

IBM Rational Software Architect RealTime Edition (RSARTE)

C++ RT Services Library

Version 1.5

Revision History

Date	Version	Description	Author
2012 May 07	1.0	Initial Release	Mattias Mohlin, Senior Software Architect
2014 May 14	1.1	Added definition of Run-to-Completion semantics, chapter about intra and inter-threaded communication, paragraph with technical information about RTS library classes	Elena Volkova, Software Engineer
2015 Jan 19	1.2	Updated for version 9.1.1	Mattias Mohlin
2015 May 14	1.3	Updated information about RTTimerId::isValid	Elena Volkova
2016 Jun 09	1.4	Updated for version 10 2016.23 Added RTActor::rtgStateEntry	Mattias Mohlin
2018 Jun 20	1.5	Updated for version 10.2 2018.24	Mattias Mohlin

Table of Contents

Introduction **5**

Target Configurations **5**

Services **7**

Communication Service 7
Message Delivery 9
Message Representation 10
Deferring and Recalling Messages 11
Non-wired Ports 12
Logging Service 13
Timing Service 13
Frame Service 15
Working with Optional Capsule Parts 16
Working with Plugin Capsule Parts 18
Accessing Model Information at Run-Time 19
Exception Service 20
External Port Service 22

Structure of Generated C++ Code **23**

Type Descriptors 23
Threads 24
Logical threads and physical threads 24

Inside the C++ RT Services Library **27**

Run-to-Completion Semantics 27
Intra-thread and Inter-thread Communication 27
Message queues 28
Message structure and freeList of messages 29
Intra-thread message sending 30
Inter-thread message sending 31
Message dispatch algorithm 32

C++ RT Services Library Class Reference **34**

RTActor 34
RTActorClass 36
RTActorRef 37
RTActorId 37
RTController 38
RTExceptionSignal 39
Frame 39
RTInSignal 44
Log 45
RTMessage 46
RTObject_class 48
RTOutSignal 49

RTProtocol....52
RTSymmetricSignal....56
RTTimerId....56
RTTimespec....57
RTTiming....58
RTTypedValue....61

This document describes the C++ RT services library, which is the run-time library used by real-time applications generated from Rational Software Architect RealTime Edition (RSARTE).

Readers of this document are assumed to have read the document "Modeling Real-Time Applications in RSARTE" which covers many of the concepts which are explained in more detail in this document.

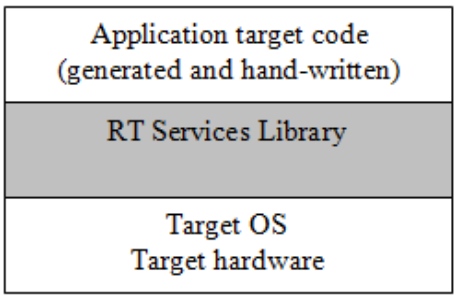
All screen shots were captured on the Windows platform.

Introduction

The RT services library is a run-time framework used by the target code that is generated from a UML real-time model. RSARTE can transform the real-time model to either C++ or C code, and hence there exists an implementation of the RT services library for each of these target languages. However, the services that are provided are the same regardless of implementation language. This document describes these services by using the C++ version of the run-time library.

The RT services library provides the run-time implementations of the UML real-time concepts that are supported by RSARTE. The implementation of some of these services requires functionality provided by the **target environment**. By target environment we mean the things that "surround" the real-time application, such as the operating system and the target hardware on which the real-time application will run.

The picture below shows the functional layering of a real-time application generated by RSARTE.



The RT services library isolates the application code from the target environment, so that the same real-time application can be built for multiple target environments. In addition to providing this platform independence, the RT services library also provides certain services which the application can use at run-time. The following categories of services are provided:

- Communication
- Timing
- Dynamic Structure
- Concurrency
- Message based processing

These services are described in the Services chapter.

Target Configurations

The RT services library should isolate the target code (at least the target code that is generated from the UML model) from all sorts of target environment differences. Since target environments exist in a large number of variations, there has to be also a large number of versions of the RT services library. Each such version of the RT

services library is called a **target configuration**, and is configured to work with a particular target environment. Here are some of the parameters that make up a target configuration:

- The operating system (name and version)
- The operating system threading configuration (single or multi-threaded)
- The processor architecture
- The target compiler (name and version)

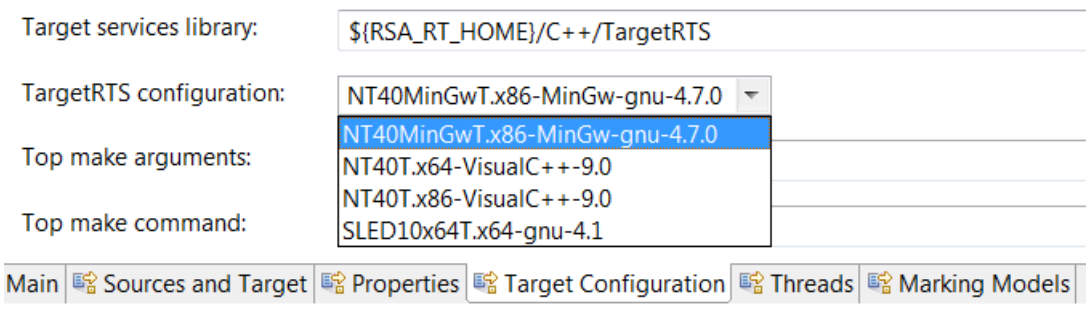
A specific target configuration of the RT services library provides a fixed value for each of these parameters. These values can be put together to form a string which uniquely identifies the target configuration in a compact way. Here is an example of such a string:

```
SLED10x64T.x64-gnu-4.1
```

target basename — libset name

The first part of the name is the **target basename** which identifies the operating system name, version and threading configuration. In the example above the operating system name is SLED (SUSE Linux Enterprise Edition) and the version is 10 (for 64 bit machines). The letter 'T' shows that the OS is multi-threaded (for single-threaded OS:es the letter 'S' is used instead).
The second part of the name, which follows after the dot, is the **libset name** which identifies the processor architecture and the target compiler. In the example the processor architecture is x64 and the compiler is the Free Software Foundation's GNU compiler version 4.1.

Target configuration strings are used throughout the C++ implementation of the RT services library. For example, you will see these strings if you look at the directories in `<InstallDir>/rsa_rt/C++/TargetRTS`. You also see these strings in the "TargetRTS configuration" drop down menu in the "Target Configuration" tab of the Transformation Configuration Editor:



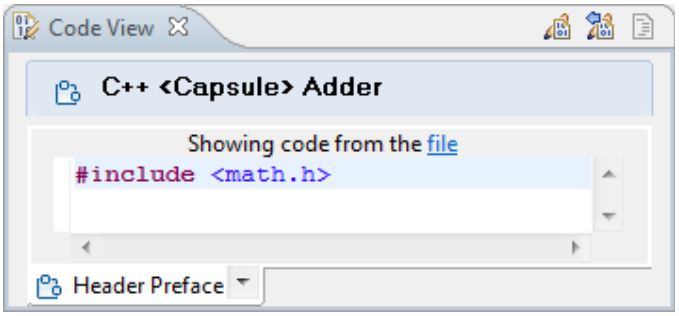
The target configuration strings shown in this drop down menu are dynamically extracted from the specified "Target services library" directory. Hence you can specify any directory that contains target configurations of the RT services library. For example, you can specify the `C++/TargetRTS` directory of the RSARTE installation to use one of the target configurations that are shipped with the RSARTE product.

You can also specify a directory where you have created your own custom version of the RT services library, adapted for a custom target configuration that is not shipped as part of the RSARTE installation. See the documentation about the Target RTS wizard to learn how you can build your own custom version of the RT services library using your platform and compiler of choice.

Services

In this chapter we will go through each of the run-time services which the RT services library provides. These services can be used by the action code that you include in your UML model, such as the code in the effect of a transition or the entry behavior code of a state. You can also access these services from any hand-written code that you include in your application.

Each run-time service has one or a few C++ header files which you must include from the C++ source file where you need to use the service. You must also ensure that the RT services library will be linked with when building the application. If the source file is generated from the UML model, required #includes are usually added automatically by the C++ code generator. However, if any #include is missing (either from the RT services library or from some other library that your code needs to use) you can simply select the context capsule and edit the "Header Preface" property using the Code View or the Code Editor:



The header preface code snippet will be copied verbatim at the top of the header file that is generated for the capsule.

Communication Service

The communication service provides the means by which capsule instances can communicate with each other using messages. Two kinds of communication are supported:

- **Asynchronous communication (send)**
  In this case the sender capsule instance is not blocked while the sent message is in transit. As soon as it has sent the message to a port, it can continue to execute. This type of communication is most commonly used since it provides a high throughput of messages, with minimal dependencies between sending and receiving capsules.
- **Synchronous communication (invoke)**
  In this case the sender capsule instance is blocked until the sent message has reached the receiver capsule instance, and that instance has replied on the

received message. Synchronous communication should only be used when the sender is unable to continue its execution until the sent event has been received (typically because the receiver must compute some data which the sender needs before it can proceed).
In some cases, the sender may not need any data from the receiver, but still want to suspend its execution until the receiver has processed the message. For those cases there exists a possibility to perform an invoke where the reply is implicitly made as soon as the message has been processed by the receiver.

There are three important rules to remember when using synchronous communication:

1. Unless the sender performs an invoke where the reply is implicitly made, the receiver **must** call a `reply()` function to reply to the sender. This has to be done while the receiver still is in the triggered transition (more precisely, it must be done before the receiver enters a new state). You can pass a return data with the reply message, which will be made available to the sender.
2. It is not allowed to perform invokes across different threads. If you need synchronous communication across threads you either have to implement it by means of two asynchronous events (call/reply) and a state (where the sender can wait until the receiver replies), or use a more low-level synchronization primitive such as a semaphore.
3. It is not allowed to perform invokes which lead to cycles. For example, if capsule A invokes capsule B which in turn tries to invoke capsule A, the invocation will fail with an error message at run-time.

Both synchronous and asynchronous communication is performed by calling functions on the port through which the message shall be sent. The port is transformed into a member variable in the C++ class that is generated from the capsule. The type of the port variable is a subclass of `RTProtocol`. This subclass is generated from the protocol that types the port, and contains one function for each event defined in the protocol. The function returns an `RTInSignal` if the event is an in-event, and an `RTOutSignal` if the event is an out-event. The `send()` and `invoke()` functions (for asynchronous and synchronous communication respectively) are found in the `RTOutSignal` class. If you want the receiver to make a reply for the invoke, you should call the version of `invoke()` which takes a reply buffer argument. Otherwise the reply will be implicitly made.

If the sender port is replicated (i.e. it has "many" multiplicity), the `send()` and `invoke()` functions will broadcast the event to all port instances, creating one message for each of them. In this case, if you instead want to send the event to a specific port instance, there are the functions `sendAt()` and `invokeAt()` which take the index of the port instance to use. Indexing is 0-based.

The send functions have an optional parameter "priority" which specifies the priority level at which the message shall be sent. The default value of this parameter is "General" which is the standard priority that is appropriate for most messages. Note

that the invoke functions do not have a priority parameter since events in that case are directly processed by the sender, without going through the event queue of the receiver. Hence, synchronously sent events can be said to have higher priority than any other event that is asynchronously sent.

Here are some examples of how to use the communication service for sending events:

```
mul.getIncrement(12).send();
```

Sends the event "getIncrement" out through the "mul" port. The event has an integer parameter and the sent message gets the value 12 for this parameter.

```
reqPort.abort().sendAt(2, High);
```

Sends the event "abort" out through the "reqPort" port. The event does not have any parameters. The port is replicated, and the message for the event will be sent through the port instance at index 2. The message is sent at priority level High.

```
RTMessage replies[ aPort.size() ];
aPort.ack().invoke(&replies);  // Hangs until all replies are available
for (int i = 0; i < aPort.size(); i++ ) {
    if (replies[i].isValid()) {
        // code to handle valid reply
        bool returnVal = (bool) replies[i].getData()
    }
    else {
        // code to handle invalid reply
    }
}
```

Synchronous sending of an event ”ack” to all port instances in the replicated port "aPort" (broadcast). Note that each receiver has to reply on the received message using code similar to

```
bPort.nack(false).reply();
```

Here the reply is made using the event "nack" which will be sent through the port "bPort". The boolean value false is passed as return data which the sender can access from the `RTMessage` object as shown above.

```
comPort.runTest().invoke();
```

Synchronous sending of an event ”runTest” through the port ”comPort”. The sender will be blocked until the receiver has processed the event, which it will do immediately, before processing other events in its event queue. The sender does not require any response value from the sender, so the reply is implicitly made as soon as the receiver has processed the event.

Message Delivery

The RT services library ensures that messages that are sent over the same

connection between a sender and a receiver are received in the same order as they were sent. This is true even if the receiver capsule instance runs in a different thread than the sender capsule instance. However, if the application is distributed so that the sender and receiver runs in different processes, this guarantee may no longer hold.

There are situations when the RT services library fails to deliver a sent message to the receiver. One such situation is when the message is sent to an unbound port (either wired or non-wired). Another situation that may occur in distributed applications is that there is a loss in the physical communication medium which causes messages to not reach their receiver.
When the RT services library is able to detect the failure to deliver a message, a run-time error will be issued. The functions for sending and invoking (and replying to) events will in this case return 0, which your code can test on. For example:

```
if (!thePort.start().send())
    log.log("Failed to send 'start' event");
```

It is good practice to always test for these error return values, especially when sending events through non-wired ports that are registered programmatically.

Another situation that may occur is that the message can reach the receiver, but when it is time to dispatch the message to the receiver no transition is found that can be triggered by the message. If this happens a virtual function `unexpectedMessage()` is called on the receiver capsule class. You can override this function to define what should happen in this case. The default implementation of the function in `RTActor` (from which all generated capsule classes inherit) is to print an error message to `stderr`.

Message Representation

The RT services library represents messages using objects of the `RTMessage` class. This class contains

- The parameter data carried by the message. The data is untyped (`void*`) and is accessed using the `getData()` function.
- The name of the event for the message, as defined in the protocol.
- The priority level at which the message was sent.

An `RTMessage` object is created by the RT services library and put in the message queue when the message is sent to the receiver capsule instance. It stays in the message queue until it becomes the first message in the queue, and then it will be dispatched (i.e. delivered) to the receiver capsule instance. You should normally treat the `RTMessage` object, and everything it contains, as read-only. This memory will be deleted by the RT services library when all code that runs when the transition is triggered is finished and control returns to the RT services library.

By default the RT services library passes all event parameter data by value in a message, which means that the data will be copied when a message is sent. This approach avoids problems with access of common data from different threads, but if

the data is big it may be too costly to copy it. In that case you may instead pass a pointer to the data as the event parameter. However, you then need to ensure that the data is sufficiently protected from simultaneous access from different threads (for example using a semaphore). You must also ensure that this data is deleted when it is no longer needed.

