



HCL DevOps Model RealTime

C++ RT Services Library

Version 1.16

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 <code>RTTimerId::isValid</code>	Elena Volkova
2016 Jun 09	1.4	Updated for version 10 2016.23 Added <code>RTActor::rtgStateEntry</code>	Mattias Mohlin
2018 Jun 20	1.5	Updated for version 10.2 2018.24	Mattias Mohlin
2018 Sep 25	1.6	Updated for version 10.3 2018.40	Mattias Mohlin
2018 Nov 27	1.7	Updated for version 10.3 2018.48	Mattias Mohlin
2019 Aug 30	1.8	Updated for version 10.3 2019.35	Mattias Mohlin
2020 Oct 14	1.9	Updated for version 10.3 2020.45	Mattias Mohlin
2021 Jun 04	1.10	Updated for version 11.1 2021.24 Type descriptors with template parameters	Mattias Mohlin
2021 Sep 20	1.11	Updated for version 11.1 2021.40 Added <code>RTFrame::incarnateCustom</code> and <code>RTActorFactory</code> . Also added new chapter about the new <code>RTInjector</code> service for dependency injection.	Mattias Mohlin
2021 Nov 11	1.12	Updated for version 11.1 2021.46 Added chapter about how to avoid copying event data, and updated chapter about type descriptors to mention move functions.	Mattias Mohlin
2022 May 19	1.13	Updated for version 11.1 2022.21 Timer API now supports the <code>std::chrono</code> library	Mattias Mohlin
2023 January 23	1.14	Updated for version 11.2 2023.04 Clarified corrected behavior when sending an event by move to a replicated port. Documented new functions <code>RTActor::unhandledMessage()</code> and <code>RTActor::messageReceivedBeforeInitialized()</code>	Mattias Mohlin
2024 February 12	1.15	Updated for version 12.0.1 Documented new JSON Decoder Documented new JSON Parser	Mattias Mohlin
2025 February 28	1.16	Updated for version 12.1.2 The logging service now supports log streams	Mattias Mohlin

Table of Contents

Introduction.....	6
Target Configurations.....	6
Services.....	8
Communication Service.....	8
Message Delivery.....	11
Message Representation.....	11
Avoiding to Copy Message Data.....	13
Deferring and Recalling Messages.....	14
Non-wired Ports.....	15
Logging Service.....	15
Log Stream.....	16
Log Port.....	17
Timing Service.....	17
Frame Service.....	20
Working with Optional Capsule Parts.....	20
Working with Plugin Capsule Parts.....	23
Accessing Model Information at Run-Time.....	24
Exception Service.....	25
External Port Service.....	27
Dependency Injection Service.....	29
Structure of Generated C++ Code.....	31
Type Descriptors.....	31
Type Descriptor Hints.....	33
Templates.....	33
Threads.....	35
Logical threads and physical threads.....	35
Inside the C++ RT Services Library.....	38
Run-to-Completion Semantics.....	38
Intra-thread and Inter-thread Communication.....	38
Message queues.....	39
Message structure and freeList of messages.....	40
Intra-thread message sending.....	41
Inter-thread message sending.....	42
Message dispatch algorithm.....	43
Encoding and Decoding.....	44
C++ RT Services Library Class Reference.....	47
RTActor.....	47
RTActorClass.....	50
RTActorFactory.....	50
RTActorRef.....	51
RTActorId.....	51

RTController.....	52
RTEExceptionSignal.....	53
RTFrame.....	54
RTInSignal.....	59
RTLog.....	60
RTMessage.....	62
RTObject_class.....	63
RTOutSignal.....	65
RTProtocol.....	68
RTSymmetricSignal.....	72
RTTimerId.....	72
RTTimespec.....	73
RTTiming.....	74
RTTypedValue.....	78

This document describes the C++ RT Services Library, which is the run-time library used by real-time applications generated from DevOps Model RealTime.

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

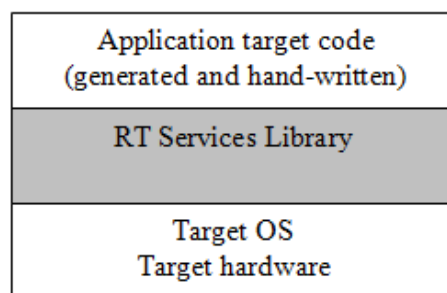
All screen shots were captured on the Windows platform.

Introduction

The RT Services Library is a run-time framework used by the target code that is generated from a UML real-time model. Model RealTime 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. The majority of 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. Note that the run-time library is often referred to as the “TargetRTS” (Target Run Time System).

The RT Services Library provides the run-time implementations of the UML real-time concepts that are supported by Model RealTime. 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 Model RealTime.



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 if needed also 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:

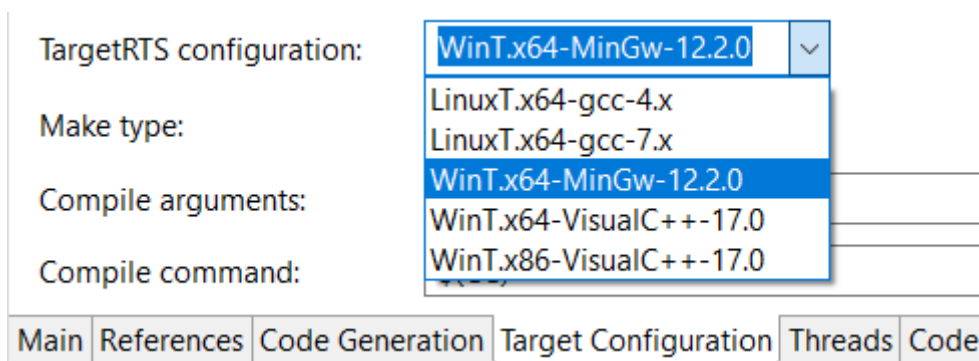
WinT.x64-VisualC++-17.0

target libset name
basename

The first part of the name is the **target basename** which identifies the operating system name, version (if significant) and threading configuration. In the example above the operating system name is Windows. The version is not specified which means that the target configuration can work on more than one version of Windows. 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 (i.e the library is compiled for 64 bits) and the compiler is Microsoft Visual C++ version 17.0.

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:

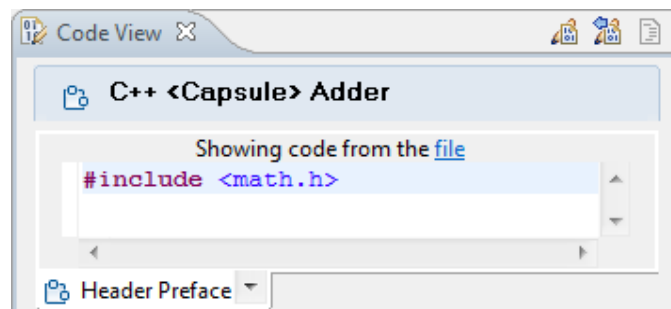


The target configuration strings shown in this drop down menu are dynamically extracted from the specified "Target services library" directory. Hence you can specify any directory that contains target configurations of the RT Services Library. For example, you can specify the `C++/TargetRTS` directory of the Model RealTime installation to use one of the target configurations that are shipped with the Model RealTime 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 Model RealTime 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 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. If you only need to use the included file in the capsule implementation file, you can instead add the `#include` in the "Implementation Preface" code snippet.

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 a response for 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 wants 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 four 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 the sender's (i.e. caller's) responsibility to allocate `RTMessage` objects that can hold the reply data. The sender is also responsible for deleting these objects when they are no longer needed. Use `RTMessage::isValid()` to ensure that a reply object corresponds to a valid reply made by the receiver.
3. 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.
4. 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 as argument 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 = new RTMessage[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
    }
}
delete[] replies;
```

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 `unhandledMessage()` is called on the receiver capsule class. In almost all cases unhandled messages are unexpected, and in those cases another virtual function `unexpectedMessage()` is called. 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 and/or sent a large number of times, it may be too costly to copy it. See [Avoiding to Copy Message Data](#) for ways how you can avoid message data from being copied.

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.

When you develop your application, do not make the assumption that an `RTMessage` object only will be copied exactly once by the RT Services Library when an event is sent. There are situations when it will be copied multiple times. For example, when using the model debugger the RT Services Library creates notifications which carry a copy of the `RTMessage` object. It is therefore important that any data object that is sent as an event parameter has a type descriptor where the copy function (e.g. a copy constructor) is able to copy the data object multiple times. Note that copying of the message object may happen even after it has been consumed by its receiver. The receiver must therefore not change it in a way that will prevent it from later being copied correctly.

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 a transition can trigger on multiple events, with different types of event parameter data, `rtData` will be an untyped pointer and you have to cast it to the expected type.

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.

Avoiding to Copy Message Data

In many cases the data carried by a message will be small enough so that copying it will not lead to any performance concerns. However, if a larger piece of data needs to be sent it may be too costly to copy it. This is especially true if the data is not only sent once, but several times.

One solution for avoiding that the RT Services Library copies the message data could be to instead pass a pointer to the data as the event parameter. However, you then need to ensure that the data (that now is shared between the sender and the receiver) is sufficiently protected from simultaneous access from different threads (for example using a semaphore or mutex). You must also ensure that this data is deleted when it is no longer needed.

If the data is movable, a better solution can be to move the data instead of copying it. To make the data movable its type descriptor must define a move function. If the data type is a C++ class (compiled with a C++ 11 compiler) the usual way to make it movable is to ensure it has a move constructor (either explicitly defined, or automatically generated by the compiler). The type descriptor move function can then simply invoke the move constructor. For example:

```

(void)new( target ) MyClass( std::move(*source) );

```

See [Type Descriptors](#) for more information about type descriptors.

You decide at the time an event is sent if its data should be copied or moved, based on if an lvalue or rvalue reference to the data is used:

```

myPort.myEvent(data).send(); //Send by copy (lvalue ref to data)
myPort.myEvent(std::move(data)).send(); //Send by move (rvalue ref to data)

```

If you attempt to send the data by move, but it is not movable, the data will anyway be copied. Also note that if you send an event on a replicated port (i.e. a port with multiplicity > 1) the data can only be moved once. In this case the data will be moved for the last port instance send, and copied for all the others.

If the capsule that receives an event needs to store its data for later use, it can also avoid a data copy by moving the received data to for example a capsule attribute:

```
someAttr = std::move(*rtdata); // Avoid copying the message object
```

For `rtData` to be movable it must be declared as non-const. A property "Const `rtdata` parameter" on the transition controls whether `rtData` should be declared as const or not.

Properties

<Transition> t1

General Qualified Name: [Move_function_2::Inner::State Machine::t1](#)

Triggers Name:

Documentation Kind:

Stereotypes ☐ Const `rtdata` parameter

Constraints

Note that there are more situations where message data gets copied, for example when invoking an event (i.e. synchronous communication), or when providing initialization data when incarnating a capsule into an optional capsule part. Currently it's not possible to avoid copying message data in those situations.

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

There are many C++ logging libraries to choose from if you need to produce logs from your realtime application at runtime. The RT Services Library also includes a simple logging service which lets you log messages in a thread-safe way to `stderr` or `stdout`. The logging service can be accessed in two ways:

- By using a [log stream](#) (available since TargetRTS version 8010)
- By using a [log port](#), i.e. a port typed by the predefined Log protocol

Log streams have certain benefits over log ports and should be the default choice for new code:

- Only capsules can have ports, which means that log ports cannot be used for code that does not run in the context of a capsule, or where the context capsule is not readily available
- Log ports can only print messages to `stderr` while log streams also allow messages to be printed to `stdout`
- Log ports are only thread-safe at the lowest level where a single string is printed. If you need to print composite log messages that consist of multiple strings, and do this from multiple threads, it's better to use log streams since they give you better control over how to avoid interleaved log messages coming from different threads.
- It's easier to mix the logging of text and data values, and to format such log messages, when using log streams. This can make logs generated by log streams more readable.

Log Stream

There are two log streams available; `Log::out` for logging message to `stdout` and `Log::err` for logging messages to `stderr`. You can use these log streams in a very similar way to how other C++ output streams are used. Here are some examples:

```
Log::out << "Count: " << count << Log::endl;
```

Prints the string "Count: " followed by the value of the variable `count` to `stdout`. Then a newline is printed and the stream is flushed. As an alternative to `Log::endl` you can use the standard `std::endl`.

