IBM Rational Rose RealTime Target Service Library

Version 1.1

Revision History

Date	Version	Description	Author
2006 March 17	1.0	Initial Release	M. Lang
2006 March 29	1.1	Added some communication performance information and reference to testing a TargetRTS port.	M. Lang

Table of Contents

IBM Rational Rose RealTime - Target Service Library

1. Introduction

1.1 Purpose

This document provides insight in to some of the key concepts of IBM Rational Rose RealTime's Target Service Library (TSL). The reader will gain an understanding of the TSL architecture, communication performance considerations, and the opportunities available to customize the TSL for specific needs and environments.

1.2 Scope

While this document is directed to the TSL, a general knowledge of the IBM Rational Rose RealTime v2003.06.00 toolset is expected. While the majority of the concepts apply to all the supported languages, this document is targeted to C++ development. For more specific information regarding the TSL and considerations for customization, reference the *Adapting Rational Rose RealTime for Target Environments* guide that is provided with the toolset (**Tools > TargetRTS Wizard > Porting Guide** or `$ROSERT_HOME\Help\rosert_cpp_ref_guide.pdf`).

1.3 Definitions, Acronyms, and Abbreviations

FIFO	First In First Out
OS	Operating System
RRT	Rose RealTime
RTS	Run Time Services
TargetRTS	Target Run Time Services
TSL	Target Service Library
UML	Unified Modeling Language
UML-RT	Unified Modeling Language Real-Time profile

1.4 Overview

This document is divided in to three main sections:

- Target Service Library Architecture - This section discusses the architecture of the Target Service Library in detail.
- Intra-thread and Inter-thread Communication - This section discusses the intra-thread and inter-thread communication mechanisms and how messages are handled using the Target Services Library.
- Configuring and Customizing the Target Service Library - This section discusses the different ways that are available for customizing the Target Service Library.

2. References

- Advanced Development Workshop for Developer C++, Release 5.2 version 1.4, March 2000, ObjecTime Limited
- Rational Rose RealTime QuickStart/Deployment Service, Version 1.1, February 2002, Rational Software Corporation
- Adapting Rational Rose RealTime for Target Environments, Version 2003.06.00, Rational Software Corporation
- DEV470 Mastering Rational Rose RealTime using C++, version 2003.06.00, IBM / Rational software

3. Target Service Library Architecture

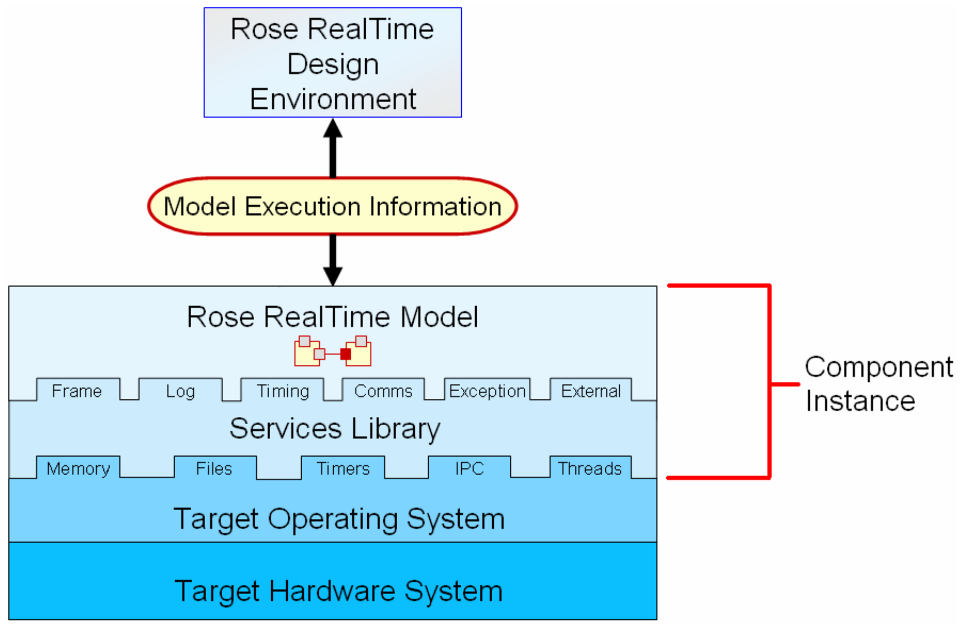
This section discusses the architecture of the Target Service Library in detail. After this section you should understand the key responsibilities of the Target Service Library, the structure and role of each of the main Target Service Library components, and how threads are used within a Target Service Library.

3.1 Key Responsibilities

3.1.1 The Target Service Library

The Target Service Library is the set of run-time services that provide a framework in which a Rational Rose RealTime model can run. It provides the run-time implementation of the UML-RT constructs used in the model.

Think of it as a layer of software that sits between a Rose RealTime model and the Operating System that the component instance, or executable, is running on.



3.1.2 Target Service Library Features

The Target Service Library has several key features and was designed for performance, portability and ease of customization.

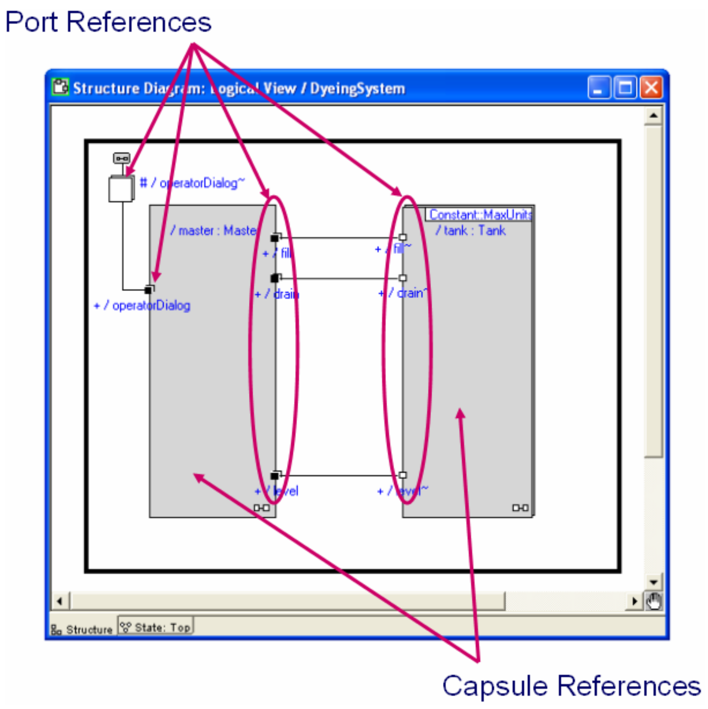
- It provides essential services used by most real-time applications, this includes asynchronous and synchronous messaging services, thread services, timing services, and log services.
- It provides support for the UML design concepts such as capsules, ports and their bindings, and finite state machines.
- It allows Rose RealTime models to be compiled into stand-alone executables.
- It supports a wide variety of platforms out-of-box and is fully user-customizable as the entire Service Library source is provided with the tool.
- It allows direct access to the underlying OS.

3.2 Structure and Role of the Components

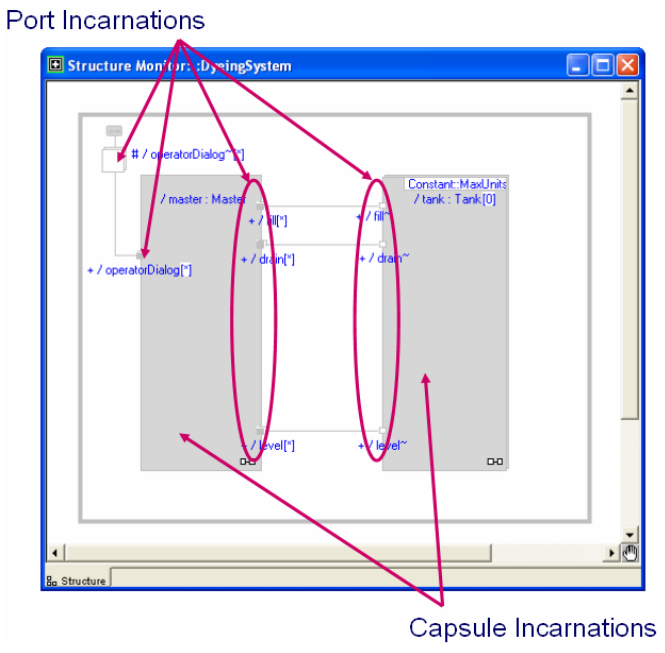
3.2.1 TSL Component Overview

The key structural and behavioral components of a UML model manifest themselves in the form of classes at the Target Service Library level. These components are capsules and ports, and they have several objects related to them.