For synchronous communication ("invoke") the event parameter data is not copied. There is no need to do that since such communication may not take place across different threads.

The RT services library makes use of type descriptors to know how to copy an object of a user-defined type. If a type does not have a type descriptor, it cannot be copied and hence cannot be sent by value with messages. See Type Descriptors for more information about type descriptors.

The function that is generated for a transition provides direct access to the message parameter data by means of an `rtData` parameter. This parameter is typed with a pointer to the parameter data type, so it can be directly used without having to cast it. Here is an example of a transition function where the trigger event has a `double` as event parameter.

```
INLINE_METHODS void Adder_Actor::transition5_increment_computed( const
double * rtdata, INC_REQ::Base * rtport )
{
//{{{USR
double inc = *rtdata;
result += inc;
//}}}USR
}
```

If you need to access the message parameter data outside of the transition function, for example in a function generated from a capsule operation, you can use the `getMsg()` function or the `msg` member variable of the capsule class (an instance of `RTActor`) to access the `RTMessage` object for the currently processed message.

Deferring and Recalling Messages

It is possible to defer the handling of a received message, in order to handle it at a later point in time instead. To do so call the `defer()` function on the `RTMessage` object for the received message.

Deferring a message puts it in a defer queue. This queue is manipulated by using the functions provided by the `RTInSignal` class. For example, it has a function `recall()` which can be used to recall a previously deferred message. When a message is recalled it is moved from the defer queue back into the message queue, so that it can later be dispatched to the receiver capsule instance again.

Note that a deferred message will stay in the defer queue until it is recalled. You must ensure that you don't forget a message in the defer queue.

Here are some examples of deferring and recalling messages:

```
msg->defer();
```

Defers the current message (i.e. the message which was most recently dispatched to the capsule instance).

```
thePort.theEvent().recall();
```

Recalls the first deferred message of the event "theEvent" on the port "thePort". It will be moved to the back of the message queue.

```
thePort.theEvent().recallAll(1);
```

Recalls all deferred messages of the event "theEvent" on the port "thePort". The recalled messages will be moved to the front of the message queue, which means they will be the next messages that get dispatched to the capsule instance.

```
thePort.theEvent().purge();
```

Deletes all deferred message of the event "theEvent" on the port "thePort". If "thePort" is replicated the function operates on all port instances.

If you need to defer a reply message for an invoked event, so that the reply can be handled in a different transition, then you cannot use the `defer()` function since the `RTMessage` object for the reply message usually is located in the same function where `invoke()` is called. What you can do instead is to make a copy of the reply message and send it to the same capsule instance again. Thereby you can defer the handling of the reply message so that it does not have to occur immediately after the invoked event has been processed by the receiver. For example:

```
RTMessage reply;
myport.myevent().invoke(&reply);
sendCopyToMe(&reply);
```

Of course, the `sendCopyToMe()` function can be used for any kind of event, and it's useful whenever you cannot fully handle an event in a single transition.

Non-wired Ports

Ports that are defined as wired are automatically connected by the RT services library when the containing capsule instance is initialized. The same is true for non-wired ports which have the "Registration Kind" property set to "Automatic" or "Automatic (Locked)". However, if this property is set to "Application" you have to programmatically register such ports as either SPP or SAP ports. The RT services library will establish the connections automatically when the ports have been registered.

The functions to use for establishing the connections are found in the `RTProtocol` class. Here is an example:

```
p1.registerSAP("aPort");
```

Registers the port "p1" as an SAP port under the name "aPort".

Logging Service

The RT services library provides a basic logging service by means of the Log protocol. To use this service you have to add a port to the capsule from which you want to print the log messages. The type of this port should be set to the Log protocol.

Ports typed by the Log protocol will in C++ become member variables typed by the `Log::Base` class. This class provides several functions for writing strings and other data types to the `stderr`. Here are some examples:

```
log.log("Hello");
```

Prints the string "Hello" followed by a carriage return to the log.

```
log.show(x);
```

Prints the contents of the variable `x` to the log without a carriage return.

The `Log::Base` class provides several overloaded versions of `log()` and `show()` to handle all the primitive C++ types. However, if you want to print an object of a user-defined type, such as a UML class, you need to provide the type descriptor of this user-defined type as a second argument. The encode function specified for the type descriptor will be invoked in order to obtain a string encoding of the object which can be printed. For example:

```
log.log(&myClass, &RTType_MyClass);
```

For more information about type descriptors see Type Descriptors.

Timing Service

The timing service is provided by means of a Timing protocol. A port that is typed by this protocol acts as a timer which will send a timeout event either at a particular point in time (absolute, or relative from now) or at periodic intervals.

The C++ class of the RT services library that implements the Timing protocol is `Timing::Base` and it contains functions for setting the timer to timeout in various ways, to cancel it etc. A timer that is set-up to timeout only once is called a **one-shot timer**. It can be set to timeout either after a specified time duration (a relative time) or at a specified point in time (an absolute time). A timer can also be set-up to timeout repeatedly at periodic intervals, a so called **periodic timer**. Although it is possible to implement such a timer by using a one-shot timer that is repeatedly reset, a periodic timer is more accurate. This is because it always takes some time to process a timeout and to reset the timer. Also, you have to take into account timer round-off errors which may either reduce or add-to the drift in timing.

Time values are represented by the `RTTimespec` class. This class also has a static function `getclock()` which populates an `RTTimespec` object according to the current time value. You can compare different `RTTimespec` objects with the usual C++ comparison operators such as '<', '>=', '==' etc. You can also perform simple time arithmetics using the '+' and '-' operators.

Here are some examples of using the timing service ("timer" is a port typed by the Timing protocol):

```
timer.informIn(RTTimespec(5, 0));
```

Sets a one-shot timer to expire in 5 seconds from now (relative time).

```
RTTimespec now;
RTTimespec::getclock(now);
timer.informAt(now + RTTimespec(5, 0));
```

Sets a one-shot timer to expire in 5 seconds from now (absolute time).

```
timer.informEvery(RTTimespec(5, 0));
```

Sets a periodic timer to expire every 5 seconds. The first timeout will occur in 5 seconds from now (relative time).

Just like other events, the timeout event that is sent when a timer expires, can have parameter data. It can also be sent with a non-default priority. Here are some examples:

```
bool b = true;
timer.informIn(RTTimespec(5, 0), &b, &RTType_bool);
```

Sets a one-shot timer to expire in 5 seconds from now (relative time). The timeout event will have a boolean parameter and the timeout message that is produced on timeout will have the value `true` for this parameter. Note that the UML definition of the timeout event does not have any parameter. It is only at the time of setting a timer that we can specify whether it should have a parameter or not. If we want it to have a parameter we therefore also have to specify the type of this parameter. This is why we have to pass the type descriptor for the `bool` type as the last argument in the call to `informIn()`.

```
bool b = true;
timer.informIn(RTTimespec(5, 0), RTTypedValue(&b, &RTType_bool), High);
```

As above, but uses an RTTypedValue object to specify the parameter data and type. The timeout event will be sent at the High priority level.

All functions that set a timer return an `RTTimerNode` object, from which you can construct an `RTTimerId` object. You need to store the `RTTimerId` object if you want to cancel the timer. Cancelling a timer ensures that it will not produce a timeout message. This is true even if the timer already has expired at the time when it is

cancelled. In this case the timeout message exists in the message queue and is waiting to be dispatched to the capsule instance from there. When the timer is cancelled the timeout message is removed from the message queue. Here is an example:

```
RTTimerId tid;
if (!(tid = timer.informIn(RTTimespec (10, 0))))
  log.log("error when setting a timer");
else
  timer.cancelTimer(tid);
```

Sets a one-shot timer to expire in 10 seconds from now. The timer is then immediately cancelled using the obtained RTTimerId object. Note that you may obtain a `null` value instead of an RTTimerId object in case setting the timer fails.

It is common that the timer is cancelled in another transition, typically because a certain message arrived before the timer expired. In this case make sure that you store the `RTTimerId` object in an attribute of the capsule, and not in a transition local variable. Also note that an `RTTimerId` object only is valid until one of the following happens

1. the timeout message for the timer is dispatched (does not apply for periodic timers), or
2. the timer is cancelled

When in doubt you may use the `isValid()` function on an `RTTimerId` object to check whether the timer id is valid or not.

Note that the RT services library maintains pointers to `RTTimerId` objects for timers that have been set. This means you must ensure that the addresses of `RTTimerId` objects do not change, as otherwise such pointers become invalid. For example, it is not safe to store `RTTimerId` objects in containers which do not guarantee that pointers to contained objects remain valid when adding or removing objects from the container (an example of such an "unsafe" container is `std::vector`). If you need to use such containers to keep track of multiple timers, you should only store pointers to `RTTimerId` objects rather than the objects themselves.

When a capsule instance is destroyed all its active timers are automatically cancelled.

Frame Service

The frame service is provided by means of a protocol Frame which can be used to type ports in your model. The service allows you to work with capsule instances dynamically in various ways, for example

- adding or removing capsule instances to/from optional and plugin capsule parts
- accessing capsule instances from capsule parts
- obtaining information about the run-time representation of the UML model, such as its structure and properties of certain run-time objects

The Frame protocol does not have any events, so all functionality is exposed in the form of functions on the C++ class of the RT services library that implements the Frame protocol. This class is `Frame::Base`.

Working with Optional Capsule Parts

The frame service allows you to incarnate new capsule instances into an optional capsule part by calling an `incarnate()` function. The capsule part is identified by means of an `RTActorRef` object. You obtain this object from the member variable that corresponds to the capsule part.

By default the type of the capsule part is used to determine which capsule to instantiate. However, you can also specify that another capsule should be instantiated, as long as that capsule is compatible with the capsule that is the type of the capsule part.

A capsule cannot have a user-defined constructor. Instead you can pass initialization data in the call to `incarnate()`. This data can be accessed by the incarnated capsule instance in its initial transition by means of the parameter called `rtdata`.

By default the incarnated capsule instance starts to run in the same logical thread as the caller. However, at the time of incarnation you can specify that the new capsule instance shall run in another logical thread. You can refer to available logical threads using the names specified in the Threads tab of the transformation configuration editor.

If the capsule part is replicated (i.e. the upper bound of its multiplicity is greater than 1) you may also want to specify the index where to insert the created capsule instance. Indexing starts at 0. If you specify -1 as index the RT services library will insert the new capsule instance at the first slot that is available. This is the last slot that was made available after a capsule instance in the capsule part was destroyed. When all such slots have been filled up, the next available slot is the one that has the lowest index number. For example, assume that you first incarnate a capsule into indexes 0,1,2,3 in a replicated capsule part, and then destroy the capsule instances in the same order (0,1,2,3). Then if you start to incarnate capsules using index -1 the capsule instances will be inserted in the following index order: 3,2,1,0,4,5,6,...

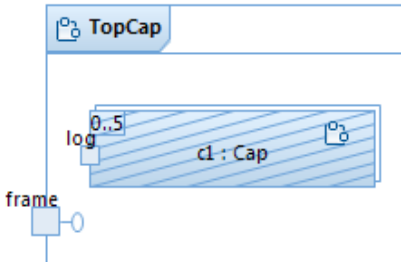
The `incarnate()` functions return an `RTActorId` object which is a handle to the instantiated capsule instance. There are many things that may go wrong when incarnating a capsule so you should always check the validity of this handle by calling the `isValid()` function. Here are some examples of reasons why an incarnation may fail and the `RTActorId` object to become invalid:

- You specified a bad index (for example an index that was higher than the upper bound of the multiplicity of the capsule part).
- All indices of a replicated capsule part were already occupied by a capsule instance, so there was no room for another one to be inserted.
- The incarnated capsule was not type compatible with the type of the capsule part.
- The `RTActorRef` object that was used did not correspond to an optional

capsule part owned by the incarnating capsule.
- There was not enough memory available to allocate a new capsule instance.

If incarnation fails you can use the `getError()` function (defined in `RTActor`) to obtain information about why it failed.

Let's look at an example of using the Frame service for incarnating a capsule.



The capsule "TopCap" has a capsule part "c1" with multiplicity 0..5 and typed by another capsule "Cap". Here are some examples of code that "TopCap" may execute to incarnate the "Cap" capsule into "c1":

```
RTActorId id = frame.incarnate(c1);
if (!id.isValid()) {
    RTController::Error error = getError();
    context()->perror(context()->strerror());
}
```

Incarnate "Cap" into "c1" at the first available index. If incarnation fails obtain the error code and print the error message.

```
int data = 14;
RTActorId id = frame.incarnate(c1, EmptyActorClass, &data);
```

Incarnate "Cap" into "c1" at the first available index. An integer value (14) is passed as initialization data which the "Cap" instance can obtain in its initial transition.

```
RTActorId id = frame.incarnate(c1, EmptyActorClass, (const void*) 0,
                               (const RTObject_class*) 0, MyThread, 3);
```

Incarnate "Cap" into "c1" at index 3. No initialization data is passed. The instantiated capsule instance will run in the logical thread "MyThread".

The function to use for destroying a capsule instance is called `destroy()`. There exists two overloads of this function, one that takes a capsule instance (i.e. an `RTActorId` object) and one that takes a capsule part (i.e. an `RTActorRef` object). The latter version of the function destroys all capsule instances that exist in the capsule part.

Here is an example:

```
RTActorId id = frame.incarnationAt(c1, 3);
if (id.isValid()) {
    if (!frame.destroy(id)) {
        context()->perror(context()->strerror());
    }
}
```

Obtains the capsule instance at index 3 in the capsule part "c1". If such a capsule instance exists and is valid, it will be destroyed.

Working with Plugin Capsule Parts

Optional capsule parts let you wait until run-time to decide what capsule instances they should contain. However, once you have incarnated a capsule instance into an optional capsule part, it will remain in that location for its entire lifetime. Sometimes you may need the flexibility for a capsule instance to move between different capsule parts, or even belong to more than one capsule part at the same time. This can be accomplished by means of plugin capsule parts.

The frame service allows you to import an existing capsule instance into a plugin capsule part by calling an `import()` function. The capsule part is identified by means of an `RTActorRef` object. You obtain this object from the member variable that corresponds to the capsule part.