```
Log::err << "Is initialized: " << std::boolalpha << isInitd << std::flush;
```

Prints the string "Is initialized: " to `stderr` followed by "true" or "false" depending on the value of the boolean variable `isInitd`. Then the stream is flushed, without printing a newline.

Note the use of standard C++ manipulators in the examples above. You can use these since log streams are based on the standard `std::ostream`.

In addition to strings, log streams can print values of all primitive C++ types and predefined types provided by the TargetRTS (such as `RTString`). They can also print values of your user-defined types, provided they have a type descriptor. To do so, you need to provide both the value and the [type descriptor](#) using an `RTTypedValue` object:

```
Log::out << "My data: " << RTTypedValue(&m, &RTType_MyType) << std::endl;
```

Prints the string "My data: " to `stdout` followed by the string encoding of the value of `m` which has a user-defined type `MyType`. Then a newline is printed and the stream is flushed.

Two special manipulators are provided for locking the log streams so that a thread has exclusive access to them. `Log::lock` will lock the log streams (both at once), and `Log::unlock` will unlock them (also both at once). While a thread has locked the log streams only that thread can print log messages to them. Other threads will be blocked if they attempt to lock the log streams while they are locked. For example:

```
Log::out << Log::lock << "This is a compound log message, " << "printed by  
thread " << context()->name() << Log::endl << Log::unlock;
```

Prints a compound log message to `stdout`. The logged message consists of three strings, and since the log streams are locked while they are printed, there is no risk for interleaved log messages if this code (or similar code that uses the log streams) also is run from other threads at the same time.

Remember to lock the log streams in all logging code that may run simultaneously from more than one thread, and do not forget to unlock them after the log message has been printed.

Log Port

A log port is a capsule port that is typed by the predefined Log protocol. It will in C++ become a member variable 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](#).

Note that to flush the log you must call `log.commit()`. If you forget to flush the log you may not see messages in the console until the underlying stream buffer is full.

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.

When setting timers, time values can also be specified using types from the `std::chrono` library. Such values will then be automatically converted to `RTTimespec` objects.

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).

```
timer.informIn(std::chrono::seconds(5));  
// or shorter with C++ 14 and using namespace std::chrono_literals;  
timer.informIn(5s);
```

Same as above but using the std::chrono library instead

```
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).

```
std::chrono::system_clock::time_point t = std::chrono::system_clock::now()  
+ std::chrono::seconds(5);  
timer.informAt(t);
```

Same as above but using the std::chrono library instead

```
timer.informEvery(RTTimespec(5, 500000000));
```

Sets a periodic timer to expire every 5.5 seconds. The first timeout will occur in 5.5 seconds from now (relative time).

```
timer.informEvery(std::chrono::milliseconds(5500));  
// or shorter with C++ 14 and using namespace std::chrono_literals;  
timer.informEvery(5500ms);
```

Same as above but using the std::chrono library instead

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 = timer.informIn(RTTimespec(10, 0));  
if (!tid.isValid())  
    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 `RTTimerNode` object in case setting the timer fails, and in this case the `RTTimerId` becomes invalid.

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.

The call to `incarnate()` allows you to pass initialization data for the capsule instance. Such data can be accessed by the incarnated capsule instance in its initial transition by means of the parameter called `rtdata`. If the thread where `incarnate()` is called is the same thread that will run the incarnated capsule instance, then the capsule's initial transition will run synchronously (blocking the caller). In this case you can modify the initialization data (provided you have unchecked the "Const `rtdata` parameter" checkbox for the initial transition) if you need to pass some information back to the caller. However, if the threads are different, the initial transition will run asynchronously based on dispatching an initialization event, and in that case you should not modify `rtdata`.

If you need to pass initialization data that is available already when the capsule instance is created, you need to define a custom capsule constructor and then instead use the `incarnateCustom()` function. This function allows you to provide the code (in the form of a lambda expression) that will invoke the 'new' operator where the initialization data can be passed to the constructor. For more information about custom capsule constructors, see the article "Custom capsule constructors" in the Model RealTime documentation.

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,...

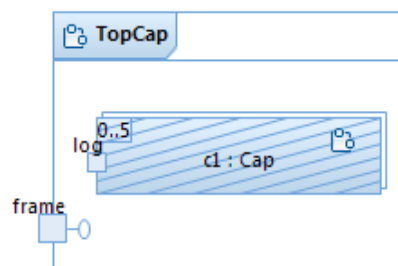
The `incarnate()` (and `incarnateCustom()`) functions return an `RTActorId` object which is a handle to the incarnated capsule instance. There are many things that may go wrong when incarnating a capsule so you should always check the validity of this handle by calling the `isValid()` function. Here 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()->pererror(context()->sterror());
}
```

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, new_RTTypedValue(
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.incarnateCustom(c1,
    RTActorFactory([this](RTController * c, RTActorRef * a, int index) {
        return new Cap_Actor(c, a, true); // User-defined constructor
    })
);
```

Incarnate "Cap" into "c1" at the first available index. A boolean value (true) is passed as initialization data in a call of a custom capsule constructor defined for "Cap".

```
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:

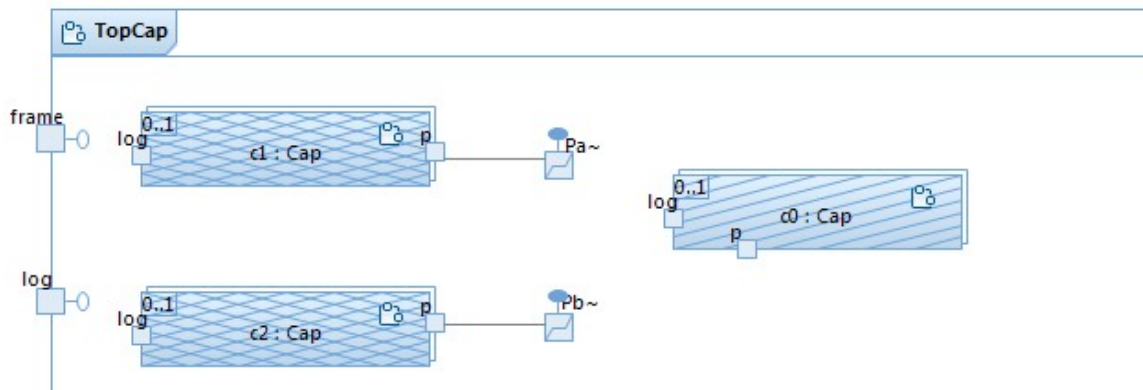
- The capsule instance must be valid. Use the `isValid()` function to check this.
- The capsule instance must be type compatible with the type of the capsule part.
- The capsule instance must not have a port that is already bound.

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 its 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");
log.commit();
```

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. Finally flush all printed log messages so they appear in the console.

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 is 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. External ports is a useful mechanism for integrating code generated by Model RealTime with external code that also is part of the real-time application. For example, you may have one part of your application that is responsible for reacting on external stimuli, for example bytes read from a socket or data read from a sensor. Such code may run in an external thread, and interact with the code generated from the UML model by means of external ports.

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.

Events sent to external ports can contain any data. Here is an example of sending a string with the raised event:

```
char* str = "external data";
extPort.raise(&str, &RTType_RTpchar);
```

External data is received in the transition that is triggered by the event called `event` of the External protocol. Here is an example of transition code that will receive the string sent above:

```
RTpchar p = *((RTpchar*) rtdata);
printf("Received external data: %s", p);
```

Sometimes data may become available in the external thread at a higher pace than what the capsule can (or want to) handle. In that case it's not convenient to pass the data to the capsule thread in the call of `raise()`. The external thread would have to maintain a data structure for storing the received data until the capsule is ready to receive it, and sending a complete data structure in the call of `raise()` would require the definition of a custom type descriptor to avoid copying it.

The external port service provides an alternative mechanism for data transfer that is more convenient in this situation. The external thread can just call an operation on the external port to push an arbitrary data object onto the external port itself. Any number of data objects can be pushed on the external port (i.e. it is not necessary that the capsule handles them one by one). Here is an example of storing a pair containing a string and an integer:

```
std::pair<std::string,int>* data = new std::pair<std::string,int>("external data", 15);
extPort.dataPushBack(data);
```

When appropriate the external thread can notify the capsule that external data is available by simply calling `raise()` without passing any arguments. If the call fails, the external thread can choose to wait a little and try again, or simply ignore it and not call `raise()` again until more external data becomes available. What to do depends on how urgent it is that the capsule gets notified about the availability of external data.

When the capsule is ready to handle the external data it can choose if it wants to handle all received data, or just some of it. It can also choose if it wants to handle the data in the same order as it arrived (FIFO) or to handle the most recently received data first (LIFO). Which approach that is best is application specific. Here is an example where it handles all received data from the above example in a FIFO manner:

```
unsigned int remaining;
```

```

do {
    std::pair<std::string,int>* data;
    remaining = extPort.dataPopFront((void**) &data);
    if (data == 0)
        break;

    // Handle received external data here...

    delete data;
}
while (remaining > 0);

```

Note that the external thread allocates the memory for the external data, while the capsule is responsible for deleting it once it has handled it.

Sometimes you may anyway choose to create a special data structure for external data. For example, you may want to avoid sending duplicate data to the capsule, or have other more specific requirements on the data structure. In that case the external thread and the capsule thread has to take care to access the shared data in a thread-safe manner. For example, you can protect it with a mutex. A mutex implementation is available for the different targets at

<InstallDir>/rsa_rt/C++/TargetRTS/src/target/<target>/RTMutex.h.

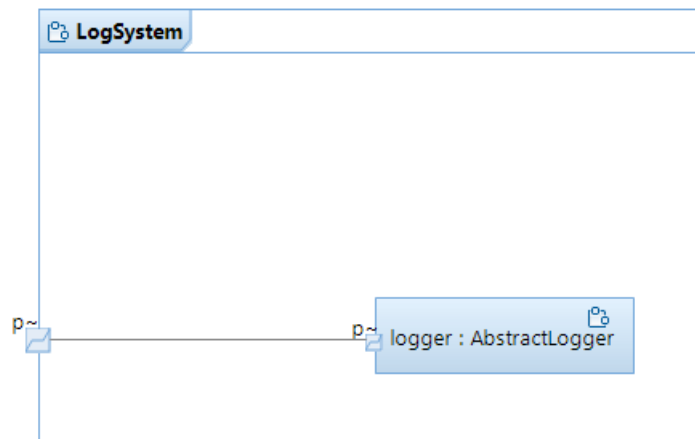
Dependency Injection Service

The TargetRTS provides a simple dependency injection service by means of the RTInjector class. You can use this service on its own, or together with a C++ dependency injection framework, to implement dependency injection in your real-time application.

