Telelogic

Rhapsody®

C++ Framework Execution Reference Manual

Rhapsody®

C++ Framework Execution Reference Manual

Before using the information in this manual, be sure to read the “Notices” section of the Help or the PDF available from **Help > List of Books**.

This edition applies to Telelogic Rhapsody 7.4 and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Frameworks and OXF Overview

Welcome to the *Rhapsody in C++ Framework Execution Reference Manual*. This guide is intended to be used by application developers as a reference manual for the framework layer classes, methods, and attributes.

Rhapsody® is an award-winning, UML-compliant, systems design, application development, and collaboration platform. Rhapsody is used by systems engineers and software developers to deliver embedded or real-time systems. Rhapsody uniquely combines a graphical UML programming paradigm with advanced systems design and analysis capabilities and seamlessly links with the target implementation language, resulting in a complete model-driven development environment, from requirements capture through analysis, design, implementation, and test.

Real-Time Frameworks

The emergence of the unified modeling language (UML) as an industry standard for modeling complex systems has encouraged the use of automated tools that facilitate the development process from analysis through coding. This is particularly true of real-time embedded systems whose behavioral aspects facilitate full life-cycle software development by way of modeling. Statecharts are natural candidates for automatic code generation, testing, and verification.

One major benefit of the object-oriented paradigm is the inherent support for abstraction-centric, reusable, and adaptable design. In particular, it is common to construct complex systems using predefined *frameworks*. A framework is a collection of collaborating classes that provides a set of services for a given domain. You `customize` the framework to a particular application by subclassing and composing instances of the framework classes. Therefore, frameworks represent object-oriented reuse.

There are several advantages to using frameworks:

- You do not need to write the application from scratch because it reuses elements of the framework.
- Frameworks structure the design of the application by providing a set of predefined abstractions, given by the classes in the framework. These classes provide architectural guidance for the system design.
- Frameworks are open and flexible designs because their classes can be customized via subclassing.

The Object Execution Framework (OXF)

Rhapsody is a visual programming environment that enables you to create an embedded software application by creating a graphical, object-oriented model and generating production-level code from that model.

Code generation in Rhapsody is framework-based: it includes a fixed, predefined framework called the OXF (**O**bject e**X**ecution **F**ramework), and the generated code reuses that framework. For example, the code generated for a reactive class reuses the event processing functionality by subclassing a framework class that embodies event processing capabilities. This has the following implications:

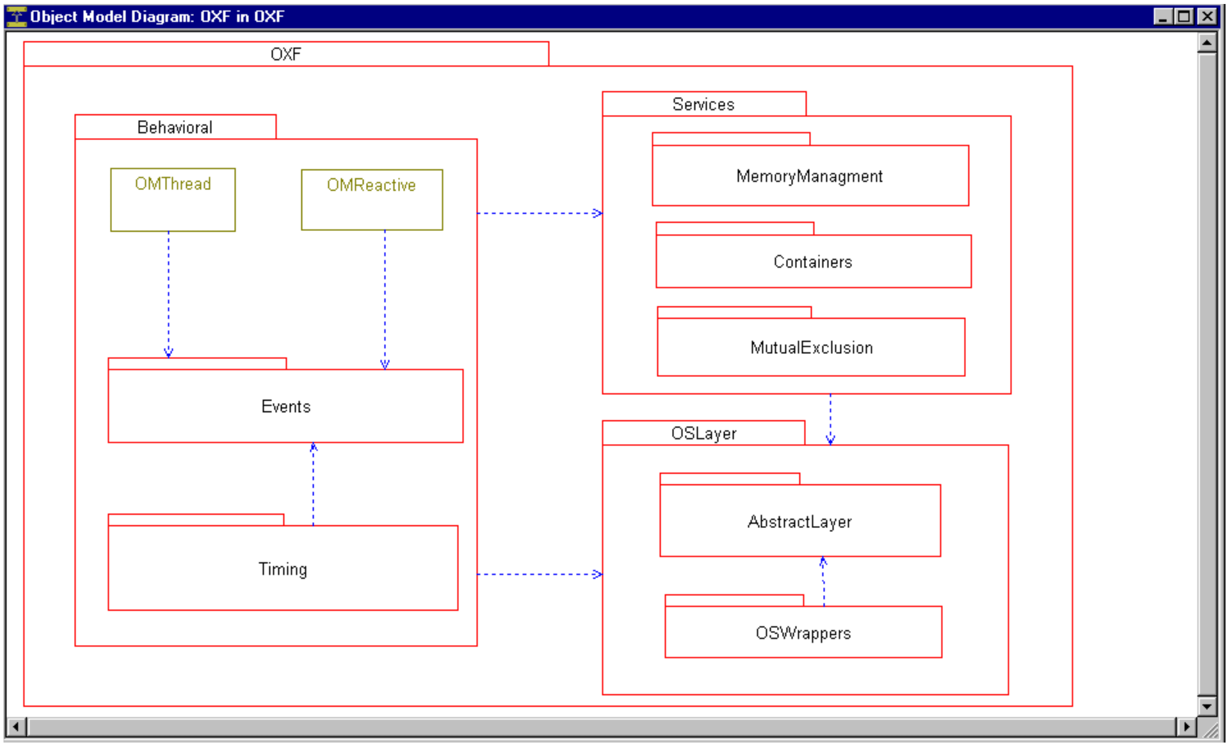
- The framework contains a set of useful real-time abstractions that structure the generated code and give concrete meaning to UML concepts (such as “active class”).
- Significant portions of functionality are factored out into the framework classes, so there is less need to generate specific code. This also eases the task of understanding the code.
- You can customize framework elements using inheritance to fit your specific needs.
- The framework has an existence of its own, which is independent of the code generator. Its classes can be used outside the code generation process, in user-class implementations, or in any other way you desire.

Working with the Object Execution Framework

You can work with the OXF at several levels. For example, you can use the OXF to:

- Create multithreaded, reactive applications. This is the most common way to use the OXF.
- Write actions (generate events, synchronize threads, manipulate relations, and so on). This does not require deep understanding of the internals; rather, you simply need to call a few methods.
- Implement reactive behaviors without a statechart. If you want to further customize the automated behavioral code, you need to understand the collaborations within the framework.
- Customize the framework. The framework classes enable you to tailor the framework for your specific needs.

The following figure shows the architecture of the framework, which is described in detail in The OXF Library.



The OXF Library

Rhapsody has one central runtime library, `OXF`, that provides run-time services required by the generated code. The other libraries under the `Share` directory of the installation enable the animation and tracing capabilities of Rhapsody.

Note

For a list of the most relevant files in the directory <install_dir>/Share/LangCpp/oxf, see the **The Framework Files** section.

