type descriptors). The type descriptor generated by the toolset is called **RTType_<class name>** and can be referenced directly in detail level code where an **RTObject_class** is required by a Services Library operation.

Example

```
int result;
ConnectParams conn_p;
ConnectParams_construct( &conn_p, <arguments> );

/* 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'.
*/
result = RTPort_send( &this->port,
     RTPort_createOutSignal( port, connect),
     RTPriority_General,
     &conn_p,
     &RTType_ConnectParams );

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

Creating and Using Classes with Attributes that are Pointers

If you provide a **CopyFunctionBody** and a **DestroyFunctionBody** (Class, C TargetRTS), the class can be sendable by value.

If you also provide the **NumElementsFunctionBody** (Attribute, C TargetRTS), the class can be marshallable (can be observed, inspected and injected).

If you do not provide any of these operations, the class should *never* be sent by value. That would cause incorrect behavior, and possibly a system crash.

Note: If a class has attributes which are pointers, ensure that the memory is managed properly by the class. Rational Rose RealTime does not create a destroy function to delete allocated memory; you will have to write your own destructor/constructor.

When attributes are pointers, there is an additional step required to make them sendable and marshallable. This extra step is required because pointers can be pointing to anything, and the Services Library cannot predict how many things the pointer references. You will have to help the Services Library determine how many things the pointer is pointing to.

Integrating an External Class (not defined in the toolset)

If you have classes defined outside of 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 with Rational Rose RealTime and used for class modeling. The are also available in the drop-down type lists, or can be used within detail level code.

Note: Any class or type defined outside the toolset can be used in your model, and depending on how the class or type is used in your model, there are a couple of ways that the class or type will have to be integrated with Rational Rose RealTime.

Integration Questions

Before integrating classes into Rational Rose RealTime, determine how the class or data type will be used within the model using the following case criteria:

1. Will objects of this type only be used to store information within a single capsule instance, or only sent by reference never to 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 inspected/modified?

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 cases #2 and #3

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

There are essentially two possibilities for handling an externally defined class or data type: either you create a class within Rational Rose RealTime with the same attributes as the external class and let Rational Rose RealTime generate the type descriptor, or you add the code yourself for describing how to **init**, **copy**, **destroy**, **encode**, and **decode** an instance of this type.

An external class can be made sendable by value without being observable and vice versa.

- Integration option 1: Describing an External Type to Rational Rose RealTime
- Integration option 2: Providing Marshalling Functions

Integration option 1: Describing an External Type to Rational Rose RealTime

If the class is described to Rational Rose RealTime it can be made marshallable (can be observed, inspected and injected).

If your external class has well defined **init**, **copy** and **destroy** methods, then the class can be (the default type descriptor will use the operations already defined on the class) sendable by value.

Example: External Definition

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 */
struct Ext_Simple
{
   int a;
   char b[80];
   float c[8];
};
```

Tasks

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

To describe an external type:

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

2 In the class **C** tab, ensure that the **GenerateClass** option is not selected.

  Because the class is already defined outside the toolset, you will not want another class to be generated, you are merely describing the type to Rose RealTime.

3 In the class **C** tab, make the header file which contains 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**.

  The next step will allow the C code generator to create marshalling functions for the external class. This is only required to encode/decode the class.

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

Note: If the external class contains pointers you will also have to follow the steps in creating attributes as arrays and pointers to correctly define the attribute and ensure that the external class has a well-formed (no memory leaks) init and destruct methods.

Integration option 2: Providing Marshalling Functions

Instead of having to redefine all the attributes defined in an external class to allow an external data type to be marshalled (as described in the integration option 1), a data type can be integrated for marshalling with Rational Rose RealTime if it already contains operations to encode and decode to and from a string of bytes.

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) and **EncodeFunctionBody** (Class, C TargetRTS).

When writing type descriptor functions, you will have access to a pointer to an instance of the class (target), and in some cases both a target and a source object instance (the source can not be modified in this case). To demonstrate how these can be used see the Integrating data example model.

Note: Ensure that the external class has well defined **init**, **copy** and **destruct** methods, and call these from within the **InitFunctionBody**, **CopyFunctionBody** and **DestroyFunctionBody** properties, respectively.

Tasks

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

2 In the class **C** tab, set the **GenerateClass** to **false**.

3 Make the header file which contains the actual class definition visible to this class by adding **#include <An_External.h>** to the **HeaderPreface** property.

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. For additional information, see **EncodeFunctionBody** (Class, C TargetRTS) for an example.

6 In the class **C TargetRTS** tab, edit the **DecodeFunctionBody** (Class, C TargetRTS) property. Add code to decode the data class. For additional information, see **DecodeFunctionBody** (Class, C TargetRTS) for an example.

Because the **GenerateClass** property was set to **false**, only a type descriptor will be generated for this new type. Moreover, it is important that the class definition in the external header file is visible to the compiler.

C Services Library

5

Contents

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 referenced configurations, so that your model can be readily ported to different target configurations.

This chapter is organized as follows:

C Services Library Framework

Taken together the classes and data types defined in the C Services Library provide an application framework - the framework in which your application will run.

At a very general level, the framework defines the skeleton of a real-time application: **messaging**, **timing**, **concurrency**, **event based processing**, **platform independence**. Your job as a Rational Rose RealTime developer is to fill in the rest of the skeleton - the classes, capsules, and protocols which are specific to your system.

The Big Advantage

Now you can understand the power of code generation. With Rational Rose RealTime, you will develop your application in a high level language using **State** diagrams and **Structure** diagrams, and automatically these elements are converted to C and placed in a framework which already 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 3 main services to our application:

- Communication Services is the basic mechanism for using message-based communication via ports.
- Timing Service provides general purpose timing facilities. It also provides an interface for implementing custom timer capsules.
- Log Service is a general purpose logging service.

Services are explained by introducing the general concepts related to the service followed by the functions that are used to implement the service. You should become familiar with the C syntax and notational conventions used in these sections as well as the *Services Library API Reference* on page 151.

Message Processing

An event is a message arriving on a capsule's port. Message-based communication is the basic mechanism for communication between capsules. Only aynchronous communication between capsules is supported in the C Services Library. Messages are also used by the Services Library to communicate with the capsules in the model.

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 Rational 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 Rational Rose RealTime models run on a machine with a single processor, or possibly in a distributed environment, with a few processors. In any case, there are almost always more capsules than processors. Thus, the capsules must share the processor in some manner.

Types of Concurrency

The underlying operating system provides preemption to allow concurrent programs to share the processor in a fair way, where 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 on).
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). Processes usually have a significant amount of protection such that if one process crashes it does not affect any other processes. 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

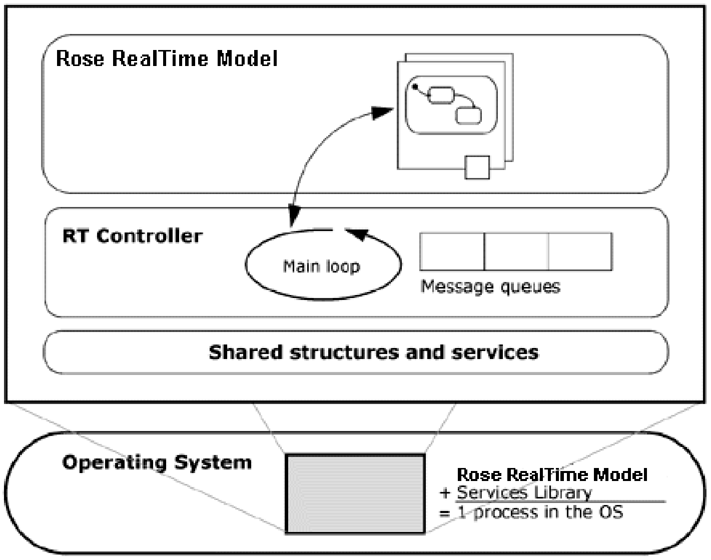
Rational Rose RealTime allows designers to make use of the underlying multi-tasking operating system so that the processing of a capsule on one thread does not block the processing of capsules on other threads. Designers can specify the physical operating system threads onto which the capsules will be 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. Figure 7 shows the basic structure of the single-threaded Services Library.

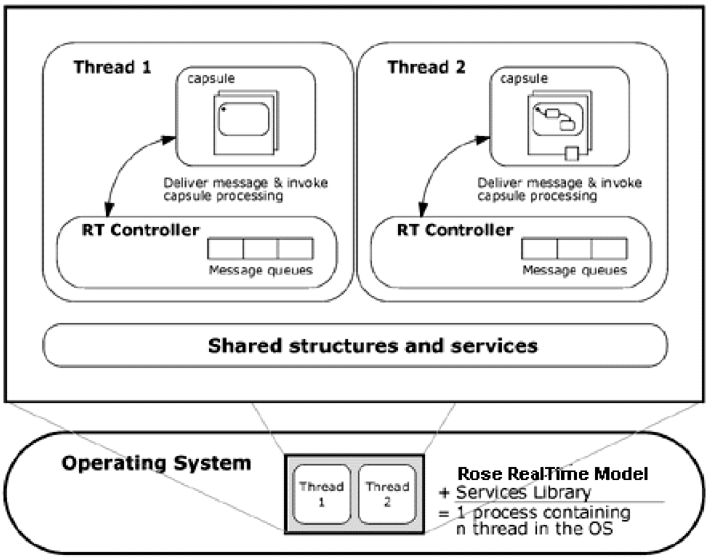
Figure 7 Single-Threaded Services Library



Multi-Threaded Services Library

In this version, capsules can belong to different logical threads. Logical threads are mapped to a set of concurrent physical threads defined by the user. 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 8.

Figure 8 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

C Services Library Framework

The capsules, capsule roles, protocols, ports and classes in a Rational Rose RealTime model will eventually be generated to C code and integrate into the C Services Library framework. The framework provides a set of pre-defined data structures and functions which you will use in the detail level code of your model.

The complete API is explained in the chapter called *Services Library API Reference* on page 151.

As well as the reference material detailed in the API the following characteristics of the C Services Library framework are important to understand:

- *Capsules are Generated as Subclasses of RTCapsule* on page 57
- *Ports are Generated as Fields of a Capsule Structure* on page 57
- *Every Capsule Instance has Access to its Controller* on page 58
- *Capsule Instances, Logical, and Physical Threads* on page 58
- *Capsule Instances Have Access to a RTMessage Object* on page 59

Capsules are Generated as Subclasses of RTCapsule

Every generated capsule structure contains, as the first field in the structure, an **RTCapsule** called **std**. Thus, for any API function that requires an **RTCapsule *** as a parameter, you can either **cast** the capsule instance's '**this**' pointer, or pass the address of the **std** field. For example, the **RTCapsule_context()** function, which requires a **RTCapsule** pointer, can be called in the following syntax:

```
/* both expressions are equivalent */

RTController * rts1 = RTCapsule_context(&this->std);

RTController * rts2 =

   RTCapsule_context((const RTCapsule *)this);
```

Ports are Generated as Fields of a Capsule Structure

The ports on the structure of a capsule are generated as **RTPort** fields in the generated capsule structure. The field is named exactly as the port is named in the model. Most communication service functions require that you specify a port as a parameter to the function. For example, the asynchronous send function has the following prototype:

```
int RTPort_send  ( const RTPort *, RTSignal, RTPriority, void *,
const RTObject_class * );
```

The first parameter is a pointer to a port. Therefore you would access the port via the capsule instance pointer this, and send a message out of that port using the following syntax:

```
/*

   Assume a port called 'control' that has

   an out signal 'ack'.

*/

RTPort_send( &this->control,

   RTPort_createOutSignal( &this->control, ack ),

   RTPriority_General,

   (void *)0, /* don't send data */

   (RTObject_class *)0 );
```

Every Capsule Instance has Access to its Controller

Each capsule instance has access to the controller for the thread on which it is running. The **RTController** class provides several functions that can be useful in a capsule's implementation. The function **RTCapsule_context()** returns a pointer to the controller instance, which can then be passed to **RTController** functions.

For example, to find out the name of the thread on which a capsule is running, you would use the following function:

```
/*
   This code would be in a capsule's
   transition
*/
char * name =
   RTController_name( RTCapsule_context( this ) );
```

Capsule Instances, Logical, and Physical Threads

As described in the *Introduction to Threads* on page 53, your application may required that certain capsule instances run on separate physical threads. Logical threads are used to represent a conceptually independent thread of execution. Logical threads may be mapped to different physical thread configurations when generating an executable. However, the mapping of capsule roles is defined purely in terms of logical threads.