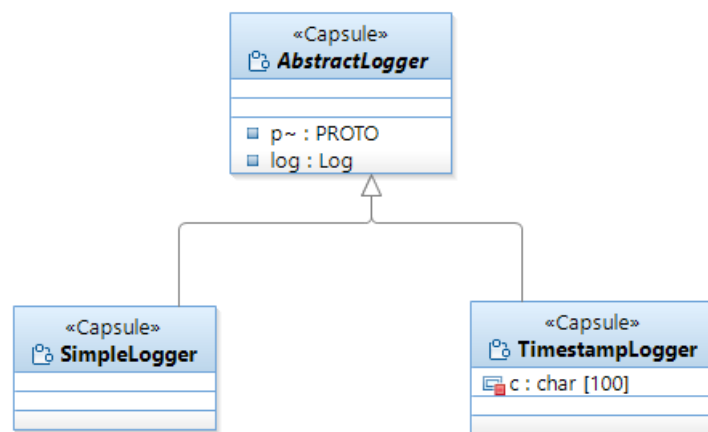
Note that "dependency" in this context refers to some form of run-time dependency, not the UML dependency concept used for expressing compile-time dependencies between elements. Examples of run-time dependencies include what capsule to use when incarnating a capsule part, what configuration data to pass to its capsule constructor, and which thread to run the incarnated capsule instance in.

The dependency injection service is strongly related to capsule factories (see [RTActorFactory](#)). A capsule factory can delegate requests to create a capsule instance to the dependency injection service. This makes it possible to centralize all dependencies of capsules in an application to a central location (for example the top capsule constructor).

To understand the value provided by the dependency injection service, let's first look at an example where it is not used. Assume we have a LogSystem capsule which internally uses a "logger" capsule part for implementing some logging functionality:



The application may have multiple implementations of logging functionality, represented by different subcapsules inheriting from **AbstractLogger**. For example:



Now assume that you sometimes want to use one logging implementation, and sometimes another. Perhaps you even want to switch dynamically from one implementation to the other depending on some run-time condition.

Without dependency injection it would be necessary to write code that decides what subcapsule of **AbstractLogger** to use when incarnating the "logger" capsule part. If the capsule part would be declared as optional such code would be a call to `RTFrame::incarnate()`, while if it's declared as fixed, you'll instead provide the code in a capsule factory, for example as a "Create Function Body" code snippet on the "logger" capsule part. The problem with both these approaches is that your application will contain hard-coded "configuration" code that is spread out in different places in the application. To change the application configuration to use different logging implementations you will have to find all such configuration code and change it.

With dependency injection it's possible to place such configuration code in a single place, possibly outside the application logic itself. For example, you can write code in the top capsule constructor which registers a create function to be used for

incarnating the "logger" capsule part:

```
RTInjector::getInstance().registerCreateFunction("/logger:0/logger",
    [this](RTController * c, RTActorRef * a, int index) {
        //return new SimpleLogger_Actor(c, a);
        return new TimestampLogger_Actor(LoggerThread, a);
    });
```

The capsule part is identified by means of its fully qualified run-time name (same as is used by the model debugger). Note that dependency injection not only let's us configure what capsule to create an instance of, but also what thread it should run in, and any initialization data to pass to its capsule constructor.

Note that the RTInjector class provides a singleton object that can be used anywhere from your application. You use this singleton both for registering create functions for capsule parts (like in the example above) and for creating a capsule instance from a capsule factory. For example, here is a simple capsule factory implementation that delegates to the RTInjector singleton for creating capsule instances:

```
#include <RTInjector.h>

class CapsuleFactory : public RTActorFactoryInterface {
public:

    RTActor* create(RTController *rts, RTActorRef *ref, int index) override {

        return RTInjector::getInstance().create(rts, ref, index);
    }

    void destroy(RTActor* actor) override {
        delete actor;
    }

    static CapsuleFactory factory;
};
```

If you need to change the configuration dynamically at run-time you can simply register another create function for the same capsule part. It will then override the previously registered create function.

Structure of Generated C++ Code

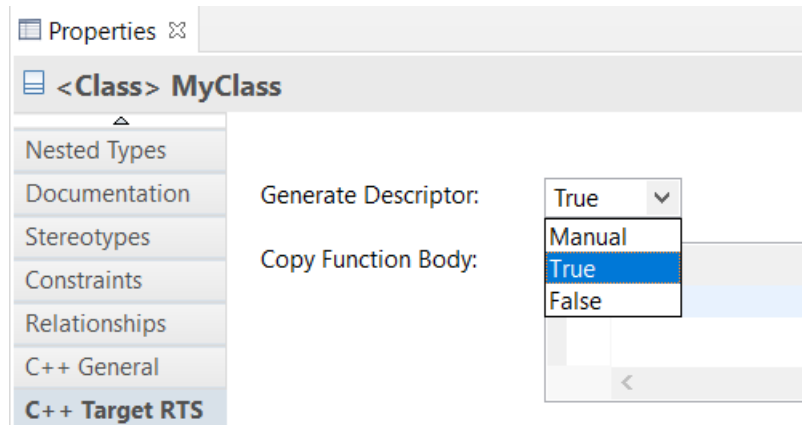
This chapter provides the information you need to know about the structure of the C++ code that is generated from an Model RealTime model. Generated C++ code makes extensive use of classes provided by the RT Services Library.

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.

If you set the property to "True" the model compiler will generate the type descriptor using the Function Body code snippets that come below the "Generate Descriptor" property. Note that it's mandatory to provide all function bodies except Move Function Body, which you only need to implement if you want the type to be movable (see [Avoiding to Copy Message Data](#) for an example when a movable type can be useful).

Note that in many cases the model compiler can generate a default type descriptor implementation and if your needs are covered by that default implementation you don't need to implement any of the Function Body code snippets.

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](#).

Type Descriptor Hints

As mentioned above the model compiler can generate type descriptors automatically for all types that are simple enough. This includes enumerations and classes with attributes typed by other simple types. However, it cannot do it for more complex types. One common case is that classes in the model are translated to typedefs or type aliases of STL container types, such as lists, vectors or maps. Such types are too complex for the model compiler to automatically generate a type descriptor for. However, there is a preference *RealTime Development – Build/Transformations – C++ – Generate type descriptors for complex types* which can be set to let the model compiler attempt to generate type descriptors also for such types. For this to work you need to specify the “Type Descriptor Hint” property so the model compiler can know something about the type. The following directives are recognized in this property:

- `@kind=vector|map|list|set`
Kind of STL container
- `@itemType=identifier|qualifiedIdentifier`
Item type of STL container (for list/set/vector). If the type is a nested type, you should use its fully qualified name, for example `MyClass::MyNestedClass`.
- `@keyType=identifier|qualifiedIdentifier`
`@dataType=identifier|qualifiedIdentifier`
Key/data types of STL container (for maps)

As an example, assume there is a typedef class with implementation type `std::map<A,B>`, or a similar type alias. Then the “Type Descriptor Hint” property of this typedef (or type alias) should contain the lines:

```
@kind=map
@keyType=A
@dataType=B
```

Templates

Contrary to typedefs a C++ type alias may have template parameters. In that case the generated type descriptor will also have the same template parameters. This

ensures that for each concrete instantiation of the type alias, a corresponding type descriptor instance will also be available.

The type descriptor functions will then be generated as template functions which makes it possible to implement them generically so they can work regardless of actual template parameter being used. If necessary you can specialize some of these template functions to use a special implementation for a certain set of actual template parameters.

When implementing a generic type descriptor for a type alias with template parameters it's often useful to be able to lookup the type descriptor of the actual type template parameters being used. The TargetRTS provides a template function `RTObject_class::fromType<T>()` for doing this. As an example assume we have defined the following type alias:

```
template<typename T, unsigned int N > using StdArray = std::array<T, N>;
```

We can then implement the encode function for this type alias so it can encode all kinds of arrays, regardless of the actual element type being used:

```
template<typename T, unsigned int N > inline int rtg_StdArray_encode( const
RTObject_class * type, const StdArray< T, N > * source, RTEncoding * coding )
{
    //{{{USR
    platform:/resource/type_descriptor_with_template_parameter/CPPModel.emx#_gcMGALo_E
    eu4j48Uy6dLVQ|Target RTS|encodeFunctionBody
    const RTObject_class *elementTypeDescriptor = RTObject_class::fromType<T>();
    if (!elementTypeDescriptor)
        return 0; // Element type descriptor not available
    int sum = 0;
    bool first = true;
    sum += coding->write_string(type->name());
    sum += coding->write_string("{");
    for (auto i = source->begin(); i != source->end(); i++) {
        if (!first)
            sum += coding->write_string(",");
        first = false;
        T element = *i;
        sum += elementTypeDescriptor->encode(&element, coding);
    }
    sum += coding->write_string("}");
    return sum;
    //}}}USR
}
```

The TargetRTS provides an implementation of the `fromType<T>()` template function for each built-in C++ type. These template specializations are found in the file `RTObject_class.h`. To make this function work also for other types, such as user-defined types or types provided by the TargetRTS, you need to write a similar specialization for them. For example:

```
template <> inline const RTObject_class* RTObject_class::fromType<StdString>() {
```

```
return &RTType_StdString; } // User-defined type: StdString
template <> inline const RTOBJECT_CLASS* RTOBJECT_CLASS::fromType<RTString>() {
return &RTType_RTString; } // Type provided by the TargetRTS: RTString
```

The generated code will define a name for the type descriptor which by default is shared with all instantiations of the template. It's set to the name of the type alias. For the above example:

```
template<typename T,unsigned int N> const char* RTName_StdArray<T,N>::name =
"StdArray";
```

You can specialize this variable definition to use a more specific name for the type descriptor for each instantiation of the template that you use. For example:

```
template <> const char* RTName_StdArray<StdString, 4>::name = "StdArray<StdString,
4>";
template <> const char* RTName_StdArray<RTString, 2>::name = "StdArray<RTString,
2>";
```

If you don't do this, and the template is instantiated more than once, there will be multiple type descriptors with the same name. This is allowed but a warning will be printed at run-time. For example:

```
WARNING: A type "StdArray" was already installed
```

There is a function `RTOBJECT_CLASS::lookup()` which can return a type descriptor from the name of the type it describes. This function will fail if there are multiple type descriptors sharing the same name, so it's recommended to set a unique name for each type descriptor.

Threads

One of the strengths of the RT Services Library is how easy it is to change the threading configuration of a generated real-time application. This is mainly accomplished by separating actual physical threads used from conceptual logical 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":

Physical threads:

-  CustomThread
-  MyThread
-  MainThread
-  TimerThread

Thread properties:

Name: CustomThread

Stack size: 20000

Priority: DEFAULT_MAIN_PRIORITY

Implementation class: RTPeerController

Logical threads:

Logical thread	Physical thread
MyThread	CustomThread

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

This chapter describes some important things to know about how the RT Services Library works. It also covers some optional utilities that your application can take advantage of.

Run-to-Completion Semantics

The RT Services Library does not preempt capsule processing. The heart of the RT Services Library are the controller objects (run by the physical threads) that are responsible for dispatching messages to the capsule instances it manages. A controller object has a message queue where it stores messages targeted at any of the managed capsule instances. The basic mode of operation of a controller object 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 triggered transition has completed processing the message and run to completion. 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 scheme 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 physical thread.

The RT Services Library implements a simple message dispatch algorithm: Find the highest-priority, non-empty message queue of the available controller objects. Then 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.

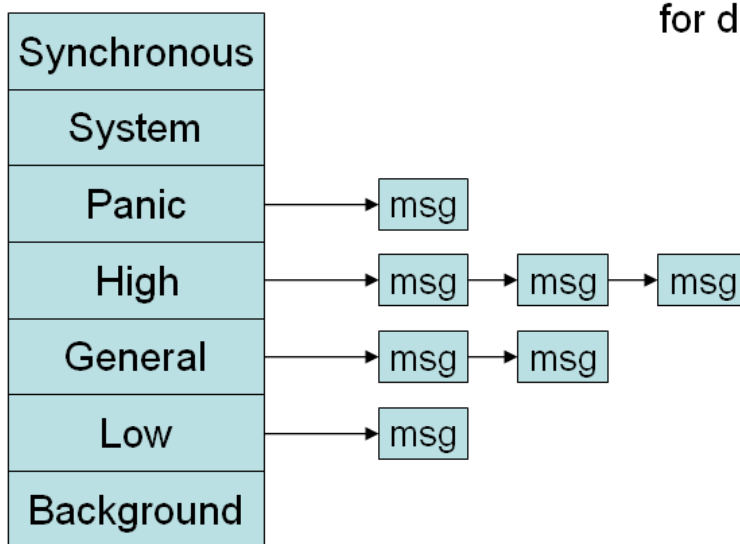
A message is delivered via a call to `rtsBehavior(signal, port)`. This call is made inside the message dispatch loop. Run-to-completion semantics is enforced as the control is not returned to the loop until the capsule instance completes execution and makes a "return".