Design View



Runtime View



3.2.2 Capsules

Each capsule contains the definitions of its interfaces to other capsules (the ports), its behavior in the form of a Finite State Machine (FSM), its attributes and operations, and other contained capsules (if they exist).

There are several Target Service Library classes that are used to reference and implement capsules:

- RTActorClass – This is the capsule class identifier
- RTActor – This class represents the run-time incarnation of a capsule.
- RTActorRef – This is a placeholder and manager of capsule incarnations.

3.2.2.1 RTActorClass

This class is created to represent the common external features (interface ports and capsule name) of each capsule in your model. Only one instance of a **RTActorClass** structure exists for all capsule instances. This way common information about the capsule class can be stored only once.

The class name must map to some unique identifier which is why there is a need for an RTActorClass instance. The class has an identifier which is required to specify the capsule class for certain operations, for example, to incarnate a capsule of a particular class.

3.2.2.2 RTActor

Every capsule when generated as C++ code is a subclass of **RTActor**. This common base class for all capsules defines attributes and operations which allow the Services Library to communicate with the running capsule instances.

RTActor::context	Gets the controller for the physical thread on which a capsule instance is executing.
RTActor::getCurrentStateString	Gets the current state name containing the executing segment.
RTActor::getError	Gets the last error value for this thread.
RTActor::getIndex	Gets the replication index of this capsule instance in the home capsule role.
RTActor::getName	Gets the capsule role name in which this capsule instance is running.
RTActor::getTypeName	Gets the capsule class name of this capsule instance.
RTActor::isType	Queries the capsule class of this capsule instance.
RTActor::logMsg	Called before every message is delivered to a capsule instance (if configured in Services Library).
RTActor::msg and RTActor::getMsg	Accesses the msg attribute.
RTActor::unexpectedMessage	Called when a message is delivered to a capsule instance for which there is no trigger defined.

Capsule incarnations are the objects that actually receive and process messages. These are the graphical objects that you design within the Rose RealTime toolset that maintain the state of individual capsule incarnations.

3.2.2.3 RTActorRef

The **RTActorRef** class maintains information about each capsule role in your model. For each capsule role in the structure of a capsule, an attribute of this type is added to the RTActor subclass generated C++ capsule class.

The main purpose of the reference object is to maintain the set of incarnations related to the named reference. Typical tasks requiring capsule references are to:

- Create a new incarnation in the reference location (within the context of a container capsule)
- Destroy one or more incarnations in the reference location
- Import or deport one or more incarnations into the reference location
- Find the set of incarnations related to a particular reference

3.2.3 Ports

Ports are objects whose purpose is to send and receive messages to and from capsule instances. They are owned by the capsule instance in the sense that they are created along with their capsule and destroyed when the capsule is destroyed. Each port has its identity, which is distinct from the identity and state of their owning capsule instance.

To specify which messages can be sent to and from a port, a port is associated with a protocol role. The protocol role is the specification of a set of the messages that can be received (in) and sent (out) from the port. The protocol role essentially defines the port type.

There are several classes related to ports:

- RTProtocol - For each protocol class in your model, two subclasses of the **RTProtocol** class are generated for each direction of the protocol or protocol roles.
- RTInSignal - This class is used to work with incoming signals defined within a protocol.
- RTOutSignal - This class is used to work with outgoing signals defined within a protocol.
- RTSymmetricSignal - This class is used for symmetric signals defined within a protocol. Since symmetric signals can be both incoming and outgoing you can perform the combined actions of both RTOutSignal and RTInSignal on these classes.

3.2.3.1 RTProtocol

Each port defined on a capsule is generated as an attribute of the generated C++ capsule class. The port attribute has the same name as the port, with the type as either the base or conjugate protocol role.

3.2.3.2 RTInSignal

Each signal defined on a protocol becomes an operation. For incoming signals the operations return an **RTInSignal** object on which you can specify what action to perform with the signal.

RTInSignal::purge	Delete all of these deferred signals for all port instances.
RTInSignal::purgeAt	Delete all of the deferred signals on a specific port instance.
RTInSignal::recall	Recall one deferred signal on all port instances.
RTInSignal::recallAll	Recall all deferred signals on all port instances.
RTInSignal::recallAllAt	Recall all deferred signals on a specific port instance.
RTInSignal::recallAt	Recall one deferred signal on a specific port instance.

3.2.3.3 RTOutSignal

Each signal defined on a protocol becomes an operation. For outgoing signals the operations return an **RTOutSignal** object on which you can specify what action to perform with the signal.

RTOutSignal::invoke	Synchronous message broadcast to all port instances.
RTOutSignal::invokeAt	Synchronous message send to a specific port instance.
RTOutSignal::reply	Used to respond to a synchronous message.
RTOutSignal::send	Asynchronous message broadcast to all port instances.
RTOutSignal::sendAt	Asynchronous message send to a specific port instance.

3.2.4 TSL Behavioral Overview

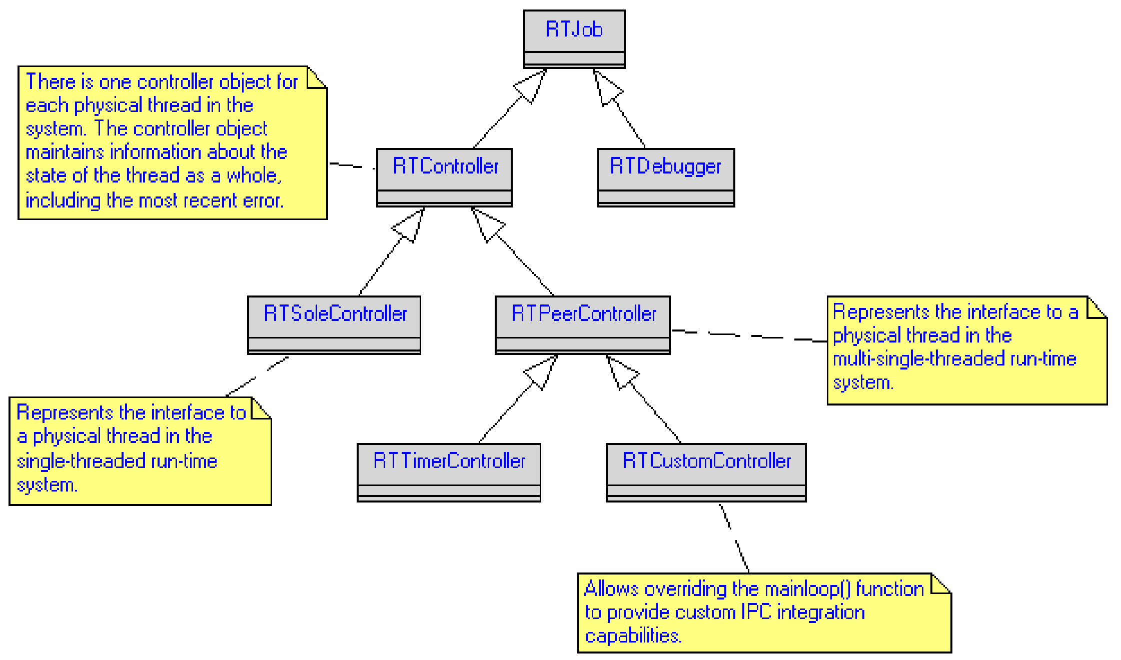
Now that we have discussed the key structural components, how do they communicate with one another? A Controller object is used to group capsules together into execution units and to handle the communication between them.

Each execution unit exhibits the following properties:

- There is only one thread of control.
- Capsules have Run-to-completion semantics with respect to every other capsule in the unit.
- All capsules in a unit share a common resource pool: a stack, message bodies, and message queues.

3.2.5 Controllers

Conceptually, all capsules are concurrent, but in reality they are all grouped together. Controllers are the conceptual objects used to group capsules together into a single unit of concurrency. The controller class hierarchy is shown below:


RTJob
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.
RTController
RTDebugger
RTSoleController
RTPeerController
Represents the interface to a physical thread in the multi-single-threaded run-time system.
Represents the interface to a physical thread in the single-threaded run-time system.
RTTimerController
RTCustomController
Allows overriding the mainloop() function to provide custom IPC integration capabilities.