The compiled OXF consists of three logical packages:

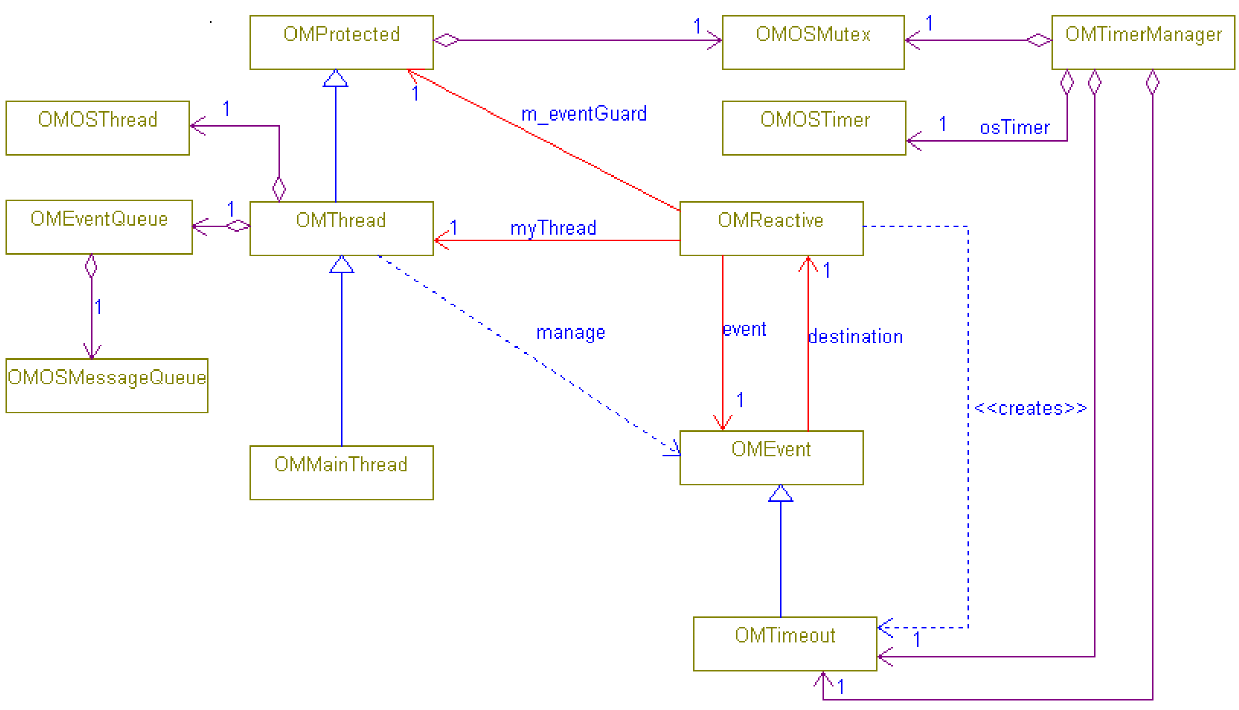
- **Behavioral package** (**`Behavioral`**)—Consists of a set of collaborative classes that form the fundamental architecture of an object-oriented, reactive, multithreaded system. For more information, see **Behavioral Package**.
- **Operating system package** (**`OSLayer`**)—Provides a thin abstraction layer through which the framework and generated code access operating system services. For more information, see **OSLayer Package**.
- **Services package** (**`Services`**)—Consists of two subpackages: `MemoryManagement` and `Containers`. For more information, see **Services Package**.

Behavioral Package

The `Behavioral` package (also known as the *active behavioral framework*) consists of a set of collaborative classes that form the fundamental architecture of an object-oriented, reactive, multithreaded system.

The `OMReactive, OMThread, OMProtected, OMEvent, OMTimeout`, and `OMTimerManager` classes are the base classes from which concrete model classes are derived. The code generator automatically derives model classes from framework classes based on their application classes.

The following figure shows the class diagram of the OXF.



OMReactive Class

Essentially, a *reactive* class is one that reacts to events; that is, it is an event consumer. A reactive class is represented in the execution framework by the `OMReactive` class (defined in `omreactive.h`), from which every generated reactive class inherits by default. Every reactive class is associated with an active class, from which its events are dispatched.

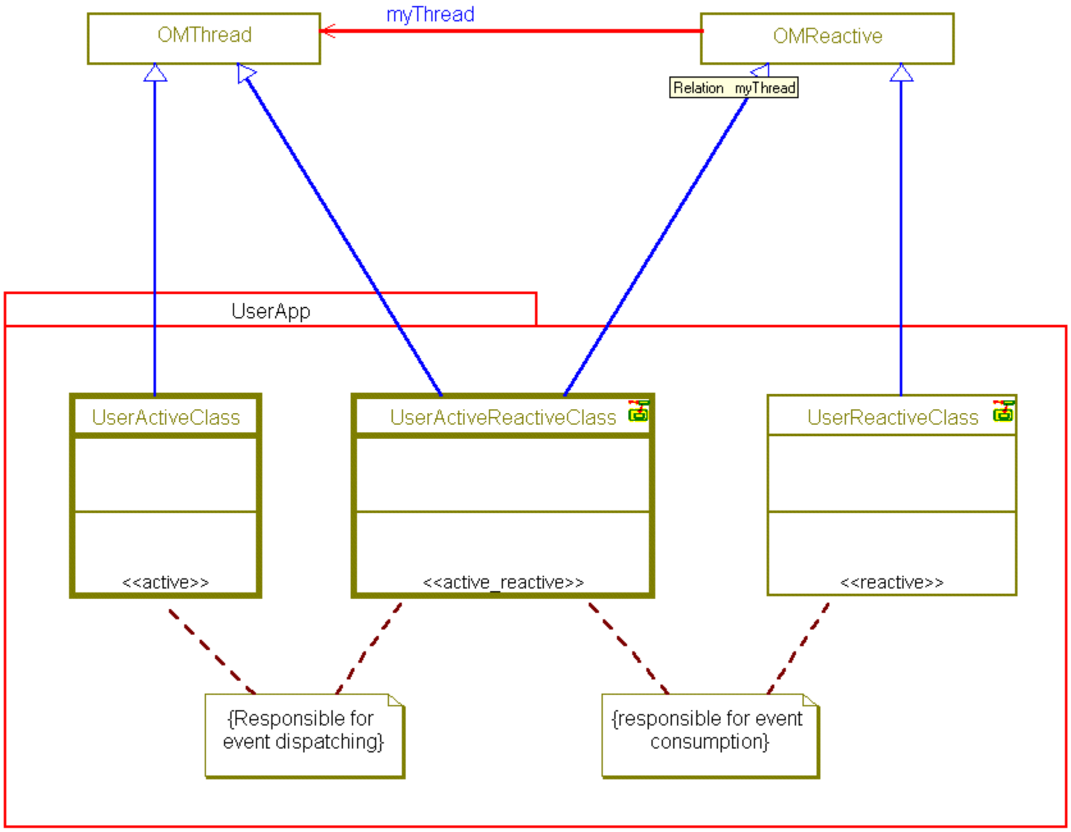
The **Active and Reactive Class** illustration shows the relationships between active and reactive class-related elements in the execution framework. In the diagram, framework classes are shown at the top, whereas representative user classes are shown at the bottom.

Each class can have events and operations defined on it. Events are significant occurrences located in time and space. In the context of statecharts and activity diagrams, events can trigger transitions between states. For detailed information on signal events, triggered operations, and timeout events, see **Event Handling**.

An instance of a reactive class accepts a given event via the **gen** operation, which queues the event in its associated manager using the **queueEvent** method. The manager will later inject it to the instance for consumption by calling the **takeEvent** method. In the general case, the reactive class and its manager are distinct objects. However, in many cases, they are one and the same.