Message queues

Each physical thread has got its own `RTController` object with its own message queue. More precisely, this is an array of message queues, one for each priority level. The controller object organizes all its message in two kinds of such 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.

`internalQ[OTRTS_NUMPRIO]` – an array of message queues for different priorities



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`:

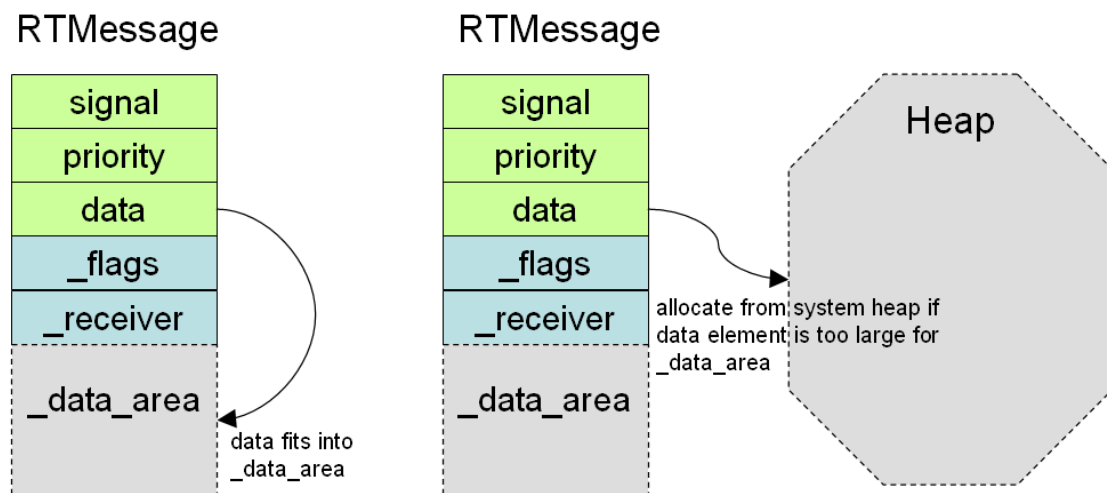
```
RTMessageQ  internalQ[ OTRTS_NUMPRIO ];
```

```
RTMessageQ  incomingQ[ OTRTS_NUMPRIO ];
```

Message structure and freeList of messages

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

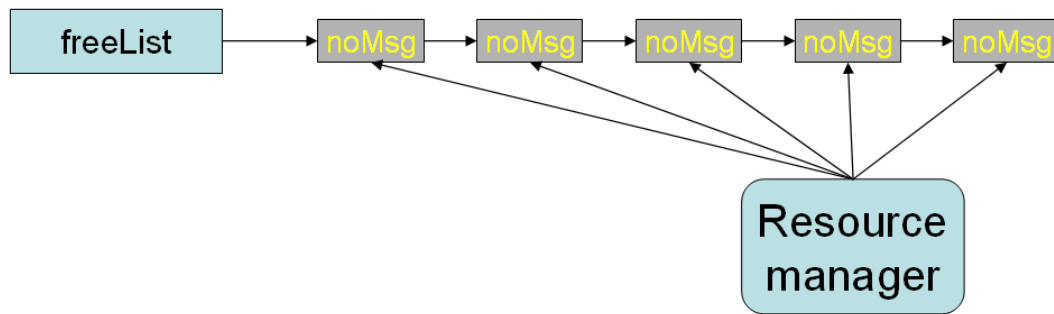
If the data to be sent fits into the message body data area, it will simply be copied into the message. If the data does not fit inside the payload area, memory for the data will be allocated in the system heap and freed after the message has been received. The size of the data area (`RTMESSAGE_PAYLOAD_SIZE`)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



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 )
```

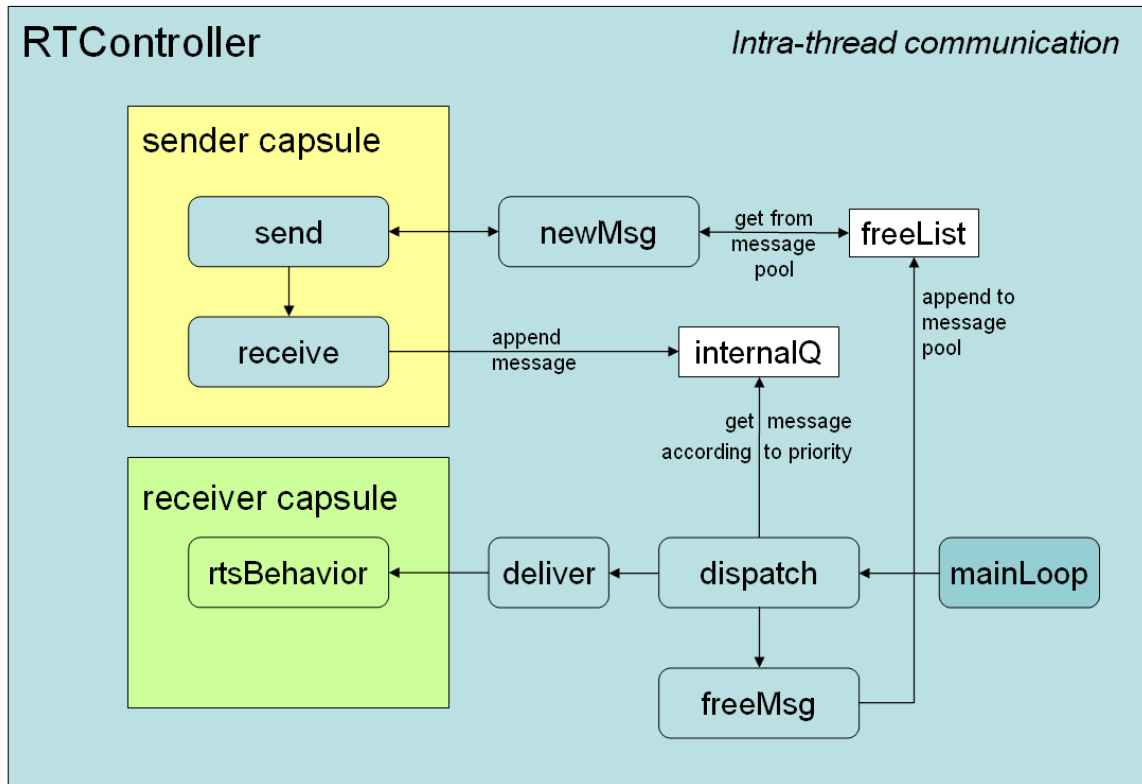
By default, 50 new messages are added. Note that once allocated, message objects are not freed back to the system heap but returned to the message pool.

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.

The 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.

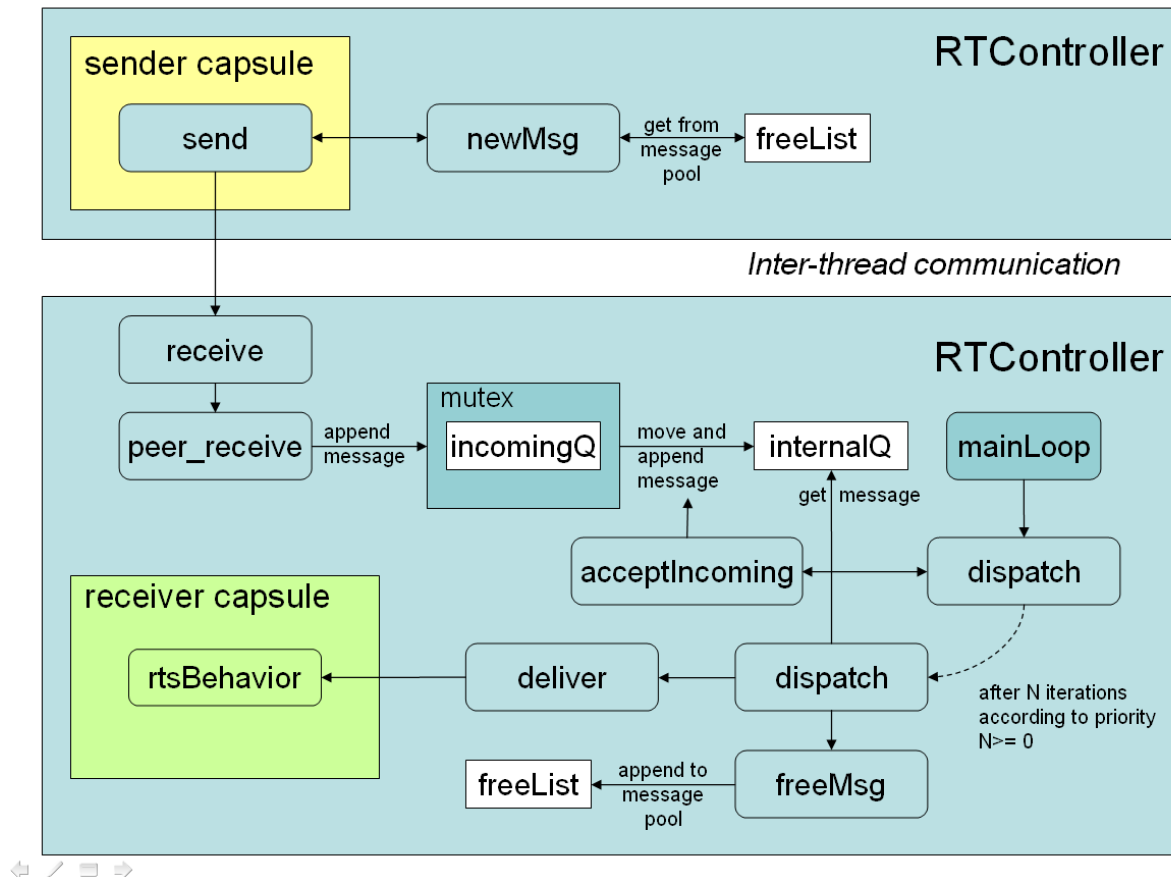
Intra-thread message sending

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

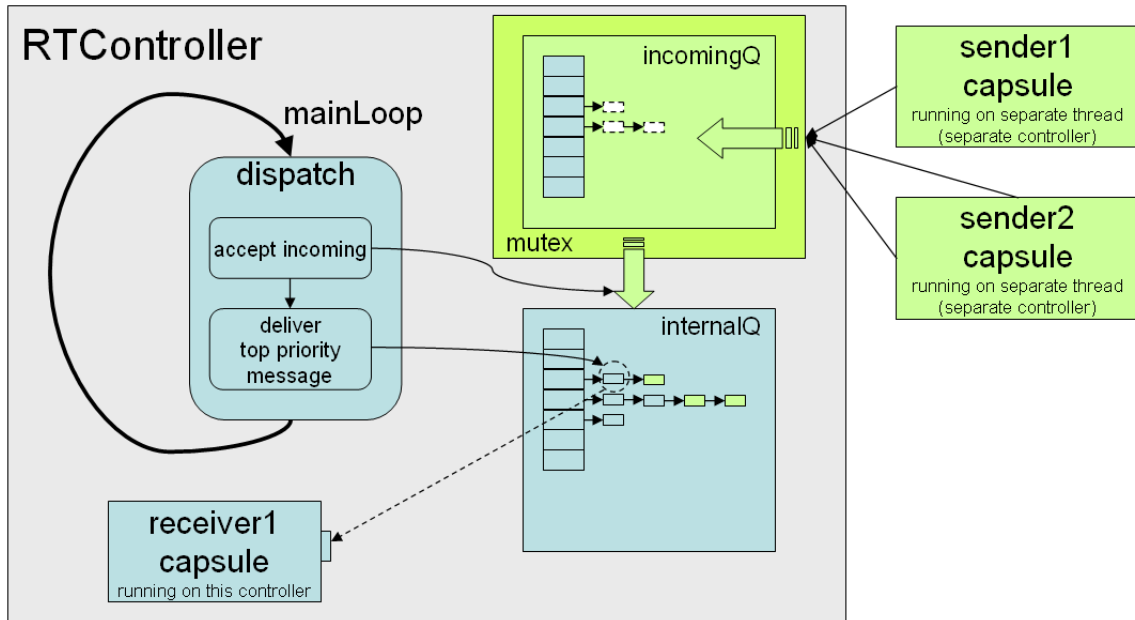
For inter-thread communication, `RTController::send` is called on the sender's controller/thread, and `RTController::receive` will be called on the receiver's controller/thread which in this case is a different object. `RTController::receive` will call `RTController::peer_receive` that will put a message 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.

Encoding and Decoding

The RT Services Library can encode messages and data values to a textual representation. This can be useful in many situations, such as when tracing information about what happens in the real-time application (for example using the model debugger) or when sending events and data outside the current process (for example when implementing a distributed application, or communicating with a cloud server). Likewise, it can also decode such a textual representation to obtain an in-memory copy of the original message or data value.

Support for encoding and decoding can be removed from the RT Services Library by undefining the macros `OBJECT_ENCODE` and `OBJECT_DECODE` respectively.

Encoding and decoding is performed by means of the `encode` and `decode` functions that are part of the [Type Descriptor](#) of a type. These functions are defined like this:

```
typedef int (*RTEncodeFunction)(const RTOBJECT_class * type,
                               const void * source,
                               RTEncoding * );
```

```
typedef int (*RTDecodeFunction)(const RObject_class * type,
                                void                * target,
                                RTDecoding           * );
```

The last parameter in these function types specifies the encoding or decoding object to use for performing the encoding or decoding. The default encoding/decoding implementation uses a compact ASCII representation (implemented by the classes `RTAsciiEncoding` and `RTAsciiDecoding`). You will for example see this representation being used when tracing using the model debugger:

You can manually invoke the encode function if you need a textual representation of a data value. For example, here is the code for encoding the data of a received message and printing it to stdout:

```
const RObject_class* mt = msg->getType();
char buf[1000];
RTMemoryOutBuffer buffer( buf, 1000 );
RTAsciiEncoding coding( &buffer );
mt->_encode_func(mt, msg->getData(), &coding);
buffer.write("", 1); // IMPORTANT: Terminate the buffer string before printing it!

std::cout << "ASCII encoding: " << buf << endl << flush;
```

Here it is assumed that the encoded string will contain less than 1000 characters, so a fixed-size memory buffer is used when encoding the data. If you don't want to make an assumption about the size of the encoded string, you can instead use the class `RTDynamicStringOutBuffer` which implements a dynamic string buffer.

In addition to the default ASCII encoding, the RT Services Library implements two more encodings:

- **Versioned ASCII representation**
This is implemented by means of `RTVAsciiEncoding` and is identical to `RTAsciiEncoding` except that the version number of the type descriptor is included. The corresponding decoder is implemented in `RTVAsciiDecoding`.
- **JSON representation**
This is implemented by means of `RTJsonEncoding`. Encoding to JSON can for example be useful when data is sent to a web server or to another real-time application which expects JSON as a standard data format. The corresponding decoder is implemented in `RTJsonDecoding`.

The JSON encoding fulfills the same API as the ASCII encoding, but in addition also provides a function for encoding an entire message object. Here is an example that encodes a received message to JSON using the `RTDynamicStringOutBuffer` mentioned above.

```
RTDynamicStringOutBuffer buf;
RTJsonEncoding coding(&buf);
coding.put_msg(msg);
cout << "Received msg: " << buf.getString() << endl << flush;
```

The JSON encoding of the event shown in the trace example above will look like this:

```
{ "event" : "event_with_class",  
  "type" : "MyClass",  
  "data" : { "a" : 8, "b" : false }  
}
```

Neither `RTJsonEncoding` nor `RTDynamicStringOutBuffer` is included by default, so you have to explicitly include these files from the RT Services Library if you want to use them.

If you need to encode and/or decode messages or data values to/from another representation, you can create your own class that inherits from `RTEncoding` and/or `RTDecoding`. These classes contain virtual functions that can be overridden to implement a custom encoding and/or decoding.

For more information about encoding and decoding, refer to [this page](#).

C++ RT Services Library Class Reference

In this chapter some important classes of the RT Services Library are described. You often need to use these within the action code of a capsule to access the services provided by the RT Services Library.

This reference does not describe private or restricted functions and variables from the RT Services Library. Some features and classes in the RT Services Library are internal to the library itself and are thus not intended to be accessed directly from your application code.

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 RT 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.

<pre>RTController * context(void);</pre>	<p>Gets the controller for the physical thread on which a capsule instance is executing.</p> <p>There are some public functions on the <code>RTController</code> class that can be accessed this way. In particular, you may find it useful for printing error information, as in the example below:</p> <pre>if(! port.ping().send()) { log.show("Error on physical thread: "); log.log(context()->name()); context()->perror("send"); }</pre>
<pre>const char * getCurrentStateString (void) const;</pre>	<p>Gets the name of the currently active state in the capsule's state machine.</p>
<pre>RTController::Error getError(void) const;</pre>	<p>Returns the most recent error within the current thread. The Error enumeration is defined within the <code>RTController</code> class.</p> <p>The error code is not reset by a subsequent successful primitive function call. It should be called immediately following the failure of an RT Services Library function call.</p>
<pre>int getIndex(void) const;</pre>	<p>Gets the replication index of this capsule instance in its "home" role (i.e. the capsule part where it was incarnated, or imported to). The</p>

	replication index is zero (0) based.
<pre>const char * getName(void) const;</pre>	Gets the name of the capsule part in which this capsule instance is running (where it was incarnated, or imported to).
<pre>const char * getTypeName(void) const;</pre>	Returns the name of the capsule from which this capsule instance was incarnated.
<pre>int isType (const char class_name) const;</pre>	<p>Queries the capsule class of this capsule instance. Returns 1 (true) if this capsule instance is of class <code>class_name</code>, and 0 (false) otherwise.</p> <pre>if (isType("PhoneManagerCapsule")) { log.log("This capsule role is of type: "); log.log(getTypeName()); }</pre>
<pre>virtual void logMsg(void);</pre>	<p>This function is called by the RT Services Library before a received message is processed by a capsule instance (if so configured).</p> <p>As implemented by the <code>RTActor</code> 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.</p> <p>To override this function, simply add a new operation to the capsule with the same name and prototype. It can also be overridden for the entire application by creating a new <code>RTActor::logMsg()</code> function, compiling it, and including it in the model using the model link options.</p>
<pre>const RTMessage * msg; const RTMessage * getMsg(void);</pre>	<p>The <code>msg</code> variable can be accessed via the <code>getMsg</code> function. A pointer to the message is returned.</p> <p>Every capsule class has a variable <code>msg</code> which contains a pointer to the current message delivered to a capsule instance. This attribute can be used within transition code to retrieve the message that was dispatched to the capsule instance.</p> <p>Examples:</p> <p>Retrieve the void * pointer to the data portion of the message:</p> <pre>const void *data_ptr = msg->data;</pre> <p>You can also use <code>getMsg()</code> to access the current message:</p> <pre>const void *data_ptr = getMsg()->getData();</pre> <p>For most cases, the data can be accessed directly using the <code>rtda</code> parameter that is passed to every transition code snippet:</p> <pre>// The following is commonly needed to make a copy of the data // that was sent with a message const ADataClass & data1 =</pre>

	<pre>*((const ADataClass *)getMsg()->getData());</pre> <p>// the above statement can be written using the rtdata // parameter available in all state transition segments. const ADataClass & data1 = *rtdata;</p>
<pre>virtual void unhandledMessage (void);</pre>	<p>This function is called by the RT Services Library when there is no transition to trigger by the current message that is about to be dispatched. This happens when the capsule's <code>rtsBehavior()</code> function is called to process a message and no corresponding trigger event is found. The default <code>unhandledMessage()</code> implementation checks if the message was received before the capsule has been initialized. This is an unusual case, but can happen in certain special situations. If so it calls <code>messageReceivedBeforeInitialized()</code>. Otherwise it calls <code>unexpectedMessage()</code>.</p> <p>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 <code>RTActor::unhandledMessage()</code> function, compiling it, and including it in the model using the model link options.</p>
<pre>virtual void unexpectedMessage (void);</pre>	<p>This function is called when the RT Services Library has concluded that the current message was unexpectedly received. The default implementation prints a message to <code>stderr</code>. This function can be overridden on a capsule class basis to provide any other functionality that may be required when an unexpected message is received.</p> <p>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 <code>RTActor::unexpectedMessage()</code> function, compiling it, and including it in the model using the model link options.</p>
<pre>virtual void messageReceivedBeforeIn itialized (void);</pre>	<p>In certain special situations a capsule may receive a message before it has been initialized, and then this function is called. The default implementation saves the message in the defer queue so it is not lost and later can be recalled when the state machine is about to be initialized. This function can be overridden on a capsule class basis to provide any other functionality that may be required, for example to print an error if your application design doesn't expect this situation to occur.</p> <p>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 <code>RTActor::messageReceivedBeforeInitialized()</code> function, compiling it, and including it in the model using the model link options.</p>
<pre>virtual void</pre>	<p>This function is called by the RT Services Library when a state in a</p>

<pre>rtgStateEntry (void);</pre>	<p>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.</p> <p>To override this function, simply add a new operation to the capsule with the same name and prototype.</p>
----------------------------------	--

RTActorClass

An instance of this struct is created to represent the common external features (e.g. service ports and capsule name) of each capsule in your model. It acts as the type descriptor for a capsule. Only one instance of an `RTActorClass` structure exists for all instances of a particular capsule. This way common metadata about the capsule can be stored only once.

You can reference this capsule information object in your code, by referencing it by name (the name of the `RTActorClass` object is the same as the name of the capsule).

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 that 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 only reference existing objects.

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

This struct 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.

RTActorFactory

The `RTActorFactory` class implements a creation policy for capsule instances where they are created by a user provided "create" function, and deleted using the regular "delete" operator. It is used when incarnating capsule instances into optional capsule parts using the `RTFrame::incarnateCustom()` function. The custom "create" function allows you to, for example, pass additional arguments to a custom capsule constructor. See [Working with Optional Capsule Parts](#) for an example.

`RTActorFactory` is an implementation of the `RTActorFactoryInterface` abstract

class. There is also another implementation of this abstract class called `RTDefaultActorFactory` which implements a default policy for creating and destroying capsule instances. The generated code uses that class as the base class when a Create or Destroy function body has been specified for a fixed capsule part. This allows for invoking a custom capsule constructor also when incarnating a fixed capsule part. There is also a variant of this class called `RTDefaultActorFactory_NoCreate` which is used in case the capsule that types the capsule part has no default create function. This is the case if it is abstract or lacks a default capsule constructor.

For more information about actor factories and custom capsule constructors, see the article "Custom capsule constructors" in the Model RealTime documentation. You can find this document in the built-in Help under *Model RealTime User's Guide – Articles – Modeling realtime applications*.

RTActorRef

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

You can reference this capsule part in 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 );
```

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

2. To find the replication factor (i.e. size) of a capsule part

<pre>int size(void)const;</pre>	<p>Returns the replication factor (size) of a capsule part. The function returns the size whether or not there is a capsule instance currently incarnated at a specific slot. That is, it returns the maximum number of capsule instances that can fit in the capsule part.</p>
-----------------------------------	---

RTActorId

The Frame service functions `Frame::incarnate` return an object of type `RTActorId` to identify a particular capsule instance. The `RTActorId` object 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 its `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 is not destroyed when accessing the pointer.