3.2.6 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. The RTController class provides all of the messaging support that a thread requires.

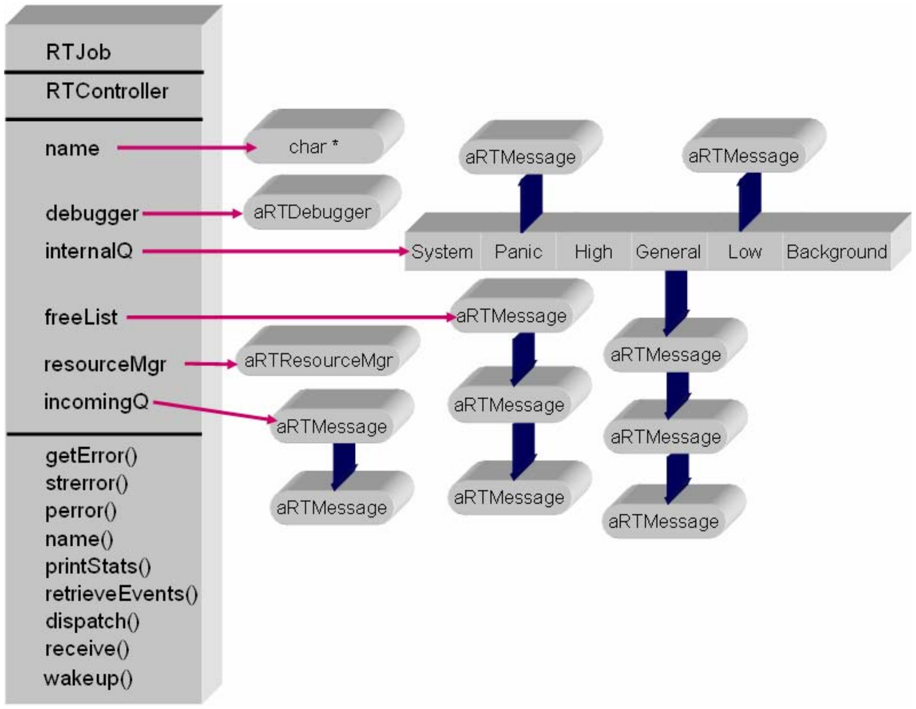
It is often said that capsules exhibit run-to-completion semantics. This statement means that the controller does not interrupt a capsule while it is executing a run-to-completion step, although the underlying OS can swap out the thread it is running on.

The run-time services library is thread-based. It delivers one message from the message queue at a time. When the capsule receives a message, a transition chain is triggered. The entire transition chain must be executed before the controller delivers the next message.

3.2.7 RTController Structure Model

The structure of the RTController is shown below. Several of the notable elements of the controller are:

- internalQ – List of messages to be dispatched to capsules managed by this controller. The controller will process the queue based on the highest priority message in a FIFO fashion.
- freeList – Manages the list of free messages available for communication.
- resourceMgr - Allocates freeList at startup and when more messages are needed. Surplus free messages are maintained in a global free queue for redistribution across threads as needed.
- incomingQ – List of messages that have arrived from other controllers, that is, from inter-thread communication.



Additional RTController member variables not shown are:

- internalPriority - variable holds the index to the highest priority message queue that is not empty. The first messages to be dispatched are taken from this queue.
- incomingPriority - is used to track the highest priority message delivered to the incomingQ but which hasn't been moved to the internalQ. It is used to force the RTController mainloop to retrieve events from the incomingQ when there are high priority inter-thread messages that have been received.

3.3 How Threads are Used

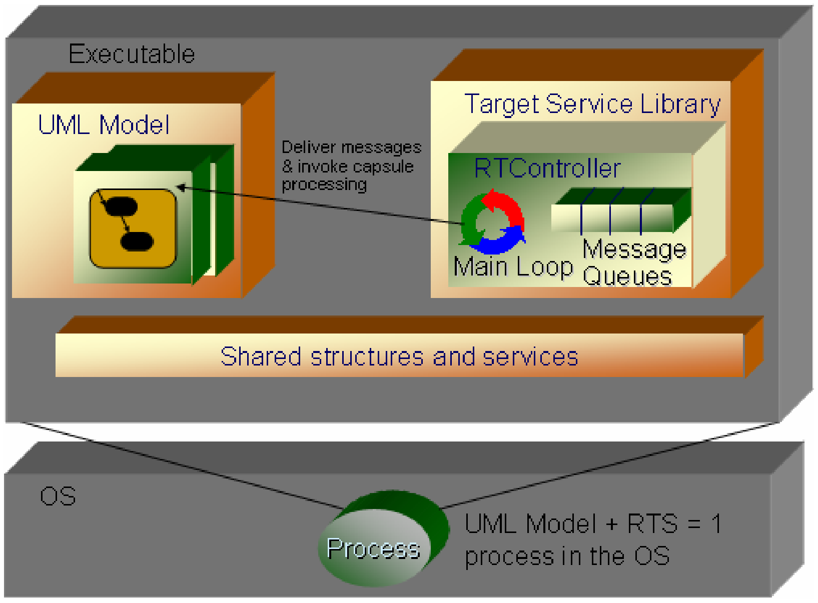
3.3.1 *Threads*

The Target Service Library supports both single-threaded and multi-threaded environments but there are differences in how the Target Service Library sets up the execution environment and handles events between the single-threaded and multi-threaded implementations.

- The multi-threaded version can handle external requests using a different thread
- The multi-threaded version must create extra threads at startup to handle system events and debugging.

3.3.2 *Single-Threaded Target Service Library*

The single-threaded version of the Target Service Library contains a single controller object which is an instance of RTSoleController. The RTSoleController polls for system events (such as timer timeouts and debugger input) and dispatches messages inside of its main loop.


Executable
UML Model
Deliver messages & invoke capsule processing
Target Service Library
RTController
Main Loop
Message Queues
Shared structures and services
OS
Process
UML Model + RTS = 1 process in the OS

The RTController shown above is actually an RTSoleController (RTController is the superclass).

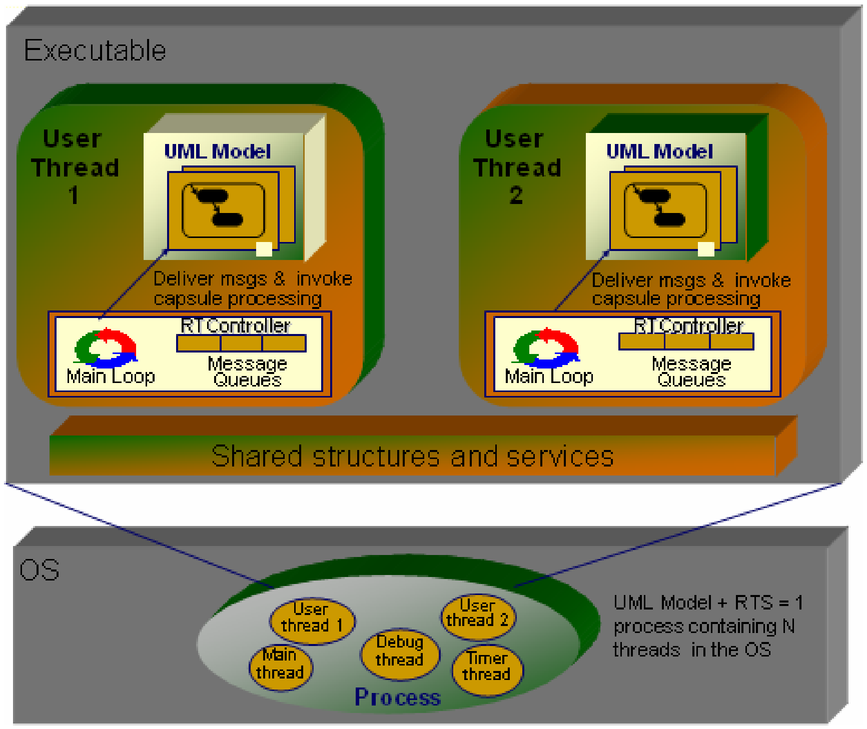
3.3.3 *Multi-Threaded Target Service Library*

In the multi-threaded version, a controller object which is an instance of RTPeerController or RTCustomController runs on each physical user thread. This instance controls the execution of the capsules on that thread and handles the dispatching and receiving of messages from other threads.