There are some rules when importing a capsule instance into a plugin capsule part:

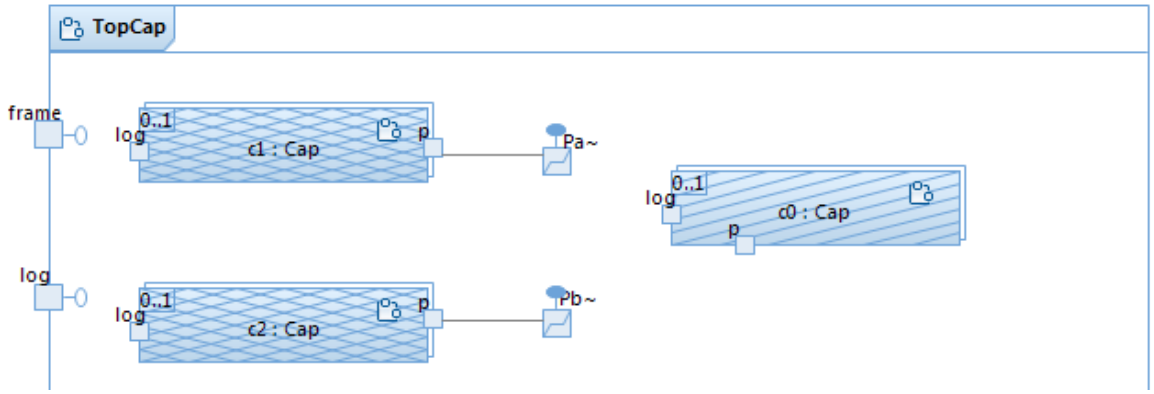
- The capsule instance must be valid. Use the `isValid()` function to check this.
- The capsule instance must be type compatible with the type of the capsule part.
- The capsule instance must not have a port that is already bound.

Violations of these rules will cause the `import()` function to fail with an error code. You can use the `getError()` function (defined in `RTActor`) to obtain information about why it failed.

If the capsule part is replicated (i.e. the upper bound of it's multiplicity is greater than 1) you need to specify the index where to insert the imported capsule instance. Indexing starts at 0.

To remove a capsule instance from a plugin capsule part call the `deport()` function.

Let's look at an example of using the Frame service for importing and deporting a capsule instance to and from a plugin capsule part.



The capsule "TopCap" has an optional capsule part "c0" and two plugin capsule parts "c1" and "c2". All capsule parts are typed by a capsule "Cap". The "TopCap" capsule may contain the following code:

```
RTActorId id = frame.incarnate(c0);
if (!frame.import(id, c1)) {
    context()->perror("Failed to import into c1!");
}
if (!frame.deport(id, c1)) {
    context()->perror("Failed to deport from c1!");
}
if (!frame.import(id, c2)) {
    context()->perror("Failed to import into c2!");
}
```

Incarnate a "Cap" capsule instance into "c0", and then import it into "c1". After that the port 'p' becomes bound to the port 'Pa'. The capsule instance can therefore not immediately be imported also to "c2", but must first be deported from "c1".

Accessing Model Information at Run-Time

The UML model that describes your real-time application contains static design-time information which sometimes is useful to access at run-time. For example, you may want to access the name of a capsule to be able to write a generic logging function. However, some parts of the UML model also has a run-time representation, which contains dynamic information. For example, a capsule part has a run-time representation by means of an `RTActorRef` object, and you may want to access dynamic information such as its current replication factor (i.e. the number of capsule instances currently stored in the capsule part).

The frame service provides several functions that let you access both static (design-time) and dynamic (run-time) model information.

The function `incarnationAt()` lets you access a capsule instance from a capsule part at a particular index. Just like `incarnate()` an `RTActorId` object is returned and you should use its `isValid()` function to determine if there actually was a valid capsule instance at the specified index. There is also a function `incarnationsOf()` which returns all capsule instances that currently exist in a capsule part. This is

useful if you want to iterate over all capsule instances.

The function `classOf()` takes a capsule instance (`RTActorId`) and returns the type descriptor for the capsule (`RTActorClass`). That is, this function obtains the dynamic type of a capsule instance. This type is not necessarily the same as the static type of the capsule part in which the capsule instance is located, but the dynamic type should at least be compatible with the static type. The `className()` function returns the capsule name from its type descriptor, and `classIsKindOf()` can be used to check if a capsule is the same as or a subclass of another capsule. There are also two useful functions `me()` and `myClass()` which return the capsule instance and capsule which the running thread executes at the moment.

Here is an example of using some of these functions:

```
RTActorId id = frame.incarnationAt(c1, 3);
const RTActorClass& cls = frame.classOf(id);
log.log(frame.className(cls));
const RTActorClass& myCls = frame.myClass();
if (frame.classIsKindOf(myCls, cls))
    log.log("Compatible");
else
    log.log("Not compatible");
```

Get a capsule instance from capsule part "c1" at index 3 and obtain the type descriptor for this capsule instance. Print the capsule name of this instance to the log. Then obtain the type descriptor for the running capsule and check whether this capsule is the same or a subclass of the capsule that is the type of the capsule instance.

Exception Service

The exception service is provided by means of a protocol Exception which can be used to type ports in your model. The service allows you to send exception events to these ports as a means to indicate that an exceptional situation has occurred. Typically, this would be an error situation which the code that raises the exception cannot handle itself. An exception message is handled by the RT services library just like any other message, i.e. it will be placed in the message queue of the receiving capsule instance. When the exception message gets dispatched a transition may get triggered which can provide the actions that are necessary to handle the exceptional situation.

The C++ class of the RT services library that implements the Exception protocol is `Exception::Base` and it contains functions for raising a number of predefined exceptions that are intended to cover common error situations. The functions return an `RTExceptionSignal` object which provides a `raise()` function for sending the exception event to the exception port.

Here is an example of code for raising an exception ("exPort" is a port typed by the Exception protocol):

```
exPort.userError(RTString("An error occurred.")).raise();
```

Raise the exception "userError" on the port "exPort". The exception message will have a string as parameter data.

Each exception is represented by a specific in-event in the Exception protocol. The UML definitions of these in-events do not specify any event parameters, but you may actually pass any data with these events using an `RTTypedValue` object. In the example above we used this possibility to pass a string with the "userError" exception event. Passing parameter data with an exception event is optional as shown in the example below:

```
exPort.error().raise();
```

Raise the exception "error" on the port "exPort". The exception message does not have any parameter data.

The following exceptions can be raised:

- arithmeticError
- error
- notFoundError
- notUnderstoodError
- serviceAccessError
- streamError
- subclassResponsibilityError
- timeSliceError
- userError

Note that the RT services library does not itself raise any exceptions. Your application is hence responsible for raising the exceptions that needs to be handled. This also means that there is no precise definition of when to use a particular kind of exception. Your application has to define the conditions for when a certain exception is raised. The general "error" exception can be used for errors that are not well described by the names of the other kinds of exceptions.

If there is no transition available to be triggered by a dispatched exception message, this is treated in the same way as for other messages (see Message Delivery).

Of course, the use of exceptions in an application is optional. You can always decide to handle errors in a different way that does not involve usage of exceptions. For example, you may define your own protocols for error handling, or you may handle errors using some other mechanism. However, the use of C++ exceptions (`throw / catch`) is often not appropriate, at least not for code that is invoked by the RT services library, since it in general will not be possible to catch the exceptions that are thrown. If you do use C++ exceptions make sure to catch them before control returns to the RT services library.

External Port Service

The external port service is provided by means of a protocol External which can be used to type ports in your model. The service allows you to send events to these ports from threads that are external to the RT services library and the code that is generated from the UML model.

The C++ class of the RT services library that implements the External protocol is `External::Base` and it contains functions that enable or disable the port to receive these external events. These functions must be called from the thread on which the capsule instance runs, i.e. an external event can only be sent to a capsule instance when it is ready to receive such an event.
Here are examples of code which is run by the capsule instance to enable and disable the reception of an event on an external port "extPort":

```
extPort.enable();
```

Enable reception of an event on the external port. This puts the port in a mode where it is ready to receive exactly one external event.

```
extPort.disable();
```

Disable the reception of an event on the external port. This puts the port in a mode where it cannot receive any external events.

When the external port receives an event, it automatically becomes disabled. It has to be enabled again to be able to receive another event.

Here is an example of code which may only execute in the external thread, and which will send an event to the external port "extPort":

```
if (extPort.raise() == 0){
    //fail
}
else {
    //pass
}
```

Sends an event to the external port. The raise() function returns zero if the event failed to be sent. The typical reason for failing is that the external port is not currently enabled to receive an external event.

External ports is a useful mechanism for integrating code generated by RSARTE with external code that also is part of the real-time application. For example, you may have a part of your application that is responsible for reacting on external stimuli, such as sensors or other kinds of notifications from the physical world. Such code may run in an external thread, and interact with the code generated from the UML model by means of external ports.

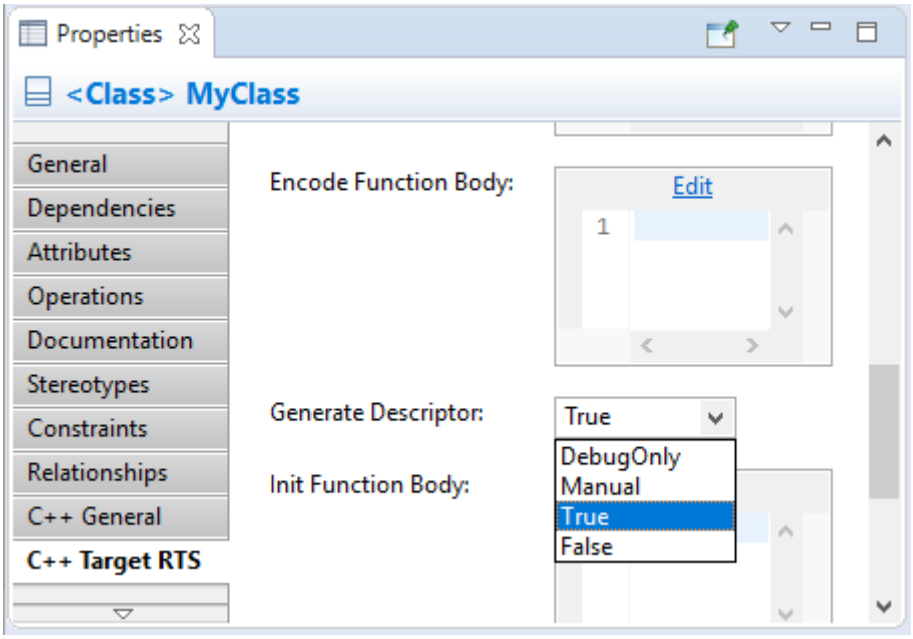
Structure of Generated C++ Code

This chapter provides the information you need to know about the structure of the C++ code that is generated from an RSARTE model. Generated C++ code makes extensive use of classes provided by the RT services library.

Type Descriptors

A **type descriptor** is meta data that is generated for each user-defined type in the model, such as a UML class. The RT services library uses information in a type descriptor to know how to initialize, copy, destroy, encode and decode objects of the corresponding type.

You can customize the generation of a type descriptor for a type by means of the "Generate Descriptor" property. This property is available in the “C++ Target RTS” tab in the Properties View.



If you set the property to "False" no type descriptor will be generated for the type. This means that you for example cannot send an object of this type by value with an event (since the RT services library does not know how it shall be copied). Another thing that is lost is the ability to print an object of this type to the log using the Log service (since the RT services library does not know how it shall be encoded).
You can set the property to "Manual" if you want to write a custom type descriptor for the type. The file that contains your type descriptor must be linked with the application or a link error will arise.

For all details about the information that is stored in a type descriptor (and which you have to provide if you write your own) see the `RTObject_class` class.

Each type descriptor is instantiated exactly once in the generated application, and are constants. Here is an example of a type descriptor constant that is generated for a class "MyClass":

```
extern const RTObject_class RTType_MyClass;
```

Whenever you need to pass a type descriptor to a function in the RT services library you shall use the address of the constant that is generated for the type descriptor.

The RT services library contains predefined type descriptors for all primitive types. The convention is that for a type `T` the type descriptor is called `RTType_T`. For example, the type descriptor `RTType_double` is the type descriptor for the `double` type.

Some functions in the RT services library take as arguments both a data value, and the type of the data. The data value is then often expressed as an untyped pointer to the data (`void*`) while the type of data is expressed using a type descriptor (`RTObject_class*`). To be able to encapsule both the data value and the type descriptor in a single object the RT services library provides a class `RTTypedValue`.

For capsules a special kind of type descriptor is generated using objects of the `RTActorClass` class. This class for example stores the name of the capsule, a reference to the type descriptor for its super capsule and information about the ports of the capsule. This information is used by various functions in the RT services library that let you access model information at run-time. See Accessing Model Information at Run-Time.

Threads

Logical threads and physical threads

Each capsule instance has its own logical thread of control. This means that conceptually it runs in its own thread, independently of other capsule instances. However, the transformation configuration editor allows you to create multiple logical threads (in the Threads tab) and each of them can run multiple capsule instances.

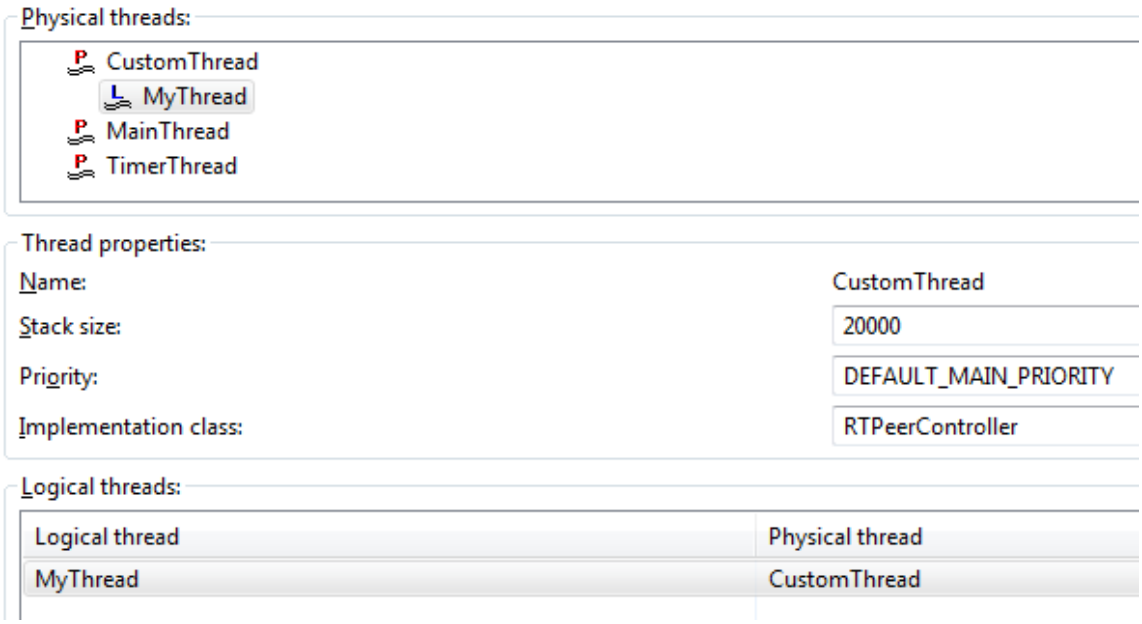
Logical threads are mapped to real physical threads in the Threads tab of the transformation configuration editor. This mapping allows you to control the total number of physical threads in the application, and how many capsule instances that each physical thread is controlling.