<pre>int isValid(void) const;</pre>	<p>Returns 0 (false) if the id refers to an invalid capsule instance, and 1 (true) otherwise.</p> <p>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 <code>Frame::incarnate</code> (or <code>Frame::incarnateCustom</code>) function. Once the capsule instance has been created, <code>isValid</code> always returns 1 (true), even if the capsule instance is subsequently destroyed.</p> <p>The example below shows how the capsule instance 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.</p> <pre>RTActorId capsule_id = frame.incarnate(terminal, LongDistanceTerminal, RTTypedValue(), callThread, 0); if(! capsule_id.isValid()) context()->pperror("Incarnation failed: "); else frame.destroy(capsule_id);</pre>
-------------------------------------	--

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 application. 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 RT 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 RT 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.

<pre>void abort(void);</pre>	<p>Calling this function on any controller will terminate the controller on which the capsule instance is running which in turn destroys all</p>
--------------------------------	--

	<p>capsule instances running on that controller. Messages that are waiting in the controller to be dispatched are deleted.</p> <p>If this is called on the main thread, then all threads are destroyed and the process quits.</p> <pre>context()->abort();</pre>
<pre>Error getError(void) const;</pre>	<p>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 an RT Services Library function call.</p>
<pre>const char * name(void) const;</pre>	<p>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 thread specification in the transformation configuration. This method is a way of allowing capsules to find out what thread they are running on.</p>
<pre>void perror(const char * error_string = "error");</pre>	<p>The optional parameter <code>error_string</code> is the string to be printed to <code>stderr</code> along with the current error string as returned by the <code>RTController::strerror</code> function. By default, the string "error" will be printed.</p> <pre>if(! aPort.ack().send()) context()->perror("Error sending ack");</pre> <p>Output: Error sending ack: Port not connected.</p>
<pre>const char * strerror(void) const;</pre>	<p>Returns a description of the current error code on the current <code>RTController</code>, that is, the controller for a physical thread.</p>

RTExceptionSignal

The Exception Service, like other run-time system services, is accessed through an exception port. Exceptions manifest themselves in the form of RT 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 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 RT Services Library does not automatically raise any exceptions by itself.

With `userError()`, you can provide any relevant data that is required to be sent 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();

```

RTFrame

The Frame service is accessed via Frame ports, declared in the structure of a capsule class. Frame ports are instances of the class `RTFrame`. 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 `RTActorId` (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 otherwise returns 0.

```
const char * className
( const RTActorClass & );
```

The function returns the name of the specified capsule class in the form of a null-terminated string.

	<p>The RT 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, having the same name as that of the capsule that it represents.</p>
<pre>const RTActorClass & classOf (const RTActorId & incarnation);</pre>	<p>The function returns the capsule class of the specified capsule instance. If an error occurs the <code>EmptyActorClass</code> is returned.</p>
<pre>int deport(const RTActorId & instance, RTActorRef & role);</pre>	<p>Removes a capsule instance from a plug-in capsule part.</p> <p><code>instance</code> is the id of the capsule instance to be removed. <code>role</code> is the capsule part from which the capsule instance is to be removed.</p> <p>The function returns false (0) if an error occurred, and true (non-0) otherwise.</p> <p>The function can fail if:</p> <ul style="list-style-type: none"> the capsule instance being removed was not present in the capsule part the capsule part is not uniquely identified.
<pre>int destroy (RTActorId & instance); int destroy (RTActorRef & role);</pre>	<p>Using <code>destroy</code> with the capsule id will destroy the capsule instance and all of its contained capsule parts.</p> <p>Instead of destroying only one capsule instance, you can destroy all instances contained in a capsule part. The capsule part can only be destroyed by the immediate container of that part.</p> <p>These functions return false (0) if an error occurred, and true (non-0) otherwise.</p> <p>Example:</p> <p>Receive the capsule instance identifier from the capsule instance to destroy:</p> <pre>RTActorId cid = rtdata; frame.destroy(cid);</pre> <p>Or you can destroy all instances by specifying the capsule part instead:</p> <pre>frame.destroy(terminal);</pre>
<pre>int import (const RTActorId & instance,</pre>	<p>Imports a capsule instance into a plug-in capsule part.</p>

<pre> RTActorRef & dest, int index = -1); int import (RTActorRef & actor, RTActorRef & dest, int index = -1); </pre>	<p><code>instance</code> is the instance id of the capsule instance which is to be imported.</p> <p><code>dest</code> 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.</p> <p><code>index</code> 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.</p> <p>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.</p> <p>These functions return false (0) if an error occurred, and true (non-0) otherwise.</p> <p>The function fails in the following cases:</p> <ul style="list-style-type: none"> • if the capsule instance 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.
<pre> RTActorId incarnate(RTActorRef & role); RTActorId 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); </pre>	<p>Creates capsule instances in optional capsule parts. This function can be used to create and run capsule instances on different logical threads.</p> <p>The functions return a valid <code>RTActorId</code> if the call is successful. To test if the call failed, use the <code>RTActorId::isValid</code> function on the returned object. For example:</p> <pre> RTActorId ind = frame.incarnate(mySubcapsule); if (ind.isValid()) ...//use index else ...//getError() to see what's wrong, don't use index </pre> <p>If the function fails you can use the <code>RTActor::getError()</code> function to find out why it failed.</p> <p>There are alternate forms of the <code>incarnate</code> function which leave out the <code>RTActorClass</code> parameter (defaulting to the class specified for the capsule part) and for sending different types of initialization data to the new instance, either as <code>RTDataObject</code></p>

<pre> 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); RTActorId incarnateCustom(RTActorRef & role, RTActorFactory & factory, int index = -1); RTActorId incarnateCustom(RTActorRef & role, RTActorFactory && factory, int index = -1); RTActorId incarnateCustom(RTActorRef & role, const void * data, const RTOBJECT_class * type, RTActorFactory & factory, int index = -1); RTActorId incarnateCustom(RTActorRef & role, const void * data, const RTOBJECT_class * type, RTActorFactory && factory, </pre>	<p>classes or anything with a type descriptor.</p> <p><code>role</code> is the name of the optional capsule part contained in the structure of the capsule instance making the incarnate call.</p> <p><code>capsule_class</code> [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 <code>EmptyActorClass</code> to explicitly specify that the incarnated class defaults to the class of the capsule part.</p> <p><code>data, type, rtdata, info</code> [optional] is the data to be sent to the created capsule instance. The data sent is accessible in the capsule instance's initial transition. Be sure to specify if no data is to be sent. See the examples below.</p> <p><code>log_thread</code> 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.</p> <p><code>index</code> 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 the first free slot.</p> <p>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, ...</p> <p>The <code>incarnateCustom</code> function works in a similar way but allows you to provide an <code>RTActorFactory</code> to more exactly control (using code) how to create the capsule instance. There is also an overload of <code>incarnateCustom</code> which accepts a temporary <code>RTActorFactory</code> object which allows you to define the factory object inline.</p> <p>To use the frame functions, create a port using the <code>Frame</code> protocol.</p> <p>Examples:</p> <p>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</p>
---	---

<pre>int index = -1);</pre>	<p>capsule part:</p> <pre>RTActorId capsule_id; capsule_id = frame.incarnate(terminal); if(! capsule_id.isValid()) context()->perror("Incarnation failed: ");</pre> <p>If you want to incarnate the incarnated class of the capsule part and send initialization data you can specify the <code>EmptyActorClass</code> variable as the second argument:</p> <pre>RTActorId capsule_id; ControlData data(15, 8.98); capsule_id = frame.incarnate(terminal, EmptyActorClass, &data, &RTType_ControlData, (RTController *)0, -1);</pre> <p>This will incarnate a capsule instance into the optional capsule part at index 0, with initialization data, on a specific logical thread.</p> <pre>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: ");</pre> <p>The following could be used to incarnate a capsule instance without initialization data, but with a specific logical thread or a replication index:</p> <pre>RTActorId capsule_id; PrinterData data(14, "ott05"); capsule_id = frame.incarnate(device, Printer, (const void *) 0, // initialization data (const RObject_class *) 0, // type descriptor PrintThread, // logical thread 0); // replication index if(! capsule_id.isValid()) context()->perror("Incarnation failed: ");</pre>
<pre>RTActorId incarnationAt (const RTActorRef & part, int index);</pre>	<p>Retrieves a particular capsule instance of a capsule part.</p> <p>Returns the instance id of the capsule instance at the specified</p>

	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();
```

<code>int purge(void);</code>	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.
<code>int purgeAt(int index);</code>	<p>If a port is replicated then this function will delete deferred messages for the event on the specified port instance only.</p> <p><code>index</code> is a port instance index on which to delete deferred messages. Returns the number of messages that were deleted from the defer queue.</p>
<code>int recall(int front = 0);</code>	<p>Recall one deferred message for the event on all port instances.</p> <p><code>front [optional]</code> 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.</p> <p>Returns the number of recalled messages.</p> <p>There is no time-limit on deferral. Applications must take precautions against forgetting messages on defer queues.</p> <p>This function recalls the first deferred message of this event on any port instance. To recall the first message of any event, use the <code>RTProtocol::recall</code> function.</p>
<code>int recallAll</code>	Recall all deferred messages for the event on all port instances.

<pre>(int front = 0);</pre>	<p>Returns the number of recalled messages.</p> <p>There is no time-limit on deferral. Applications must take precautions against forgetting messages on defer queues.</p> <p>This function recalls ALL deferred messages of this event on ALL port instances. To recall all messages of any event, use the <code>RTPProtocol::recallAll</code> function.</p>
<pre>int recallAt (int index, int front = 0);</pre>	<p>Recall one deferred message for the event on a specific port instance. Returns the number of recalled messages.</p> <p>There is no time-limit on deferral. Applications must take precautions against forgetting messages on defer queues.</p> <p>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 <code>RTPProtocol::recallAt</code> function.</p>
<pre>int recallAllAt (int index, int front = 0);</pre>	<p>Recall all deferred messages for the event on a specific port instance. Returns the number of recalled messages.</p> <p>There is no time-limit on deferral. Applications must take precautions against forgetting messages on defer queues.</p> <p>This function recalls ALL deferred messages of this event on a specific port instance. To recall all messages of any event, use the <code>RTPProtocol::recallAllAt</code> function.</p>

RTLog

The Log service is accessed via [ports typed by the Log protocol](#) or by means of [log streams](#). Log ports become in C++ instances of the TargetRTS struct `Log::Base`. A log port only provides functions to call and does not pass any information in the reverse direction. Two static log stream objects are also provided, `Log::err` and `Log::out`, for writing log messages to `stderr` and `stdout` respectively.

Currently all log service output is directed to `stderr`. (for log ports) and `stderr` or `stdout` (for log streams). It's currently not possible to print log messages to files. This means that the `open()`, `clear()`, and `close()` functions should not be used.

Note that to flush the log you must call `log.commit()` (for log ports) or use the `Log::endl` or `std::endl` manipulator (for log streams). If you forget to flush the log you may not see messages in the console until the underlying stream buffer is full.