Since all C capsule instances are created when a model is run, the mapping of capsule instances to logical threads must be provided at design time. The top level capsule is where you defined the logical threads and map the capsule instances to logical threads. The top level capsule is always mapped to the MainThread, and you cannot map it to any other. See *Capsule To Logical Thread Mapping (Capsule, C Executable)* on page 139 for details of how to work with logical threads.

Note: You must use this same process to map a timer capsule role to its own logical thread. This logical thread can then be mapped to a separate physical thread. You then have a timer capsule running on its own physical thread.

The mapping from logical thread to physical thread is performed on a component. The component uses the logical thread information contained within the top level capsule assigned to that component, and allows you to map the logical threads defined in the top level capsule to physical threads. See *PhysicalThreads (Component, C Executable)* on page 141 for details of how to work with physical threads.

Note: Only logical threads defined on the top level capsule are considered by the component.

Capsule Instances Have Access to a RTMessage Object

Every capsule has an attribute **msg** which is 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. In your detail level code, you will first retrieve the message using **RTCapsule_getMsg(this)**; then use the **RTMessage** methods to query the message.

Log Service

The Log Service is organized as follows:

- *Implementation Functions* on page 59
- *Characteristics* on page 59

Implementation Functions

RTLog

Characteristics

The Log service is a stream of **ASCII** text in which system or application events can be recorded. The Log output is directed to the stream **RTSTDIO_STREAM**, which is defined as **stdout** in:

$ROSERT_HOME/C/TargetRTS/src/include/RTPriv/Stdio.h

You can change it to **stderr**, but this change will also affect all calls to **RTStdio_put** used internally in the Services Library.

There is a **Log** method for each basic C data type, and a generic **Log** method for user-defined data types. Each call to a **Log** method involves locking **RTStdio**, writing the resulting text, flushing the output, and unlocking **RTStdio**.

Communication Services

Communication Services is organized as follows:

Implementation Functions

RTMessage, **RTPort**, **RTPriority**

Concepts

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

The Communication Service is accessed by calling the **RTPort** functions. The port name is the user defined name of the port declared in the model. The named port is generated as a field 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.

A service request results in the creation of an instance of **RTMessage**. This message is delivered by the Services Library to the port at the other end of the connection. It is eventually processed by the behavior of the capsule containing that port.

Primitives

This service is used for messages 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 calling the **RTCapsule_getMsg()** method.

Upon processing a message received at a particular end port, the **RTMessage_getPortIndex()** method returns an index to the particular port instance that received the message. Calling **RTPort_sendAt()** on the port instance returned by **RTMessage_getPortIndex()** results in a send to only that particular port instance.

Communication Service properties

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

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.

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.

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 to 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.

Note: 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:

- **RTPriority_Panic** - highest priority available to users; to be used only for emergencies
- **RTPriority_High** - for high-priority processing
- **RTPriority_General** - for most processing
- **RTPriority_Low** - for low-priority
- **RTPriority_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 the 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 [published|unpublished] 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 fails).

SPPs are often replicated, with their multiplicity specifying the maximum number of clients that can be bound to the server at run-time; otherwise, no SAPs can be bound. By default, a SAP or SPP is automatically registered under its reference name when the capsule containing that SAP/SPP is initialized.

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 deregistering, an SPP with a larger multiplicity registering.

Registration String

A registration string is used to identify a unique name and service under which SAPs and SPPs will connect, and can be of any length > 0.

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 according to the algorithm described in the previous section.

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

The Timing Service is organized as follows:

- *Implementation Functions* on page 64
- *Characteristics* on page 64
- *Usage* on page 65
- *Timer Thread Configurations* on page 66
- *Customizing the Timing Service* on page 66
- *Timing Precision and Accuracy* on page 67

Implementation Functions

RTTimespec, **RTTimerId**, **RTPort_informIn**, **RTPort_cancelTimer**, **RTPort_isTimerValid**

Characteristics

The timing services provide a way for a user to specify a timeout. After a timeout has occurred, a timeout message is then delivered to a timing port on the originating capsule. It's possible to keep track of a specific timeout through it's **RTTimerId** which is returned by the **RTPort_informIn** function. You may then check that the timeout is still valid, or cancel it via this index. The timing services also allow one to get the current system time into a **RTTimespec** structure, and to perform arithmetic operations on that structure.

Usage

The implementation of a timing service is very much dependent on the timing interface provided by an operating system. For this reason, the timing solution provided with the C Language Add-in is tailored to be easily customizable. The Services Library does not contain the timing algorithms and data structures, instead the Services Library acts like a dispatcher of timing messages by calling timing functions which have been registered on a controller. The **RTController** functions allow registration of timing functions with a controller, thus when subsequent timing requests are received by the controller, the controller calls the timing functions that have been registered.

The C Language Add-in provides a generic timer implementation which is supplied in the **RTCClasses** package. The complete implementation for the timers is in the classes and data structures in this package, this allows easy customization of timers from within the toolset.

Follow these steps to add timing services to your model:

1 Decide which timer configuration you require. Do you want a timer on:

- each thread
- one timer for all threads
- the timer on it’s own thread servicing all other threads

You must consider the timing requirements for your application, and take into consideration the overhead of having the timer capsule on its own thread.

2 Drag the appropriate timer capsule (Timer or SelfTimer) from the **RTCClasses** package into the structure of a capsule in your model.

Note: The **RTCClasses** package should be included by default in all models. If you are migrating or have deleted the package, you can share the package back into your model by selecting the **Logical View** and right-clicking, then select **File > Share**. Browse to the $ROSERT_HOME/C directory, and select the file called RTCClasses.rtlogpkg. This will share the package that contains the C timer implementation capsules and data classes.

3 Create a port based on the **CTiming** protocol on each capsule which will be using the timing services.

4 Use the **RTPort_informIn** function to request a timer.

5 When the timer expires, a **timeout** signal will be sent via the **CTiming** protocol port that was passed as an argument to the **RTPort_informIn** function.

6 Add a transition to your model to handle the receipt of **timeout** signals.

Timer Thread Configurations

Every application has different requirements in terms of timing. For this reason it will be important to consider how timers will be configured in your model. You should carefully consider the performance requirements you require from your timers; depending on this requirement you can chose from the following common timer configurations:

- **One timer for the entire model.** The timer runs on main thread and services all threads in the model. This behavior is implemented by the Timer capsule provide in the **RTCClasses::TimerPackage**. If you add this capsule to your model it will register with all threads and by default be incarnated on the main thread.
- **One timer per thread.** The timer runs on the thread it is assigned to and provides timing services on that thread only. This behavior is provided by the **SelfTimer** capsule provided in the **RTCClasses::TimerPackage**.
- **One timer for whole model, but runs on it's own thread.** In this case, use the **Timer** capsule but map to it's own logical and physical thread.

Customizing the Timing Service

Although there are platform independent timing capsules available with Rational Rose RealTime, you may wish to implement your own timing capsules. A timer capsule does not need to have any internal structure, or any state machine. It registers certain functions with the Services Library during system start-up, and those functions modify the behavior of a thread when it would normally perform a wait. Instead of performing a wait it can perform a timed wait, and then send timeout messages when a timeout occurs.

Note: To understand how to implement a timer capsule, you can browse through the **Timer** capsule provided in the **RTCClasses** package.

To create a timer capsule, you must create functions with the following signatures:

-
  ```
  RTTimerId informIn( RTCapsule * this, RTPort * replyToPort,
  RTTimespec * timeout, void * data, const RTObject_class *
  type )
  ```

  Creates a **timeout** and puts it into the active timeout queue.

-
  ```
  int cancel( RTCapsule * this, RTTimerId timerId )
  ```

  Cancels a **timeout**.

- `int valid( RTCapsule * this, RTTimerId timerId )`

  Checks if a **timeout** is still active.

- `void sleep( RTController * this )`

  Checks for expired timers and sends **timeout** messages. Does timed wait on lowest timeout in queue until expiry or a signal.

- `void wakeup( RTController * this )`

  Signals a sleeping thread to wake up.

- void setup(void)

  This function performs all the initialize functions necessary for the timer to be fully operational towards any timing requests it may receive. The setup function is called before any initial transition in the model.

You will also need to allocate the supporting structures for your functions, like timer queues and **mutexes**, and you will probably want to create some supporting functions to modularize your code as well.

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 Windows.

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.

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.

Accessing the Error Value

The error enum identifier for the current error can be obtained via **RTController_getError()**. A description of the current error code can be accessed by calling the operation **RTController_strError()** on the current controller object. The controller object for any capsule can be retrieved by calling the **RTCapsule_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( !   RTPort_send( &this->port,

        RTPort_createOutSignal( port, start ),

        RTPriority_General,

        &sendData,

        &RTType_SomeClass ) )

{

switch( RTController_getError(sendingController) )

   {

      case RTController_noConnect:

         RTPort_informIn( timingPort, 1, 0, 0, 0 );/*try later*/

         break;
      default:

         RTStdio_putString( "Unexpected send error: ");

         RTStdio_putString( RTController_strError( sendingController
) );
```

```
            RTStdio_putString( "\n" );
            break;
        } /* switch */
    } /* if */
```

Error Enumeration

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

```
typedef enum _RTController_PrimitiveError
{
    RTController_ok, /* all */
    RTController_internalError, /* all */
    RTController_unexpectedStatus, /* debugger ops */
    RTController_unexpectedPrimitive, /* debugger ops */
    RTController_cannotSetTimer,
    RTController_cannotRegTimer,
    RTController_unauthorizedMemoryAllocation,
    RTController_alreadyDeferred, /* CommDefer */
    RTController_badClass, /* CommDeliver, CommSend */
    RTController_badId, /* TimerInform */
    RTController_badOperation, /* LayerDeregister */
    RTController_badMessage, /* CommSend */
    RTController_badSignal, /* CommSend */
    RTController_badState, /* CommSend */
    RTController_badValue, /* TimerInform */
    RTController_dereg, /* LayerDeregister */
    RTController_noConnect, /* TimerInform */
    RTController_noMem, /* all */
    RTController_prio, /* CommSend */
    RTController_reg, /* LayerRegister */
    RTController_tooManySAPs /* LayerRegister */
} RTController_PrimitiveError;
```

RTController_alreadyDeferred

A message can only be deferred one time within the chain of transitions it triggers. Subsequent calls fail and set the error code to this value.

RTController_badClass

This is not set anywhere at this point. The error should be set when there is an incompatible subclass detected.

RTController_badId

The **cancelTimer** primitive of the timing service requires a valid timer identifier returned by the **informIn** primitive. 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.

RTController_badOperation

This is not set anywhere at this point. It should be set when the controller attempts to execute a primitive it's not permitted to execute.

RTController_badMessage

This is set when a capsule receives a message for which it has no event handling.

RTController_badSignal

An unindentifiable message without any capsule information occurred. Since the message was destined for the controller, but did not fall into the types of message a controller knows how to handle, it is a bad signal.

RTController_badState

A capsule has entered an invalid state.

RTController_badValue

This is not currently set anywhere. An invalid argument has been passed to the controller.

RTController_cannotRegTimer

Registering a timer service has failed since there is already another timer registered.

RTController_cannotSetTimer

Attempting to set an interval timer has failed.

RTController_dereg

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

RTController_internalError

This error signifies that there are null function pointers for necessary functions, unbound end ports on sends, or unrecognized controller wait options when checking for events.

RTController_noConnect

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

RTController_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.

RTController_ok

So far, no error conditions have occurred. This value is set during controller construction.

RTController_prio

Send and **informIn** primitives accept an argument which 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.

RTController_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.

RTController_unauthorizedMemoryAllocation

A call to **RTMemoryUtil_new** has been made to allocate memory, yet the Run Time system is in the executing state, so no memory allocations should occur at this point.

RTController_unexpectedStatus

The controller has its state field set to something unrecognizable.

RTController_unexpectedPrimitive

The debugger is trying to trace a controller operation, but the primitive that the controller is doing is unrecognizable by the debugger.

Running Models on Target Boards 6

Contents

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 exist in the target operating system before you try to run a model on your target RTOS.

This chapter is organized as follows:

Overview

The C Services Library ships with supported configurations for a set of target processors, operating systems, and compilers. See the *Installing and Configuring Guide* for a list of the referenced target configurations. You may have to configure and customize the libraries included with Rational Rose RealTime 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 Toolchain Functionality

A functioning development environment must be in place prior to building and running models with Rational Rose RealTime. You should be able to compile, load, and execute non-Rational 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 included with your compiler.

You will need to configure 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.

Testing your Toolchain

To ensure that your toolchain is configured properly, 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 configured properly. Your RTOS usually includes 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. Refer to **Step 3** for additional information.

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

Step 3: Verify main.c

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 following file:

$RTS_HOME/src/target/<target name>/Main/main.c.

On many configurations (platforms), this is the code for the **main** function, which simply passes **argc** and **argv** directly. However, on other configurations, 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/TORNADO2/Main/main.c 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.c.

If your configuration 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 Exec) property.

Step 4: Try manual loading

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

Note: With some target operating systems, when a Rational Rose RealTime model is built, you are not finished. In some cases, as with pSOS+, the Rational Rose RealTime model is built as a library and you have to compile and link the board support package with the Rational 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.

Note: To compile for a specific configuration, ensure that a C Executable component is created in Rational Rose RealTime with the correct **TargetConfiguration** set to the library for your configuration. This will instruct the code generator which build scripts and libraries to use.

After 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 the file main.c 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) 1993-2002 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 configured properly and that your RTOS is configured correctly.

Step 5: Running with Observability

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
```

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

When the model is started with -**obslisten** it will not start running the model until you have connected to the model via the toolset, and clicked 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) 1993-2002 Rational Software
rosert: observability listening not enabled
```

```
****************************************************************** *
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 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**, then right-click on the Component instance and select **Attach Target**. The **RTS Browser** will appear. Click the **Start** button and you can use the observability tools to watch the execution of your model.