In the generated code each logical thread is available as a variable typed by the `RTController` class, and named according to what is specified in the Threads tab. This variable is assigned to a corresponding physical thread in generated C++ code. This means that when you refer to a logical thread in your application code, you actually have access to the corresponding physical thread represented by an `RTController` object.

The function that maps logical threads to physical threads is called `_rtg_mapLogicalThreads()` and is generated into the Unit C++ file (by default called `UnitName.cpp`). Next to this function you will also find the generated functions `_rtg_createThreads()` and `rtg_deleteThreads()` which contain the code for creating and deleting the physical threads that you have added in addition to the

default MainThread and TimerThread.

Here is an example of using the Threads tab in the transformation configuration editor to add one additional physical thread "CustomThread" and a logical thread "MyThread":



When you incarnate an optional capsule part you can specify that the new capsule instance shall run in one of the logical threads you have specified in the Threads tab. The top capsule instance always executes in the MainThread, and every capsule instance contained in a fixed capsule part always executes in the same thread as the owner capsule instance. See Working with Optional Capsule Parts for an example of incarnating a capsule instance that is run by a custom logical thread.

The mapping you choose between logical threads and physical threads can have a significant impact on the application performance. Capsule instances that may perform long-running tasks are beneficial to run in separate physical threads, because while a capsule instance performs a task, all the other logical threads that are mapped to the same physical thread have to wait until the running capsule instance returns control to the RT services library (see Run-to-Completion Semantics). But at the same time there is a practical limit on how many physical threads the target environment can support, and you must ensure that the application stays within those limits.

Another input to your choice of how to map logical threads to physical threads is how the capsule instances communicate data with each other. Capsule instances that run in the same physical thread do not need to worry about thread synchronization of data that is shared between them. However, if you use shared data between capsule instances run by different physical threads you have to ensure that all access to such data is done in a thread-safe manner. Read more about this in Intra-thread and Inter-thread Communication.

Some target environments only support one thread. In this case the macro `USE_THREADS` will be unset when compiling generated C++ code. This will remove all code that deals with multiple threads, for example the functions `_rtg_createThreads()` and `rtg_deleteThreads()`.

To access the `RTController` object that represents the executing thread, you may call the function `context()` that is available in the `RTActor` class. The `RTController` class provides several useful functions, for example functions for accessing the most recent error that has taken place in the thread. Here is an example that involves usage of some of the `RTController` functions:

```
RTController* c = context();
if (c->getError() != RTController::ok) {
    c->perror("An error occurred on thread ");
    log.log(c->name());
    c->abort();
}
```

Check if there is a recent error in the context thread. If so it is printed, followed by the name of the context physical thread (as specified in the Threads tab of the transformation configuration editor). Finally, terminate the context thread (this will destroy all capsule instances run by that thread).

Inside the C++ RT Services Library

Run-to-Completion Semantics

The RT Services Library does not preempt capsule processing. The heart of the RT Services Library is a controller object that dispatches messages to capsules. Its basic mode of operation is to take the next message from its 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 RT Services Library until the capsule's transition has completed processing the message. Each capsule instance processes only one message at a time. It processes the current message to the completion of the transition chain (which can consist of several code snippets, such as a guard condition, an exit action, a transition effect, a choice point condition, and an entry action) and then returns control to the RT Services Library to wait for the next message to be dispatched. This is referred to as run-to-completion semantics. Typically, the code snippets involved in a transition chain should be short and complete quickly, to result in rapid handling of messages.

Intra-thread and Inter-thread Communication

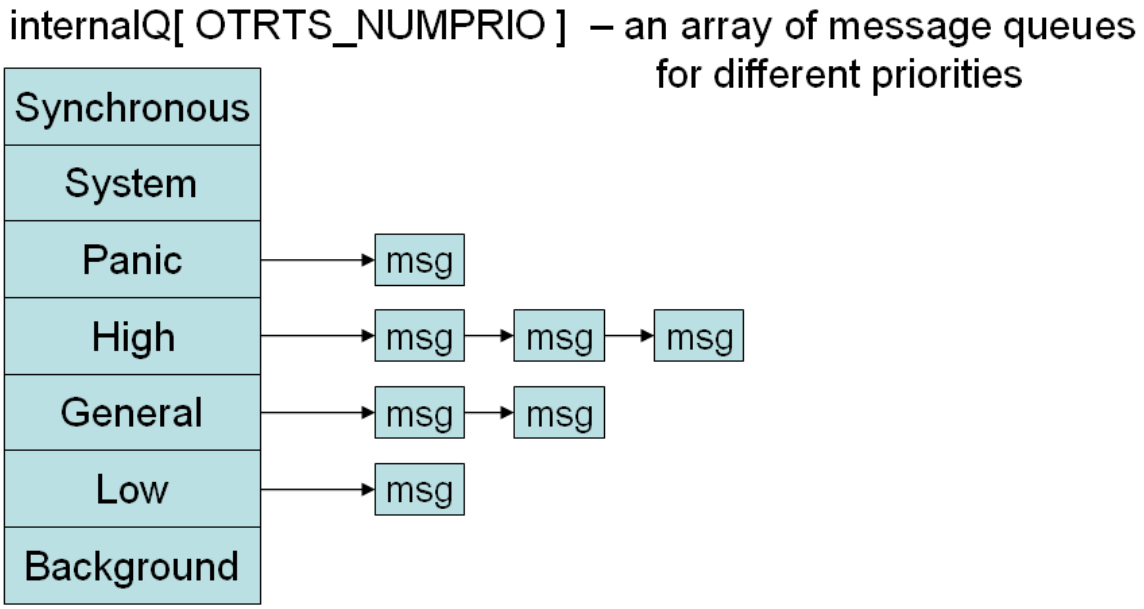
It’s important to understand the intra-thread and inter-thread communication mechanisms and how messages are handled using the RT Services Library.

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, and message sending across threads is approximately 10-20 times slower. Optimal designs therefore place capsule instances that have intense message communication with each other on the same thread.

The RT Services Library implements 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. Then repeat, once the recipient capsule completes processing the message.
A message is delivered via a call to `rtsBehavior(signal, port)`. This call is made inside the message dispatch loop. Run-to-completion semantics is enforced as the control is not returned to the loop until the capsule completes execution and makes a “return”.

Message queues

Each physical thread has got its own `RTController` object with its own message queues. There are two kind of message queues: `internalQ` and `incomingQ`.
`InternalQ` holds messages that are ready to be dispatched by its controller. Messages going between capsules incarnated on the same physical thread (the same `RTController` object) will be placed into `internalQ`. Below is an illustration of what this queue may look like at a particular point in time, when 7 messages are ready to be dispatched.



As can be seen in the picture the queue is categorized according to priority level to ensure that messages with higher priority get dispatched before messages with lower priority.

`IncomingQ` is used for communication between threads. It holds messages coming from other physical threads (other `RTController` objects). Messages from `incomingQ` will first be moved to `internalQ` at an appropriate time in the controller loop and then dispatched by the controller in the same way as other internal messages. `IncomingQ` is organized in the same way as `internalQ`:

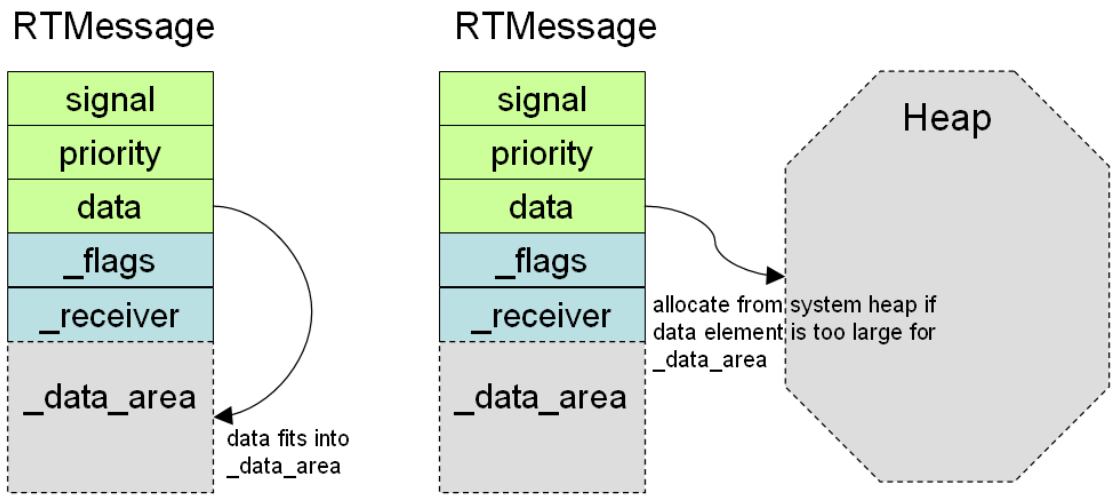
```
RTMessageQ   internalQ[ OTRTS_NUMPRIO ];

RTMessageQ   incomingQ[ OTRTS_NUMPRIO ];
```

Message structure and freeList of messages

Each message in the message queue is an object of the class `RTMessage` holding signal id (i.e. the id of the protocol event), priority, data pointer, information about port and receiver, flags, and data area of size `RTMESSAGE_PAYLOAD_SIZE`.

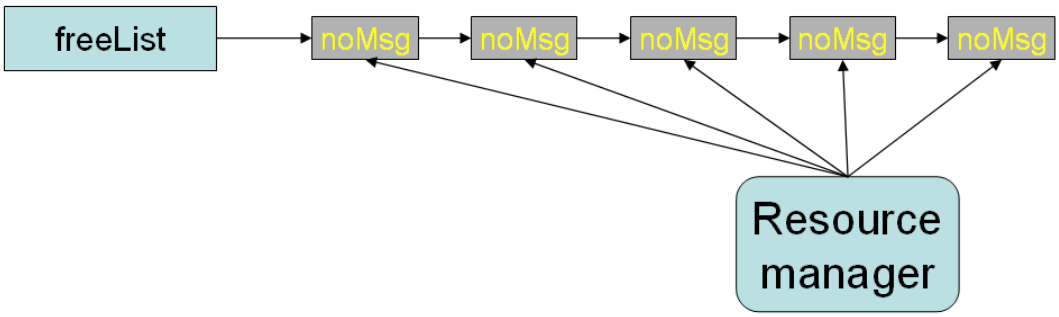
If data to be sent fits into the message body data area, it will simply be copied into the message. If the data does not fit inside the payload area, memory for the data will be allocated in the system heap and freed after the message has been received. The size of the data area (`RTMESSAGE_PAYLOAD_SIZE`) is configurable and the default size is 100 bytes.



Each `RTController` object has a pool of allocated `RTMessage` objects that are free of data and that are used when a new message should be sent. This pool is called the ”free list”:

```
RTMessage * freeList
```

A resource manager object (`RTResourceMgr`) allocates the initial `freeList` queue of message objects at startup. The resource manager will allocate extra message objects when a low threshold is passed and return these extra message objects to the message pool as a high threshold is over.



When a thread needs to send a message, the resource manager takes the next free message object in the `freeList` of the corresponding `RTController` object and returns it to the controller that will fill in all message information and send the message. If there is no free message in the list, a new bunch of messages are allocated in the system heap with the call to the function

```
unsigned msgAlloc( RTMessage * &, unsigned howMany = 50U )
```

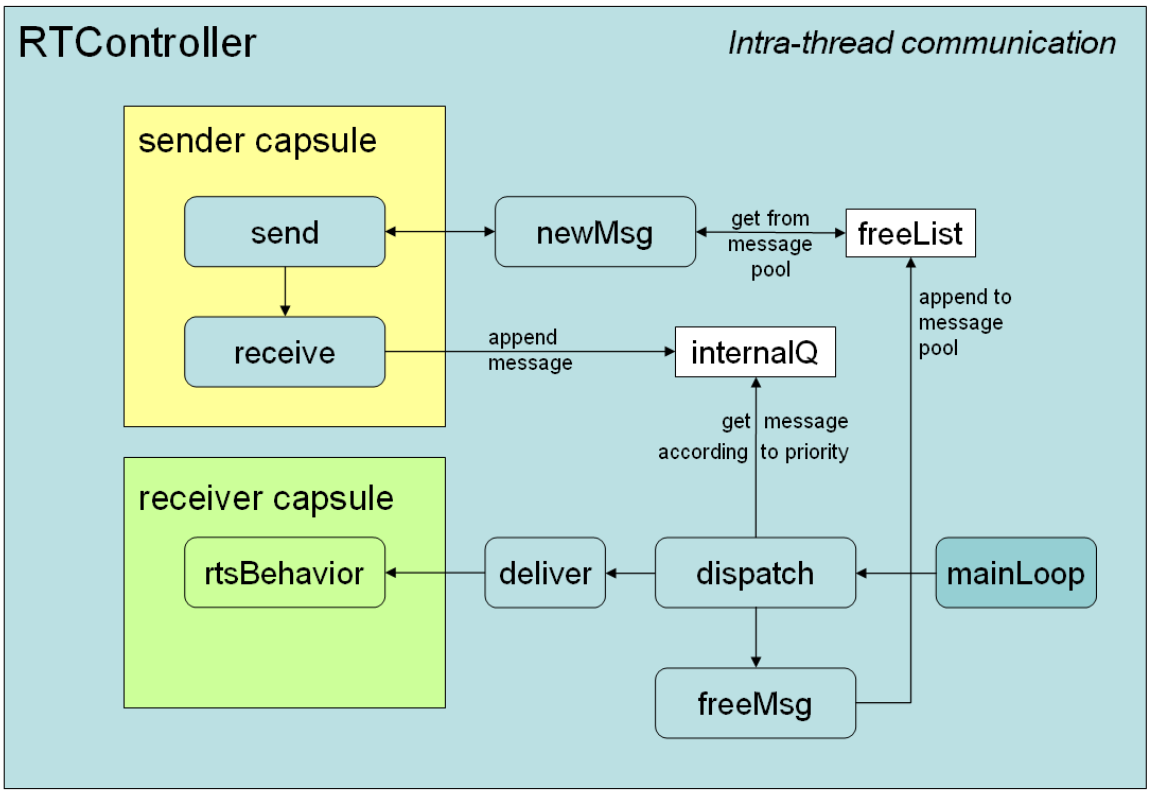
By default, 50 new messages are added. Note that once allocated, message objects are not freed back to the system heap but returned to the message pool.

Message objects are released from the message pool in `RTController::freeMsg` when it reaches `maxFreeListSize` (100 by default), and only `minFreeListSize` (20 by default) messages are left in the pool.

Free queue thresholds and data buffer size are statically configurable values and it is necessary to rebuild the RT services library if these are changed.