There are also several system threads that may be created to handle timers and debugging.

- A timer thread is required and is used for the timing services
- The debug thread is optional and is only used when the Services Library debugger is enabled.

If you have created a multi-threaded model, in addition to the MainThread and system threads, there will be a user thread created for each physical thread that you have specified in your model. All capsules that are not explicitly incarnated onto a specific thread will run in the Main Thread.

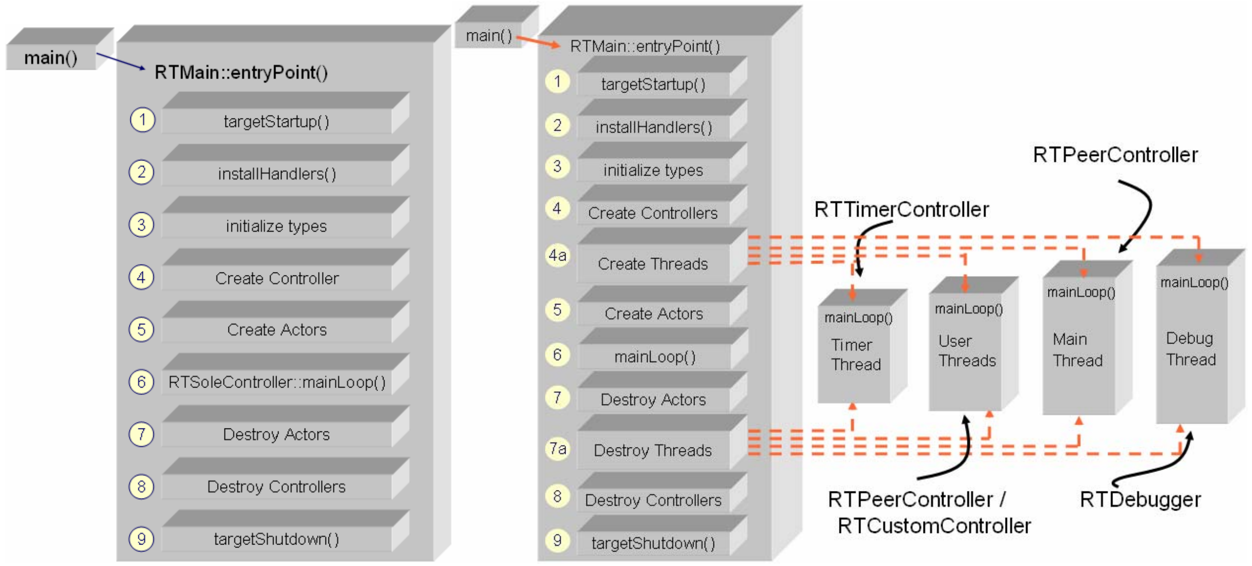


The RTControllers shown above are either RTPeerController or RTCustomController (RTController is the superclass).

3.3.4 Initialization of the Target Service Library

When a Rose RealTime model is first executed, it is initiated from the main() function just like any other C++ program. The main() function calls RTMain::entryPoint() which takes care of all of the setup and tear down of the model executable.

The single-threaded and multi-threaded versions have a few differences in how they are initialized.



Single Threaded Initialization **Multi-Threaded Initialization**

4. Intra-thread and Inter-thread Communication

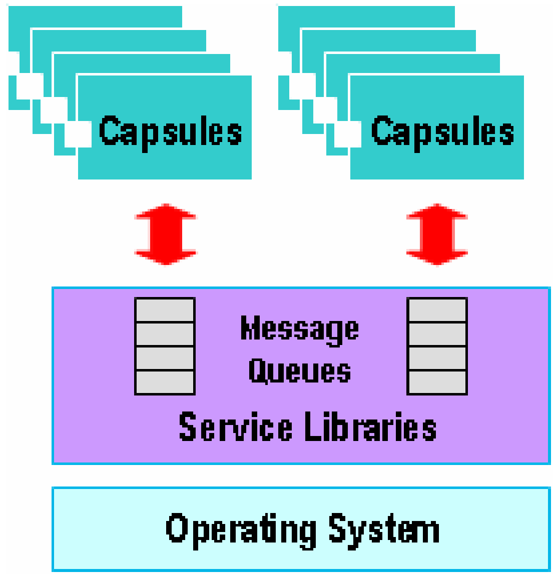
This section discusses the intra-thread and inter-thread communication mechanisms and how messages are handled using the Target Services Library.

From the application's perspective, there is no difference between sending a message within a thread and sending a message across threads; the code to send the message is still the same. There are, however, some performance implications, approximately a 10-20 X hit. Optimal designs place capsule instances that have "an intense message dialog" on the same thread.

The capsule behaviors are control independent and generated from the design model. They are defined through state machines (high level) and detailed code (detail level) and communicate by sending and receiving messages.

The Service Libraries are linked with the generated code to create an application executable. The TSL provides OS abstraction, thus making capsule designs independent of OS selection.

COTS software that the user buys.



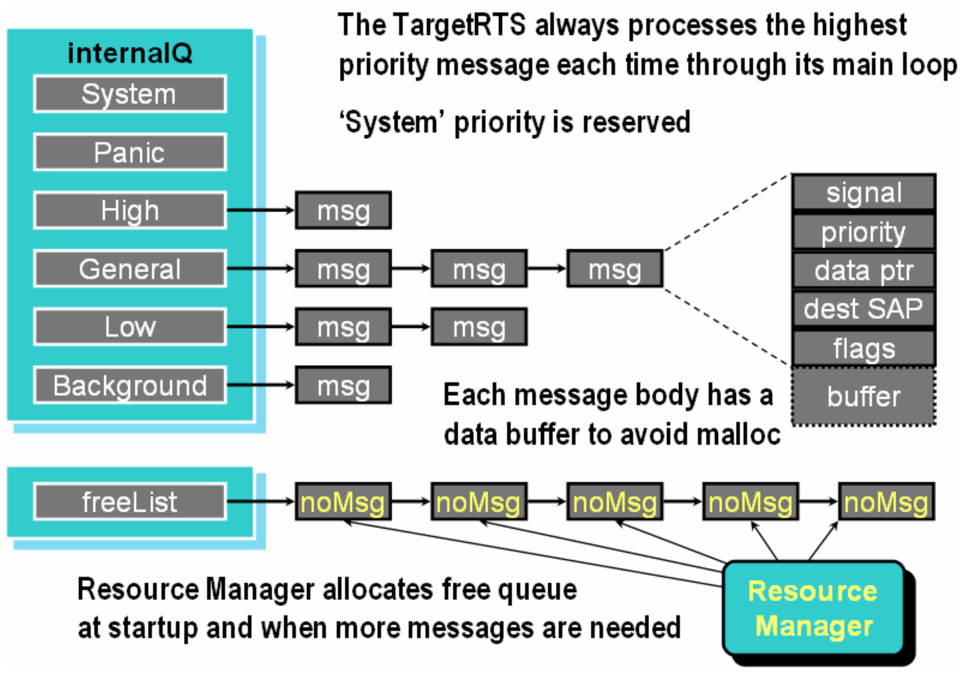
The TSL provides for a simple message dispatch algorithm, find the highest-priority, non-empty message queue, take the message from the head of that queue and deliver it to the recipient capsule. Repeat once the recipient capsule completes processing the message.

Message delivery is via a call invocation to `rtsBehavior(signal,port)`. This call is made inside the message dispatch loop. Run to completion is enforced as the control is not returned to the loop until the capsule completes execution and makes a "return".

4.1 Intra-thread Communication

4.1.1 Message Structures

All queues shown are dedicated to a particular thread. Similar structures would be created for each kernel thread defined.