The target guide is meant to help developers build, compile, debug, and deploy their models to a target system. The C Services Library is at the heart of the C Language Add-in. It will be essential that you understand its architecture if you are to start optimizing and configuring it for your project requirements.

Command Line Model Debugger 7

Contents

This chapter is organized as follows:

Overview

The Services Library debugger provides a mechanism to allow UML for Real-Time models executing on the Services Library to be debugged at the UML for Real-Time concept level. The Services Library debugger does not provide source-level debugging. Source code debugging requires an external source level debugger for C, such as gdb.

Note: Some versions of the Services Library libraries are supplied with the command line debugger disabled for optimum efficiency. You can recompile the Services Library source code to configure the Services Library without the debugger. This saves some space in the executable model. For additional information, see *Configuring and Customizing the Services Library* on page 111.

Note: The debugger must be configured for Observability to be enabled.

Starting the Run-time System Debugger

URTS_DEBUG parameter

You can use the **URTS_DEBUG** parameter to initialize the Services Library 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 Services Library 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. Use semicolons (;) to separate multiple commands.

Differences Between Single-threaded and Multi-threaded Services Library Debugger

In single-threaded mode - that is, when using a Services Library 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 threads related to timing and external layer 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** 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 that can be passed to the debugger (**-URTS_DEBUG**).

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. Rational Rose RealTime provides an interface within the toolset that allows you to specify start-up 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

- **tasks** - prints the list of tasks (threads)
- **detach <taskId>** - do not monitor a thread specified by **taskId**. Allows the thread to run freely.
- **attach <taskId>** - monitor a thread specified by **taskId**. TaskIds of the different physical threads in the model can be determined using the `tasks` command.

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 <testId>** - prints the run-time statistics for thread `taskId`.

Control Commands

- **exit** - terminates the Services Library process
- **go [<n>]** - delivers n messages
- **step [<n>]** - delivers n messages
- **quit** - quits debug mode. Allows all tasks to run freely.

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).

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** command. 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 set-up:

```
Rational Rose RealTime C Target Run Time System
Release 6.40.C.00 (+c)
Copyright (c) 1993-2002 Rational Software
rosert: observability listening not enabled

RTS debug: ->
```

Thread Commands

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

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: ->det 1
Task 1 detached

RTS debug: ->
```

Informational Commands

saps

Lists all registered unwired ports (SAPs and SPPs).

```
RTS debug: ->saps
Name: prot2
 SAP: Compile_OnTop[0]/prot2[0]
 SPP: echo2[0]/prot2[0]

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 include the <*depth*> parameter, you must include the <*capsuleId*> parameter as well.

Each capsule is displayed in the following format:

```
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 and roles. 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)


Components:
   0: servus


RTS debug: ->
```

printstats <taskId>

Prints information about the number of messages delivered, outstanding messages, and a breakdown of messages by priority. The alias **stats** is mapped to this command.

```
RTS debug: ->print 0
main
No error.
   messages[Synchronous] : 0
   messages[System] : 0
   messages[Panic] : 0
   messages[High] : 0
   messages[General] : 1
   messages[Low] : 0
   messages[Background] : 0
```

For 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.

Tracing Commands

log <category> <detail-level>

The **log** command turns **ON** the logging of all 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
```

The following example shows an example of a message trace:

```
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**
- Layer: **Register SAP**, **Deregister SAP**, **Register SPP**, **Deregister SPP**
- Timer: **Cancel**, **InformIn**

Note that the detail levels are as follows:

- **none** - suppresses all log messages
- **all** - logs all events as described above.

Control Commands

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).

Inside the C Services Library 8

Contents

This chapter includes 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 111.

This chapter is organized as follows:

Organization of the Services Library Source

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.

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 and the libset name.

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

For example:

```
TORNADO2S.ppc-cygnus-2.7.2-960126

SUN5T.sparc-gnu-2.7.1

NT40T.x86-VisualC++-6.0
```

Target Base Name

The target base name identifies the operating system, and it's configuration and version. For this reason the target base name is made up of three parts which describe the operating system (OS), the os name, the os version, and the os configuration (single (S), multi-threaded (T)):

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

For example:

```
TORNADO2S -> Tornado 2.x Single-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, and the compiler version.

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

For example:

```
ppc-cygnus-2.7.2-960126 ->
PowerPC processor using Cygnus version 2.7.2-960126

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:

```
TORNADO2S.ppc-cygnus-2.7.2-960126 ->
For the Tornado 2.x Single-threaded RTOS running on a PowerPC processor
using Cygnus version 2.7.2-960126
```

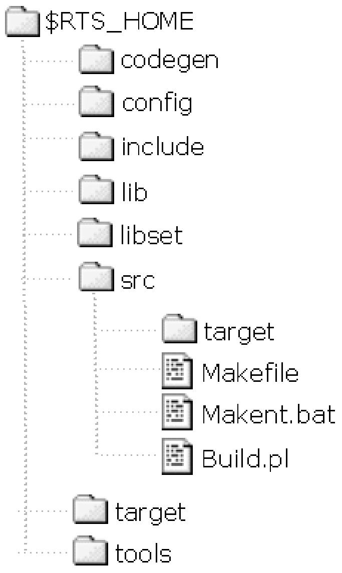
This naming scheme is used throughout the C Services Library.

Directory Structure

The source structure basically contains directories that mirror the convention described in the library *Configuration Naming Convention* on page 94. For example the libset directory contains libset specific files (processor, compiler), the same goes for the target directory (operating system).

To better understand the directory structure, browse through it yourself.

Figure 9 Example C Services Library Directory Structure



codegen

This directory contains scripts for compiling models on different configurations (platforms).

include

This directory contains interface definitions for library classes and structures.

config

This directory contains platform specific (operating system and compiler) configurations. Each platform (see *Platform Name (or configuration)* on page 94) has its own directory that contain the platform specific scripts and configuration files.

target

This directory contains target (operating system) configurations. Each target (see *Target Base Name* on page 94) has its own directory that contain the target specific scripts and configuration files.

lib

This directory contains the compiled libraries.

libset

This directory contains processor and compiler specific configurations. Each libset (see *Libset Name* on page 94) has its own directory that contain the libset specific scripts and configuration files.

src

This directory contains the generic (platform independent) source files for the Services 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 (see *Target Base Name* on page 94) has its own directory that contains target specific source files.

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 the following 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 most compilers, as they can use the default $RTS_HOME/include/RTLibSet.h file.

Any macros defined in these files will override the corresponding macro defaults which appear in $RTS_HOME/include/RTPubl/Config.h. The macros and their default values are listed in the following pages.

Note: In the following section, in general, defining a symbol with the value 1 enables the feature the symbol represents, defining it with the value 0 disables the feature, and leaving it undefined means it will get a default value from $RTS_HOME/include/RTPubl/Config.h.

DEFAULT_DEBUG_PRIORITY

Possible values: Any valid thread priority for the OS in question

Default value: Dependant upon OS and values in RTTarget.h file

Description: Thread priority of the debug thread to be used as a parameter to the OS call used to create the **Debug** thread.

DEFAULT_MAIN_PRIORITY

Possible values: Any valid thread priority for the OS in question

Default value: Dependant upon OS and values in RTTarget.h file

Description: Thread priority of the debug thread to be used as a parameter to the OS call used to create the **Main** thread.

DEFAULT_TIMER_PRIORITY

Possible values: Any valid thread priority for the OS in question

Default value: Dependant upon OS and values in **RTTarget.h** file

Description: Thread priority of the debug thread to be used as a parameter to the OS call used to create the **Timer** thread.

INTERNAL_LAYER_SERVICE

Possible values: 0 or 1

Default value: 1

Description: This enables SAP/SPP functionality. If a model has no Services (unwired ports), and relies solely on wired ports, you can disable this option to save space.

MAX_NUM_SPPS

Possible values: 0 or more

Default value: 10

Description: This defines how many SPPs are possible in the model. You can lower this value, if required, or if you require more than the default, you can increase it.

RTS_NAMES

Possible values: 0 or 1

Default value: 1

Description: Target Observability and debugging require a lot of strings to make the Run Time System presentable to a human being. If you want to save space in your final shippable executable, you can compile out a lot of these strings by setting this macro to 0.

Turning this definition off will minimize footprint. It is up to those who make the models that use this configuration to not use the API that refers to the names of objects, or at least capture these calls in the following code blocks:

```
#if RTS_NAMES
code...
#endif
```

TIMING_SERVICE

Possible values: 0 or 1

Default value: 1

Description: Enables all the code required for supporting a timing service.

TO_OVER_TCP

Possible values: 0 or 1

Default value: 1

Description: This flag should be set to 1 when Target Observability is run over TCP, 0 otherwise. It is used to compile code required for the **tcp** stack and the supporting functionality.

USE_THREADS

Possible values: 0 or 1

Default value: not set, must be defined in the platform headers (usually RTTarget.h)

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.

LOG_MESSAGE

Possible values: 0 or 1

Default value: 1

Description: Controls whether the debugger will log the contents of messages.

MULTIPLE_PRIORITIES

Possible value: 0 or 1

Default values: 1

Description: When this feature is enabled, the Services Library creates multiple priority queues as opposed to one priority queue. Higher priority messages will be processed before lower priority messages.

OVERRIDE_BASIC_SIZES

Default value: undefined

Possible values: defined or undefined

Description: If defined in RTTarget.h, can be used to override the basic sizes of many **RTS** types. Check in $RTS_HOME/include/RTPubl/Config.h for the types that are meant to be overridden.

OBJECT_DECODE

Possible values: 0 or 1

Default value: 1

Description: Enable the conversion of strings to objects, needed for Target Observability and message injection.

OBJECT_ENCODE

Possible values: 0 or 1

Default value: 1

Description: Enable the conversion of objects to strings, needed for Target Observability and variable inspection.

STDIO_ENABLED

Possible values: 0 or 1

Default value: 1

Description: If you disable this define, you can remove all I/O operations that the Target Services Library generally performs. This can save a substantial amount of code space, and makes a lot of sense to disable if your final target doesn't have any **stdio** output mechanism.

RTS_CLEANUP_MECHANISM

Possible values: 0 or 1

Default value: 1

Description: During system shutdown, you might want to clean up all the resources the Target Services Library allocated during start-up and during the execution of the model. This might be especially important if you use a tool like Purify and want to match up all the allocations and deletions, and only see the inconsistencies. But, if you are more concerned with saving space, disabling it gets rid of a substantial amount of cleanup code.

RTS_COMPATIBLE

Possible values: 521 or undefined

Default value: 521

Description: If this value is set to 521, then the **ROOM** macros in $RTS_HOME/include/RTPubl/UMLRT.h used in the officially published interface of ObjecTime Developer 5.2.1 will continue to work.

RTS_MEMORY_POLICY

Possible values: **RTS_CAN_ALLOCATE**, **RTS_WARN_ALLOCATE**, **RTS_NEVER_ALLOCATE**

Default value: **RTS_CAN_ALLOCATE**

Description: Generally, you don't want to allocate memory in a Real-Time system after system initialization has completed. By toggling this flag, you can easily check if it is being allocated for some reason, or explicitly forbid it, making the call fail. Alternatively, if your system is not excessively concerned about memory allocations after start-up, you can allow it.

MESSAGE_DEFERRAL

Possible values: 0 or 1

Default value: 1

Description: If enabled, activates the capability to defer processing a message received until a later time. An explicit **RTMessage_defer** call must be made to actually defer a message.