The processing of events is normally defined by a statechart or activity diagram, but you can define an arbitrary event-consumption behavior for a reactive class by overriding the `consumeEvent` method.

Active and Reactive Class



For more information on the `OMReactive` class, see **OMReactive Class**.

OMThread Class

An active object is defined in the UML as "an object that owns a thread and can initiate control activity." The `OMThread` class (defined in `omthread.h`) is the base class in the framework for every active class. User active classes inherit from `OMThread`, which has the following responsibilities:

- Runs an event loop on its own thread
- Dispatches events to client reactive classes

For more information, see **Active and Reactive Class** and **Event Handling**.

A thread is represented by `OMOSThread`, which wraps an operating system thread.

`OMThread` contains code that manages an event queue. It executes an infinite event dispatching loop, taking events from the queue and injecting them to the target instances. Every user class that inherits from `OMThread` acquires this default behavior.

Active classes encapsulate the notion of event-driven tasks; that is, an active class is a task that performs event management. It is not necessarily reactive, but every reactive object needs an active object to manage (queue and dispatch) its incoming events.

You can customize `OMThread` so it uses a different event dispatching mechanism via inheritance. For example, you could define a class `MyThread` that uses two event queues instead of one. `myThread` would inherit from `OMThread`, overriding the **execute**, **queueEvent**, **cancelEvent**, and **cancelEvents** methods. You can then tune the code generator to use `myThread` instead of `OMThread` during code generation, meaning that classes marked "active" will automatically inherit from `myActive` instead of `OMThread`.

OMMainThread Class

The `OMMainThread` class (defined in `omthread.h`) is a special case of `OMThread`—it defines the default active class for an application. `OMMainThread` inherits from `OMThread` and is a singleton—only one instance is created.

OMDelay Class

The `OMDelay` class (defined in `omthread.h`) is used to delay a calling thread. A timeout is asynchronous, which means that the thread is not waiting for a timeout—the timeout is dispatched to a reactive class that can handle it. By using `OMDelay`, a task can block a thread.

`OMDelay` is normally used by the application. If a reactive instance creates an `OMDelay`, it will get a timeout after the specified delay time.

You call `OXF::delay` to create an instance of `OMDelay`.

OMProtected Class

Resources in a class can be monitored by declaring them *guarded*, which allows only one operation to access the resource at any given time. A *protected* class can be used to model an exclusive resource: at any given moment, only a single copy of a single guarded operation (of the class) can be executing.

The `OMProtected` class (defined in `omprotected.h`) is the base class for all protected objects. It supports the operations `lock` and `unlock` using `OMOSMutex`.

One central characteristic of real-time system design is the existence of resources that, in the presence of concurrency, must be managed. The OXF includes abstractions for concurrency control mechanisms.

`OMOSMutex` is a wrapper class for an operating system mutex. It supports the operations `lock` and `unlock`. A mutex is used for managing exclusive resources.

OMGuard Class

The `OMGuard` class (defined in `omprotected.h`) is an enter-exit object (its work is performed in `CTOR` and `DTOR`) used to guard a section of code. Several macros (defined in `omprotected.h`) are used to start and stop the guard.

OMEvent Class

The `OMEvent` class (defined in `event.h`) is the base class for all events defined in Rhapsody. The code generator implicitly derives all events from `OMEvent`. *Events* are significant occurrences located in time and space. In the context of statecharts and activity diagrams, events can trigger transitions between states.

The Rhapsody execution framework supports three types of events:

- Signal events (or “events”)
- Triggered operations (or “synchronized events”)
- Timeout events (or “timeouts”)

For detailed information on events, see **Event Handling**.

OMTimeout Class

Timeouts are a specialization of class `OMEvent`. The `OMTimeout` class (defined in `event.h`) implements timeouts issued by statecharts or activity diagrams within reactive classes. The system timer manages the timeouts and sends them to the requesting object—the object that issued the timer request.

Timeouts are either created by instances entering states with timeout transitions or delay requests from user code.

For more information on timeouts, see **OMTimeout Class**, and **Event Handling**.

OMTimerManager Class

The `OMTimerManager` is responsible for managing the timeout. How it is called to do its job depends on the tick timer (`OMOSTimer`) implementation in the operating system adapter. In most implementations, there is an additional thread that provides timer support for the application. If the timer uses a separate thread, then for a single-threaded application, the Rhapsody-generated application will have two threads—one thread for the application and one thread for the timer manager.

The `OMTimerManager` class (defined in `timer.h`) manages timeout requests and issues timeout events to the application objects. `OMTimerManager` is a singleton object in the execution framework.

The timer manager has a timer, class `OMThreadTimer`, that notifies it periodically whenever a fixed time interval has passed. At any given moment, the timeout manager holds a collection of timeouts that should be posted when their time comes. Each time the timer manager is notified by its timer, it examines the collection and sends the due timeout to the originating object. The timeout objects themselves are passive in the sense that they do not contain timers.

The timer manager has a timer, class `OMThreadTimer`, that notifies it periodically whenever a fixed time interval has passed. `OMThreadTimer` is a subclass of `OMTimerManager` that does the actual work of dispatching the timeouts to the reactive classes (that is, generating the timeouts to the reactive classes).

For more information on the `OMTimerManager` class, see **OMTimerManager Class**.

Customizing Timeout Manager Behavior

By customizing the framework, you can create a class that inherits from the framework base class, overrides the behavior of the base class, and modifies code generation. All other classes from the same type will then inherit from the user class instead of inheriting directly from the framework base class. For example, you can customize the behavior of the timeout framework by overriding the **schedTm** and **unschedTm** methods so each active class has its own timeout manager. See **OXF Reference Pages**, for detailed information about these methods.

OMThreadTimer Class

The `OMThreadTimer` class (defined in `timer.h`) inherits from `OMTimerManager` and performs the actual work of dispatching timeouts to the reactive classes (that is, generating the timeouts to the reactive classes).

OMTimerManagerDefaults Class

The `OMTimerManagerDefaults` class (defined in `timer.h`) is used to define values for the following timer attributes:

- `defaultTicktime` specifies the default value for the basic system tick, in milliseconds.
- `defaultMaxTM` specifies the limitation on the maximum number of timeouts that can exist in the system. Timeouts are preallocated at system initialization.

OSLayer Package

The typical embedded software application created in Rhapsody is designed to work with a real-time operating system (RTOS). Rhapsody includes a number of adapters that cover the more common RTOSes. In addition, you can customize the Rhapsody installation to accommodate a specific OS/RTOS targeted for use with the embedded software application. This involves interfacing with the `OSLayer` package, defined specifically for this purpose.

The operating system package (`OSLayer`) consists of two packages:

- AbstractLayer Package
- OSWrappers Package

AbstractLayer Package

The operating system `AbstractLayer` package (OSAL) provides a thin abstraction layer through which the framework and generated code access operating system services. Each one represents an operating system object.

The behavioral framework and generated code are RTOS-independent (as are all other parts of the framework). RTOS independence is achieved via the set of adapter classes that comprise the OSAL. The OSAL is the only RTOS-dependent package within the OXF, and serves as the only interface to the RTOS. By "plugging-in" different OSAL implementations, the user application can run on different operating systems.