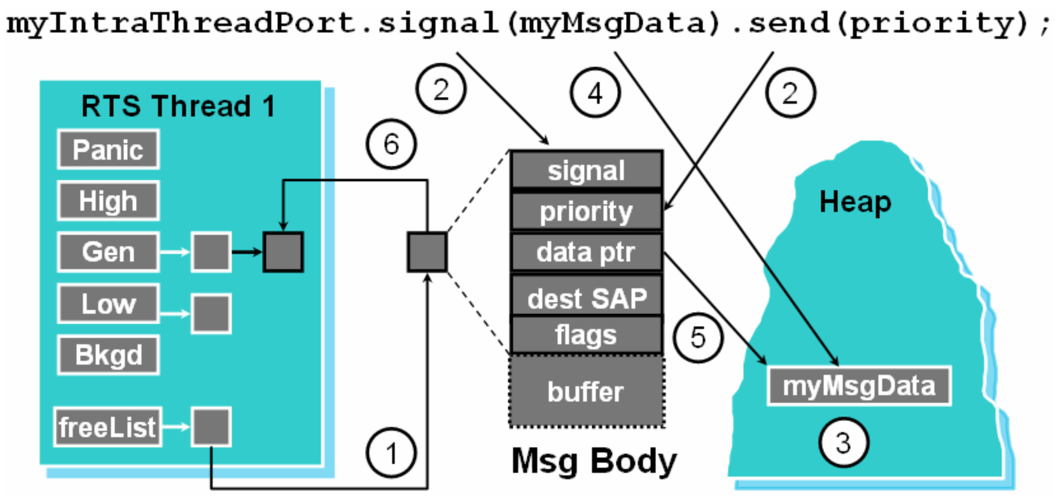
internalQ
System
Panic
High
General
Low
Background
msg
The TargetRTS always processes the highest priority message each time through its main loop
'System' priority is reserved
signal
priority
data ptr
dest SAP
flags
buffer
Each message body has a data buffer to avoid malloc
freeList
noMsg
Resource Manager
Resource Manager allocates free queue at startup and when more messages are needed

The Resource Manger is responsible for allocating extra message bodies when a low threshold is passed and also returning extra message bodies to a global pool as a high threshold is passed. When a thread needs more, the managed pool is checked first and if sufficient messages are not available, a malloc is made on the system heap. Note that once allocated, message bodies are not freed back to the system heap but managed in the global pool; therefore, the system will eventually reach equilibrium and no further dynamic memory allocation will be required.

Free queue thresholds and data buffer size are configurable values and a service library recompile is required when these are changed.

4.1.2 Message Send

The following example is the "worst case scenario" for a send by value and the data specified is too large to fit in the buffer so memory must temporarily be allocated from the system heap. Typically, the data buffer size value will hold the data avoiding a heap operation. Often, the malloc is the single most expensive part of the operation.

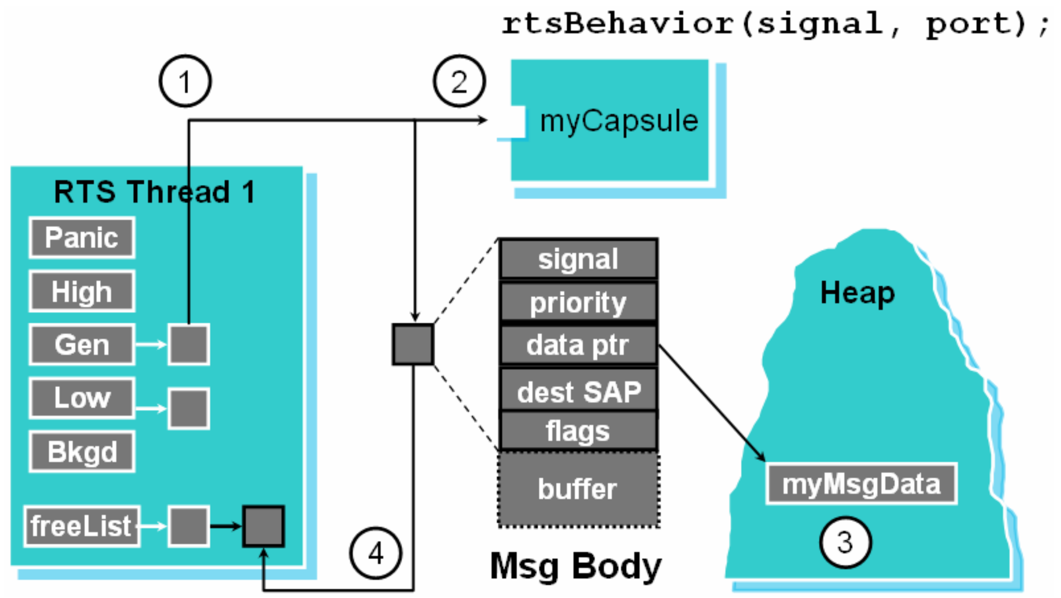


1) Get an empty message body from this thread's free queue.
2) Copy the signal and priority (in this case a General priority) in to the message body directly.
3) Malloc from system heap for temporary data buffer (only required if data element is too large for built in buffer).

   Note that the malloc is only required for a large block of sent data. Typically, sent data fits in the message body buffer (the size of the buffer is configurable in the TargetRTS).

4) Data value copied into temporary buffer.
5) Pointer to temporary data buffer is updated in message body.
6) Enqueue the completed message body in the appropriate queue as indicated by the priority value in the send statement.

4.1.3 Message Receive



1) Message at the head of the highest non-empty queue is processed next (General priority queue is shown in this example).

2) Capsule instance is identified by the destSAP field in the message body and the capsule behavior is called via `rtsBehavior(signal, port)`.

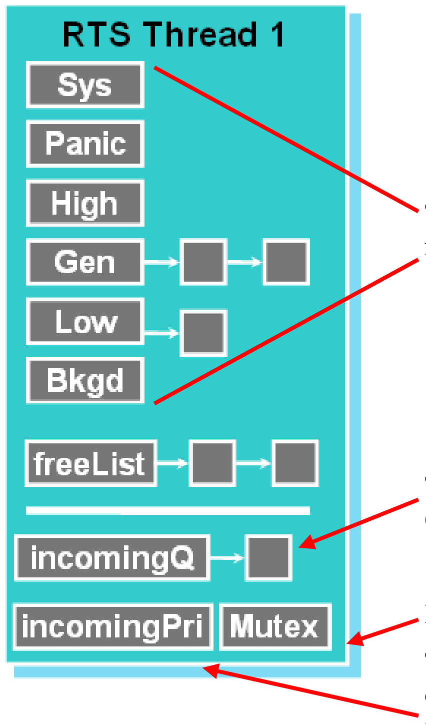
3) Upon completion of message processing (indicated by a return from the called capsule instance) the temporary memory buffer, allocated when the message was sent, is released. Note that if the capsule's behavior kept a pointer to this data, it is now looking at garbage.

4) Message body is released to the receiving thread's free queue. Any free queue imbalance is corrected on an as needed basis. Typically, balance is achieved through normal message exchange between threads.

4.2 Inter-Thread Communication

4.2.1 Message Structures

In addition to the internal queue structure used for intra-thread communication, inter-thread communication requires an incoming queue to hold the messages, an indication of highest priority and a mutex to protect the message.



The internalQ is a list of messages to be dispatched to capsules managed by this controller.

The incomingQ is actually a set of priority-ordered queues (presented as a single queue here for simplicity).

Inter-thread queues are protected by mutual exclusion.

The incomingPriority is used to track the highest priority message delivered to the incomingQ but which hasn't been moved to the internalQ.

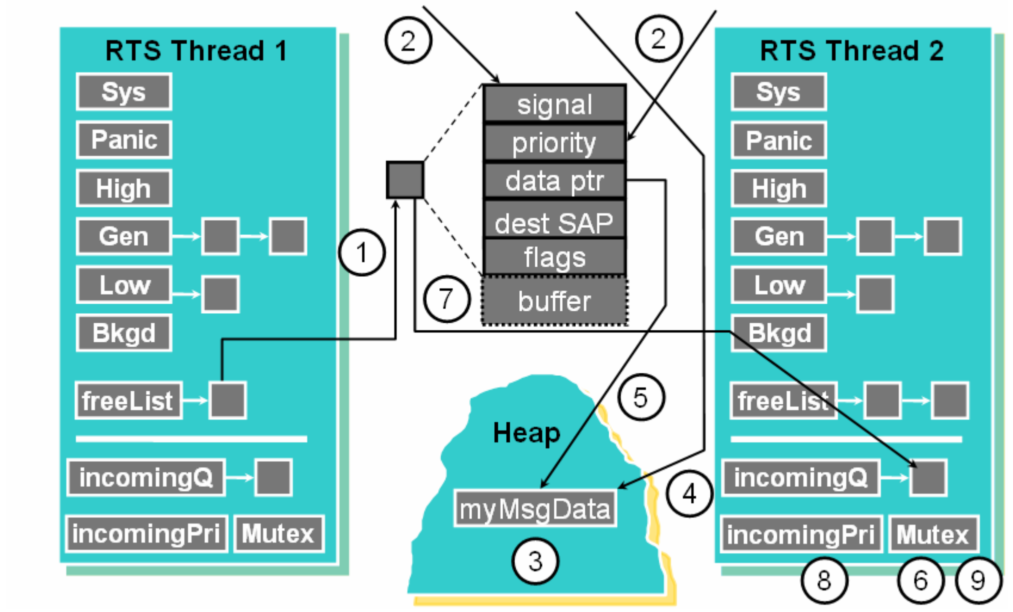
Note that the incomingQ represents a set of priority ordered queues with each corresponding to an internal queue. As the entire incomingQ is flushed, all external messages will be added into the appropriate internal priority based queue.