OTRTSDEBUG

Possible values: **DEBUG_NONE** or **DEBUG_VERBOSE**

Default value: **DEBUG_VERBOSE**

Description: Determines whether the Services Library debugger should be enabled. If set to **DEBUG_VERBOSE**, makes it possible to log all important internal events such as the delivery of messages, the creation and destruction of capsules, and so on.

If set to **DEBUG_NONE**, neither logging, Target Observability nor the Services Library debugger will be available.

PURIFY

Possible values: 0 or 1

Default value: 0

Description: Set this flag to 1 to indicate 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 properly.

RTS_INLINE

Possible values: inline or blank

Default value: blank

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

INLINE_CHAINS

Possible values: inline or <blank>

Default value: <blank>

Description: This variable is used to indicate whether transition code chains are 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 on the average.

Note: This gain is incurred only once for each capsule class. This feature depends on whether your compiler supports inlining.

INLINE_METHODS

Possible values: inline or <blank>

Default value: inline

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.

RTMESSAGE_PAYLOAD_SIZE

Possible values: Any numerical value >= 0

Default value: 36

Description: This defines the size of the payload area in each **RTMessage**, where small objects are copied for better performance. When sending typed data by value, and the data to be sent fits into the payload area, the data will simply be copied into the message. If the data does not fit inside the payload area, memory will be allocated for the data and free'd after the message has been received. If set to 0, there will be no payload area.

SEND_BY_VALUE

Possible values: 0 or 1

Default value: 1

Description: Determines whether the Services Library has the code compiled into it that will allow for sending typed data by value, instead of just sending a pointer. If this is turned off, ensure that your model does not use type descriptors (for example, sending data by value in **RTPort_send()** functions).

OBSERVABLE

Possible values: 0 or 1

Default value: 1

Description: Determines whether the Services Library has the code compiled into it that will allow for 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 Rational 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 1 Definitions for Minimum Services Library Configuration

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

Note: Not all C compilers support inlining.

Note: Disabling the **LOG_MESSAGE** definition will turn off the logging capability of the Services Library. Log messages will no longer appear when the model is running.

Additional definitions that affect important functionality from the C Services Library can also be turned off. Ensure that your model does not rely on any of these services before removing them from the Services Library.

Table 2 Additional Minimum Configuration Definitions

Definition	Default	Minimum Configuration
SEND_BY_VALUE	1	0
MULTIPLE_PRIORITIES	1	0
TIMING_SERVICE	1	0
INTERNAL_LAYER_SERVICE	1	0
MESSAGE_DEFERAL	1	0

See *Changing Pre-processor Macros* on page 112 for a description of how to modify configuration parameters and rebuild a Services Library.

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 106
- *General C Performance Notes* on page 109
- *Additional Design Considerations* on page 109
- *Toolchains* on page 110

Capsule Instances and Capsule Behavior

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.

Unnecessary Sends

Problem:

Sending on replicated ports involves a send on every replication.

Recommendation:

If you have a replicated port with only a few known connections, calling **RTPort_sendAt()** on only the connected instances may be much quicker than the broadcast approach used by **RTPort_send()**.

Sending Typed Data by Value in Messages

Problem:

When typed data is sent by value in a message, the data is deep copied before being sent. For large data structures (such as a large user-defined data type), this operation involves several memory copies and possible allocations and de-allocations if the data does not fit inside of the message's payload area.

Recommendation:

For best performance when sending between capsules within the same memory space, you should consider sending pointers instead of objects. This will introduce more complexity into the design and coding (with respect to memory management and thread issues), 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 can be 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 on the same thread and to a capsule on a different thread, subsequent messages on 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 (I/O) functions (such as **printf** and **scanf**) are quite expensive (about **100 x function** call overhead)

Recommendation:

In performance-critical software, these I/O 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 I/O when the system is idle.

Problem:

Dynamic creation and destruction of objects, particularly of large complex user-defined data types, is expensive (relative to a function call).

Recommendation:

Do not dynamically create objects on the critical data path. Preallocation 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 Rational 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 which 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 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 Rational 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.

Toolchains

As a project moves through its lifecycle, 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, for the 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

9

Contents

This chapter discusses the different ways that are available for configuring and customizing the C Service Library.

This chapter is organized as follows:

Configuration and Customization Explained

The difference between configuring and customizing is that configuring modifies pre-defined parameters built-in to the Services Library to increase speed, or reduce size of your model. Whereas with customization, you change the behavior of the Services Library by adding source files, or by overriding existing operations.

There are several different ways of changing the functionality of the Services Library:

- *Configuration Options* on page 111
- *Customization Options* on page 112

Configuration Options

- *Changing Pre-processor Macros* on page 112

  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 114

  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 or Adding Operations and Classes* on page 115

  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 *Organization of the Services Library Source* on page 93, the *Configuration Naming Convention* on page 94 and *Directory Structure* on page 95.

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/RTPubl/Config.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:

- $RTS_HOME/libset/<*libset*>/RTLibSet.h

  To change for a specific processor and compiler.

- $RTS_HOME/include/RTLibSet.h

  To change for all **libsets** that do not have their own RTLibSet.h file.

- $RTS_HOME/target/<*target*>/RTTarget.h

  To change for all libraries for an operating system.

It is usually preferable to perform a **libset** configuration; that is, to reconfigure only for a specific processor and compiler.

How

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**.

For this example we will assume that our current platform is:

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

To reconfigure the Services Library, build, and update your model to use the new library:

1 Choose a name for the new **libset**. Typically, you can append to the existing **libset** name. In this example let's name the new **libset**:

```
sparc-gnu-2.8.1-minimal
```

2 Create a new directory called $RTS_HOME/libset/sparc-gnu-2.8.1-minimal.

3 Create a new directory called $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 stop the logging messages, add the following:

```
#define LOG_MESSAGE 0
```

6 Build the new Services Library for the platform SUN5T.sparc-gnu-2.8.1-minimal (see *Building the Services Library* on page 117).

7 Update components in model to use the new Services Library (see *Updating a Component to use a Different Services Library* on page 118).

Changing Build Options

Before you Start

Ensure that you understand the *Organization of the Services Library Source* on page 93, the *Configuration Naming Convention* on page 94 and *Directory Structure* on page 95.

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**:

- $RTS_HOME/libset/<*libset*>/libset.mk

  To change for a specific processor and compiler.

- $RTS_HOME/target/<*target*>/target.mk

  To change for all libraries for an operating system.

- $RTS_HOME/config/<*platform*>/config.mk

  To change for a specific configuration (platform).

It is usually preferable to perform a **libset** configuration; that is, to reconfigure only for a specific processor and compiler.

How

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
```

To modify to build a Services Library with debug symbols:

1. Chose a name for the new **libset**, usually you can just append to the existing **libset** name. In this example let's name the new **libset**:

   ```
   sparc-gnu-2.8.1-debug
   ```

2. Create a new directory called $RTS_HOME/libset/sparc-gnu-2.8.1-debug.

3 Create a new directory called $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 it with **-g**.

   LIBSETCCEXTRA should now look like the following:

```
LIBSETCCEXTRA=-g -finline -finline-functions -mv8 \
                 -Wall -Winline -Wwrite-strings
```

6 Build the new Services Library (see *Building the Services Library* on page 117).

7 Update components in model to use the new Services Library (see *Updating a Component to use a Different Services Library* on page 118).

Now you have a Services Library with debug information. You can use your source level debugger to step through the code.

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

Any Services Library modification has to be done on the target level; that is, in the Services Library itself and not in the model.

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), initialize specific libraries or structures that are needed by the Services Library, or initialize signal handlers.

- **RTPeerController_mainloop()**, **RTSoleController_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.

How

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>/<class>

The **target/<*target*>** base directory mirrors the $RTS_HOME/src directory. Thus it must have a directory for each class that has an overridden operation. When the library is built, existing files in directories in $RTS_HOME/src/target/<*target*> are used instead of the corresponding files in **$RTS_HOME/src** . For additional information, see *Organization of the Services Library Source* on page 93.

Tasks

In this example, we will override the **RTCapsule_logMsg()** operation by creating a new target configuration. You can also override for an existing target configuration but you will not be able to easily go back and forth between the original libraries and the customized versions.

For this example we will assume that our current platform is:

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

To override the RTCapsule_logMsg() operation:

1 Choose a name for the new target. Typically, you append to the existing target name. In this example, let’s name the new target as follows:

```
SUN5NEWT
```

2 Create a new directory called $RTS_HOME/target/SUN5NEWT.

3 Create a new directory called $RTS_HOME/config/SUN5NEWT.sparc-gnu-2.8.1

4 Create a new directory called $RTS_HOME/src/target/SUN5NEW/Capsule

5 Copy all the files and sub-directories 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 Copy the file that contains the **logMsg()** operation from the generic source directory to the new target source directory:

   From $RTS_HOME/src/Capsule/logMsg.c `to` $RTS_HOME/src/target/SUN5NEW/Capsule/logMsg.c

7 Edit $RTS_HOME/src/target/SUN5NEW/Capsule/logMsg.c .

8 Build the new Services Library (see *Building the Services Library* on page 117).

9 Update components in model to use the new Services Library (see *Updating a Component to use a Different Services Library* on page 118)

Building the Services Library

Whenever you create a new **libset** or target, you have to build the new configuration of the Services Library. 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>.<libset>***).

Assuming we are using a custom configured OSE Diab C compiler version 4.1a for the Motorola PowerPC platform, the name of our re-configured platform is OSE401T.ppc603-Diab-4.1a-Debug.

To build this Services Library:

Unix:

```
cd $ROSERT_HOME/C/TargetRTS/src
make CONFIG=OSE401T.ppc603-Diab-4.1a-Debug
```

Windows:

```
cd %ROSERT_HOME%\C\TargetRTS\src
nmake CONFIG=OSE401T.ppc603-Diab-4.1a-Debug
```

After rebuilding the Services Library, you must rebuild your Rational Rose RealTime models to link against the new Services Library libraries (see *Updating a Component to use a Different Services Library* on page 118).

Note: If your new Services Library changed the debugging, logging, or Target Observability functionality, visibility into the model may be removed. Debugging the resulting model via the toolset may no longer be possible.

Updating a Component to use a Different Services Library

After you build a new Services Library, ensure that your components reference the new library.

To update a component to use a different Services Library:

1 Open the **Component Specification** dialog.

2 On the **C Compilation** tab, click **Select...** .

3 A list displays the built libraries found in the current Services Library root ($RTS_HOME) directory. If your library build was successful, it will appear in this list.

4 Select your library and click **OK**.

5 Rebuild your model.

Model Properties Reference 10

Contents

This chapter is organized as follows:

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 the C Language Add-in model properties. The model properties provide the language-specific information required to map your model to C. The C properties are grouped into the following property sets:

- *C Model Element Properties* on page 121
- *C TargetRTS Properties* on page 128
- *C Generation Properties* on page 131
- *C Compilation Properties* on page 134
- *C Executable Properties* on page 139
- *C Library Properties* on page 145
- *C External Library Properties* on page 147

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 **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 **Class::C::ConstructFunctionName** property is defined as:

```
${name}_construct
```

If the class name is **NewClass1**, this property will be expanded by the code generator to:

```
NewClass1_construct
```

The following symbols are recognized by the C Code generator and are expanded:

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 certain property fields. Environment variables are not interpreted by the code generator, instead they are passed as is into the generated files. Naturally, environment variables do not make sense in .c 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 as opposed to hard-coded values.

Pathmap symbols are expanded by the code generator into the generated source code. So these can also be used to avoid having to hard code paths into a component.

The following properties are usually defined using environment variables or pathmap symbols:

- *InclusionPaths (Component, C Compilation)* on page 137
- *TargetServicesLibrary (Component, C Compilation)* on page 137
- *InclusionPaths (Component, C External Library)* on page 148
- *Libraries (Component, C External Library)* on page 148
- *UserLibraries (Component, C Executable)* on page 145
- *UserObjectFiles (Component, C Executable)* on page 145

Note: Other properties can be defined with environment variables, but these are the ones you will have to modify the most often.

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**. The following lists contain a summary of the C properties grouped by model the element to which they are associated.

Class

- *GenerateClass (Class, C)* on page 122
- *ClassKind (Class, C)* on page 123
- *ImplementationType (Class, C)* on page 123
- *ConstructFunctionName (Class, C)* on page 123
- *GlobalPrefix(Class, C)* on page 124
- *HeaderPreface (Class, C)* on page 124
- *HeaderEnding (Class, C)* on page 124
- *ImplementationPreface (Class, C)* on page 124
- *ImplementationEnding (Class, C)* on page 125

Attribute

- *AttributeKind (Attribute, C)* on page 125
- InitializerKind (Attribute, C)

Association end

- *InitializerKind (Role, C)* on page 125
- *InitialValue (Role, C)* on page 126

Capsule

- *GenerateConstructFunction (Capsule, C)* on page 126
- *GlobalPrefix (Capsule, C)* on page 126
- *HeaderPreface (Capsule, C)* on page 126
- *HeaderEnding (Capsule, C)* on page 126
- *ImplementationPreface (Capsule, C)* on page 127
- *ImplementationEnding (Capsule, C)* on page 127

Dependency

- *KindInHeader (Uses, C)* on page 127
- *KindInImplementation (Uses, C)* on page 127

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 does not need to be generated.

For example, it is common to create a class within the toolset which is a place-holder 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 be generated even if the class is not.

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: **struct**, **union**, **typedef**, **none**. By default classes are generated as a **struct**.

If **ClassKind = union**, this will generate a C union.

Note: Type descriptors cannot be generated for unions. In addition, default values of all possibilities of the union and constructor operations are not used with this type of class.

If **ClassKind = typedef**, the **ImplementationType (Class, C)** property is used to specify the type.

If **ClassKind** set to none is used for backwards compatibility. If you don't want a class to be generated use the **GenerateClass (Class, C)** property to turn off code generation.

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]`.

ConstructFunctionName (Class, C)

Use this property to configure the name of the constructor function for the generated class. The default name for the **construct** function is **${name}_construct**, where **${name}** is the name of the class. For example, if your class is called **ConfigData** then the generated function would be called **ConfigData_construct**.

If this property is blank, a constructor function is not generated.

The default constructor function is generated to initialize each attribute defined in the class. Each attribute is initialized with its initial value, or by calling the **construct** function for the attribute. This is configurable for each attribute by using the **InitializerKind (Attribute, C)** property.

GlobalPrefix(Class, C)

This is a global prefix by which you wish to prefix all generated class operations. The default is empty.

Adding a global prefix will minimize conflicts with operations defined on other classes and make your detail code more intuitive. A common value for this property is **${name}_**. This will prefix each operation with the name of the class on which it is defined.

HeaderPreface (Class, C)

Specifies the text that will appear 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 Generated Code* in the Toolset Guide.

HeaderEnding (Class, C)

Specifies the text that will appear 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 Generated Code* in the Toolset Guide.

ImplementationPreface (Class, C)

Specifies the text that will appear 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 Generated Code* in the Toolset Guide.

ImplementationEnding (Class, C)

Specifies the text that will appear 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 Generated Code* in the Toolset Guide.

AttributeKind (Attribute, C)

Specifies whether the attribute is generated as a field of the generated **struct**, or as a **#define** defined within the file generated for the class. Options are **normal** and **constant**.

If set to **constant**, a **#define** will be generated using the name of the attribute as the name of the macro and the initial value as the value.

If an attribute is set to **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. Also ensure that the dependency **KindInHeader (Uses, C)** property is set to **inclusion**.

InitializerKind (Attribute, C)

Use this property to configure how the attribute is initialized. Possible values are **assignment** and **call construct function**. When the owner class generates and uses a construct function, then the construct will try and initialize its attributes however it can.

If **InitializerKind = assignment** then in the owners construct the attribute will be initialized with the attributes initial value (**Attribute::Detail Page::Initial value**).

If **InitializerKind = call contruct function** then the classes' **construct** function will call the attributes' **construct** function.

InitializerKind (Role, C)

Use this property to configure how the generated attribute for the association end is initialized. The values and usage of this property is described in **InitializerKind (Attribute, C)**.

InitialValue (Role, C)

If the association end (Role) **InitializerKind (Role, C)** property is set to **assignment** then this value is used to initialize the generated attribute.

GenerateConstructFunction (Capsule, C)

Specifies if a **construct** function is to be generated for the capsule to initialize all of its attributes with either their initial values or by calling the attributes' `construct` functions.

GlobalPrefix (Capsule, C)

This text field represents the string that all operations will be prefixed with. By default this is blank. A good value for this is **${name}_**, as this will prefix all functions that serve as operations with the name of the capsule followed by an underscore then the operation name. This is the convention used in the C Services Library.

HeaderPreface (Capsule, C)

Specifies a block of C code to include in the generated code of the capsule class header, after any generated **#include**'s and before the generated capsule declarations. Code can include: comments, **#define**'s, **#include**'s, declarations, and so on.

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 Generated 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 Generated 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, after any generated **#include**'s and before the generated capsule definitions. Code can include: comments, **#define**'s, **#include**'s, declarations, and so on.

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 Generated 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 Generated 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

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. The following lists contain a summary of the C TargetRTS properties grouped by model element to which they are associated.

Class

- *GenerateDescriptor (Class, C TargetRTS)* on page 128
- *Version (Class, C TargetRTS)* on page 128
- *InitFunctionBody (Class, C TargetRTS)* on page 129
- *CopyFunctionBody (Class, C TargetRTS)* on page 129
- *DestroyFunctionBody (Class, C TargetRTS)* on page 129
- DecodeFunctionBody (Class, C TargetRTS)
- *EncodeFunctionBody (Class, C TargetRTS)* on page 130

Attribute

- *GenerateDescriptor (Attribute, C TargetRTS)* on page 130
- *TypeDescriptor (Attribute, C TargetRTS)* on page 130
- *NumElementsFunctionBody (Attribute, C TargetRTS)* on page 130

AssociationEnd

- *GenerateDescriptor (Role, C TargetRTS)* on page 131
- *TypeDescriptor (Role, C TargetRTS)* on page 131
- *NumElementsFunctionBody (Role, C TargetRTS)* on page 131

GenerateDescriptor (Class, C TargetRTS)

If selected, the C code generator will create a type descriptor 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 not selected, the data type cannot be sent by value in messages and will not 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 **RTstruct_init**.

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 Generated Code* in the 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 types copy 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 *Using Code Sync to Change Generated 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.

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 Generated 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 **RTDecoding** object from which the stream of bytes can be retrieved and then used to create a new object of this type. The decode function **target** argument is an already allocated object that should be initialized with the new data.

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 Generated 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 **RTEncoding** object from which the stream of bytes should be passed from the object that is being encoded. The encode function **source** argument is an already allocated object that is to be encoded.

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 Generated 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 will not be able to encode/decode the attribute.

TypeDescriptor (Attribute, C TargetRTS)

Specifies an explicit descriptor for the attribute. Normally, the code generator will determine 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 deep copied and observable in the execution monitors.

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 Generated 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 will not be able to encode/decode the attribute.

TypeDescriptor (Role, C TargetRTS)

Specifies an explicit descriptor for the attribute. Normally the code generator will determine 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 additional information, see *NumElementsFunctionBody (Attribute, C TargetRTS)* on page 130.

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 Generated Code* in the Toolset Guide.

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 132
- *CodeGenDirName (Component, C Generation)* on page 132
- *ComponentUnitName (Component, C Generation)* on page 132
- *CommonPreface (Component, C Generation)* on page 132
- *CodeGenMakeType (Component, C Generation)* on page 133
- *CodeGenMakeCommand (Component, C Generation)* on page 133
- *CodeGenMakeArguments (Component, C Generation)* on page 134
- *CodeGenMakeInsert (Component, C Generation)* on page 134
- *CodeSyncEnabled (Component, C Generation)* on page 134

OutputDirectory (Component, C Generation)

The output path can be changed to allow you to set the directory into which the generated files 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 C code for the component elements. This directory will be generated as a subdirectory of the output directory identified in the **OutputDirectory (Component, C Generation)**.

ComponentUnitName (Component, C Generation)

Specifies the name of the source files generated for the component itself.

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 **InclusionPaths (Component, C Compilation)**. Inclusions items can be added and deleted as required.

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 having normal system include files specified before user **include**s. 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 **Class Specification** dialog. In either case, the directory search list comes 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.

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 classes 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 external and type declarations; otherwise, multiple definitions are reported at link time.

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 Generated Code* in the Toolset Guide.

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. For example, if using **nmake** on Windows, then **MS_nmake** must be selected as the **make** type.

Leaving the entry as **<default>** allows 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 Windows) is substituted for **<default>**. If you require another **make** type, then explicitly specify the **make** type in this field.

CodeGenMakeCommand (Component, C Generation)

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 allows 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) is substituted. If you require a different **make** utility, then explicitly specify the **make** type in this field.

CodeGenMakeArguments (Component, C Generation)

Any flags supported to be passed to the **make** utility.

CodeGenMakeInsert (Component, C Generation)

The overrides file 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)

The flag which is used to enable or disable Code Sync for a component, from the component's **Generation** tab.

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 134
- *CompilationMakeCommand (Component, C Compilation)* on page 135
- *CompilationMakeArguments (Component, C Compilation)* on page 135
- *CompilationMakeInsert (Component, C Compilation)* on page 135
- *CompileCommand (Component, C Compilation)* on page 136
- *CompileArguments (Component, C Compilation)* on page 136
- InclusionPaths (Component, C Compilation)
- *TargetServicesLibrary (Component, C Compilation)* on page 137
- *TargetConfiguration (Component, C Compilation)* on page 138

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 using **nmake** on Windows, then **MS_nmake** must be selected as the **make** type.

Leaving the entry as **<default>** allows 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 Windows) is substituted for **<default>**. If you require another **make** type, 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 **make** name must be the exact name of the **make** command. The **make** name must be the exact name of the **make** command. By default the default **make** command is **$defaultMakeCommand** which allows 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. If you require a different **make** utility, explicitly specify the **make** type in this field.

CompilationMakeArguments (Component, C Compilation)

Any flags supported to be passed to the make utility.

CompilationMakeInsert (Component, C Compilation)

The overrides file is a makefile fragment which is included in the compilation makefile that allows for the addition of user-defined dependencies, compile, and link options.

Although you can add any valid **make** information to the overrides **makefile** the following example is how the overrides **makefile** while is most commonly used.

To quickly add object or library files to the link line of a component, add the object file or library to the link flags field.

Note: When quickly adding object or library files, no dependency is automatically generated for the object or library files. This means that if the object or library files change, it will not automatically cause your component to become out-of-date, and require re-compiling.

Use the overrides **makefile** to add these dependencies by using the **USER_DEPS** and **USER_OBJS** macros within your **makefile**.

CompileCommand (Component, C Compilation)

The **CompileCommand** property replaces the pre-configured compiler shell command defined in libset.mk. You would normally leave this entry (for example, usually set to **$(CC)**) and use the default compiler specified in the **libset makefile**.

When building your model, a compiler compiles the generated code and a linker will link the executable. By default, when you specify the Service Library, you identify the **make** files 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 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 (for example, gnu vs. Greenhills), or a different release of a compiler, change the Service Library specification instead. The **make** files used will pass flags understood by the compiler/linker. As well, the pre-compiled Service Library 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.

Note: Enclose directory names with embedded spaces in double quotes (").

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 required for searching all the directories.

We recommend that 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: **Pathmap** variables, those defined within the toolset, cannot be used to specify indirect inclusion paths because they are not substituted into the generated **makefiles**. ONLY use environment/shell variables because they will be visible to the **makefiles** (which are built outside the toolset) when the build occurs.

TargetServicesLibrary (Component, C Compilation)

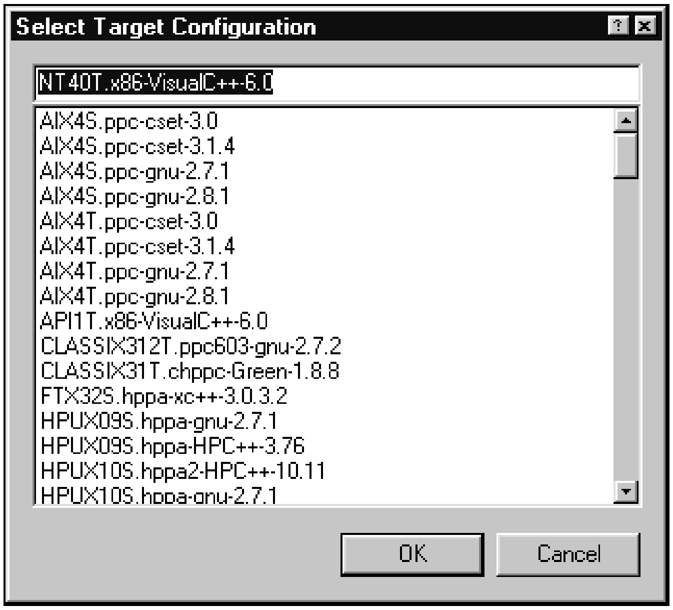
The text field specifies the path to the root directory for the specific Services Library desired. This can be any valid directory name. This name must be specified as a full path to the root directory of the Services Library root.

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 Rational Rose RealTime home directory $ROSERT_HOME/C/TargetRTS.

TargetConfiguration (Component, C Compilation)

This property is used to uniquely identify the configuration of the Services Library used to compile and link a component. By clicking the **Select...** button on the **Component Specification** dialog, the following dialog appears:



```
NT40T.x86-VisualC++-6.0
AIX4S.ppc-cset-3.0
AIX4S.ppc-cset-3.1.4
AIX4S.ppc-gnu-2.7.1
AIX4S.ppc-gnu-2.8.1
AIX4T.ppc-cset-3.0
AIX4T.ppc-cset-3.1.4
AIX4T.ppc-gnu-2.7.1
AIX4T.ppc-gnu-2.8.1
API1T.x86-VisualC++-6.0
CLASSIX312T.ppc603-gnu-2.7.2
CLASSIX31T.chppc-Green-1.8.8
FTX32S.hppa-xc++-3.0.3.2
HPUX09S.hppa-gnu-2.7.1
HPUX09S.hppa-HPC++-3.76
HPUX10S.hppa2-HPC++-10.11
HPUX10S.hppa-gnu-2.7.1
```

Select a Target Configuration from the list. The list was created from the entries found in $ROSERT_HOME/$RTS_HOME/lib. 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:

```
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 controls the aspects of generating an executable from a C model. C Executable properties apply only to components which of type C Executable. The following lists contain a summary of the C Executable properties grouped by the model element to which they are associated.