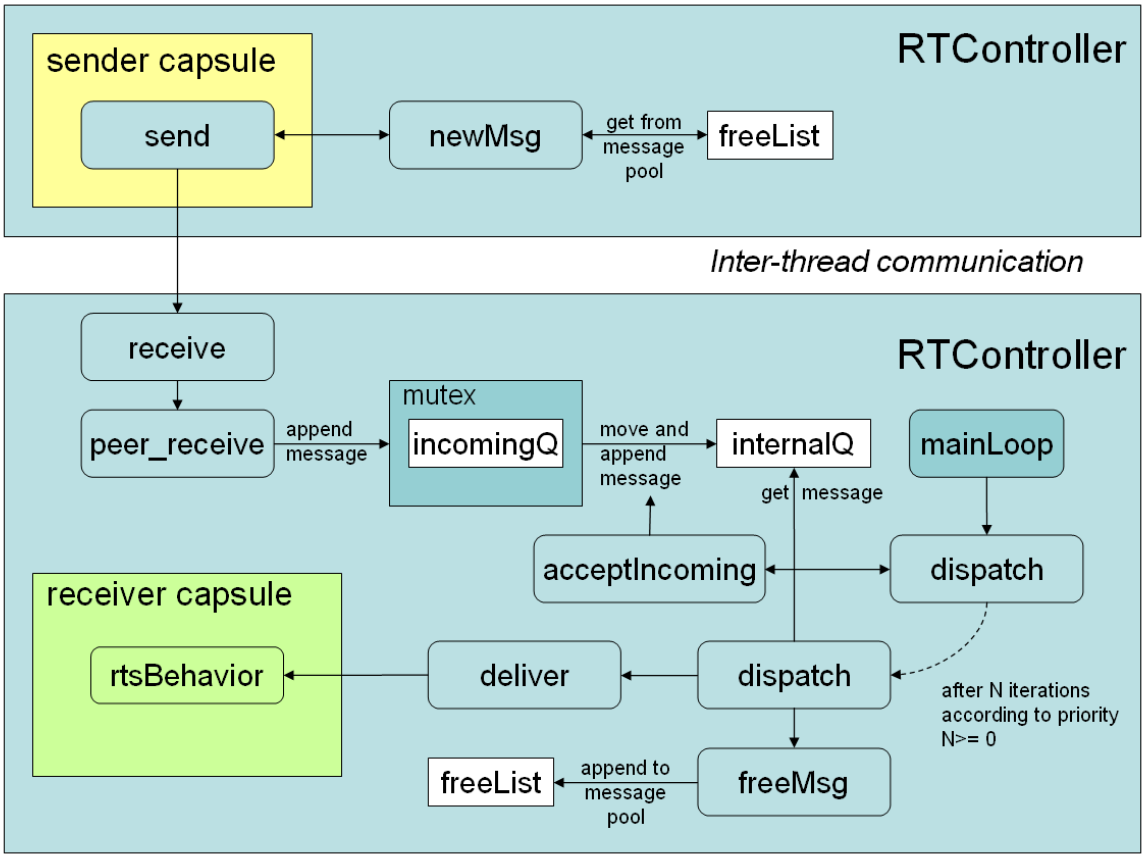
Intra-thread message sending

When sending a message, the sending capsule calls `RTController::send` on its controller, which in turn calls `RTController::receive` on the controller of the receiving capsule. If both capsules are incarnated on the same physical thread, meaning they are running on the same controller, the message will be appended to the appropriate `internalQ[message_priority]` message queue according to the priority indicated in the send statement. When the sending capsule has executed its current transition to completion, the control is returned to the `mainLoop`, which then calls `RTController::dispatch` function that will dispatch messages from `internalQ`.



Inter-thread message sending

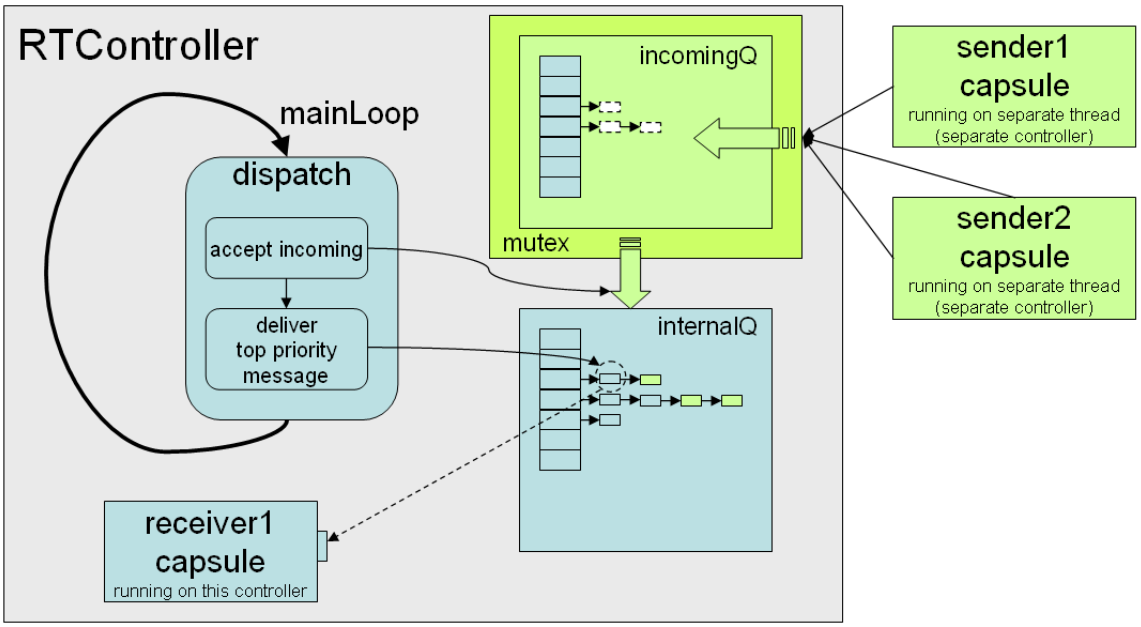
For inter-thread communication, `RTController::send` is called on the sender's controller/thread, and `RTController::receive` will be called on receiver's controller/thread which in this case is a different object. `RTController::receive` will call `RTController::peer_receive` that will put a message into `incomingQ` on the receiver's controller object. Access to `incomingQ` is protected by a mutex in the `peer_receive` function. This ensures that several threads sending messages to the same receiver thread will not interfere during access to the receiver's `incomingQ` variable.



Message dispatch algorithm

Each `RTController` object runs its own `mainLoop` that is dispatching messages from `incomingQ`. In common cases, each dispatch iteration performs the following actions:

- checks if there are any messages in `incomingQ` and moves all messages to the end of `internalQ`.
- takes the first message with the highest priority in `internalQ` and removes it from `internalQ`
- delivers this message to the receiver capsule by calling `receiver->rtsBehavior(signal_id, port_id)` which may trigger a transition in the receiver capsule's state machine
- frees the message



Messages from the incoming queues are moved to the corresponding internal queues in the beginning of each dispatch iteration no matter which priorities they have. Priorities are considered only when messages are selected from `internalQ` for delivery.

Since the access to `incomingQ` is protected by a mutex, it is guaranteed that no external thread will put any new messages to `incomingQ` while `RTController::acceptIncoming` function is updating the queue. The same mutex also guarantees that different external threads will not get access to `incomingQ` at the same time.

C++ RT Services Library Class Reference

The C++ RT Services Library Class Reference is a reference to the structures and classes that you will need to use within the action code of a capsule to access the services provided by the C++ RT Services Library.

This reference does not describe private or restricted functions and variables 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 from a user's application.

RTActor

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

Since all action code added to a capsule is generated as part of a capsule class, the action code has direct access to some useful variables and functions that are defined in RTActor. You should only be calling the functions of RTActor or using variables that are defined below.

Note that most variables on RTActor are private, because one capsule may not manipulate another capsule's attributes. Use the public or protected functions instead.

`RTController * **context**(void);`	Gets the controller for the physical thread on which a capsule instance is executing.
	There are some public functions on the `RTController` class that can be accessed this way. In particular, you may find it useful for printing error information, as in the example below:
	`if( ! port.ping().send() )`
	`{`
	`  log.show("Error on physical thread: ");`
	`  log.log( context()->name() );`
	`  context()->perror("send");`
	`}`

`const char * **getCurrentStateString** ( void ) const;`	Gets the name of the currently active state in the capsule's state machine.
`RTController::Error **getError**( void ) const;`	Returns the most recent error within a particular thread. The Error enumeration is defined within the `RTController` class. The error code is not reset by a subsequent successful primitive function call. It should be called immediately following the failure of a Services Library function call.
`int **getIndex**( void ) const;`	Gets the replication index of this capsule instance in its "home" role (i.e. the capsule part where it was incarnated). The replication index is zero (0) based.
`const char *`	Gets the name of the capsule part in which this capsule instance is

getName(void) const;	running (where it was incarnated).
const char * **getTypeName**(void) const;	Returns the class name of this capsule instance.

int **isType** (const char * class_name) const;	Queries the capsule class of this capsule instance. Returns 1 (true) if this capsule instance is of class class_name, and 0 (false) otherwise.
	`if ( isType("PhoneManagerCapsule") )`
	`{`
	`  log.log("This capsule role is of type: ");`
	`  log.log( getTypeName() );`
	`}`

virtual void **logMsg**(void);	This function is called by the Services Library before a received message is processed by a capsule instance (if so configured). As implemented by the `RTActor` class, this function prints to the log every message that was delivered to the capsule, depending on the debug level. Since this function is defined as virtual, it can be useful in some circumstances to override it within a capsule class in order to provide some alternative general processing for each message. To override this function, simply add a new operation to the capsule with the same name and prototype (it takes no parameters and returns void). It can also be overridden for the entire model by creating a new `RTActor::logMsg()` function, compiling it, and including it in the model using the model link options.

const RTMessage * **msg**;
 	
const RTMessage * **getMsg**(void);
	The `msg` attribute can be accessed via the `getMsg` function. A pointer to the message is returned.
	Every capsule class has an attribute `msg` which contains a pointer to the current message delivered to a capsule instance. This attribute can be used within transition detail level code to retrieve a message that was sent to the capsule instance.
	Examples:
	Retrieve the void * pointer to the data portion of the message:
	`const void *data_ptr = msg->data;`
	You can also use getMsg() to access the current message:
	`const void *data_ptr = getMsg()->getData();`
	For most cases, the data can be accessed directly using the rtdata parameter that is passed to every transition code segment:
	`// The following is commonly needed to make a copy of the data`
	`// that was sent with a message`
	`const ADataClass & data1 =`
	`  *((const ADataClass *)getMsg()->getData());`
	
	`// the above statement can be written using the rtdata`
	`// parameter available in all state transition segments.`
	`const ADataClass & data1 = *rtdata;`

`virtual void` **`unexpectedMessage`** `(void );`	This function is called by the Services Library when there is no transition found that is triggered by the current message about to be delivered. This happens when the capsule's `rtsBehavior()` function is called to process a message and no corresponding trigger event is found. The default `unexpectedMessage()` behaviour prints a message to `stderr`. This function can be overridden on a capsule class basis to provide any additional functionality that may be required. To override this function, simply add a new operation to the capsule class with the same name and prototype. It can also be overridden for the entire model by creating a new `RTActor::unexpectedMessage()` function, compiling it, and including it in the model using the model link options.
`virtual void` **`rtgStateEntry`** `(void );`	This function is called by the Services Library when a state in a capsule state machine becomes active. This happens immediately after the state has become active, before the entry action of that state executes. The default implementation of this function does nothing. The function can be overridden on a capsule that wants to perform some common actions whenever a new state in its state machine becomes active. To override this function, simply add a new operation to the capsule with the same name and prototype.

RTActorClass

This class is created to represent the common external features (service 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 metadata about the capsule class can be stored only once.

You can reference this capsule information object in detail level code, by referencing it by name.

The `RTActorClass` object is commonly required when using the `Frame::incarnate` function. When incarnating (i.e. creating a new capsule instance) you always have to specify which capsule class should be instantiated in an optional capsule part.

Below the first parameter is the capsule part (`RTActorRef`) and the second the capsule class (`RTActorClass`):

```
frame.incarnate( aCapsulePartName, ACapsuleClass );
```

You should not create new instances of `RTActorClass`, but you can reference existing class objects.

Objects of type `RTActorClass` cannot be passed as message data, but it is safe to pass an address within a process.

This class does not have any functions available, and is only used in conjunction with

the frame service to refer to specific capsule classes for manipulating the dynamic structure of a model.

RTActorRef

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

You can reference this capsule part in detail-level code of the containing capsule by referencing the capsule part by name. There are basically only two reasons why you would want to directly access capsule parts:

1. To incarnate a capsule instance into a capsule part

For example, to specify which part to incarnate a capsule into, you would use the name of the capsule part directly in the incarnate function:

```
frame.incarnate( devices, Device );
```

Where `devices` is the target capsule part in which you want to incarnate a capsule of type `Device`.

2. To find the replication of a capsule part

It is also commonly useful to use the replication size of a replicated capsule part.

`int` **`size`**`( void )const;`	Returns the replication size of a capsule part. The function returns the replication size whether or not there is a capsule instance currently incarnated at a specific slot.

RTActorId

The Frame service functions `Frame::incarnate` return an object of type `RTActorId` to identify a particular capsule instance. The `RTActorId` object instance is used as a handle to import the capsule instance into a plug-in capsule part, and to destroy or deport a capsule instance.

In a capsule that has a Frame SAP called "frame", the capsule gets it's RTActorId as follows:

```
RTActorId id = frame.me();
```

Note that `RTActorId` is a pointer to the capsule. If the capsule is destroyed, the pointer is invalid and the functions that use it will crash. It is important to guarantee, at the application level, that the capsule will not be destroyed.

`int` **`isValid`**`(void) const;`	Returns 0 (false) if the id refers to an invalid capsule instance, and 1 (true) otherwise. This function should not be used to test for the state of a capsule instance (regardless of whether it is still alive). It should only be used immediately after a call to the `Frame::incarnate` function. Once the capsule instance has been created, `isValid` always returns 1 (true), even if the capsule instance is subsequently destroyed.

	The example below shows how the capsule instance id is checked after calling the incarnate function. If the incarnation fails an error message is printed to the log, and if the incarnation is successful the capsule instance is immediately destroyed. `RTActorId capsule_id = frame.incarnate( terminal, LongDistanceTerminal, RTTypedValue(), callThread, 0 ); if( ! capsule_id.isValid() ) context()->perror("Incarnation failed: "); else frame.destroy( capsule_id );`

RTController

The `RTController` is an abstract class that defines the interface to a group of executing capsule instances within a single thread of concurrency. There is one controller object for each physical thread in the system. The controller object maintains information about the state of the thread as a whole, including the most recent error. Since the majority of functions 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.

From within a capsule instance, you can retrieve a pointer to its controller by calling the `RTActor::context` function. You can also use `RTActor::getError` to obtain the error value maintained by the controller.