In general, each target environment requires a custom implementation of the OSAL. For detailed information about customizing the OSAL for a specific RTOS, see the *RTOS Interface Guide*. The `os.h` specification file includes the interfaces for the OSAL.

Note

Some environments can use the same adapter. For example, although VxWorks™ PPC860 and VxWorks Pentium® III are different environments, they use the same adapter. The same is true for Windows NT® and Windows CE®.

Classes

The `AbstractLayer` package defines classes that describe basic operations and entities used by the operating system, including the following:

- `OMOSThread` provides basic threading features. It provides two create thread methods so you can create either a simple thread or a wrapper thread.
- `OMOSMessageQueue` allows independent but cooperating tasks (active classes) within a single CPU to communicate with each other.
- `OMOSTimer` acts a building block for `OMTimerManager`, which provides basic timing services for the execution framework.
- `OMOSMutex` protects critical sections within a thread using binary mutual exclusion. Mutexes are used to implement protected objects.
- `OMOSEventFlag` synchronizes threads. Threads can wait on an event flag by calling `wait`. When some other thread signals the flag, the waiting threads proceed with their execution.
- `OMOSSemaphore` allows a limited number of threads in one or more processes to access a resource. The semaphore maintains a count of the number of threads currently accessing the resource.
- `OMOSSocket` represents the socket through which data is passed between Rhapsody and an instrumented application.
- `OMOSConnectionPort` used for interprocess communication between instrumented applications and Rhapsody.
- `OMOSFactory` provides abstract methods to create each type of operating system entity. Because the created classes are abstract, the factory hides the concrete class and returns its abstract representation. The factory is implemented as a static global variable to ensure that only one instance of a given `OSFactory` can exist.

The operating-specific header files implement the abstract classes defined by `AbstractLayer` package for the target system.

OSWrappers Package

The `OSWrappers` package holds the concrete implementation of the OSAL for each supported RTOS.

Services Package

This section describes the `Services` package, which consists of the following subpackages:

- **MemoryManagement Package**
- **Containers Package**

MemoryManagement Package

The framework supports two memory management packages:

- A plug-in memory manager (`OMMemoryManager`). This class is defined in the `ommemorymanager.cpp/h`. For custom adapters, you must add these files to the OXF makefile.
- A static memory manager that enables you to define static memory pools for user classes and events (defined in `MemAlloc.h`).

See **OMMemoryManager Class** for detailed information about this class's methods.

Containers Package

The `Containers` package is a set of template and non-template classes used by Rhapsody to implement relationships (associations and aggregations) in the application's object model. Each container class is suitable for different relation attributes. Note that some of the containers (such as `OMStack, OMQueue` and `OMHeap`) are not used for relation implementation. They are used internally in the framework, and can also be used directly by the client application.

The OXF container classes provide the default implementation for the relations in the object model. Note that the Rhapsody code generator can be parameterized to use an "off-the-shelf" container library, e.g., RogueWave™, MFC, or the Standard Template Library (STL), instead of its "native" container library. The relation implementation with STL containers is supported "out-of-the-box" by Rhapsody.

Rhapsody uses containers to implement to-many relations between objects. These include relationships of one object to many, or many objects to many. Rhapsody automatically selects the appropriate container to implement the behaviors of various relations based on the multiplicities, access, and ordering of classes and objects involved. Typical containers are lists, stacks, heaps, static arrays, collections, and maps, each of which has its own set of behaviors. For example, arrays allow random access, whereas lists do not.

The OXF supports the following container types:

- `OMAbstractContainer`—An abstract, type-safe container.
- `OMCollection`—A type-safe, dynamically sized array. See **OMCollection Class** for more information.
- `OMHeap`—A type-safe, fixed size heap implementation. See **OMHeap Class** for more information.
- `OMIterator`—A type-safe iterator over an `OMAbstractContainer` (and derived containers). See **OMIterator Class** for more information.
- `OMList`—A type-safe, linked list. See **OMList Class** for more information.
- `OMMap`—A type-safe map, based on a balanced binary tree (`log(n)` search time). See **OMMap Class** for more information.
- `OMQueue`—A type-safe, dynamically sized queue. It is implemented on a cyclic array, and implements a FIFO (first in, first out) algorithm. See **OMQueue Class** for more information.
- `OMString`—A string class. See **OMString Class** for more information.
- `OMStack`—A type-safe stack that implements a LIFO (last in, first out) algorithm. See **OMStack Class** for more information.
- `OMStaticArray`—A type-safe, fixed-size array. See **OMStaticArray Class** for more information.

In addition to these containers, the OXF supports `omu*` containers, which are containers that are not implemented with templates. The use of template-free containers reduces the size of the generated code considerably.

The `OMU*` containers are as follows:

- `OMUAbstractContainer`—An unsafe (typeless) abstract container. All derived containers hold `void*`. See **OMUAbstractContainer Class** for more information.
- `OMUIterator`—An iterator over `OMUAbstractContainer` and derived containers. See **OMUIterator Class** for more information.
- `OMUList`—A typeless list. See **OMUList Class** for more information.
- `OMUCollection`—A typeless, dynamically sized array. See **OMUCollection Class** for more information.
- `OMUMap`—A typeless map. See **OMUMap Class** for more information.

Event Handling

This section describes event handling within the OXF. It describes the following topics:

- Events
- Timeouts

Events

Each class can have *events* and operations defined on it. In the context of statecharts and activity diagrams, events can trigger transitions between states.

The Rhapsody execution framework supports three types of events:

- **Signal events (or “events”)**—Asynchronous stimuli communicated between instances that can have parameters. Signal events are implemented by class `OMEvent`.
- **Triggered operations (or “synchronous events”)**—Stimuli that can trigger transitions synchronously (without queueing them first).
- **Timeout events (or “timeouts”)**—Signal the expiration of a time interval after a certain state was entered. Timeout events are implemented by class `OMTimeout`.

Generating and Queuing an Event

The following sequence diagram shows the generation and queuing of an event.

The sequence to generate and queue an event is as follows:

1. A client class creates the event.
2. The client class calls the **gen** method of the reactive class that should consume the event.
3. The **setDestination** method sets the destination attribute to the specified `OMReactive` instance.
4. The **queueEvent** method asks the thread to queue the event by calling the `put` method (defined in `omthread.cpp`).
5. The `put` method inserts the event into the thread's event queue.

Dispatching an Event

The following sequence diagram shows a dispatched event.

The method `OMThread::execute` is responsible for the event loop. This sequence diagram shows the main sequence of events that are done inside this method.

The event loop is as follows:

1. `execute` calls the `get` method to get the first event from the event queue.
2. If the event is not a NULL event, `execute` calls the **getDestination** method to determine the `OMReactive` destination for the event.
3. `execute` calls the **takeEvent** method to request that the reactive object process the event. `takeEvent` calls the `consumeEvent` method, which does the following:

   a. It calls `isBusy` to determine whether the object is already consuming an event. If the object is not busy, **consumeEvent** does the following:

      Sets the `sm_busy` flag to `TRUE`

      Calls **getIId** to get the event ID

Passes the value of `lId` to **rootState_dispatchEvent** to dispatch that event

b. **consumeEvent** calls `shouldCompleteRun` to see if there are any null transitions to take after the event has been consumed. If there are null transitions to be taken, the method calls `runToCompletion` to take them.

c. **consumeEvent** calls `undoBusy` to reset the `sm_busy` flag to `FALSE`.