Component

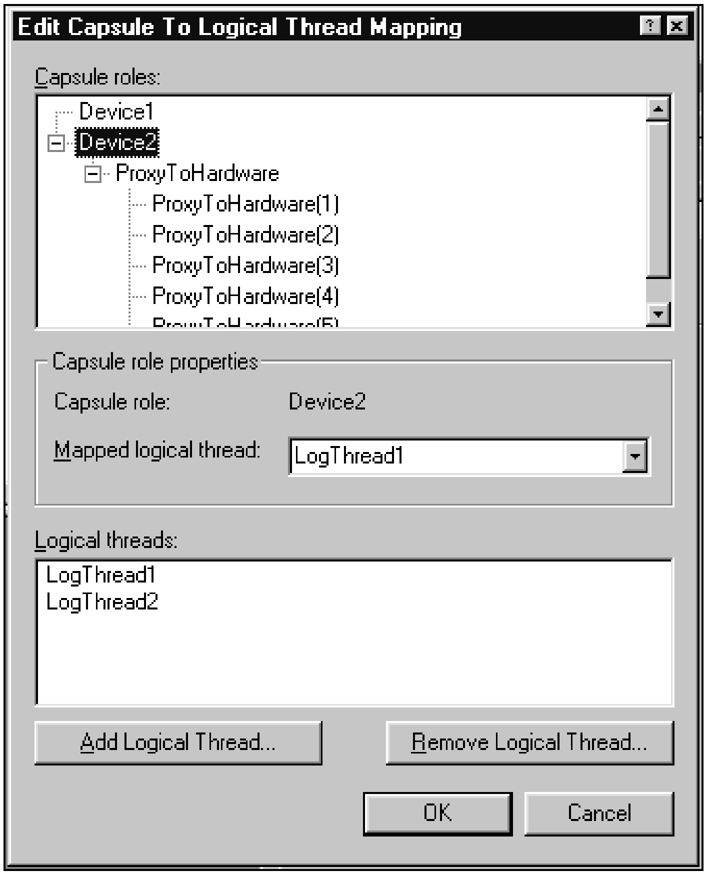
Capsule

Capsule To Logical Thread Mapping (Capsule, C Executable)

In C, logical threads are defined in the **Logical View** and physical threads are defined in the **Component View**. Individual capsule instances can be mapped to any logical thread and logical threads can be mapped to any physical thread.

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.

The definition and mapping of capsule roles to logical threads is done on the top level capsule of your C model. This property is edited with an advanced property editor which provides a graphical interface. To open the dialog, click the **Edit...** button to the right of the property name. The following dialog will appear:



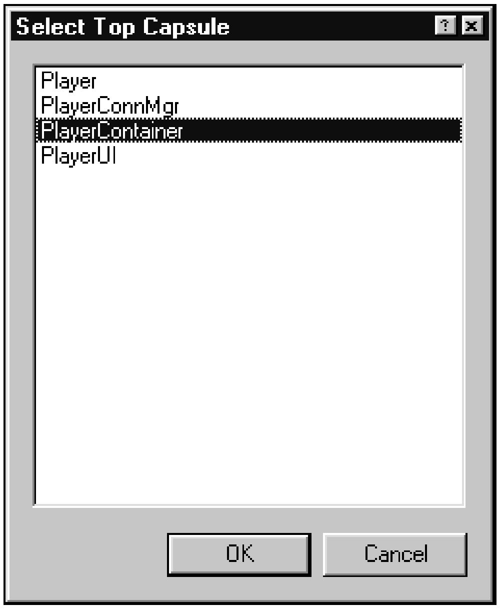
This dialog automatically shows all the contained capsule roles of the current capsule. If the **Mapped logical thread** field is set to **<default>**, the capsule role will run on the same logical thread as its parent. Thus, assigning **Device2** to **LogThread2**, shown in the example dialog above, will cause, unless overridden in a contained capsule, all contained capsules to be assigned to the same thread as its containing capsule.

You can define capsule to logical thread mappings on any capsule, however when mapping logical threads to physical threads on a component, the component will only look at the logical threads defined on the top level capsule.

TopCapsule (Component, C Executable)

Specifies the top capsule to compile 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 the **Select...** button, a dialog shows a list of all capsules referenced by the component. Select the desired top level capsule, and click **OK**.

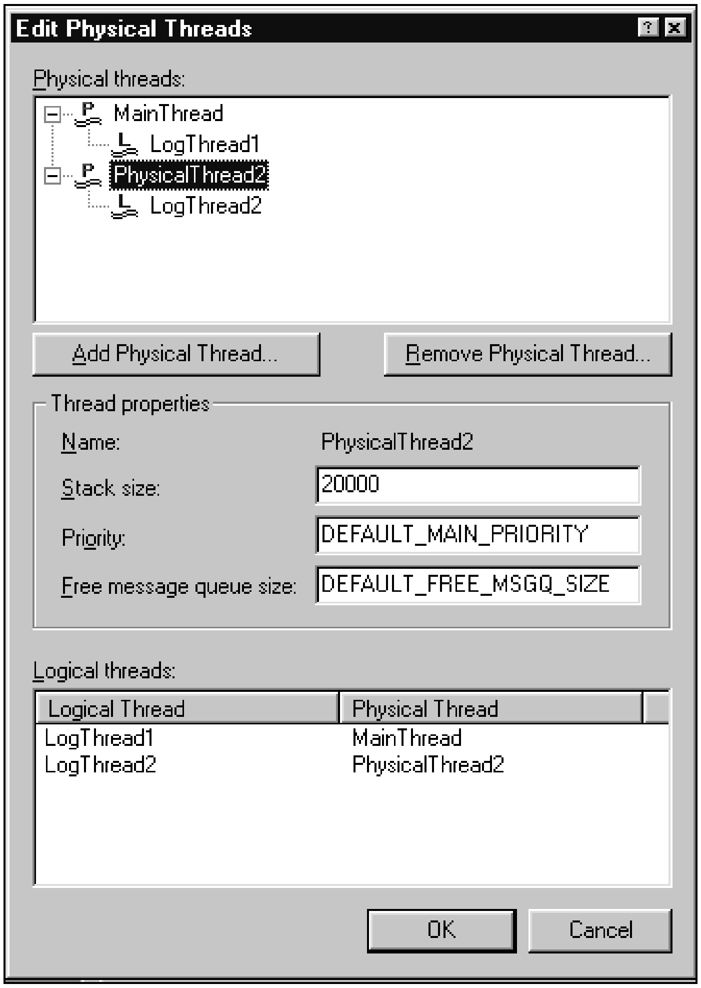


Player
PlayerConnMgr
PlayerContainer
PlayerUI

PhysicalThreads (Component, C Executable)

For some configurations (platforms), the Services Library supports multiple threads. By default, all logical threads are assigned to a pre-defined thread called **MainThread**. The top-level capsule is always placed on the **MainThread**. The C Services Library is responsible for allocating all capsule instance to the appropriate threads as defined by the thread's configuration.

This property is modified using a property editor which provides a graphical interface. To open the dialog, click the **Edit...** button to the right of the property name. The following dialog will appear:



Name: PhysicalThread2

Stack size: 20000

Priority: DEFAULT_MAIN_PRIORITY

Free message queue size: DEFAULT_FREE_MSGQ_SIZE

Logical threads:

Logical Thread	Physical Thread
LogThread1	MainThread
LogThread2	PhysicalThread2

The dialog automatically populates the list with logical threads defined on the capsule assigned as the top-level capsule for this component.

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.

By default, every configuration contains the following physical thread:

- **MainThread** - here 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.

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.
Free message queue	The number of messages allocated for inter-capsule messaging on this thread.

Note: Although stack size is configurable, for some target operating systems, this stack size is effective at the time the main thread is created 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. For 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.

Using Physical Thread Trade-offs

Before choosing to create another physical thread in your model, consider the following costs:

- memory overhead for the thread stack space, and control objects created by the C Services Library required for each physical thread.
- processing overhead; inter-thread message sending is generally an order of magnitude slower that intra-thread messaging.

ExecutableName (Component, C Executable)

You can specify the name, or a name with an absolute path, of the executable 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 configurations (platforms) do not allow the passing of command line arguments to an executable at load time (namely, on some real-time operating systems). For this case, the default arguments provides a mechanism for getting execution arguments into the executable. You can use **RTMain_argv()** to retrieve any passed command line argument within your model. Enter a comma-separated list of quoted arguments into this field.For example:

```
"134.434.344.4","barneyht","delay=98"
```

The default arguments field is only 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 replaces the pre-configured linker shell command defined in the file 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 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
```

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 files that are 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. To re-use libraries that have already been built within a Rational Rose RealTime model, use an External Library component.

Component

- *LibraryName (Component, C Library)* on page 146
- *BuildLibraryCommand (Component, C Library)* on page 146
- *BuildLibraryArguments (Component, C Library)* on page 146

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 configuration (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 for more information about the Services Library directory, see **TargetServicesLibrary (Component, C Compilation)**.

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 archive utility. They are passed as is to the archiver.

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 within a C Executable. C External Library properties apply only to components which of type C External Library.

Component

- *GenerateClassInclusions (Component, C External Library)* on page 147
- *CodeGenDirName (Component, C External Library)* on page 147
- *InclusionPaths (Component, C External Library)* on page 148
- *Libraries (Component, C External Library)* on page 148

GenerateClassInclusions (Component, C External Library)

Ensure that this property is not set if you do not 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 else in the model, or in an external file. Typically, this property should remain set.

CodeGenDirName (Component, C External Library)

This property is only required if **GenerateClassInclusions (Component, C External Library)** is set 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 to which this external library references.

Having this prefix ensures that all inclusions generated for model elements that reference elements in the external library are prefixed with this value. This will reduce the chance of having inclusion conflicts. For example if this property is set to **rtg**, then inclusions are 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. See Environment variables and pathmap symbols.

If **ComputeDependencies** is set to Yes, then the **make** depends utility is 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: We recommend that you use environment variables instead of hard-coded paths. Alternatively, you can use the pre-defined **code gen** variables, such as **$@**. Environment variables are recongnized by the **make** utility.

Libraries (Component, C External Library)

Specifies the location and names of the libraries that this external component represents. This libraries listed in this field are 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
```

It is recommended that you use **pathmap** symbols or environment variables for *pathnames* in this property. See *Environment Variables and Pathmap Symbols* on page 121.

If **GenerateDependencies** 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**).

Services Library API Reference

11

Contents

This chapter is organized as follows:

Overview

The C Services Library Class Reference is a reference to the structures and abstract data types that you use within the detailed code of a capsule to access the services provided by the C Services Library.

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 users application.

For each of the classes listed in this reference, only the operations and attributes explicitly detailed in this chapter represent the supported interface to the C Services Library.

Minimally Configured Services Library

If you have reconfigured the Services Library which has resulted in the removal of functionality from the library, **some functions of the interfaces defined in this API may no longer be available in the minimally configured Services Library**. For details on configuring the Services Library refer to the C Target Guide.In addition, the functions described in this section refer to the pre-processor macros on which they depend.