`void` **`abort`**`( void );`	Calling this function on any controller will terminate the controller on which the capsule instance is running which in turn destroys all capsule instances running on that controller. Messages that have not been dispatched are deleted. If this is called on the main thread then all threads are destroyed, and the process quits. `context()->abort();`
`Error` **`getError`**`( void ) const;`	Returns the value of the most recent error within the thread. The error code is not reset by a subsequent successful primitive function call. It should be called immediately following the failure of a Services Library function call.
`const char *` **`name`**`( void ) const;`	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 in the transformation configuration. This method is a way of allowing capsules to find out what thread they are running on.
`void` **`perror`**`( const char *`	The optional parameter `error_string` is the string to be printed to `stderr` along with the current error string as returned by the

`error_string = "error" );`	`RTController::strerror` function. By default, the string `"error"` will be printed.
	`if( ! aPort.ack().send() )`
	`  context()->perror("Error sending ack");`
	
	`Output:`
	`Error sending ack: Port not connected.`

`const char *` **`strerror`**`( void ) const;`	Returns a description of the current error code on the current `RTController`, that is, the controller for a physical thread.

RTExceptionSignal

The Exception Service, like other run-time system services, is accessed through an exception port. Exceptions manifest themselves in the form of Services Library messages arriving on appropriate Exception Service ports. Any capsule class that needs to raise or handle exceptions must define an exception port in its structure. Exception Service ports are instances of the class `Exception`.

Exceptions are defined as events in the Exception Service protocol, and an application can raise these exceptions by sending the events to an exception port.

```
RTExceptionSignal arithmeticError( const RTTypedValue & );
RTExceptionSignal error( const RTTypedValue & );
RTExceptionSignal notFoundError( const RTTypedValue & );
RTExceptionSignal notUnderstoodError( const RTTypedValue & );
RTExceptionSignal serviceAccessError( const RTTypedValue & );
RTExceptionSignal streamError( const RTTypedValue & );
RTExceptionSignal subclassResponsibilityError( const RTTypedValue & );
RTExceptionSignal timeSliceError( const RTTypedValue & );
RTExceptionSignal userError( const RTTypedValue & );
```

`int` **`raise`**`( void );`	Call this function to raise the exception that is represented by the specific `RTExceptionSignal` object.
	
	The exception must be raised by the application. The Services Library does not automatically raise any exceptions by itself.
	
	With `userError()`, you can provide any relevant data that is required to send along with the exception.
	
	Example:
	`// How to handle service errors using the exception service`
	`if( ! myPort.start().send() )`
	`  ex.userError( RTString("Send on ring port failed.")).raise();`

Frame

The Frame service is accessed via Frame service ports, declared in the structure of a capsule class. Frame service ports are instances of the class Frame. The functions

take, as their parameters, either of:

- static capsule part names `RTActorRef` (design-time names of capsule parts), and capsule class names `RTActorClass`
- dynamic capsule instance `idsRTActorId` (generated at run time)

The Frame class also provides a number of query primitives that can be used to get information about the structure of the model. These functions may be useful in some circumstances, particularly for writing generic capsules that must deal with very dynamic structures.

`int` **`classIsKindOf`**
`(const RTActorClass & subClass,`
` const RTActorClass & superClass`
`);`
	Tests whether a particular capsule class is a subclass of another.
	`subClass` is the name of the capsule class in question. `superClass` is an capsule class which, if it is the same as, or a superclass of, the class in question, the method returns 1, and returns 0 otherwise.

`const char *` **`className`** `( const RTActorClass & );`	The function returns the name of the specified capsule class in the form of a null-terminated string. The Services Library stores run-time information about each capsule class in the model using a separate class (often referred to as a metaclass). The information is contained within an `RTActorClass` object. There is one object for each capsule class, having the same name as that of the class that it represents.
`const RTActorClass &` **`classOf`** `(const RTActorId & incarnation);`	The function returns the capsule class of the specified capsule instance. If an error occurs the `EmptyActorClass` is returned.
`int` **`deport`**`( const RTActorId &` `instance, RTActorRef & role);`	Removes a capsule instance from a plug-in capsule part. `instance` is the id of the capsule instance to be removed. `role` is the capsule part name from which the capsule instance is to be removed. The function returns false (0) if an error occurred, and true (non-0) otherwise. The function can fail if: • the capsule instance being removed was not present in the capsule part • the capsule part is not uniquely identified.

`int` **`destroy`**
`( RTActorId  & instance );`
`int` **`destroy`**
`( RTActorRef & role );`
	Using destroy with the capsule id will destroy the capsule instance and all of its contained capsule parts.
	Instead of destroying one capsule instance, you can destroy all

	instances contained in a capsule part. Any and all instances of this capsule part will be destroyed. The capsule part can only be destroyed by the immediate container of that part.
	These functions return false (0) if an error occurred, and true (non-0) otherwise.
	Example:
	Receive the capsule instance identifier from the capsule instance to destroy:
	`RTActorId cid = rtdata;`
	`frame.destroy( cid );`
	Or you can destroy all instances by specifying the capsule part instead:
	`frame.destroy( terminal );`

`int` **`import`**
`( const RTActorId & instance,`
`  RTActorRef & dest,`
`  int index = -1 );`
`int` **`import`**
`( RTActorRef & actor,`
`  RTActorRef & dest,`
`  int index = -1 );`
	Imports a capsule instance into a plug-in capsule part.
	`instance` is the instance id of the capsule instance which is to be imported.
	`dest` is the name of the capsule part into which the capsule instance will be imported. The capsule part must be in the immediate decomposition frame within the calling capsule instance.
	`index` is the replication index within the plug-in capsule part into which the capsule instance is to be imported. If unspecified, the capsule instance is imported into the first available index.
	Using an alternate form of the function, you can provide a capsule part name instead of the capsule instance id. To use this form of import, the capsule part must not be replicated, and a valid capsule instance for this part must be active. This function will import the capsule instance held by the part (only one, since the part is not replicated) into the destination capsule part.
	These functions return false (0) if an error occurred, and true (non-0) otherwise.
	The function fails in the following cases:
	• if the capsule identified no longer exists.
	• the class of the capsule instance is not a compatible class.
	• a port of the instance that is bound in the imported capsule port is already bound elsewhere.
	• the target capsule part is not uniquely identified.

`RTActorId` **`incarnate`**`( RTActorRef & role);` `RTActorId`	Creates capsule instances in optional capsule parts. This function must be used to create and run capsule instances on

```
incarnate( RTActorRef & role,
  const RTActorClass &
capsule_class);

RTActorId
incarnate( RTActorRef & role,
  const void * data,
  const RTObject_class * type,
  RTController * log_thread,
  int index );

RTActorId
incarnate( RTActorRef & role,
  const RTActorClass &
capsule_class ,
  const void * data,
  const RTObject_class * type,
  RTController * log_thread,
  int index );



RTActorId
incarnate( RTActorRef & role,
  const RTDataObject & rtdata,
  RTController * log_thread = 0,
  int index = -1 );

RTActorId
incarnate( RTActorRef & role,
  const RTActorClass &
capsule_class ,
  const RTDataObject & rtdata,
  RTController * log_thread = 0,
  int index = -1 );

RTActorId
incarnate( RTActorRef & role,
  const RTTypedValue & info,
  RTController * log_thread = 0,
  int index = -1 );

RTActorId
incarnate( RTActorRef & role,
  const RTActorClass &
capsule_class ,
  const RTTypedValue & info,
  RTController * log_thread = 0,
  int index = -1 );
```

different logical threads.

The functions return a valid `RTActorId` if the call is successful. To test if the call failed, use the `RTActorId::isValid` function on the returned object. For example:

```
RTActorId ind = frame.incarnate( mySubcapsule );
if ( ind.isValid() )
...//use index
else
...//getError() to see what's wrong, don't use index
```

If the function fails you can use the `RTActor::getError` function to find out why it failed.

There are alternate forms of the incarnate function which leave out the `RTActorClass` parameter (defaulting to the class specified for the capsule part) and for sending different types of initialization data to the new instance, either as `RTDataObject` classes or anything with a descriptor.

`role` is the name of the optional capsule part contained in the structure of the capsule instance making the incarnate call.

`capsule_class [optional]` is the name of the class that should be instantiated into the optional capsule part. If absent, the incarnated class defaults to the class of the capsule part. You can also use the predefined variable `EmptyActorClass` to specify that the incarnated class defaults to the class of the capsule part.

`data, type, rtdata, info [optional]` is the data to be sent to the created capsule instance. The data sent is accessible in the capsule instance initial transition. Be sure to specify if no data is to be sent. See the examples below.

`log_thread` is the name of the logical thread (given in the thread configuration in the transformation configuration) where you want the incarnated capsule instance to run. If no thread is specified the capsule instance is incarnated in the thread of the caller.

`index` is the replication index into which the new capsule instance should be incarnated. This is only valid when incarnating capsule instances into replicated capsule parts. Indexing begins at 0, that is index 0 is the first capsule instance. If specified as -1, the capsule will be incarnated in first free slot.

The first free slot is the last slot number that was made available after a capsule was deleted. For example, assume you created and then deleted capsules 0, 1, 2, 3 in this order. The list of free slots is as follows in order: 3, 2, 1, 0, 4, 5, 6, 7, 8, 9.

To use the frame functions, create a port using the `Frame` protocol.

	Examples:
	This will incarnate a capsule instance into the optional capsule part named 'terminal'. No initialization data is sent to the capsule instance. The incarnated capsule defaults to the type of the capsule part:
	`RTActorId capsule_id;`
	`capsule_id = frame.incarnate( terminal );`
	
	`if( ! capsule_id.isValid() )`
	`context()->perror("Incarnation failed: ");`
	If you want to incarnate the incarnated class of the capsule part and send initialization data you can specify the `EmptyActorClass` variable as the second argument:
	`RTActorId capsule_id;`
	`ControlData data( 15, 8.98 );`
	
	`capsule_id = frame.incarnate( terminal, EmptyActorClass,`
	`&data, &RTType_ControlData, (RTController *)0, -1 );`
	
	This will incarnate a capsule instance into the optional capsule part at index 0, with initialization data, on a specific logical thread.
	`RTActorId capsule_id;`
	`PrinterData data( 14, "ott05" );`
	`capsule_id = frame.incarnate(`
	`device, // capsule part name`
	`Printer, // capsule`
	`&data, // initialization data`
	`&RTType_PrinterData, // type descriptor`
	`callThread, // logical thread name`
	`0 ); // index`
	
	`if( ! capsule_id.isValid() )`
	`  context()->perror("Incarnation failed: ");`
	The following could be used to incarnate a capsule instance without initialization data, but with a specific logical thread or a replication index:
	`RTActorId capsule_id;`
	`PrinterData data( 14, "ott05" );`
	
	`capsule_id = frame.incarnate( device, Printer,`
	`(const void *) 0, // initialization data`
	`(const RTObject_class *) 0, // type descriptor`
	`PrintThread, // logical thread`
	`0 ); // replication index`
	
	`if( ! capsule_id.isValid() )`
	`  context()->perror("Incarnation failed: ");`

`RTActorId` **`incarnationAt`**	Retrieves a particular capsule instance of a capsule part.

`( const RTActorRef & part, int index );`	Returns the instance id of the capsule instance at the specified capsule role index. The index is zero-based.

RTInSignal

This class is used to work with incoming events defined within a protocol. As explained in `RTProtocol`, each event defined on a protocol becomes a function. For incoming events the functions return an `RTInSignal` object on which you can specify what action to perform with the event.

The only actions defined on incoming events are to manipulate the defer queue, that is to retrieve specific messages for the event that have been deferred.

For example if a message was deferred at some point in a capsule's behavior:

```
getMsg()->defer();
```

you can later recall the message by calling:

```
aPort.ack().recall();
```

`int` **`purge`**`( void );`	If a port is replicated then the purge function will delete all deferred messages for the event on all port instances. Returns the number of deleted messages from the defer queue.
`int` **`purgeAt`**`( int index );`	If a port is replicated then this function will delete deferred messages for the event on the specified port instance only. `index` is a port instance index on which to delete deferred messages. Returns the number of messages that were deleted from the defer queue.
`int` **`recall`**`( int front = 0 );`	Recall one deferred message for the event on all port instances. `front [optional]` is a boolean int that indicates whether the message should be recalled to the front of the system message queue. If false, or left unspecified, the message is sent to the back of the message queue. By recalling to the front, it is possible to avoid overtaking of messages. Returns the number of recalled messages. There is no time-limit on deferral. Applications must take precautions against forgetting messages on defer queues. This function recalls the first deferred message of this event on any port instance. To recall the first message of any event then use the `RTProtocol::recall` function.
`int` **`recallAll`** `( int front = 0 );`	Recall all deferred messages for the event on all port instances.

	Returns the number of recalled messages. There is no time-limit on deferral. Applications must take precautions against forgetting messages on defer queues. This function recalls ALL deferred messages of this event on ALL port instances. To recall all messages of any event, use the `RTProtocol::recallAll` function.

`int recallAt`
`( int index,`
`  int front = 0 );`
	Recall one deferred message for the event on a specific port instance. Returns the number of recalled messages.
	There is no time-limit on deferral. Applications must take precautions against forgetting messages on defer queues.
	This function recalls the first deferred message of this event on a specific port instance. To recall the first message of any event, use the `RTProtocol::recallAt` function.

`int recallAllAt`
`( int index,`
`  int front = 0 );`
	Recall all deferred messages for the event on a specific port instance. Returns the number of recalled messages.
	There is no time-limit on deferral. Applications must take precautions against forgetting messages on defer queues.
	This function recalls ALL deferred messages of this event on a specific port instance. To recall all messages of any event, use the `RTProtocol::recallAllAt` function.

Log

The system log is accessed via ports using the Log protocol, which are instances of the class Log. A log port only takes incoming messages and does not pass any information in the reverse direction. The functions available for accessing the system log are listed below.

Currently all log service output is directed to `stderr`. This means that the `open()`, `clear()`, and `close()` functions should not be used.