<pre>void show(const char * data); void show(char data); void show(double data);</pre>	<p><code>log</code> writes data as an ASCII string to the log with a trailing carriage return.</p>
--	--

<pre> void show(float data); void show(int data); void show(long data); void show(short data); void show(unsigned data); void show(ushort data); void show(ulong data); void show (const RTDataObject & data); void show (const void * data, const RTOBJECT_class * type); void show (const RTTypedValue & data); void log(const char *); void log(const RTString &); void log(char); void log(double); void log(float); void log(int); void log(long); void log(short); void log(const RTDataObject &); void log (const void *, const RTOBJECT_class *); void log(const RTTypedValue &); </pre>	<p><code>show</code> writes data as an ASCII string to the log with NO leading or trailing carriage returns.</p> <p><code>data</code>, <code>type</code> is the object, type information, or simple type that should be written to the log.</p> <p>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 encode function, passing the ASCII encoder as argument.</p> <p>The only difference between the <code>log()</code> and <code>show()</code> functions is that <code>log()</code> outputs a carriage return after the data is output to the log.</p> <p>Examples:</p> <pre> // 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); </pre>
<pre> void cr(void); void crtab(int num_tabs = 1); void space(void); void tab(void); void commit(void); </pre>	<p>These are various functions that can be used to output predefined characters to the log. <code>commit()</code> will output all buffered characters in the log.</p> <p><code>num_tabs</code> 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.</p> <pre> log.cr(); log.space(); log.tab(); log.commit(); // The previous commands can be supplied using show() log.show("\n \t"); log.commit(); </pre>
<pre> template <typename T> Base& operator<<(const T& value) Base& operator<<(const RTTypedValue & info); Base& operator<<(std::ostream& (*manip)(std::ostream&)); Base& operator<<(const RTDataObject </pre>	<p>Insertion operator function used with log streams to write text or data to the log. A data value of a user-defined type can be printed to the log by using an <code>RTTypedValue</code> object that provides both the value and the type descriptor.</p> <p>Standard C++ stream manipulators can be used for formatting etc.</p>

& obj);	
----------	--

RTMessage

This class is the data structure used within the RT 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 event 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. The RT Services Library will delete the `RTMessage` object when control returns from the transition that was triggered by dispatching the message. Applications should therefore treat the `msg` field of an `RTActor` and all data addressed beyond that pointer as read-only.

<pre>int defer(void) const;</pre>	<p>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.</p> <p>Deferred messages can be recalled using the functions defined on the RTInSignal class.</p> <p>In the transition where a message is to be deferred you would defer the message as follows:</p> <pre>getMsg()->defer();</pre>
<pre>void * getData(void) const;</pre>	<p>Returns an untyped pointer to the data that was sent along with a message.</p> <p>Note that it is recommended to use the predefined <code>rtdata</code> parameter to access the typed data of a message in a transition. For example:</p> <pre>const ADataType & dt = *rtdata;</pre> <p>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.</p> <pre>const ADataType & dt2 = *(ADataType *) (getMsg()->getData());</pre>
<pre>int getPriority(void) const;</pre>	<p>Returns the priority of the message.</p>
<pre>const char * getSignalName(void) const;</pre>	<p>Returns the name of the protocol event of the message.</p> <pre>log.show("Event named: ");</pre>

	<code>log.log(getMsg()->getSignalName());</code>
<code>const RTOBJECT_class * getType(void) const;</code>	Returns a pointer to an <code>RTOBJECT_class</code> which contains the type information that describes the data in the message, or <code>(RTOBJECT_class *)0</code> if the event does not carry any data.
<code>int isValid(void) const;</code>	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 an <code>RTOutSignal::invoke()</code> function call.
<code>RTOutSignal * sap(void) const;</code>	Returns a pointer to the port instance on which this message was received, or <code>(RTProtocol *)0</code> if called in the initial transition. <code>// 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();</code>
<code>int sapIndex(void) const; int sapIndex0(void) const;</code>	Returns the index of the port on which the message was received. The <code>sapIndex</code> function returns a one-based index (index values begin at 1) while <code>sapIndex0</code> is 0 based. Use to send a message to a particular port instance, as follows: <code>int idx = msg->sapIndex0(); port.hello().sendAt(idx);</code>

RTOBJECT_class

The `RTOBJECT_class` is a structure that contains information describing a data type. These [type descriptors](#) may be generated automatically for most types in the model. The RT Services Library uses the information in the type 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 Model RealTime 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 RT 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 Model RealTime 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
<code>_super</code>	The base type of this type
<code>_name</code>	The name of this type
<code>_version</code>	The version of this type
<code>_size</code>	The byte size of this type (sizeof)
<code>_init_func</code>	The default constructor for this type
<code>_copy_func</code>	The copy constructor for this type
<code>_decode_func</code>	The decode function for this type
<code>_encode_func</code>	The encode function for this type
<code>_destroy_func</code>	The destructor for this type
<code>_num_fields</code>	The number of fields or array elements
<code>_fields</code>	The field types or array element type

Whenever data is passed to the RT 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 RT 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;
```

If the model compiler is unable to automatically generate a type descriptor for a class, because it contains an attribute with a too complex type, you will be warned when building the model. For example:

```
14:05:41 : WARNING : CPPModel::CustomEncoding::impl : Unable to find a descriptor
for the type of this property, deselect 'Generate Descriptor' or specify a valid
descriptor in the 'Type Descriptor' property.
```

You can provide the type descriptor for a generated class to any RT Service Library function that requires it.

RTOutSignal

This class is used for working 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 is 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
}
```

	<p>The receiver of the invoke must use <code>RTOutSignal::reply()</code> to respond to the invoke. Data can be optionally sent back with the reply.</p> <pre>rtport->nack().reply();</pre> <p>However, if the version of <code>invoke</code> without reply buffer parameter is used, then the reply is implicitly made when the receiver has processed the message.</p>
<pre>int invokeAt(int index, RTMessage * replyBuffer); int invokeAt(int index);</pre>	<p>Synchronous message send to a specific port instance.</p> <p>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.</p> <p><code>index</code> is the port replication index of the port instance on which the message should be sent.</p> <p><code>replyBuffer</code> 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 <code>RTMessage::isValid()</code> once the invoke returns.</p> <p>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.</p> <p>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.</p> <pre>RTMessage reply; aPort.ack().invokeAt(0, &reply); if(reply.isValid()) // code to handle valid reply else // code to handle invalid reply</pre> <p>The receiver of the invoke must use <code>RTOutSignal::reply()</code> to respond to the invoke. Data can be optionally sent back with the reply.</p> <pre>rtport->nack().reply();</pre> <p>However, if the version of <code>invokeAt</code> without reply buffer parameter is used, then the reply is implicitly made when the receiver has</p>

	processed the message.
<pre>int reply (void);</pre>	<p>Used to respond to a synchronous message.</p> <p>Returns 1 (true) if the reply is successful, and 0 (false) otherwise.</p> <p>If the caller has provided a reply buffer when invoking the message, then the receiver must use <code>RTOutSignal::reply</code> to respond to the invoke. Data can be optionally sent back with the reply.</p> <pre>rtport->nack().reply();</pre>
<pre>int send(int priority = General);</pre>	<p>Asynchronous message broadcast to all port instances.</p> <p>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).</p> <p><code>priority</code> [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.</p> <p>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 <code>RTOutSignal::sendAt()</code> function.</p> <pre>// In this case the ack signal does not require // data to be sent with the signal. aPort.ack().send();</pre> <p>It is always good practice to check the return codes.</p> <pre>if(! aPort.ack().send()) context()->perror("Error with send");</pre> <p>You can also send data with a message.</p> <pre>// Sending some data by value, that is a copy of the data is // sent. SomeDataClass mdata("123-4356", "Ottawa"); aPort.Info(mdata).send();</pre>
<pre>int sendAt(int index, int priority = General);</pre>	<p>Asynchronous message send to a specific port instance.</p> <p>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.</p>

	<p><code>index</code> is the port replication index of the port instance on which the message should be sent.</p> <p><code>priority</code> [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.</p> <p>This function is used instead of <code>RTOutSignal::send()</code> to send a message to a specific instance of a replicated port.</p> <pre>// 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);</pre>
--	---

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 a member variable of the generated C++ capsule class. This port variable has the same name as the port, and is typed with either the base or conjugate protocol class, depending on if the port is declared as conjugated or not.

<pre>void bindingNotification (int on_off);</pre>	<p>Use this function to request notification of the creation and destruction of bindings to this port. The events sent to the port by the RT Services Library when binding notifications are enabled are <code>rtBound</code> and <code>rtUnbound</code>.</p> <p><code>on_off</code> 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.</p> <p>No messages are sent for ports which are bound prior to the call of this function.</p>
<pre>int bindingNotificationRequested (void) const;</pre>	<p>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.</p>

<code>int deregisterSAP(void);</code>	<p>Deregisters an unwired end port (SAP). Returns 1 (true) if the deregistration of the service name was successful, and 0 (false) otherwise.</p> <p>When an SAP is deregistered when it is currently connected to a SPP, the connection is terminated.</p>
<code>int deregisterSPP(void);</code>	<p>Deregisters an unwired end port (SPP). Returns 1 (true) if the deregistration of the service name was successful, and 0 (false) otherwise.</p> <p>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.</p>
<code>char * getRegisteredName (void) const;</code>	<p>Get the name that an unwired port has registered with the layer service.</p>
<code>int indexTo(RTActor *) const;</code>	<p>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.</p> <p>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 <code>role1</code> is a capsule part in the structure of a capsule on which this code is run:</p> <pre>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:");</pre>
<code>int isBoundAt(int index) const;</code>	<p>Return true (1) if the given replication index (0-based) is connected to another port and false (0) if it is not connected.</p>
<code>int isIndexTo(int index, RTActor *) const;</code>	<p>Returns true (1) if the given port instance index (0 based) is bound to the specified capsule instance.</p>
<code>int isRegistered(void) const;</code>	<p>Returns true (1) if an unwired port has been registered with the layer service and false (0) otherwise.</p>
<code>int purge (void);</code>	<p>Empties the defer queue of all port instances without recalling any deferred message. Returns the number of messages deleted from the defer queue.</p>

	To delete deferred messages for one port instance only, use <code>RTPProtocol::purgeAt</code> .
<code>int purgeAt(int index);</code>	<p>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.</p> <p>To delete deferred messages for all port instances, use <code>RTPProtocol::purge</code>.</p>
<code>int recall(void);</code>	<p>Use this function 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).</p> <p>Calling <code>recall</code> on a port gets the first deferred message from one of the port instances. Messages are recalled behind other queued messages.</p> <p>There is no time-limit on deferral so applications must take precautions against forgetting messages on defer queues.</p> <p>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 <code>RTPProtocol::recallAt()</code> function.</p> <p>The first deferred message on any instance of the replicated port named <code>port1</code> is recalled as follows:</p> <pre>port1.recall();</pre>
<code>int recallAll (void);</code>	<p>Calling <code>recallAll</code> 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.</p> <p>To recall all messages on only one instance of a port with replication factor > 1, use the <code>RTPProtocol::recallAllAt()</code> function.</p>
<code>int recallAllAt(int index, int front = 0);</code>	To recall all deferred messages on a specified port instance <code>index</code> . <code>front</code> specifies whether recalled messages should be queued ahead (non-zero) or behind (0) other queued messages. Returns the number of recalled messages.
<code>int recallAllFront(void);</code>	To recall all deferred messages on all port instances. Recalls to the front of the queue. Returns the number of recalled messages.
<code>int recallAt(int index,</code>	To recall a deferred message on a specific instance of this port for processing. <code>front</code> specifies whether recalled messages should be

<pre>int front = 0);</pre>	<p>queued ahead (non-zero) or behind (0) other queued messages. Returns the number of recalled messages (either 0 or 1).</p> <p>The first deferred message on the port instance at index 3 of the replicated port named port1 is recalled as follows:</p> <pre>port1.recallAt(3);</pre>
<pre>int recallFront(void);</pre>	<p>This function recalls the first deferred message on any port instance. Calling <code>recallFront</code> 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).</p>
<pre>int registerSAP (const char * service);</pre>	<p>Registers an unwired end port (SAP) with the layer service (as a "client").</p> <p>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.</p> <p><code>service</code> is a string that is used to identify a unique service name under which SAPs and SPPs will connect.</p> <p>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.</p> <p>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.</p> <p>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 RT Services Library.</p>
<pre>int registerSPP (const char * service);</pre>	<p>Registers an unwired end port (SPP) with the layer service (as the "provider").</p> <p>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</p>

	<p>1.</p> <p><code>service</code> is a string that is used to identify a unique service name under which SAPs and SPPs will connect.</p> <p>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.</p> <p>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.</p> <p>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 RT Services Library.</p>
<pre>int size(void) const;</pre>	<p>Returns the replication factor (i.e. the size as defined by the multiplicity) of the port.</p> <p>Remember that port instances are indexed in the RT 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.</p> <pre>for(int i = 0 ; i < port.size(); i++) port.ack().sendAt(i);</pre>

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 in 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.