4. `execute` calls the **isDeleteAfterConsume** method to determine whether the event should be deleted. If the **deleteAfterConsume** attribute is `TRUE, execute` calls the **Delete** method to delete the event.

Canceling a Single Event

Events are canceled when the event destination is deleted.

Canceling All Events to a Destination

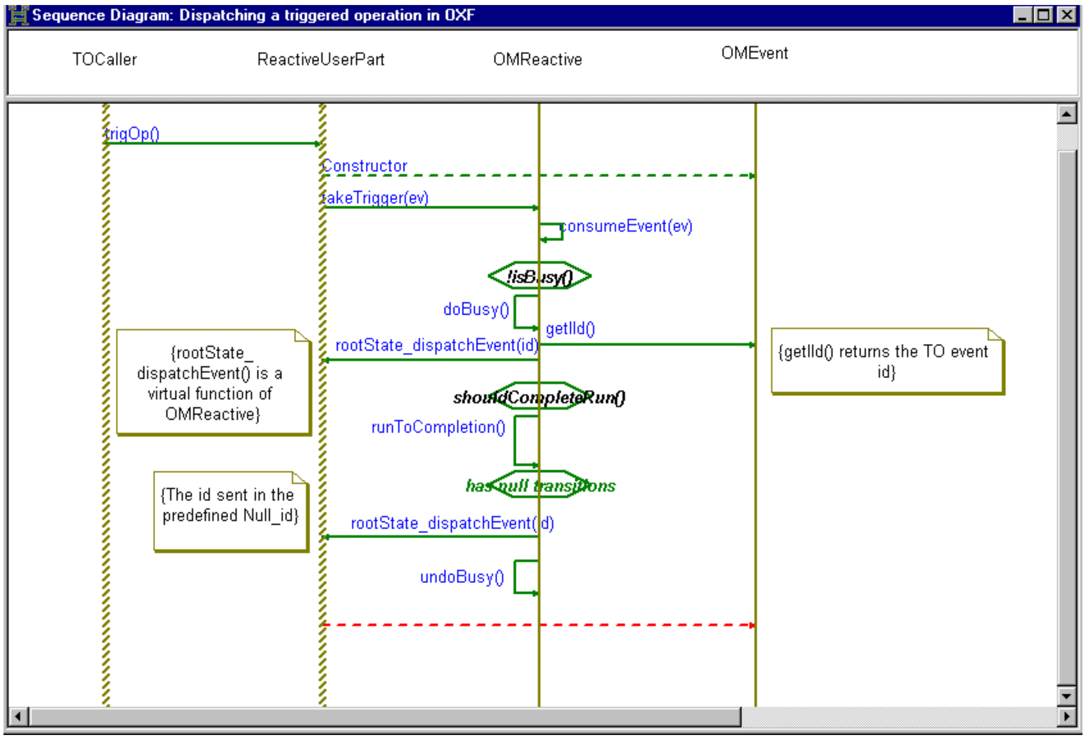
The **cancelEvents** method cancels all the events targeted for a specific `OMReactive` instance. It calls `getMessageList` to get a list of all events in the thread's event queue.

For each event in the message list:

1. **cancelEvents** calls **getDestination** to determine the destination `OMReactive` instance.
2. If the event's destination matches the destination parameter passed to **cancelEvents**, the method calls **cancelEvent** to cancel the event.
3. **cancelEvent** calls **setIId** to set the event ID to **OMCancelledEventId**.

Dispatching a Triggered Operation

The following sequence diagram shows a dispatched triggered operation (synchronous event).



The sequence for dispatching a triggered operation is as follows:

1. The **takeTrigger** method is called for the triggered operation.
2. `takeTrigger` calls the **consumeEvent** method to consume the event.
3. `consumeEvent` does the following:

   a. It calls `isBusy` to determine whether the object is already consuming an event. If the object is not busy, **consumeEvent** does the following:

      Sets the `sm_busy` flag to `TRUE`

      Calls **getIId** to get the event ID

      Passes the value of `lId` to **rootState_dispatchEvent** to dispatch that event

   b. **consumeEvent** calls `shouldCompleteRun` to see if there are any null transitions to take after the event has been consumed. If there are null transitions to be taken, the method calls `runToCompletion` to take them.

c. **consumeEvent** calls `undoBusy` to reset the `sm_busy` flag to `FALSE`.

4. **takeTrigger** calls the **shouldTerminate** and **setShouldDelete** methods. If `(shouldTerminate() && shouldDelete())` is 1 (or `TRUE`), `takeTrigger` deletes the event.

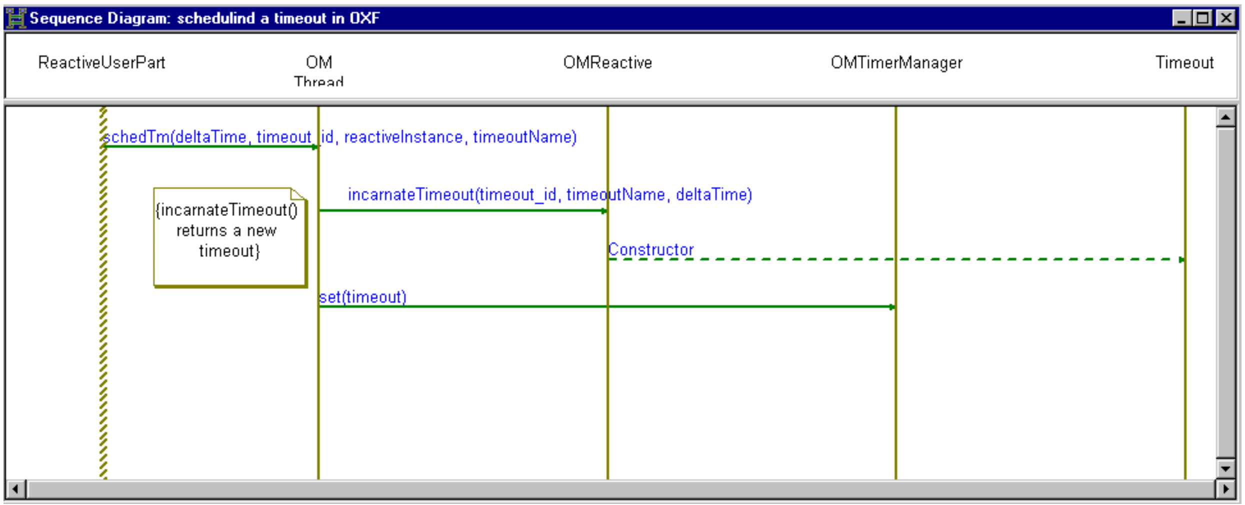
Timeouts

A *timeout* is a special kind of event that signals that a specified amount of time has elapsed since a state was entered. The entry point for timeout scheduling is an active object, which creates the timeout and passes it to the timeout manager, an instance of class `OMTimerManager`. Each time `OMTimerManager` is notified by its timer, it examines the collection of timeouts and queues the due timeouts in the appropriate manager (the active object), where they are treated for dispatching like any other event. The timeout objects themselves are passive in the sense that they do not contain timers.