`void show( const char * data );`
`void show( char data );`
`void show( double data );`
`void show( float data );`
`void show( int data );`
`void show( long data );`
`void show( short data );`
`void show( unsigned data) ;`
`void show( ushort data) ;`
`void show( ulong data) ;`
`void show`
`( const RTDataObject & data );`
`void show`
`( const void * data,`
	`log` writes data as an ASCII string to the log with a trailing carriage return.
	`show` writes data as an ASCII string to the log with NO leading or trailing carriage returns.
	`data, type` is the object, type information, or simple type that is to be written to the log.
	The log knows how to display simple types, but it can also display any user-defined type as well. To display a user-defined type, it must have a type descriptor defined with a function to encode the object. The log will simply call this

`  const RTObject_class * type);`
`void` **`show`**
`( const RTTypedValue & data);`
 	
`void` **`log`**`( const char * );`
`void` **`log`**`( const RTString & );`
`void` **`log`**`( char );`
`void` **`log`**`( double );`
`void` **`log`**`( float );`
`void` **`log`**`( int );`
`void` **`log`**`( long );`
`void` **`log`**`( short );`
`void` **`log`**`( const RTDataObject & );`
`void` **`log`**
`( const void *,`
`  const RTObject_class * );`
`void` **`log`**`( const RTTypedValue & );`
	encode function.
	The only difference between the `log()` and `show()` functions is that `log()` outputs a carriage return after the data is output to the log.
	Examples:
	`// Print as an ASCII string the contents of a class`
	`log.show( &SubscriberData, &RTType_SubscriberData );`
	
	`// Print a string`
	`log.show( "Timer has expired" );`
	
	`// Print an int`
	`log.show( 19 );`

`void` **`cr`**`( void );`
`void` **`crtab`**`( int num_tabs = 1 );`
`void` **`space`**`( void );`
`void` **`tab`**`( void );`
`void` **`commit`**`( void );`
	These are various functions that can be used to output predefined characters to the log. `commit()` will output all buffered characters in the log.
	`num_tabs` is the number of tabs to insert, the default is one (1). Tab settings are defined by the system and cannot be altered by the user.
	`log.cr();`
	`log.space();`
	`log.tab();`
	`log.commit();`
	
	`// The previous commands can be supplied using show()`
	`log.show( "\n \t" );`
	`log.commit();`

RTMessage

This class is the data structure used within the Services Library to represent messages that are communicated between capsule instances. The messages that are sent between capsules contain a required event name (which identifies the message), an optional priority (relative importance of this message compared to other unprocessed messages on the same thread - defaults to General), and optional application data.

You will most often use the functions 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.

Applications should treat the `msg` field of an `RTActor` and all data addressed beyond that pointer as read-only.

`int` **`defer`**`( void ) const;`	Defer the current message against the receiving port's defer queue. Returns true (1) if the message was successfully deferred and false (0)

	otherwise. An error will be returned if you try to defer an invoked message or a message which has already been deferred. Deferred messages can be recalled using the functions defined on the `RTInSignal` class. In the transition where a message is to be deferred you would defer the message as follows: `getMsg()->defer();`

`void *`
`getData``( void ) const;`
	Returns an untyped pointer to the data that was sent along with a message.
	Note that it is recommended to use the predefined `rtdata` parameter to access the typed data of a message in a transition:
	`const ADataType & dt = *rtdata;`
	In cases where there are multiple triggers for a transition you will have to cast the received data depending on the event that triggered the transition.
	`const ADataType & dt2 = *(ADataType *)(getMsg()->getData());`

`int` **`getPriority`**`( void ) const;`	Returns the priority of the message.

`const char *`
`getSignalName``( void )`
`const;`
	Returns the name of the protocol event of the message.
	`log.show("Event named: ");`
	`log.log(getMsg()->getSignalName());`

`const RTObject_class *` **`getType`**`( void ) const;`	Returns a pointer to an `RTObject_class` which contains the type information that describes the data in the message, or (`RTObject_class *)0` if no type was specified.
`int` **`isValid`**`( void ) const;`	Returns 1 (true) if the message has been initialized from a valid event and potentially some data, and 0 (false) otherwise. This method is intended to verify that the returned message has been properly filled by the reply to a `RTOutSignal::invoke()` function call.

`RTOutSignal *`
`sap``( void ) const;`
	Returns a pointer to the port instance on which this message was received, or `(RTProtocol *)0` if called in the initial transition.
	`// find out where the message was received,`
	`// and send a message back on that same port`
	`RTProtocol * port = msg->sap();`
	`if( port != (RTProtocol *)0 )`
	`  ((MyProtocol::Base *) port)->hello().send();`

`int` **`sapIndex`**`( void )`
`const;`
`int` **`sapIndex0`**`( void )`
`const;`
	Returns the index of the port on which the message was received. The `sapIndex` function returns a one-based index (index values begin at 1) and `sapIndex0` is 0 based.
	Use to send a message to a particular port instance, as follows:
	`int idx = msg->sapIndex0();`
	`port.hello().sendAt( idx );`

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 model. 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 data structures can be used in models even if they cannot be expressed in RSARTE 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 Services Library can decide when a payload area of a message is large enough to hold the data to be sent.
- Any user-defined type can be sent (by value), using the copy and destroy functions in the type descriptor, and inspected via the observability interface using the init, encode and decode functions.

The important thing to remember is that RSARTE will generate these descriptors for most classes which are defined using basic types (see the list defined in the `RTObject_class.h` file located in `<install dir>/rsa_rt/C++/TargetRTS/include`). If classes contain more complicated structures you can write your own type descriptor functions.

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`

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 by the model debugger, or sent to another process.

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 function that requires it.

RTOutSignal

This class is used to work with outgoing events defined within a protocol. As explained in `RTProtocol`, each event defined on a protocol becomes a function. For outgoing events the functions return an `RTOutSignal` object on which you can specify what action to perform on the event. For example, to send an event first initialize the `RTOutSignal` by calling the function on the port, then specify an action to perform on the event:

```
port.ack().send();
port.hello( 1089 ).sendAt(1);
```

`int` **`invoke`**`(`
`RTMessage * replyBuffers);`
 	
`int` **`invoke`**`();`
	Synchronous message broadcast to all port instances.
	The function returns the number of replies received. If it returns 0 (false), the call failed. An invoke call can fail if the port is not connected (no connection to the receiver end port). An error will be returned if you try to invoke an event across a physical thread boundary.
	`replyBuffers` is a user-supplied message object that stores the reply message(s) resulting from the invoke. The user is responsible for allocating and deleting this data when it is no longer required. Typically a local variable will be declared to hold the returned message(s). To verify the validity of the returned message(s) call `RTMessage::isValid` once the invoke returns.
	If a port is replicated all port instances will be invoked. Use `RTOutSignal::invokeAt` to invoke a specific port instance.
	An invoke is a synchronous send, and the sender is blocked until each receiver has processed the message and sent back a reply. Run-to-completion semantics are enforced, such that an invoke has the same semantics as a function call. Note that the data field is not copied on invoke (since the receiver always runs in the same thread).
	Do not use invoke in the initial transition of a capsule as the system may still be processing initialization messages. Also, because of its blocking nature, invoke cannot be used across threads or capsules

	connected through a network.
	`RTMessage replies[ aPort.size() ];`
	`aPort.ack().invoke( &replies );`
	`for( int i = 0; i < aPort.size(); i++ )`
	`{`
	`  if( replies[i].isValid() )`
	`  //code to handle valid reply`
	`  else`
	`  //code to handle invalid reply`
	`}`
	The receiver of the invoke must use `RTOutSignal::reply` to respond to the invoke. Data can be optionally sent back with the reply.
	`rtport->nack().reply();`
	However, if the version of `invoke` without reply buffer parameter is used, then the reply is implicitly made when the receiver has processed the message.

`int invokeAt( int index, RTMessage * replyBuffer );`
 	
`int invokeAt( int index );`
	Synchronous message send to a specific port instance.
	The function returns 1 (true) if the call is successful and 0 (false), if the call failed. An invoke can fail if the port is not connected (no connection to the receiver end port). An error will be returned if you try and invoke across a physical thread boundary.
	`index` is the port replication index of the port instance on which the message should be sent.
	`replyBuffer` is a user-supplied message object that stores the reply message(s) resulting from the invoke. The user is responsible for allocating and deleting this data when it is no longer required. Typically a local variable will be declared to hold the returned message(s). To verify the validity of the returned message(s) call `RTMessage::isValid` once the invoke returns.
	An invoke is a synchronous send, and the sender is blocked until each receiver has processed the message and sent back a reply. Run-to-completion semantics are enforced, such that an invoke has the same semantics as a function call. Note that the data field is not copied on invoke.
	Do not use invoke in the initial transition of a capsule as the system may still be processing initialization messages. Also, because of its blocking nature, invoke cannot be used across threads or capsules connected through a network.
	`RTMessage reply;`
	`aPort.ack().invokeAt( 0, &reply );`
	`if( reply.isValid() )`
	`  // code to handle valid reply`
	`else`
	`  // code to handle invalid reply`

	The receiver of the invoke must use `RTOutSignal::reply` to respond to the invoke. Data can be optionally sent back with the reply. `rtport->nack().reply();` However, if the version of `invokeAt` without reply buffer parameter is used, then the reply is implicitly made when the receiver has processed the message.
`int reply ( void );`	Used to respond to a synchronous message. Returns 1 (true) if the reply is successful, and 0 (false) otherwise. If the caller has provided a reply buffer when invoking the message, then the receiver must use `RTOutSignal::reply` to respond to the invoke. Data can be optionally sent back with the reply. `rtport->nack().reply();`

`int send(`
`  int priority =`
`General );`
	Asynchronous message broadcast to all port instances.
	The function returns a count of the successful sends (remember that ports can be replicated in which case this function will broadcast to all port instances). A send can fail if the port is not connected (no connection to the receiver end port).
	`priority [optional]` specifies the priority at which this message should be sent. The default priority is General. A message priority is interpreted as the relative importance of a message with respect to all other unprocessed messages on a thread. The priority evaluates to one of the defined global priority values.
	Since a port can be replicated, the send function effectively sends a message through all instances of the port (broadcast). If you want to send to only one instance of a replicated port, use the `RTOutSignal::sendAt` function.
	`// In this case the ack signal does not require`
	`// data to be sent with the signal.`
	`aPort.ack().send();`
	It is always good practice to check the return codes.
	`if( ! aPort.ack().send() )`
	`  context()->perror("Error with send");`
	You can also send data with a message.
	`// Sending some data by value, that is a copy of the data is`
	`// sent.`
	`SomeDataClass mdata("123-4356", "Ottawa");`
	`aPort.Info( mdata ).send();`

`int sendAt(`
`  int index,`
`  int priority =`
`General );`
	Asynchronous message send to a specific port instance.
	The function returns 1 (true) if the call succeeded and 0 (false) otherwise. A send call can fail if the port is not connected (no connection to the receiver end port) or an invalid replication index

	was provided.
	`index` is the port replication index of the port instance on which the message should be sent.
	`priority [optional]` specifies the priority at which this message should be sent. The default priority is General. A message priority is interpreted as the relative importance of a message with respect to all other unprocessed messages on a thread. The priority evaluates to one of the defined global priority values.
	This function is used instead of `RTOutSignal::send` to send a message to a specific instance of a replicated port.
	`// In this case the ack signal does not require`
	`// data to be sent with the signal.`
	`aPort.ack().sendAt(5);`
	
	`// send to the port instance on which the current`
	`// message was received`
	`int idx = getMsg()->sapIndex0();`
	`rtport->ack().sendAt( idx );`

RTProtocol

For each protocol class in your model, two subclasses of the `RTProtocol` class are generated, one for each direction of communication using the events defined in the protocol. These two subclasses are called the base and conjugate protocol class respectively. 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, and is typed with either the base or conjugate protocol class.

`void` **`bindingNotification`** `( int on_off );`	Use this function to request notification of the creation and destruction of bindings to this port. The events sent to the port by the Services Library when binding notifications are enabled are `rtBound` and `rtUnbound`. `on_off` is a boolean int. If called with 1 (true) the port will receive messages as ports become bound or unbound. Calling the function with 0 (false) will prevent such messages from being sent, but will not purge any messages already queued. No messages are sent for ports which are bound prior to the call of this function.

`int`
`bindingNotificationRequested`
`( void ) const;`
	Use this function to request status of notification for this port. Returns 1 (true) if notification has been enabled for this port, and 0 (false) otherwise.

`int` **`deregisterSAP`**`( void );`	Deregisters an unwired end port (SAP). Returns 1 (true) if the deregistration of the service name was successful, and 0 (false)

	otherwise. When an SAP is deregistered when it is currently connected to a SPP, the connection is terminated.
`int` **`deregisterSPP`**`( void );`	Deregisters an unwired end port (SPP). Returns 1 (true) if the deregistration of the service name was successful, and 0 (false) otherwise. When an SPP is deregistered all connected port instances are disconnected from all connected SAPs. Although the SAPs are disconnected they remain registered, and available to be re-connected.
`char *` **`getRegisteredName`** `( void ) const;`	Get the name that an unwired port has registered with the layer service.

`int` **`indexTo`**`( RTActor * )`
`const;`
	Find the smallest replication index (0-based) which is connected to the given capsule instance. The result is -1 if there is no such index or the id is invalid.
	This example demonstrates how to find the port index that is connected to a newly incarnated capsule part. The port is a replicated port and role1 is a capsule part in the structure of a capsule on which this code is run:
	`RTActorId aid = frame.incarnate(role1);`
	`int port_index;`
	`if( aid.isValid() )`
	`{`
	`  port_index = port.indexTo( aid );`
	`  if( port_index != -1)`
	`  port.Signal().sendAt(port_index);`
	`} else`
	`  context()->perror("Error incarnating role1:");`

`int` **`isBoundAt`**`( int index )` `const;`	Return true (1) if the given replication index (0-based) is connected to another port and false (0) if it is not connected.

`int` **`isIndexTo`**`(`
`  int index,`
`  RTActor * ) const;`
	Returns true (1) if the given port instance index (0 based) is bound to the specified capsule instance.

`int` **`isRegistered`**`( void )` `const;`	Returns true (1) if an unwired port has been registered with the layer service and false (0) otherwise.
`int` **`purge`** `( void );`	Empties the defer queue of all port instances without recalling any deferred message. Returns the number of messages deleted from the defer queue. To delete deferred messages for one port instance use `RTProtocol::purgeAt`.

`int purgeAt( int index );`	Empties the defer queue of a specified port instance without recalling any deferred messages. Returns the number of messages deleted from the port instance defer queue. To delete deferred messages for all port instances use `RTProtocol::purge`.
`int recall( void );`	To recall a deferred message on all instances of this port for processing. Recalls from the back of the queue. Returns the number of messages recalled from the defer queue (either 0 or 1). Calling `recall` on a port gets the first deferred message from one of the port instances. Messages are recalled behind other queued messages. There is no time-limit on deferral so applications must take precautions against forgetting messages on defer queues. This function recalls the first deferred message on any port instance. To recall the first message on one specific port instance of a replicated port, use the `RTProtocol::recallAt` function. The first deferred message on any instance of the replicated port named port1 is recalled as follows: `port1.recall();`
`int recallAll ( void );`	Calling `recallAll` on a port will get all the deferred messages from each of the port instances. Messages will be recalled starting from the back of the main queue. Returns the number of recalled messages. To recall all messages on only one instance of a port with replication factor > 1, use the `RTProtocol::recallAllAt` function.

`int recallAllAt(`
`  int index,`
`  int front = 0 );`
	To recall all deferred messages on a specified port instance `index`. `front` specifies whether recalled messages should be queued ahead (non-zero) or behind (0) other queued messages. Returns the number of recalled messages.

`int recallAllFront( void );`	To recall all deferred messages on all port instances. Recalls to the front of the queue. Returns the number of recalled messages.