This design provides optimal performance by minimizing the number of mutex claims for incomingQ flushing. In highly concurrent systems with multiple threads, the overhead for inter-threads sends can be reduced significantly on a per message basis.

4.2.2 Message Send

The following example is the "worst case scenario" for a send by value and the data specified is too large to fit in the buffer so memory must temporarily be allocated from the system heap. Typically, the data buffer size value will hold the data avoiding a heap operation. Often, the malloc is the single most expensive part of the operation.

```
myInterThreadPort.signal(myMsgData).send(priority);
```



An inter-thread message send begins just like an intra-thread message send.

1) Get an empty message body from this thread's free queue.

2) Copy signal and priority in message body directly.

3) Malloc from system heap for temporary data buffer (only required if data element is too large for built in buffer).

   Note that as in the intra-thread case, the malloc is only required for a large block of sent data. Typically, sent data fits in the message body buffer (the size of the buffer is configurable in the TargetRTS).

4) Data value copied into temporary buffer.

5) Pointer to temporary data buffer is updated in message body.

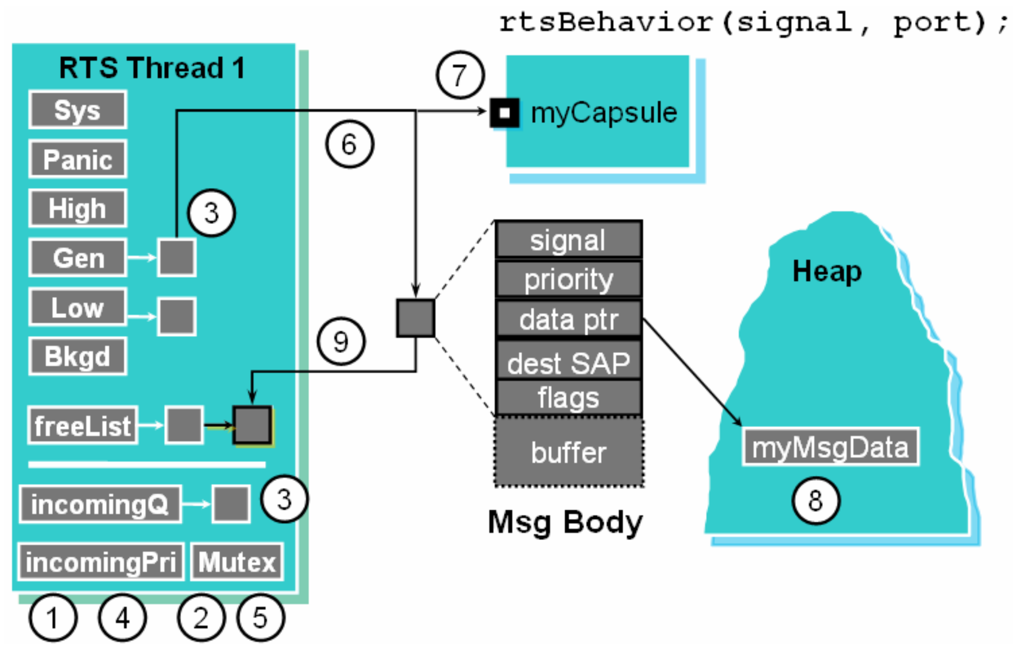
6) Claim mutex for the receiving side's incoming queue. Note that we've waited until we need it to minimize blocking.

7) Message body is enqueued in the receiving side's incoming queue in the queue of the correct priority.

8) Update the incoming priority indicator to reflect the highest priority of all messages enqueued in the receiving side's incoming queue.

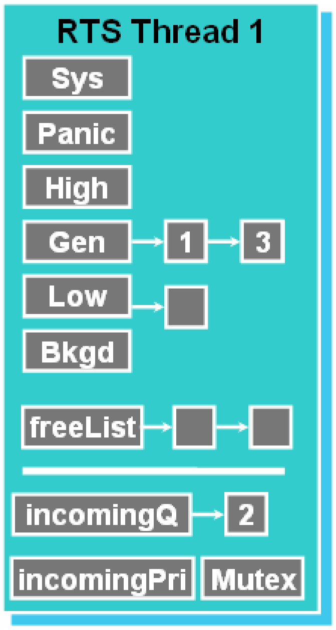
9) Free the mutex for the receiving side's incoming queue.

4.2.3 Message Receive



1) First check to see if a message in the incoming queue has a higher priority that all internal messages. In this example that is the case.
2) Need to claim the mutex to ensure protected fields can be updated safely.
3) Flush all messages from the incoming queue, regardless of their priorities, and add these messages to the corresponding internal priority based queues.
4) Reset incoming priority indicator to empty.
5) Release the mutex. Note that we only hold a mutex as long as absolutely necessary to minimize blocking.
6) Message at the head of the highest non-empty internal queue is processed next (General priority message just flushed from incoming queue).
7) Receiving capsule instance is identified by the destSAP field in the message body. Capsule behavior is called: `rtsBehavior()`.
8) Upon completion of message processing, indicated by a return from the called capsule instance, the temporary memory buffer that was allocated when the message was sent, is released.
9) Message body is released to the receiving thread’s free queue; any free queue imbalance is corrected on an as needed basis. Typically balance is achieved through normal message exchange between threads.

4.2.4 Message Ordering



1) General priority message 1 is queued

2) General priority message 2 arrives from another thread

3) General priority message 3 is queued by this thread

Message 1 is processed first

Message 3 is processed before 2

Messages in the incoming queues are moved to the corresponding internal queue, if the highest priority external message is higher than all queued internal messages

This slide allows a discussion of the fact that time sequence of messages is not guaranteed for messages of the same priority that are arriving from the same thread and other threads within a burst.

The design is done this way to reduce the number of mutex claims. In this case several messages may be received and sorted by priority for a single mutex claim.

Interestingly for most designs this is exactly the dispatch algorithm wanted: exhaust the work at a given priority on the internal queues, before looking for “new” work on the external queues.

4.3 Performance

From the application’s perspective, there is no difference between sending a message within a thread and sending a message across threads, the code is still the same:

```
portName.outSignal(data).send(priority);
```

There is however some performance implications where inter-thread communication is approximately 10-20 times more “expensive” than intra-thread communication[1]. But consider that the inter-thread performance includes the cost of two mutex claims and a context switch, which would be the equivalent cost if conventional cross thread protection was used, for example, semaphores.

For optimal performance:

- Virtual function call on simple object is fastest – This does not take into account the logic to identify the receiver object, which can involve a lengthy search and is dependent on the design, or the logic to identify the appropriate response that is required.
- Intra-thread send is fastest for an indication – The identity of the receiver object is determined by the port connections and the appropriate response is determined by the receiver’s state machine.[2]
- Intra-thread invoke is fastest for a query – The synchronous invoke is supported for intra-thread capsule interactions, but it’s suggested that use be restricted to special situations that require synchronization, such as thread-local resource allocation.
- Avoid inter-thread interactions wherever possible – Optimal designs should attempt to place capsule that have “an intense message dialog” on the same thread.

4.3.1 Threading Recommendations

A key strategy is to minimize the number of physical threads as these consume a lot of memory resources (stack, queues, and so on). Also, the more physical threads that are defined, the more CPU cycles are spend on context switching. Intra-thread sends are much more efficient than inter-thread sends so keep as many capsules on as little a number of threads as possible.

The capsule concurrency technology offered by IBM Rational Rose RealTime, helps reduce the need for physical threads where you only need threads for “general logic” and for each blocking I/O interface. A sanity watchdog thread is also useful in more sophisticated designs.