The ID of a timeout event is always `Timeout_Event_id`. This enables event consumers to distinguish timeouts from other events. Timeouts can be distinguished from one another by a special ID called `timeoutId`.

Scheduling a Timeout

The following sequence diagram shows a scheduled timeout.

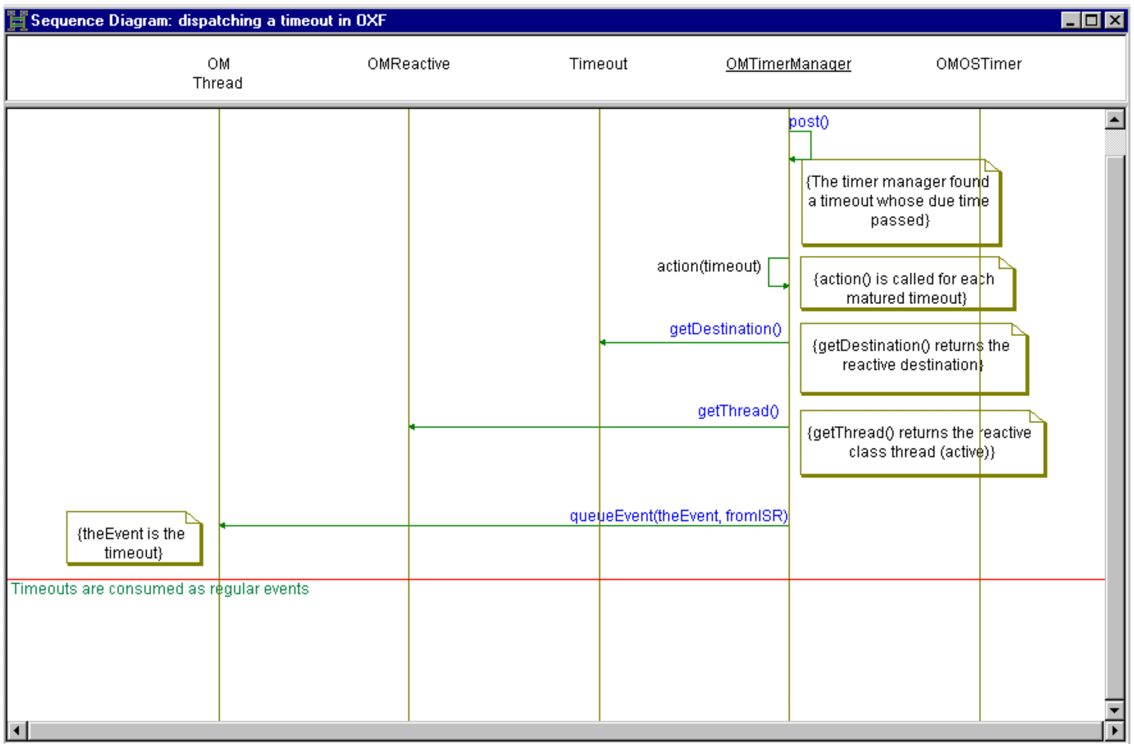


To schedule a timeout, follow these steps:

1. A user class calls the **schedTm** method to create a timeout request.
2. The **schedTm** method calls the **incarnateTimeout** method to create a timeout request for the reactive object.
3. The constructor for the `OMTimeout` class, **OMTimeout**, creates a new timeout event.
4. The **schedTm** method delegates the timeout request to `OMTimerManager`.
5. The **schedTm** method calls the **set** method to delegate the timeout request to `OMTimerManager`.

Dispatching a Timeout

The following sequence diagram shows a dispatched timeout.



To queue the timeout event, follow these steps:

1. The `timeTickCbk` method (private) is called to increment `m_Time`, the accumulated or current time.
2. The `timeTickCbk` method calls `post` (private) to get the next scheduled timeout request from the heap, trim the heap, and move the timeout to the matured list.
3. The **getDestination** method returns the reactive destination.
4. The **getThread** method returns the reactive class thread.
5. The `post` method calls the **queueEvent** method to queue the timeout request to the relevant thread as an event.

After the timeout event reaches the head of the event queue, the **takeEvent** method is used by the event loop (within the thread) to request that the reactive object process the event.

Unscheduling a Timeout

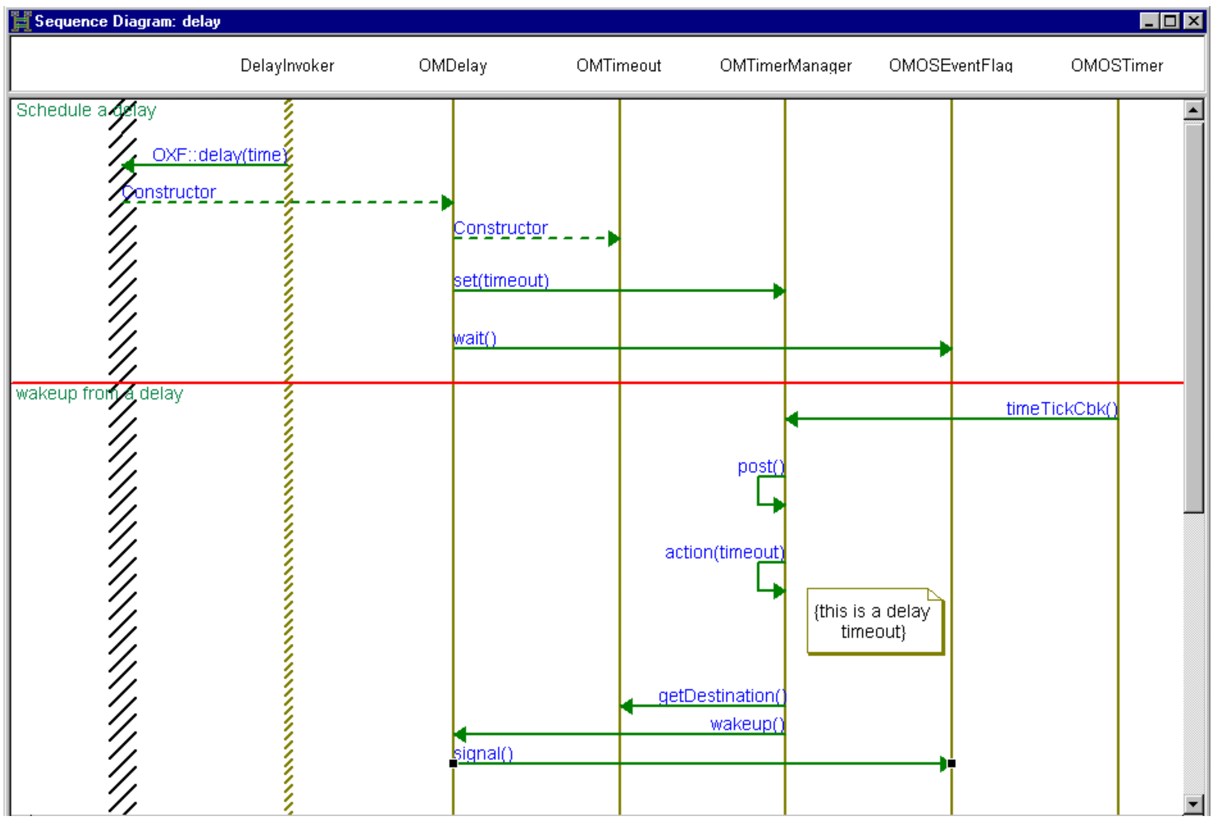
You unschedule a timeout in the following cases:

- When a state that caused the timeout is exited before the timeout expires
- During the cancellation of events upon the destruction of an `OMReactive` instance

A user class calls the **unschedTm** method to cancel a timeout request. If the timeout request was posted but not consumed, it is marked as a canceled event (an event that is not delegated to its destination). If the timeout request was not posted, it is removed from the timeout manager.