RTCapsule

Every capsule - when generated as C code - is a subclass of **RTCapsule**; thus, the first field of every generated capsule's instance data is a **RTCapsule** named **std**. In any user code where capsule instance data exists through the **this** pointer, one can access a pointer to the **RTCapsule** information in the following ways:

```
(RTCapsule *)this /* or */

&this->std
```

This common base class for all capsules defines attributes and operations which allows 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 **RTCapsule**. Under the Rational Rose RealTime paradigm, you should only be calling the operations of **RTCapsule** or using attributes that are defined below:

Note: The attributes and operations on **RTActor** are private. One capsule may not manipulate another capsule's attributes.

Attributes

msg and RTCapsule_getMsg	Contains a pointer to the current message which triggered a transition. Neither it, nor the object is points to, should be modified.
rts and RTCapsule_context	Contains a pointer to the controller for the physical thread on which a capsule instance is executing.

Operations

RTCapsule_getCurrentStateString	Gets the current state name containing the executing segment.
RTCapsule_getIndex	Gets the replication index of this capsule instance in the home capsule role.
RTCapsule_getName	Gets the capsule role name in which this capsule instance is running.
RTCapsule_getTypeName	Gets the capsule class name of this capsule instance.

msg and RTCapsule_getMsg

const RTMessage * msg;

const RTMessage * RTCapsule_getMsg(const RTCapsule *);

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.

```
int theData = *(int *)
   RTMessage_getData( (RTCapsule *) this)->msg);

RTSignal theSignal =
   RTMessage_getSignal( RTCapsule_getMsg( &this->std ) );
```

Explanation of the **RTMessage** primitives used in the above example can be found in **RTMessage** section.

rts and RTCapsule_context

RTController * RTCapsule_context(const RTCapsule *);

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. You may find it useful for printing error information, as in the example below.

Examples

```
int result =
   RTPort_send( port,RTPort_createOutSignal( port, hey ),
   RTPriority_General, (void *)0, (RTObject_class *)0 );


if( ! result )
{
   RTController * context = RTCapsule_context( this );
   log.show("Error on physical thread: ");
   RTLog_show( RTController_name( context ) );
   RTController_perror( context, "send");
}
```

RTCapsule_getIndex

int RTCapsule_getIndex(const RTCapsule *);

Return Value

The replication index of this capsule instance in its "home" role (where it was incarnated). The replication value is zero-based (0).

RTCapsule_getName

const char * RTCapsule_getName(const RTCapsule *);

Return Value

The name of the capsule role in which this capsule instance is running (where it was incarnated).

Note: Unavailable in certain Services Library configurations. For additional information, see *RTS_NAMES in* the *C Target Guide*.

RTCapsule_getTypeName

const char * RTCapsule_getTypeName(const RTCapsule *);

Return Value

Returns the class name of this capsule instance.

Note: Unavailable in certain Services Library configurations. For additional information, see *RTS_NAMES in* the *C Target Guide*.

RTCapsule_getCurrentStateString

const char * RTCapsule_getCurrentStateString(const RTCapsule *) ;

Return Value

The name of the current state containing the executing segment.

Note: Unavailable in certain Services Library configurations. For additional information, see *RTS_NAMES* in the *C Target Guide*.

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.

Also, with regards to the timing service, controllers serve as the interface between user-designed timing actors and the Services Library.

Note: From within a capsule instance, you can retrieve a pointer to its controller by calling the **RTCapsule_context** operation.

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_registerTimer	Register a RoseRT defined timer capsule as the timer service for this controller.
RTController_overrideSyncMethods	Override this controller's interface to going to sleep and waking up, in order to implement a RoseRT defined timing service.

RTController_getError

RTController_PrimitiveError RTController_getError(const RTController *);

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. Call it 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 * RTController_strError(const RTController *);

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 RTController_perror(const RTController *, const char *);

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

```
int result =
   RTPort_sendAt( &this->aPort, 0,
      RTPort_createOutSignal( aPort, ack ),
      RTPriority_General,
      (const void *)0,
      (const RTObject_class *)0 );
if( ! result )
      RTController_perror(
      RTCapsule_context( (RTCapsule *)this ),
      "Error sending ack");
```

Output

```
Error sending ack: Port not connected.
```

RTController_name

const char * RTController_name(const RTController *);

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.

Note: Unavailable in certain Services Library configurations. For additional information, see *RTS_NAMES in* the *C Target Guide*.

RTController_registerTimer

int RTController_registerTimer(RTController * *this*, **void *** *timer*, **RTController__informIn** *informInFn*, **RTController__cancel** *cancelFn*, **RTController__valid** *validFn* **);**

Remarks

This function allows for the integration of a RoseRT designed timer capsule with the Services Library. To implement the functionality of the timing service, a timer capsule must provide **informIn**, **cancelTimer** and **isTimerValid** primitives. These are fed in through this primitive to register these particular timing services on the physical thread that this **RTController** interfaces to.

Return Value

This function returns positive logic values indicating its level of success.

Example

```
/* register this capsule as the timing service
for this particular thread */
if( ! RTController_registerTimer(
   RTCapsule_context( this ),
   (void *)this,
   MyCapsule_informIn,
   MyCapsule_cancel,
   MyCapsule_valid ) )
      RTController_perror("Thread already has timing service");
```

RTController_overrideSyncMethods

int RTController_overrideSyncMethods(RTController * this, RTController__sleep sleepFn, RTController__wakeup wakeupFn);

Remarks

This function services as the other half of the timer capsule registration puzzle. In order to integrate timing functionality into the **RTController**, it is important that the timer capsule provides a mechanism for the controller's synchronization methods to be overridden. Otherwise, if a thread goes to sleep when there are no messages to process or **timeouts** to work upon, how will it wake up when it is given a timer request to handle? See the **CTimer** class for an example of how to implement sync methods.

Return Value

This function returns positive logic values to indicate success.

Example

```
/* override my thread's synch methods */
if( !
   RTController_overrideSyncMethods(
   RTCapsule_context( this ),
   MyCapsule_nap, MyCapsule_awaken ) )
      RTController_perror( RTCapsule_context( this ), "cannot override
      sync methods" );
```

RTController_abort

void RTController_abort(RTController *);

Remarks

Calling this operation on any controller will terminate the current process. The top-level capsule instance is destroyed, which in turn destroys all capsule instances in the system, messages that have not been processed are deleted, all threads are destroyed, and the process quits.

Examples

```
RTController_abort( RTCapsule_context( &this->std ) );
```

RTLog

The Log service is a stream of ASCII text in which system or application events can be recorded.

Note: Currently all log service output is directed to **stdout**.

Operations

Log show primitives	Writes an ASCII string to the log with no leading or trailing carriage returns

Log show primitives

void RTLog_show_string(const char * *data***);**

void RTLog_show_char (char *data***);**

void RTLog_show_double(double *data***);**

void RTLog_show_float (float *data***);**

void RTLog_show_int (int *data***);**

void RTLog_show_uint (unsigned int *data***);**

void RTLog_show_long (long *data***);**

void RTLog_show_ulong (RTulong *data***);**

void RTLog_show_short (short *data***);**

void RTLog_show_ushort(RTushort *data* **);**

void RTLog_show_ptr (const void ***data* **);

void RTLog_show_data (const void * *data***, const RTObject_class *** *type***);**

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. For a user-defined type to be displayable, it must have type information defined with a function to encode the object. The log will simply call this encode function.

RTLog_show_string() prints a string and is always available, even if **OBJECT_ENCODE** Services Library configuration parameter is turned off.

RTLog_show_data() prints the value of the user-defined data type and is available only if **OBJECT_ENCODE** and **STDIO_ENABLED** are turned on.

Examples

```
/* Print as an ASCII string the contents of a class */
RTLog_show_data( &SubscriberData, &RTType_SubscriberData );

/* Print a string */
RTLog_show_string( "Timer has expired" );

/* Print an int */
RTLog_show_int( 19 );
```

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 priority, 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 **RTCapsule** 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_getSignal	Returns the signal of the message.
RTMessage_copyData	Copies the data (by value) into a local buffer.
RTMessage_getPort	Retrieves a pointer to the port which received the message.
RTMessage_getPortIndex	Finds the index of the port on which the message was received (0 and 1 based).

RTMessage_getPriority

RTPriority RTMessage_getPriority(const RTMessage *);

Return Value

Returns the value of the priority of the message.

RTMessage_getSignal

RTSignal RTMessage_getSignal(const RTMessage *);

Return Value

Returns the value of the signal of the message.

RTMessage_copyData

int RTMessage_copyData(const RTMessage *, void * ***buffer*****, int** ***size*****);**

Return Value

Returns the number of bytes copied into buffer. If the size of the buffer (specified by the size parameter) is not large enough, **RTMessage_copyData** shall return 0.

Note: Unavailable in certain Services Library configurations. For additional information, see *SEND_BY_VALUE* in the *C Target Guide*.

Example

```
SomeDataClass buffer;

/* copy RTMessage data into buffer */
int copied = RTMessage_copyData(
   RTCapsule_getMessage(),
   &buffer,
   sizeof( SomeDataClass ) );
```

RTMessage_getSignalName

const char * RTMessage_getSignalName(const RTMessage *);

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.

Note: Unavailable in certain Services Library configurations. For additional information, see *RTS_NAMES* in the *C Target Guide*.

Example

```
log_show_string( RTMessage_getSignalName( RTCapsule_getMsg(this));
```

RTMessage_getData

const void * RTMessage_getData(const RTMessage *);

Return Value

Returns the pointer to the data that was sent along with a message.

```
aDataType dt =
   *(aDataType *)RTMessage_getData( RTCapsule_getMsg( this ) );
```

RTMessage_getType

const RTObject_class * RTMessage_getType(const RTMessage *);

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 only data pointer sent.

Note: Unavailable in certain Services Library configurations. For additional information, see *SEND_BY_VALUE* in the *C Target Guide*.

RTMessage_getPortIndex

int RTMessage_getPortIndex(const RTMessage *);

Return Value

Returns the index of the port on which the message was received. The **RTMessage_getPortIndex** function returns a zero-based index (index values begin at 0).

Example

Use to send a message to a particular port instance, as follows:

```
const RTMessage * msg = RTCapsule_getMsg( this );
/* reply to message */
RTPort_sendAt(
   port,
   RTMessage_getPortIndex( msg ),
   RTPort_createOutSignal( port, reply ),
   RTPriority_High,
   &someData,
   &RTType_typeOfSomeData );
```

RTMessage_getPort

RTPort * RTMessage_getPort(const RTMessage *);

Return Value

Returns a pointer to the port instance on which this message was received, or (**RTPort ***) 0 if called in the initial transition.

RTMessage_defer

int RTMessage_defer(const RTMessage *);

Return Value

Returns true (1) if the message was successfully deferred, and false (0) otherwise. An error is returned if you defer an invoked message, or a message which has already been deferred.

Remarks

Deferred messages can be recalled using the recall functions defined on **RTPort**.

Example

In the transition where a message is to be deferred you would defer the message as follows:

```
RTMessage_defer( this->std.msg );
```

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.