Logical threads[3] also help reduce the number of physical threads by allowing designers more flexibility in specifying precise threading decisions. There is no cost for using logical threads and they are a great tool to allow teams to delay decisions on physical thread mappings until preliminary performance data is in and intelligent mapping decisions can be made.

[1] This is a general comment as actual numbers will vary depending on the reference target, chip set, chip speed, and optimization settings.

[2] This same concept applies to invokes as well.

[3] This is a concept unique to IBM Rational Rose RealTime.

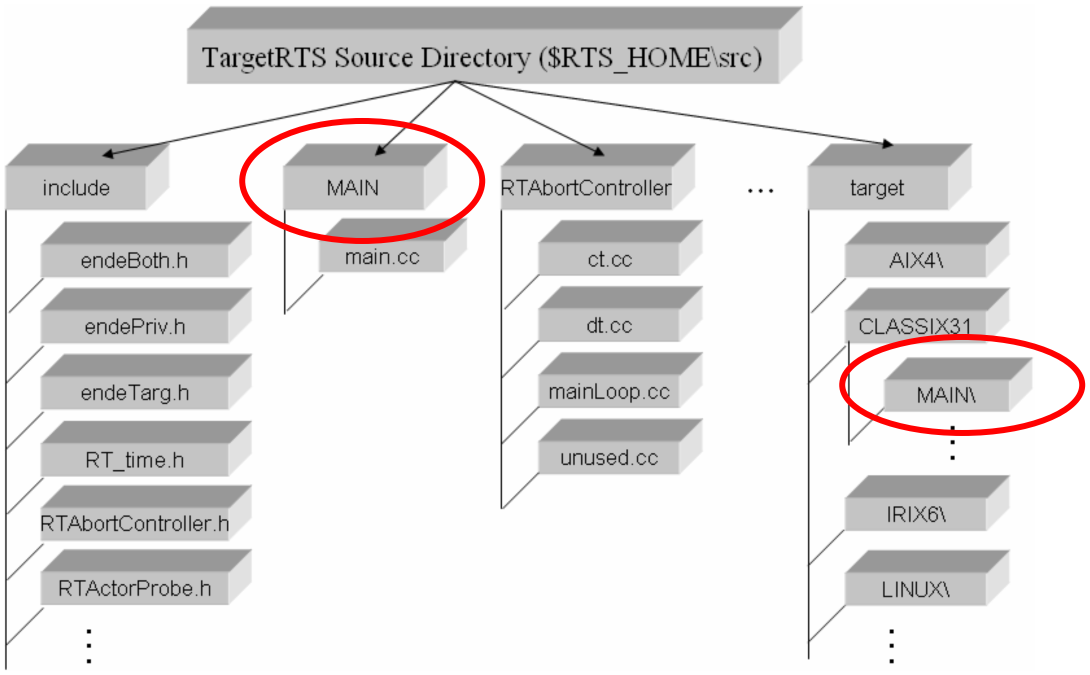
5. Configuring and Customizing the Target Service Library

This section discusses the different ways that are available for customizing the Target Service Library. After this section you should know what configuration options are available and how to change them, understand how to change the classes, functions and definitions contained in the Target Service Library source code, understand how to compile and link customizations in with your model, and how to compile the Target Service Library.

5.1 Organization of the TSL Source

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

Much of the configurability of the Service Library is done at the source code file level, where target-specific source files override common source files.



5.2 Changing the TSL Functionality

There are several different ways of changing the functionality of the Target Service Library and requires the recompilation of the TSL, the model or both.

- You will need to recompile the TSL and model if you; reconfigure the TSL by overriding the preprocessor macros defined in RTConfig.h or modify the TSL build options in the libset.mk, target.mk, config.mk make files, or customize the TSL by overriding or adding source to $RTS_HOME/src/target/<target>.
- You will need to recompile only the model if you override pre-defined capsule methods from within the toolset, or override or add TSL methods and classes using the overrides makefile.

5.3 Predefined Configuration Options

There are several configuration options that can be used to change the behavior of the Target Service Library. This is not an complete list, but most of the following options are defined in the $RTS_HOME/include/RTConfig.h file.

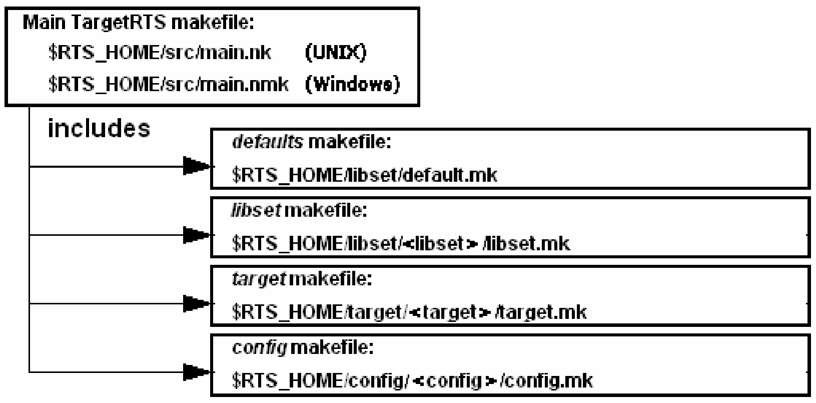
- **OTRTSDEBUG** - Determines how much of the TSL debugger should be present; DEBUG_VERBOSE enables the TargetRTS debugger, DEBUG_TERSE limits the amount of debug information, and DEBUG_NONE eliminates any debug information output and will reduce the executable size, while increasing performance.
- **LOG_MESSAGE** - Determines whether the RTActor::logMsg function is called after delivering each message.
- **DEFER_IN_ACTOR** - Determines if a defer queue is maintained for every capsule or the default of each controller. Maintaining a defer queue for every capsule will improve speed of defers and recalls but adds memory overhead.
- **RTS_COUNT** - Used to keep track of the number of messages sent, the number of capsules incarnated, and other statistics. Naturally, keeping track of statistics adds overhead
- **PURIFY** - This hides some allocation and deallocation events from tools like Rational Purify but will cause degradation in performance
- **RTMESSAGE_PAYLOAD_SIZE** - Reserve this many bytes in RTMessage for small objects. When data must be copied, objects that are no larger than this will use that space in the message itself rather than allocated on the heap.
- **MINIMUM_FREE_MSGQ_SIZE** - When freeing a message, keep at least this many messages in the Controller's free list.
- **DEFAULT_FREE_MSGQ_SIZE** - When freeing a message, keep at most this many messages in the Controller's free list.
- **INLINE_CHAINS** – When this option is specified, transition code chains are inserted directly into the code. This improves the performance of the executable, but it also causes the size of an actor to be slightly larger (about .5K).
- **USE_THREADS** - Determines whether the single-threaded or multi-threaded version of the TargetRTS is used. This option is not defined in the RTConfig.h file and should be set in the $RTS_HOME/target/<target>/RTTarget.h file.

For specifying operating system specific definitions, options should be overridden in the $RTS_HOME/target/<target>/RTTarget.h file.

For specifying compiler specific definitions, options should be overridden in the $RTS_HOME/libset/<libset>/RTLibSet.h file.

5.4 Predefined build options

The build options used to compile both the Services Library and model can be configured in one of the makefiles.


Main TargetRTS makefile:
$RTS_HOME/src/main.nk (UNIX)
$RTS_HOME/src/main.nmk (Windows)
includes
defaults makefile:
$RTS_HOME/libset/default.mk
libset makefile:
$RTS_HOME/libset/<libset>/libset.mk
target makefile:
$RTS_HOME/target/<target>/target.mk
config makefile:
$RTS_HOME/config/<config>/config.mk

The default.mk, libset.mk, target.mk, and config.mk makefiles are used to compile both the TargetRTS libraries and the model. The target.mk, libset.mk and config.mk makefiles override the defaults defined in $ROSERT_HOME/libset/default.mk. These are the makefiles that you can edit.

The main.nmk (nmake for Windows) or main.mk (make for UNIX) is the main definition for compiling the TargetRTS libraries. These makefiles should not be customized, and will not be discussed further in this document.

The default.mk file contains the default macro definitions that may be overridden by the platform-specific makefiles.

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

The libset.mk file contains the definition specific to the compiler.

The config.mk file contains the definition specific to the combination of the compiler, operating system, and TargetRTS configuration.