Delaying a Timeout

The following sequence diagram shows a delayed timeout.



To schedule the delay, follow these steps:

1. The `OMDelay` constructor creates a delay.
2. The `set` method delegates a timeout request to `OMTimerManager`.
3. The delay waits until the timeout is over, at which point the `timeTickCbk` method (private) is called. The `timeTickCbk` method increments `m_Time`, the accumulated or current time.
4. The `timeTickCbk` method calls `post` (private) to get the next scheduled timeout request from the heap, trim the heap, and move the timeout to the matured list.
5. The `action` method sends a matured timeout request to the relevant thread, where it is then inserted into the thread's event queue. Because the timeout is a delay (`isNotDelay = False`), the thread is the receiver.
6. The `action` method calls `getDestination`, which returns the current value of the `destination` attribute (an `OMReactive` instance).
7. The `action` method calls `wakeup`, which resumes processing after the delay time has expired.
8. `signal()` actually wakes up the thread blocking on the event flag.

Miscellaneous Topics

This section provides information on miscellaneous topics, including active classes; state machines; model debugging, testing, and analysis; configuring execution framework properties; and the list of OXF files.

Active and Reactive Classes

An *active* object is one that runs on its own task (thread), with a message queue available on the task object. A *reactive* object is one that has a mechanism for consuming events and triggered operations. In Rhapsody, an object is reactive if it fulfills any of the following conditions:

- Has a statechart
- Receives events and triggered operations
- Is a composite

Using Rhapsody, you can:

- Create active classes and objects that are not reactive.
- Create and control the behavior of reactive classes or objects with or without a statechart.

Active Classes that are Not Reactive

To create an active class that is not reactive, do the following:

1. Create a class and set its concurrency to active. If the class is active but not reactive, you must call `start()` to activate the event loop.
2. Override the `OMThread::execute` method, which implements the event loop. If you override a framework method, do not animate the overridden method.

Reactive Classes that Consume Events Without Statecharts

To create a reactive class that consumes events without a statechart, do the following:

1. Create a class.
2. Add an event reception or a triggered operation to the class.
3. Override the `OMReactive::`**consumeEvent** method, which implements the event consumption algorithm.

For more information on the **consumeEvent** method, see **consumeEvent**.

Classes with Statecharts Only as Documentation of Behavior

You can create statecharts as behavioral documentation only—without generating code for them.

To create a statechart for documentation only, do the following:

1. Create a class and give it a statechart.
2. Set the `ImplementStatechart` property for the class (under `CG::Class`) to `Cleared`.

Modifying Class Event Consumption

To add functionality to a class's event consumption, do the following:

1. Create a class and give it a statechart.
2. Override the `OMReactive::`**consumeEvent** operation to implement the additional functionality.

State Machines

Rhapsody supports UML state machines (which are mapped to Rhapsody statecharts), which are inspired by, and are very similar to, Harel statecharts. This includes hierarchical state decomposition (orthogonal or states), parameter-carrying events, time events, pseudo states (initial, history, join, fork, junction, and choice), completion transitions, entry and exit actions, and other features. It also includes an asynchronous event-handling model as defined in the UML—each class that has a statechart is reactive, so it has an associated event manager (an active class). The event manager queues events as they arrive, and later dispatches them into the reactive class for processing according to its statechart.

The kinds of events supported in Rhapsody were described in previous sections. As explained, time events are realized in timeouts (`OMTimeout`), which are specialized events (`OMEvent`). Timeouts can be used as transition triggers, written as `tm(n)`. This signals to the event that *n* milliseconds have passed since the transition's source state was entered.

The UML defines run-to-completion semantics for statecharts. It asserts that events are consumed one by one, where the processing of the next event does not start until the previous one has been fully consumed. Thus, each event can be viewed as transforming the statechart from one stable configuration to another. In Rhapsody, the consumption of a given event includes the ("internal") injection of all (enabled) completion transitions—the latter do not enter the event queue. This complies with the UML requirement that completion transitions be dispatched before any other queued event.

Model Debugging, Testing, and Analysis

The correctness of real-time systems has an extra dimension to it vis-à-vis other systems—in addition to functional or logical correctness, real-time systems typically carry timing requirements that must be met. The process of testing a system in that respect is called *schedulability analysis*.

There are two primary ways of accomplishing this:

1. Empirically, by injecting test data into the system and measuring its reactions.
2. Theoretically, by applying a mathematical analysis method, which can calculate the overall performance given enough timing information about the system components. *Rate monotonic analysis* is an example of such a method. This kind of analysis is usually done using special tools.

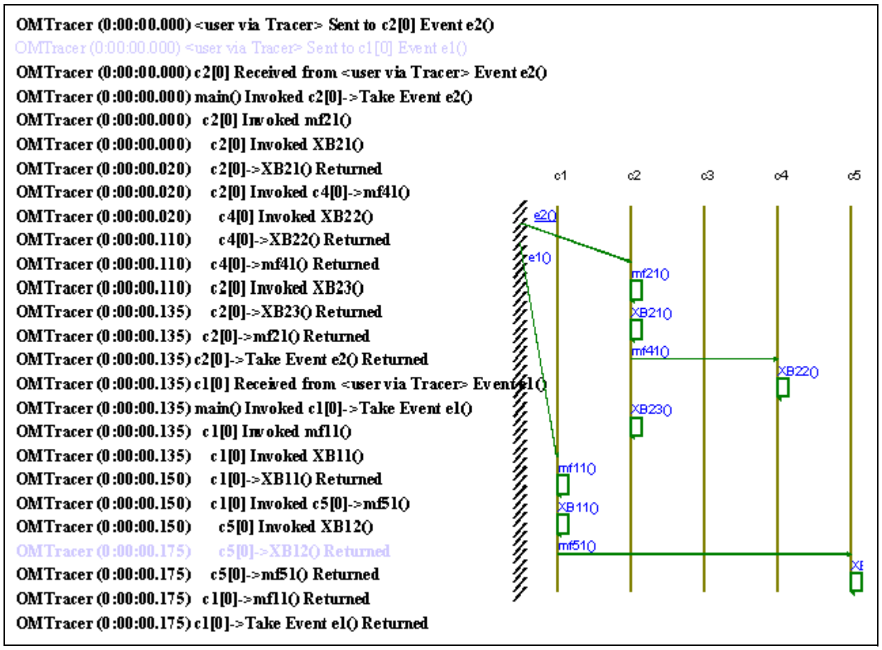
Rhapsody facilitates model-level debugging through animated statecharts and sequence diagrams. You can step through the application at an “object-oriented granularity” (operation call, one event processing, the whole event queue) and visually observe the effect on the statechart (for example, change of active state), and on the sequence diagram (for example, message/event arrows are drawn as they are sent). These capabilities are supported by various framework elements that are beyond the scope of this manual.