Note: The toolset will generate these descriptors for most classes which are defined using basic types (see below for the list). If classes contain more complicated structures you can write your own type descriptor functions from within the toolset. See C Target RTS properties for more information on this subject.

```
/* 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.

RTPeerController

RTPeerController is a refinement of the **RTController** class which represents the interface to a physical thread in the multi-threaded run-time system. To implement a timer capsule that plugs into the C Services Library, you may need to use the following primitives.

Operations

RTPeerController_timedWait	Allows for a means of doing a timed wait. This is useful for implementing your own timing service.
RTPeerController_waitForEvents	Put the **RTPeerController** in the phase of waiting for events to happen (either timer **timeout**, message arrival or timer request).

RTPeerController_timedWait

int RTPeerController_timedWait(RTPeerController *, RTTimespec *);

This function allows for a thread to block on a timed wait. The thread will be awoken by either an external event like a message delivered to a controller (return 0) or by the time expiry (return 1).

Example

```
int weTimedOut =
RTPeerController_timedWait( RTCapsule_context(), &time );
```

RTPeerController_waitForEvents

void RTPeerController_waitForEvents(RTPeerController *);

This is the means by which an **RTController** in the multi-threaded Services Library goes to sleep. Using this method to go to sleep ensures that if the thread receives a message, it shall be woken up in order to deliver it.

RTPort

For each port specified on a capsule, an **RTPort** is generated within the instance data. **RTPort** serves as the interface to most of the primitives of the communications, layer and timing services of the Service Library. A **RTPort** instance contains a list of all the individual instances of that port that may be bound (at runtime or through connectors) to one or more **RTPort** instances.

Operations

RTPort_send	Broadcast a message across the entire port using the communications and/or internal layer service.
RTPort_sendAt	Send a message solely on this index using the communications and/or internal layer service.
RTPort_enqueue	**Enqueue** a message onto a port without having to be bound to it. This is very useful to implement a timing capsule in order to deliver **timeout** messages.
RTPort_getCardinality	Returns the cardinality of the port.
RTPort_isBound	Returns the bound status of the port instance specified by index.
RTPort_getRegisteredName	Returns the name of the registration that the unwired port has registered as.
RTPort_isRegistered	Determines the registration status of the port instance specified by the index.
RTPort_registerAs	Registers an unwired port by name.
RTPort_deregister	Deregisters an unwired port.
RTPort_recall	Recall a message that came in on this port and that was deferred.
RTPort_recallAt	Recall a message that came in on this port instance and was deferred.
RTPort_recallAll	Recall all messages that came in on this port that were deferred.
RTPort_recallAllAt	Recall all messages that came in on this port instance that were deferred.
RTPort_purge	Purge all messages from the defer queue that came in on this port.
RTPort_purgeAt	Purge all messages from the defer queue that came in on this port instance specified by index.
RTPort_informIn	Request a one-shot timer to expire in a specified amount of time.
RTPort_cancelTimer	Cancel a timer that was created on this timer port.

RTPort_isTimerValid	Determine if a timer (that was created on this timer port) is valid.
`RTPort_createOutSignal`	Create an out signal local to the protocol.
`RTPort_createInSignal`	Create an in signal local to the protocol.

RTPort_getCardinality

int RTPort_getCardinality(const RTPort *);

Return Value

Returns the cardinality 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.

RTPort_purge

int RTPort_purge(const RTPort *);

Return Value

Returns the number of deleted messages from the defer queue.

Remarks

To delete deferred messages for one port instance use **RTPort_purgeAt**.

Note: Unavailable in certain Services Library configurations. For additional information, see *MESSAGE_DEFERRAL in* the *C Target Guide*.

RTPort_purgeAt

int RTPort_purgeAt(const RTPort *, int *index* **);**

Parameters

index

The port index for which deferred messages should be purged.

Return Value

Returns the number of deleted messages from the port instance defer queue.

Remarks

To delete deferred messages for all port instances use **RTPort_purge**.

Note: Unavailable in certain Services Library configurations. For additional information, see *MESSAGE_DEFERRAL in* the *C Target Guide*.

RTPort_recall

int RTPort_recall(const RTPort * *);*

Return Value

Returns the number of recalled messages (either 0 or 1).

Remarks

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 from the front of the defer queue.

There is no time limit on deferral, therefore applications must take precautions against forgetting messages on defer queues.

This operation recalls the first deferred message on any port instance. To recall the first message on one port instance of a replicated port, use the **RTPort_recallAt** operation.

Note: Unavailable in certain Services Library configurations. For additional information, see *MESSAGE_DEFERRAL in* the *C Target Guide*.

RTPort_recallAt

int RTPort_recallAt(const RTPort *, int *index* **);**

Return Value

Returns the number of recalled messages (either 0 or 1).

Parameters

index

Port instance index for which to recall a deferred message.

Note: Unavailable in certain Services Library configurations. For additional information, see *MESSAGE_DEFERRAL in* the *C Target Guide*.

RTPort_recallAll

int RTPort_recallAll(const RTPort *);

Return Value

Returns the number of recalled messages.

Remarks

This operation recalls all deferred message on any port instance. Messages are recalled from the front of the defer queue.

Note: Unavailable in certain Services Library configurations. For additional information, see *MESSAGE_DEFERRAL in* the *C Target Guide*.

RTPort_recallAllAt

int RTPort_recallAllAt(const RTPort *, int *index***);**

Return Value

Returns the number of recalled messages from a given port instance.

Remarks

Calling **RTPort_recallAllAt** on a port will get all the deferred message from the port instance indicated by index.

Note: Unavailable in certain Services Library configurations. For additional information, see *MESSAGE_DEFERRAL in* the *C Target Guide*.

RTPort_send

int RTPort_send (const RTPort *, RTSignal, RTPriority, void *, const RTObject_class *);

Remarks

Construct a message (of particular signal, data, priority, and type) and send it across the specified port. If the port has cardinality greater than 1, then the message is broadcast across each of the instances.

Returns

RTPort_send returns the number of successful messages sent. If all messages are sent properly, then this value should be equivalent to **RTPort_getCardinality()**.

RTPort_sendAt

int RTPort_sendAt(const RTPort *,int, RTSignal, RTPriority, void *, const RTObject_class *);

Remarks

Construct a message (of particular signal, data, priority, and type) and send it across the specified port instance.

Returns

Positive logic values indicate success.

RTPort_enqueue

RTMessage * RTPort_enqueue(const RTPort *, int *index*, **RTSignal, RTPriority, const void *** *data*, **RTCapsule *** *fromCapsule*, **const RTObject_class *** *type* **);**

Remarks

Construct a message (of particular signal, data, priority and type) and enqueue it upon the port instance specified by the port and index. The **RTCapsule** field is for the capsule doing the enqueuing, so that the Services Library knows what controller to allocate the message from.

Returns

A pointer to the constructed message. This can be useful in designing a timer service, as after you deliver the message through an **RTPort_enqueue**, you still may be able to cancel the timer request if a **cancelTimer** request is made by following this pointer.

RTPort_registerAs

int RTPort_registerAs(RTPort *, 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. The port knows whether or not to register itself as a published or unpublished unwired port based upon the appropriate code generation model properties.

Note: The protocols referenced by the unwired ports cannot be verified by the toolset, since there are no connectors. At runtime, protocol compatibility is not preformed and it is possible to register a SAP and SPP with the same name but incompatible protocols.

Parameters

service

This parameter is a string that is used to identify a unique name and service under which the unwired ports will connect.

The pointer to the registration name, such as *service*, is stored in the SAP/SPP registration table. **RTPort_registerAs** does not make a copy of the registration name. Therefore, the application should never change the service contents.

Remarks

If this operation is invoked on an unwired port 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 an unwired port is registered, it does not necessarily mean that the port has been connected to another unwired port. The successful completion of the register operation simply indicates that the name has been registered.

Note: Unavailable in certain Services Library configurations. For additional information, see *INTERNAL_LAYER_SERVICE* in the *C Target Guide*.

RTPort_deregister

int RTPort_deregister(RTPort *);

Return Value

Returns 1 (true) if the deregistration of the service name was successful, and 0 (false) otherwise.

Remarks

When an unwired port is deregistered if it is currently connected to another unwired port, the connection is terminated.

Note: Unavailable in certain Services Library configurations. For additional information, see *INTERNAL_LAYER_SERVICE* in the *C Target Guide*.

RTPort_isBound

int RTPort_isBound(const RTPort *, int *index* **);**

Return Value

Returns 1 (true) if the port instance specified by the index is bound (either through a connector or a layer registration).

RTPort_getRegisteredName

const char * RTPort_getRegisteredName(const RTPort *, int *index***);**

Return Value

Returns the name of the service by which an unwired port instance specified by the **RTPort** and index parameters has registered as. If the port is not unwired, or it the port has not yet registered, this function returns `(const char *)0`.

Note: Unavailable in certain Services Library configurations. For additional information, see *INTERNAL_LAYER_SERVICE* in the *C Target Guide*.

RTPort_isRegistered

int RTPort_isRegistered(const RTPort *, int *index* **);**

Return Value

Returns positive logic values to indicate if the unwired port instance specified by index is registered under a particular name. If the port is unwired, this method returns 0.

Note: Unavailable in certain Services Library configurations. For additional information, see *INTERNAL_LAYER_SERVICE* in the *C Target Guide*.

RTPort_informIn

RTTimerId RTPort_informIn(const RTPort * *this***, long** *sec***, long** *nsec***, RTPriority** *prio***, void *** *data***, const RTObject_class *** *type* **);**

Remark

RTPort serves as an interface to a registered timing service which hooks up to particular **RTController** objects on a global or per-thread basis. This function sets a timer to expire in **sec** seconds and **nsec** nano-seconds. When this timer expires, the timer capsule shall enqueue a message on the specified port, with the specified data, priority and type.

Returns

An **RTTimerId** object that represents the timer entry in the timing capsule. This **RTTimer** instance may be used to cancel or query the timer's status.

Note: Unavailable in certain Services Library configurations. For additional information, see *TIMING_SERVICE* in the *C Target Guide*.

RTPort_cancelTimer

int RTPort_cancelTimer(const RTPort * *this* **, RTTimerId** *id***);**

Remark

The **RTPort** serves as an interface to a registered timing service which hooks up to particular **RTController** objects on a global or per-thread basis. This method serves as a means of cancelling a timer request indicated by the id, which was returned by an **RTPort_informIn** call.

Returns

This function returns positive logic to indicate its success.

Note: Unavailable in certain Services Library configurations. For additional information, see *INTERNAL_LAYER_SERVICE* in the *C Target Guide*.

RTPort_isTimerValid

int RTPort_isTimerValid (const RTPort * *this***, RTTimerId** *id***);**

Remark

This operation checks the **RTPort_informIn** request made upon the port passed as a parameter that the given timer id is still an outstanding timer (for example, if the timer is to expire). The **RTPort** serves as an interface to a registered timing service which hooks up to particular **RTController** objects on a global or per-thread basis.

Returns

A boolean logic value that indicates if the timer specified by the **RTTimerId** is valid.

Note: Unavailable in certain Services Library configurations. For additional information, see *INTERNAL_LAYER_SERVICE* in the *C Target Guide*.

RTPort_createInSignal

RTPort_createInSignal(port, signal)

This operation is not a function, but a macro used to define a local signal given the name of a port and a signal. By using macros, if the name of a protocol class changes, all of the capsule user code that uses these signals does not need to be updated. This method is meant to be used for In signals only, to distinguish between triggers when forwarding messages.

RTPort_createOutSignal

RTPort_createOutSignal(port, signal)

This operation is not a function, but a macro used to define a local signal given the name of a port and a signal. By using macros, if the name of a protocol class changes, all of the capsule user code that uses these signals does not need to be updated. This method is meant to be used for out signals only, to create signals that are to be used in **RTPort_send**, **RTPort_sendAt** and **RTPort_enqueue** operations.

RTPriority

Priorities are abstracted through the **RTPriority** enumeration. The following priorities are available (from highest to lowest):

- RTPriority_System
- RTPriority_Panic
- RTPriority_High
- RTPriority_General
- RTPriority_Low
- RTPriority_Background

These priorities must be specified the following primitives:

- RTPort_send
- RTPort_enqueue
- RTPort_sendAt
- RTPort_informIn

RTSoleController

RTSoleController is a refinement of the **RTController** class which represents the interface to a physical thread in the single-threaded run-time system. In order to implement a timer capsule that plugs into the C Services Library, you may need to use the following primitives.

Operations

RTSoleController_waitForEvents	Put the **RTSoleController** in the phase of waiting for events to happen (either timer **timeout**, input/output, or **ipc** events).

RTSoleController_waitForEvents

void RTSoleController_waitForEvents(RTSoleController *);

Remarks

This is the means by which an **RTController** in the single-threaded Services Library goes to sleep. Using this method to go to sleep ensures that if the thread receives a message, it shall be woken up in order to deliver it.

RTSignal

This class is the encapsulation of signals within the Services Library. All signals are defined locally, so they must be specified with regards to the **RTPort** that they apply to. All of the operations upon **RTSignals** can be found in the **RTPort** module.

RTTimerId

The Rose RealTime Timing services use **RTTimerId** as an identifier for timer requests. The timer identifier is returned by a request to **RTPort_informIn**. The timer identifier can be used subsequently to cancel the timer by calling **RTPort_cancelTimer**.

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_addTo	Arithmetic operators
RTTimespec_lessEqualTo	Comparison operators
RTTimespec_clock_gettime	Returns the current time

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;
t1.tv_sec = 1;
t1.tv_nsec = 0;
```

This class is used most often in conjunction with the Timing Service to specify time values.

RTTimespec_clock_gettime

void RTTimespec_clock_gettime(RTTimespec *);

Parameters

tspec

The values of this **RTTimespec** parameter are filled in with the current time.

Example

```
RTTimespec t;
RTTimespec_clock_gettime( &t );
```

RTTimespec_lessEqualTo

int RTTimespec_lessEqualTo (const RTTimespec *, const RTTimespec);

Remark

To check for equality, just use the built in operator == for the structures.

Return Value

Nonzero if the first object is less than or equal to the second object; otherwise 0.

RTTimespec_addTo

void RTTimespec_addTo (RTTimespec *, const RTTimespec *);

Add the value of the second **timespec** to the first. Since the first parameter is not const, the value is saved in it.

Index