5.5 Overriding TSL Source

The implementation of the Services Library is in the $RTS_HOME/src directory. Any method in the Services Library can be overridden by placing a target specific version of the method into the $RTS_HOME/src/target/<target_base> subdirectory:

The most common reason for overriding methods in the Target Services Library is to change the way in which it is initialized, modify the main message processing, or add platform specific implementations (porting)

Some interesting methods in the Target Services Library that could be candidates for overriding are:

- mainLoop()
- targetStartup()/targetShutdown()
- retrieveEvents()
- wakeUp()

5.5.1 The mainLoop() method

The mainLoop() function is typically overridden if you want a message handling strategy that is different than the Rose RealTime default, if you need to receive messages from other applications, if you need to enhance the signal handling performed by the Services Library, or if you need to perform regular sanity checks or audits

To create a controller object that has its own mainLoop() function but is not directly overriding one of the Rose RealTime controllers, create a subclass of RTPeerController, change the mainLoop() function, then specify the new subclass when you assign your threads in the thread browser (this is done through the properties dialog on a physical thread).

If you override the mainLoop() function on a Rose RealTime controller class, all controllers incarnated from that class will have the same overrides. This may or may not be desirable.

5.5.2 The targetStartup()/targetShutDown() method

The targetStartup()/targetShutdown() methods are typically overridden to initialize and cleanup device drivers that are specific to the target environment, to setup and cleanup any OS specific services (such as clock timings etc.), to initialize any target specific libraries or structures that are needed by the Target Services Library, or to initialize signal handlers.

5.5.3 Overriding other TargetRTS code

These are not the only methods that you can override. Because of the way that the Target Services Library code is organized, you can override any Target Services Library method

5.6 Customizing the Target Service Library

As we have seen the Target Services Library can be customized for several reasons. How and which Target Services Library files are to be modified will depend on the type of customization required.

Type of customization	Files to be modified
Overriding preprocessor macros defined in RTConfig.h	• `$RTS_HOME/libset/<libset>/RTLibSet.h` or • `$RTS_HOME/target/<target>/RTTarget.h`
Modifying the Services Library build files	• `$RTS_HOME/libset/<libset>/libset.mk` or • `$RTS_HOME/config/<target>.<libset>/config.mk` or • `$RTS_HOME/target/<target>/target.mk`
Overriding Services Library source files	• `$RTS_HOME/src` • `$RTS_HOME/src/target/<target>/`

After it is determined which files are to be modified, the next decision is how to modify them. Refer to the *Adapting Rational Rose RealTime for Target Environments* porting guide for using the TargetRTS wizard.

5.6.1 Preferred Method

The preferred method is to create a new libset and/or target. The pros to this approach are that its light weight and the original Service Libraries can still be used. The instructions for this approach are:

- Make a copy of the following directories and rename them to something appropriate:
  - (for libset port): $RTS_HOME/config/<target>.<libset> and $RTS_HOME/libset/<libset>
  - (for target port): $RTS_HOME/target/<target>
- Then modify the files needed within the new config, libset, and/or target directories.
- If Services Library source files are to be overridden add them to $RTS_HOME/src/target/<target> and when the Service Library is built, these directories are searched first.

5.6.2 Temporary Solutions

One temporary solution is to make a copy of the entire $RTS_HOME directory then modify the files in place. The pros to this approach are that it is very simple and can be used if you don't have write access to $ROSERT_HOME. A consequence is that it is very heavy weight as you will need enough disk space to hold a duplicate copy of $RTS_HOME (greater then 35MB).

Another temporary solution is to modify the files in place. The consequence here is that it is difficult to manage in that the original files and libraries will be lost.

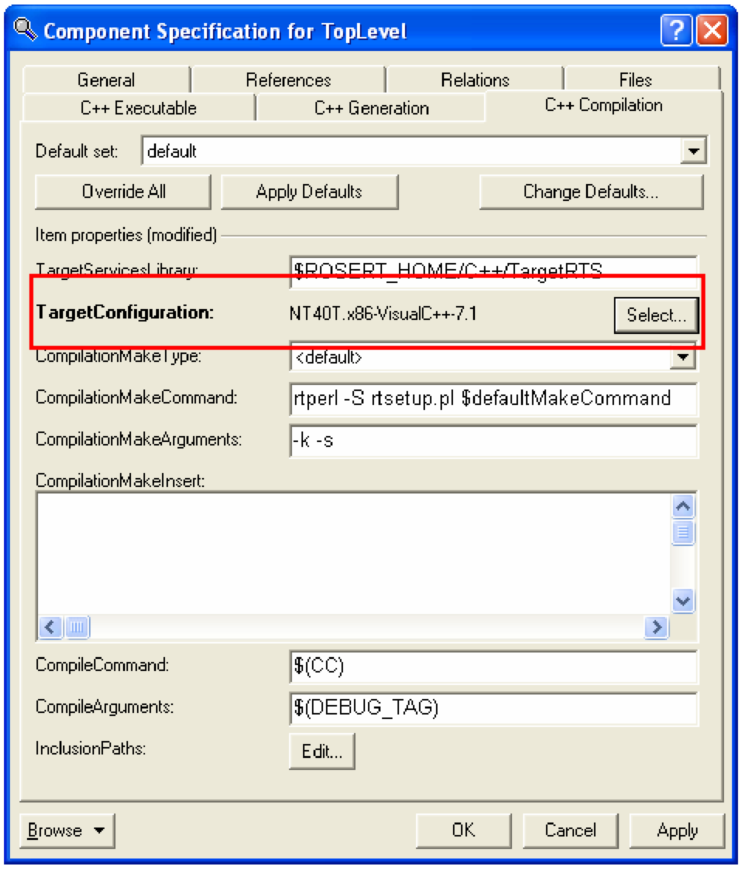
5.7 Compiling the Target Service Library

Once changes have been made to the Target Services Library the last step is to build it. In addition to the TargetRTS wizard, you can also build from the command line. The Target Services Library can be built from $RTS_HOME/src. Run the make utility with the platform name that is to be built: make CONFIG=<target>.<libset>

- Example (UNIX): make CONFIG=SUN5T.sparc-gnu-2.8.1Debug
- Example (WINNT): nmake CONFIG=NT40T.x86-VisualC++-7.1

5.8 Using the new Target Service Library

After the Services Library is rebuilt, your model must be rebuilt in order to link with the new Services Library. Select the new Target Configuration from the Component Specification.



Default set: default

Item properties (modified)

TargetServicesLibrary: $ROSERT_HOME/C++/TargetRTS

TargetConfiguration: NT40T.x86-VisualC++-7.1

CompilationMakeType: <default>

CompilationMakeCommand: rtperl -S rtsetup.pl $defaultMakeCommand

CompilationMakeArguments: -k -s

CompilationMakeInsert:

CompileCommand: $(CC)

CompileArguments: $(DEBUG_TAG)

InclusionPaths:

5.9 Overriding Predefined Capsule Operations

Some Target Services Library methods can be overridden from the toolset. The advantages to this approach are there is no need to modify Target Services Library files, you can scope behavior changes to specified capsules, and there is no need to recompile the Target Services Library.

Methods that can be overridden are:

- unexpectedMessage()
- unexpectedState()
- logMsg()

To override any of these functions, add an operation from the capsule class with the same name and prototype.

5.9.1 Process for Overriding Operations

It is possible to override any Target Services Library method without having to recompile the Target Services Library.

- Copy the function file that you plan to override from the directory where it is located in the TargetRTS source tree ($RTS_HOME/TargetRTS/src/…) to a local directory.

- Make the changes that are required to the <function.cc> file then create an override makefile.
- From the toolset, configure the model to use the override makefile then compile the model.

5.10 Override Makefile - Example

This example shows an override makefile that will compile and link a custom mainLoop() function into a model.

```
LOCAL_DIRECTORY = $(COURSE_DIR)/Exercises/M9-Customizing
USER_OBJS = mainLoop$(OBJ_EXT)

mainLoop$(OBJ_EXT) : $(LOCAL_DIRECTORY)/mainLoop.cc
                     $(OTCOMPILE_CMD) \
                     $(USER_CC) $(CC_HEAD) $(RTUPDATE_CCFLAGS) \
                     $(RTUPDATE_INCPATHS) $(GENERATE_INCPATHS) \
                     /TP $(LOCAL_DIRECTORY)/mainLoop.cc $(CC_TAIL)
```

5.11 Testing the TargetRTS Port

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

For more information, reference the *Testing the TargetRTS Port* section in the *Adapting Rational Rose RealTime for Target Environments* porting guide.