<pre>int isValid(void);</pre>	<p>Returns true (1) if the timer identifier is a valid timer id, and 0 (false) otherwise.</p> <p>This function only tells if <code>RTTimerId</code> points to a valid (existing) timer object and if <code>RTTimerId</code> 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 <code>timeout</code> event has not yet been delivered; or if the timer has been canceled or a non-periodic timer has timed out.</p> <p>The function can be used to test the result of a timer request.</p> <pre>RTTimerId tid = timer.informIn(RTTimespec(4,0)); if(! tid.isValid()) context()->perror("Error when setting timer");</pre> <p>The function can also be used to test if a timer still exists before cancelling it:</p> <pre>if(tid.isValid()) timer.cancelTimer(tid);</pre> <p>Note, that the <code>cancelTimer</code> function also checks the validity of <code>tid</code> and it will return a failure result (0) if you call it with a non-valid timer id.</p>
---------------------------------	--

RTTimespec

`RTTimespec` is used to create time 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, and `tv_nsec` is the number of nanoseconds.

<pre>long tv_sec; long tv_nsec;</pre>	<p><code>tv_sec</code> is the number of seconds for the timer setting, and <code>tv_nsec</code> is the number of nanoseconds. There are 10^9 nanoseconds in one second.</p> <p>This will initialize an <code>RTTimespec</code> with one second.</p> <pre>RTTimespec t1(1,0);</pre> <p>This struct 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 <code>RTTimespec</code> constructor.</p> <pre>timer.informIn(RTTimespec(5,0));</pre>
<pre>RTTimespec & operator= (const RTTimespec & t1);</pre>	<p>The <code>RTTimespec</code> assignment (=) operator re-initializes an existing <code>RTTimespec</code> object with new second and nanosecond values.</p>

<pre> 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); </pre>	<p>Arithmetic operators.</p> <p>t1, t2 are RTTimespec objects to add or subtract.</p> <pre> RTTimespec t1(2,0), t2; RTTimespec::getclock(t2); t2 += t1; </pre>
<pre> int operator== (const RTTimespec & t1, const RTTimespec & t2); int operator!= (const RTTimespec & t1, const RTTimespec & t2); int operator<= (const RTTimespec & t1, const RTTimespec & t2); int operator> (const RTTimespec & t1, const RTTimespec & t2); int operator>= (const RTTimespec & t1, const RTTimespec & t2); int operator< (const RTTimespec & t1, const RTTimespec & t2); </pre>	<p>Comparison operators.</p> <p>Return non-zero if the objects meet the comparison condition; otherwise 0.</p> <p>t1, t2 are RTTimespec objects to compare.</p> <pre> RTTimespec t1(2,0), t2(3,0); if (t1 < t2) //t1 is less than t2 </pre>
<pre> static void getclock (RTTimespec & tspec); </pre>	<p>Returns the current time. The values of the tspec parameter are filled in with the current time.</p> <p>This is a class-scoped function.</p> <pre> RTTimespec t; RTTimespec::getclock(t); </pre>
<pre> RTTimespec(void); RTTimespec(long sec, long nsec); RTTimespec(const RTTimespec & ts); RTTimespec(const RTime & t); </pre>	<p>Constructs an RTTimespec object with seconds and nanoseconds, or with a value of another RTTimespec object or an RTime object.</p> <p>An RTTimespec of two seconds can be created and passed to the Timing Service informEvery() as follows:</p> <pre> // 2 seconds RTTimespec t(2 , 0); // 6am coordinated universal time (UTC) RTime abst(6,0,0); timer.informEvery(t); timer.informAt(RTTimespec(abst)); </pre>

RTTiming

Timing ports are instances of the class RTTiming.

<pre> void adjustTimeBegin(void); void adjustTimeEnd(const RTTimespec & delta); </pre>	<p>Used to adjust the internal real-time system clock.</p> <p>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 RT Services Library will make adjustments to its internal data structures so that relative timeouts are not affected by the system clock change. Use <code>adjustTimeBegin()</code> to stop the timing service and <code>adjustTimeEnd()</code> to restart it at the new time.</p> <p>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 <code>timer</code> and that there are operating system primitives for reading and writing the system clock which we here call <code>sys_getclock()</code> and <code>sys_setclock()</code>, respectively.</p> <pre> 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 RT Services Library timer service timer.adjustTimeEnd(delta); } </pre>
<pre> int cancelTimer (RTTimerId &tid); </pre>	<p>Cancels a pending timer. Returns true (1) if a pending timeout request was cancelled, and false (0) if no such request was found.</p> <p><code>tid</code> is the identifier of the timer that was provided when the service request was made.</p> <p>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.</p> <p>If <code>timer</code> is the name of a timing port, you can create, and subsequently cancel, a timeout request as follows:</p> <pre> RTTimerId tid = timer.informEvery(RTTimespec(2, 0)); </pre>

	<pre>timer.cancelTimer(tid);</pre>
<pre>RTTimespec currentTime (void) const;</pre>	<p>Determines the current absolute time.</p> <p>It is recommended, for performance reasons, that you use the <code>RTTimespec::getClock</code> functions instead of <code>currentTime</code>.</p> <pre>RTTimespec ctime = timer.currentTime();</pre>
<pre>RTTimerNode * informAt (const RTTimespec & when, const void * data, const RTOBJECT_class * type, int prio = General); RTTimerNode * informAt (const std::chrono::system_clock::time_point & when, const void * data, const RTOBJECT_class * type, int prio = General); RTTimerNode * informAt (const RTTimespec & when, int prio = General); RTTimerNode * informAt (const std::chrono::system_clock::time_point & when, int prio = General); RTTimerNode * informAt (const RTTimespec & when, const RTDataObject & data, int prio = General); RTTimerNode * informAt (const std::chrono::system_clock::time_point & when, const RTDataObject & data, int prio = General); RTTimerNode * informAt (const RTTimespec & when, const RTTypedValue & info, int prio = General); RTTimerNode * informAt (const std::chrono::system_clock::time_point & when, const RTTypedValue & info, int prio = General);</pre>	<p>Starts a timer which will expire at a particular absolute point in time.</p> <p>A timer handle is returned which can be used to construct an <code>RTTimerId</code>. This can be used to cancel the timer prior to expiry. A null pointer is returned if the timer request fails.</p> <p><code>when</code> is the desired absolute time when the timer is to expire. It can either be specified using an <code>RTTimespec</code> or an <code>std::chrono::system_clock::time_point</code>.</p> <p><code>data, info, type [optional]</code> 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.</p> <p><code>prio [optional]</code> is the priority at which the timeout message will be delivered. This parameter is optional.</p> <pre>RTTimespec now; RTTimespec::getClock(&now); timer.informAt(now + RTTimespec(15, 0));</pre>

<pre> RTTimerNode * informEvery (const RTTimespec & delta, const void * data, const RObject_class * type, int prio = General); RTTimerNode * informEvery (const std::chrono::nanoseconds & delta_ns, const void * data, const RObject_class * type, int prio = General); RTTimerNode * informEvery (const RTTimespec & delta, const RTDataObject & data, int prio = General); RTTimerNode * informEvery (const std::chrono::nanoseconds & delta_ns, const RTDataObject & data, int prio = General); RTTimerNode * informEvery (const RTTimespec & delta, const RTTypedValue & info, int prio = General); RTTimerNode * informEvery (const std::chrono::nanoseconds & delta_ns, const RTTypedValue & info, int prio = General); RTTimerNode * informEvery (const RTTimespec & delta); RTTimerNode * informEvery (const std::chrono::nanoseconds & delta_ns); </pre>	<p>Starts a periodic timer. A timer handle which can be used to construct an <code>RTTimerId</code> is returned. It can be used to cancel the timer prior to expiry. A NULL pointer is returned if the timer request fails.</p> <p><code>delta</code> represents the desired time interval (in seconds and nanoseconds as an <code>RTTimespec</code>) 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.</p> <p><code>delta_ns</code> represents the desired time interval (in nanoseconds as an <code>std::chrono::nanoseconds</code>).</p> <p><code>data</code>, <code>info</code>, <code>type</code> [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.</p> <p><code>prio</code> [optional] is the priority at which the timeout message will be delivered. This parameter is optional.</p> <pre> if(! timer.informEvery(RTTimespec(10, 0)) log.log("error requesting a periodic timer"); </pre> <p>If the timer is to be cancelled, then an <code>RTTimerId</code> 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.</p> <pre> 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); </pre>
<pre> RTTimerNode * informIn (const RTTimespec & delta, const void * data, const RObject_class * type, int prio = General); RTTimerNode * informIn (const std::chrono::nanoseconds & delta_ns, const void * data, const RObject_class * type, </pre>	<p>Starts a timer which expires after some time has passed from the current time (i.e. a time interval).</p> <p>A timer handle which can be used to construct an <code>RTTimerId</code> object is returned. It can be used to cancel the timer prior to expiry. A NULL pointer is returned if the timer request fails.</p> <p><code>delta</code> represents the desired time interval (in seconds and nanoseconds as an <code>RTTimespec</code>) from the current time at which the timer will expire. Timer intervals should be ≥ 0. If the timer</p>

<pre> int prio = General); RTTimerNode * informIn (const RTTimespec & delta, int prio = General); RTTimerNode * informIn (const std::chrono::nanoseconds & delta_ns, int prio = General); RTTimerNode * informIn (const RTTimespec & delta, const RTDataObject & data, int prio = General); RTTimerNode * informIn (const std::chrono::nanoseconds & delta_ns, const RTDataObject & data, int prio = General); RTTimerNode * informIn (const RTTimespec & delta, const RTTypedValue & info, int prio = General); RTTimerNode * informIn (const std::chrono::nanoseconds & delta_ns, const RTTypedValue & info, int prio = General); </pre>	<p>interval is equal to zero, the timer will expire immediately.</p> <p><code>delta_ns</code> represents the desired time interval (in nanoseconds as an <code>std::chrono::nanoseconds</code>).</p> <p><code>data, info, type [optional]</code> 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.</p> <p><code>prio [optional]</code> is the priority at which the timeout message will be delivered. This parameter is optional.</p> <pre> // request a timer to expire in 10 seconds if(! timer.informIn(RTTimespec(10, 0))) log.log("error requesting a timer"); </pre> <p>If the timer is to be cancelled, then an <code>RTTimerId</code> 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.</p> <pre> if(! (tid = timer.informIn(RTTimespec(10, 0)))) log.log("error requesting a timer"); // this could be done in an other transition timer.cancelTimer(tid); </pre>
---	--

RTTypedValue

`RTTypedValue` is a struct which is used to encapsulate a data value and its type descriptor object. 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 the 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 RT Services Library will use the type descriptor for class A.