`int recallAt(`
`  int index,`
`  int front = 0 );`
	To recall a deferred message on a specific instance of this port for processing. `front` specifies whether recalled messages should be queued ahead (non-zero) or behind (0) other queued messages. Returns the number of recalled messages (either 0 or 1).
	The first deferred message on the port instance at index 3 of the replicated port named port1 is recalled as follows:
	`port1.recallAt( 3 );`

`int` **`recallFront`**`( void );`	This function recalls the first deferred message on any port instance. Calling `recallFront` on a port gets the first deferred message from one of the port instances, starting from the first (instance 0). Messages are recalled to the front main queue. Returns the number of recalled messages (either 0 or 1).
`int` **`registerSAP`** `( const char * service );`	Registers an unwired end port (SAP) with the layer service (as a "client"). Returns 1 (true) if the registration of the service name was successful, and 0 (false) otherwise. The registration can fail if this function is called on a port instance which is not an unwired end port. If this SAP is already registered with this same name, the function returns 1. `service` is a string that is used to identify a unique service name under which SAPs and SPPs will connect. If this function is invoked on an SAP which is already registered with a different name, then the original registered name is automatically deregistered, and the SAP is registered with the new name. When an SAP is registered, it does not necessarily mean that the port has been connected to an SPP. The successful completion of the register function simply indicates that the name has been registered. For example, if the SAP is registered with no corresponding SPP, the connection is only made later when an SPP is registered. The SAP registration is buffered until an SPP is registered with the same service name. If application registration has been selected for an SAP (protected unwired end port) or SPP (public unwired end port) registration is handled automatically by the Services Library.
`int` **`registerSPP`** `( const char * service );`	Registers an unwired end port (SPP) with the layer service (as the "provider"). Returns 1 (true) if the registration of the service name was successful, and 0 (false) otherwise. The registration can fail if this function is called on a port which is not an unwired end port. If this SPP is already registered with this same name, the function returns 1. `service` is a string that is used to identify a unique service name under which SAPs and SPPs will connect. If this function is invoked on an SPP which is already registered with a different name, then the original registered name is automatically deregistered, and the SPP is registered with the new name. When an SPP is registered, it does not necessarily mean that the port has been connected to an SAP. The successful completion of the register function simply indicates that the name has been

	registered. For example, if an SPP is registered with no corresponding pending SAP registrations, the connection will be made later when an SAP is registered. The SPP registration is buffered until an SAP is registered with the same service name. If application registration has been selected for an SAP (protected unwired end port) or SPP (public unwired end port) registration is handled automatically by the Services Library.

`int size( void ) const;`	Returns the multiplicity factor of the port.
	Remember that port instances are indexed in the Services Library as 0 based. That means that if a port has a cardinality of N, you should only reference instances using index numbers 0..N-1.
	`for( int i = 0 ; i < port.size(); i++ )`
	`  port.ack().sendAt( i );`

RTSymmetricSignal

This class is used for symmetric events defined within a protocol. A symmetric event is defined by having both an incoming and an outgoing event with the same name and data class.

As explained in `RTProtocol`, each event defined on a protocol becomes a function. Since symmetric events can be both incoming and outgoing you can perform the combined actions of both `RTOutSignal` and `RTInSignal` on these classes.

```
port.talk.send();   // to send the talk event
port.talk.recall(); // to recall all deferred talk events
```

RTTimerId

Timing services use `RTTimerId` as an identifier for timer requests. The timer identifier is returned by a request to `Timing::informIn`, `Timing::informAt` or `Timing::informEvery`. The timer identifier can be used subsequently to cancel the timer.

`int isValid( void );`	Returns true (1) if the timer identifier is a valid timer id, and 0 (false) otherwise.
	This function only tells if `RTTimerId` points to a valid (existing) timer object and if `RTTimerId` can be used to manipulate the timer. This function does not return the exact state of the timer. For example, it does not differ if the timer is still active, or if the timer has already timed out but the `timeout` event has not yet been delivered; or if the timer has been canceled or a non-periodic timer has timed out.
	The function can be used to test the result of a timer request.
	`RTTimerId tid = timer.informIn( RTTimespec(4,0) );`
	`if( ! tid.isValid() )`
	`  context()->perror("Error when setting timer");`

	The function can also be used to test if a timer still exists before cancelling it:
	`if( tid.isValid() )`
	`  timer.cancelTimer(tid);`
	Note, that the `cancelTimer` function also checks the validity of `tid` and it will return a failure result (0) if you call it with a non-valid timer id.

RTTimespec

The `RTTimespec` class is used to create timer values for passing to the Timer Service. It is designed 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.

`long` **`tv_sec;`**
`long` **`tv_nsec;`**
	`tv_sec` is the number of seconds for the timer setting, and `tv_nsec` is the number of nanoseconds. There are 10^9 nanoseconds in one second.
	This will initialize an `RTTimespec` with one second.
	`RTTimespec t1(1,0);`
	This class is used most often in conjunction with the Timing Service to specify time values. For example to set a one-shot timer to expire in 5 seconds you would use the `RTTimespec` constructor.
	`timer.informIn( RTTimespec(5,0) );`

`RTTimespec &` **`operator=`** `( const RTTimespec & t1 );`	The `RTTimespec` assignment (=) operator re-initializes an existing `RTTimespec` object with new second and nanosecond values.

`RTTimespec &` **`operator+=`**
`( const RTTimespec & t1 );`
`RTTimespec &` **`operator-=`**
`( const RTTimespec & t1 );`
`RTTimespec` **`operator+`**
`( const RTTimespec & t1,`
`  const RTTimespec & t2 );`
`RTTimespec` **`operator-`**
`( const RTTimespec & t1,`
`  const RTTimespec & t2 );`
	Arithmetic operators.
	`t1, t2` are `RTTimespec` objects to add or subtract.
	`RTTimespec t1(2,0), t2;`
	`RTTimespec::getclock( t2 );`
	`t2 += t1;`

`int` **`operator==`**
`( const RTTimespec & t1,`
`  const RTTimespec & t2 );`
`int` **`operator!=`**
`( const RTTimespec & t1,`
`  const RTTimespec & t2 );`
`int` **`operator<=`**
`( const RTTimespec & t1,`
`  const RTTimespec & t2 );`
	Comparison operators.
	Return non-zero if the objects meet the comparison condition; otherwise 0.
	`t1, t2` are `RTTimespec` objects to compare.
	`RTTimespec t1(2,0), t2(3,0);`

`int operator>`
`( const RTTimespec & t1,`
`  const RTTimespec & t2 );`
`int operator>=`
`( const RTTimespec & t1,`
`  const RTTimespec & t2 );`
`int operator<`
`( const RTTimespec & t1,`
`  const RTTimespec & t2 );`
	`if (t1 < t2)`
	`  //t1 is less than t2`

`static void getclock`
`( RTTimespec & tspec );`
	Returns the current time. The values of the `tspec` parameter are filled in with the current time.
	This is a class-scoped function.
	`RTTimespec t;`
	`RTTimespec::getclock( t );`

`RTTimespec( void );`
`RTTimespec(`
`  long sec, long nsec);`
`RTTimespec(`
`  const RTTimespec & ts );`
`RTTimespec(`
`  const RTTime & t );`
	Constructs an `RTTimespec` object with seconds and nanoseconds, or with a value of another `RTTimespec` object or an `RTTime` object.
	An `RTTimespec` of two seconds can be created and passed to the Timing Service `informEvery()` as follows:
	`// 2 seconds`
	`RTTimespec t( 2 , 0 );`
	`// 6am coordinated universal time (UTC)`
	`RTTime abst( 6,0,0 );`
	
	`timer.informEvery( t );`
	`timer.informAt( RTTimespec( abst ) );`

RTTiming

All service methods through which the user requests timeout, at an absolute time or in a time interval, return a unique handle which can be used to construct an `RTTimerId`. The timer id is used to identify a timing request to be cancelled.

Timing ports are instances of the class `RTTiming`.

`void`
`adjustTimeBegin( void );`
 	
`void adjustTimeEnd(`
`const RTTimespec & delta );`
	Used to adjust the internal real-time system clock.
	If there is a need to adjust system time, you must stop the timing service, compute the new time, and then restart the timing service with the new time. The Services Library will make adjustments to its internal data structures so that relative timeouts are not affected by the system clock change. Use `adjustTimeBegin` to stop the timing service and `adjustTimeEnd` to restart it at the new time.
	The application must coordinate time changes through a capsule with a timing port (meaning that it has access to the timing service). The example below encapsulates the clock adjustment behaviour in a capsule function. We assume that the timing port is

	called `timer` and that there are operating system primitives for reading and writing to the system clock which we here call `sys_getclock()` and `sys_setclock()`, respectively.
	`void AdjustTimeCapsule::setClock`
	`  ( const RTTimespec & new_time )`
	`{`
	`  RTTimespec old_time;`
	`  RTTimespec delta;`
	
	`  // Stop Services Library timer service`
	`  timer.adjustTimeBegin();`
	
	`  sys_getclock( old_time );`
	`  sys_setclock( new_time );`
	
	`  delta = new_time;`
	`  delta -= old_time;`
	
	`  // Resume Services Library timer service`
	`  timer.adjustTimeEnd( delta );`
	`}`

`int` **`cancelTimer`**
`( RTTimerId &tid );`
	Cancels a pending timer. Returns true (1) if a pending timeout request was cancelled, and false (0) if no such request was found.
	`tid` is the identifier of the timer that was provided when the service request was made.
	Note that this function guarantees that no timeout message will be received from the cancelled timer, even if the timer has already expired, that is, was waiting to be processed, when the call was performed.
	If `timer` is the name of a timing port, you can create, and subsequently cancel, a timeout request as follows:
	`RTTimerId tid = timer.informEvery( RTTimespec( 2, 0 ) );`
	`timer.cancelTimer( tid );`

`RTTimespec` **`currentTime`** `( void ) const;`	Determines the current absolute time. It is recommended, for performance reasons, that you use the `RTTimespec::getclock` functions instead of `currentTime`. `RTTimespec ctime = timer.currentTime();`

`RTTimerNode *` **`informAt`**
`( const RTTimespec & when,`
`  const void * data,`
`  const RTObject_class * type,`
`  int prio = General );`
 	
`RTTimerNode *` **`informAt`**
`( const RTTimespec & when,`
`  int prio = General );`
	Starts a timer which will expire at a particular absolute point in time.
	A timer handle is returned which can be used to construct an `RTTimerId`. This can be used to cancel the timer prior to expiry. A NULL pointer is returned if the timer request fails.
	`when` is the desired absolute time when the timer is to expire.

`RTTimerNode * informAt ( const RTTimespec & when, const RTDataObject & data, int prio = General );`
 	
`RTTimerNode * informAt ( const RTTimespec & when, const RTTypedValue & info, int prio = General );`
	`data, info, type [optional]` is the message data that will be added to the timeout message and delivered to the capsule when the timer expires. These parameters are optional.
	
	`prio [optional]` is the priority at which the timeout message will be delivered. This parameter is optional.
	
	`RTTimespec now;`
	`RTTimespec::getClock( &now );`
	`timer.informAt( now + RTTimespec( 15, 0 ) );`

`RTTimerNode * informEvery ( const RTTimespec & delta, const void * data, const RTObject_class * type, int prio = General );`
 	
`RTTimerNode * informEvery ( const RTTimespec & delta, const RTDataObject & data, int prio = General );`
 	
`RTTimerNode * informEvery ( const RTTimespec & delta, const RTTypedValue & info, int prio = General );`
 	
`RTTimerNode * informEvery ( const RTTimespec & delta );`
	Starts a periodic timer. A timer handle which can be used to construct an `RTTimerId` is returned. It can be used to cancel the timer prior to expiry. A NULL pointer is returned if the timer request fails.
	
	`delta` represents the desired time interval from the current time at which a periodic timer will expire. The timer interval should be >= 0. If the timer interval is equal to zero, the timer will expire immediately.
	
	`data, info, type [optional]` is the message data that will be added to the timeout message and delivered to the capsule when the timer expires. These parameters are optional.
	
	`prio [optional]` is the priority at which the timeout message will be delivered. This parameter is optional.
	
	`if( ! timer.informEvery( RTTimespec( 10, 0 ) ) log.log("error requesting a periodic timer");`
	
	If the timer is to be cancelled, then an `RTTimerId` object must be constructed for use when cancelling the timer. Ensure that if the timer is going to be cancelled in another transition, that the timer id is saved in a capsule attribute, and not in a transition local variable.
	
	`if( ! (tid = timer.informEvery( RTTimespec( 10, 0 )))) log.log("error requesting a periodic timer");`
	
	`// this could be done in an other transition`
	`timer.cancelTimer( tid );`

`RTTimerNode * informIn ( const RTTimespec & delta, const void * data, const RTObject_class * type, int prio = General );`
 	
`RTTimerNode * informIn ( const RTTimespec & delta, int prio = General );`
 	
`RTTimerNode * informIn ( const RTTimespec & delta, const RTDataObject & data, int prio = General );`
	Starts a timer which expires after some time has passed from the current time (i.e. a time interval).
	
	A timer handle which can be used to construct an `RTTimerId` object is returned. It can be used to cancel the timer prior to expiry. A NULL pointer is returned if the timer request fails.
	
	`delta` represents the desired time interval (in seconds and nanoseconds as an `RTTimespec`) from the current time at which the timer will expire. Timer intervals should be >= 0. If the timer interval is equal to zero, the timer will expire immediately.
	
	`data, info, type [optional]` is the message data that will be

`RTTimerNode * informIn ( const RTTimespec & delta, const RTTypedValue & info, int prio = General );`	added to the timeout message and delivered to the capsule when the timer expires. These parameters are optional.
	`prio [optional]` is the priority at which the timeout message will be delivered. This parameter is optional.
	`// request a timer to expire in 10 seconds`
	`if( ! timer.informIn( RTTimespec( 10, 0 ) ) )`
	`  log.log("error requesting a timer");`
	If the timer is to be cancelled, then an `RTTimerId` must be constructed for use when cancelling the timer. Ensure that if the timer is going to be cancelled in another transition, that the timer id is saved in a capsule attribute, and not in a transition local variable.
	`if( ! (tid = timer.informIn( RTTimespec( 10, 0 ))))`
	`  log.log("error requesting a timer");`
	`// this could be done in an other transition`
	`timer.cancelTimer( tid );`

RTTypedValue

`RTTypedValue` is a struct which is used to encapsulate a data and type pair. For each generated class a structure named `RTTypedValue_<class name>` is generated. The only time you will have to use this structure is when sending subclass data with an event that was defined with a data class of the parent class. For example, given class A and a subclass B, with the event `ack` defined with a data class of A, you would have to use to following syntax to send B with the `ack` event:

```
B subclass;
port.ack(RTTypedValue_A(subclass,&RTType_B)).send();
```

If you do not explicitly specify the type descriptor for class B, the Services Library will use the type descriptor for class A.