Stepping through an application is a good way to test the functional aspects of a system. But most importantly for real-time applications, you can use Rhapsody for empirical schedulability analysis, as follows:

1. First, assign estimated durations for the execution of operations.
2. Next, write a driver that simulates the injection of external events into the system. The driver can be a script or a statechart that generates events.
3. Next, activate the driver and the system reacts as programmed, simulating the time required to perform the operations. While running, Rhapsody generates an animated sequence diagram and a time-stamped trace. You can inspect these outputs to see if the deadlines have been met. The Time-Stamped Execution Trace and Sequence Diagram shows sample trace information.

This performance simulation can be run either on the development host or on the target machine. If you run it on a target machine, you have the advantage of measuring response times of the real target operating system.

Time-Stamped Execution Trace and Sequence Diagram



The duration of operations is an example of a Quality of Service (QoS) parameter. There are many QoS parameters that are relevant to schedulability analysis. For example, in the level of classes, QoS parameters include jitter, minimum arrival time, average arrival time, execution time, blocking time, and so on. Values for these parameters are needed to perform schedulability analysis in both the empirical and theoretical ways.

One important goal of future real-time extensions to the UML is to identify an appropriate set of QoS timeliness properties. The natural mechanism to do that would be UML-tagged values.

Rhapsody has an extensible property mechanism that closely corresponds to the notion of UML-tagged values. In fact, the QoS parameters mentioned previously, as well as some others, are currently supported as properties, but they are only informative.

Configuring Framework Properties

You can configure some of the OXF properties directly from within Rhapsody.

The Framework Files

The Rhapsody in C++ framework files are located in the directory `<install_dir>/LangCpp/oxf`. The following table lists some of the more important OXF files.

Key Framework Files

File	Description
`AMemAloc.h`	Contains declarations for the abstract interface for static memory allocation
`event.h`	Contains declarations for the `OMEvent`, `OMStartBehaviorEvent`, and `OMTimeout` classes
`event.cpp`	Contains the implementation of the `OMEvent`, `OMStartBehaviorEvent`, and `OMTimeout` classes
`MemAlloc.h`	Contains declarations for static memory allocation
`omabscon.h`	Contains declarations of the abstract container classes (`OMAbstractContainer` and `OMIterator`)
`omcollec.h`	Contains the declaration of the `OMCollection` class, which is an unordered, unbounded container based on a dynamic version of `OMStaticArray`
`omcon.h`	Contains common declarations for the basic `OMContainer` library
`omheap.h`	Contains the declaration of the `OMHeap` class
`omiotypes.h`	Contains the generic stream types mapped to either the vendor streams or standard library streams, based on the `OM_STL` compilation flag
`omlist.h`	Contains the declaration of the `OMList` class
`ommap.h`	Contains the declaration of the `OMMap` class
`ommemorymanager.h`	Contains declarations for the classes that support the new memory management functionality introduced in Version 3.0.1
`ommemorymanager.cpp`	Contains the implementation of the memory management functionality

Key Framework Files (Continued)

File	Description
omoutput.h	Contains reporting messages for OMNotifyToError and OMNotifyToOutput
omoutput.cpp	Contains reporting messages for OMNotifyToError and OMNotifyToOutput
omprotected.h	Contains declarations for the OMProtected and OMGuard classes, and the guard macros
omqueue.h	Contains the declaration of the OMQueue class, which is an unordered, bounded, or unbounded queue
omreactive.h	Contains declarations for the OMReactive class and the GEN macros
omreactive.cpp	Contains the implementation of the OMReactive class
omstack.h	Defines a stack template
omstatic.h	Contains the declaration of the OMStaticArray class
omstring.h	Contains definitions of the string types
omstring.cpp	Contains the implementation of the string types
omthread.h	Contains declarations for the OMThread, OMMainThread, and OMDelay classes
omthread.cpp	Contains the implementation of the OMThread, OMMainThread, and OMDelay classes
omtypes.h	Contains declarations for the basic types
os.h	Contains declarations for the operating system package
oxf.h	Contains declarations for the Behavioral package, OXF::init, and isRealTimeModel
oxf.cpp	Contains the implementation of the execution framework layer, OXF::init, and OXF::start
rawtypes.h	Contains declarations of the basic types
state.h	Contains declarations for abstract state behaviors
state.cpp	Contains the implementation of state behaviors
timer.h	Contains declarations for the OMTimerManager, OMThreadTimer, and OMTimerManagerDefaults classes
timer.cpp	Contains the implementation of the OMTimerManager, OMThreadTimer, and OMTimerManagerDefaults classes

Key Framework Files (Continued)

File	Description
`<x>os.h`	Contains declarations for the concrete operating system (for example, `ntos.h`, `PsosOS.h`, `VxOS.h`, and `linuxos.h`)
`<x>os.cpp`	Contains the implementation of the concrete operating system (for example, `ntos.cpp`, `PsosOS.cpp`, `VxOS.cpp`, and `linuxos.cpp`)
`<x>oxf.mak`	Contains the make files for the concrete operating system (for example, `bc5oxf.mak`, `linuxoxf.mak`, `msceoxf.mak`, and `msoxf.mak`)

Customizing the Framework

The Rhapsody framework was designed so it could be easily customized by creating classes that inherit from the framework classes. You could do this within Rhapsody by creating a class that inherits from an external class that represents the framework.

For example, to modify the active thread that Rhapsody uses, create a class in the model called `OMThread` and set its `CG::Class::UseAsExternal` property to `Checked`. You could then create a new class in the model, `MyThread`, that defines the `OMThread` class as a superclass. By modifying `MyThread`, you can modify the framework virtual operations or add more attributes to the framework classes.

To have the code generator use the customized behavior, set the appropriate properties (such as `CPP_CG::Framework::ActiveBase`). It is important to note that following this process facilitates upgrading to new releases of Rhapsody because no changes are done in the framework code itself. Note that all changes in the framework for a given release are documented in the *Upgrade Guide*. Before upgrading to a new version, review the changes to determine whether they impact your framework customization.

Note

The Rhapsody code generator gives special treatment to the classes specified in the framework base class properties. You should always use the framework base class properties if a base class is derived from a framework class.

OXF Reference Pages

This section contains reference pages for the classes and methods that comprise the OXF. Note that only the public and protected methods are documented.

For ease-of-use, the classes are presented in alphabetical order. Within each class, the methods are listed in the following order:

1. Constructor
2. Destructor
3. Operators
4. Methods, listed in alphabetical order.

The classes are as follows:

- OMAbstractMemoryAllocator Class
- OMAbstractTickTimerFactory Class
- OMAndState Class
- OMCollection Class
- OMComponentState Class
- OMDelay Class
- OMEvent Class
- OMFinalState Class
- OMFriendStartBehaviorEvent Class
- OMFriendTimeout Class
- OMGuard Class
- OMHeap Class
- OMInfiniteLoop Class
- OMIterator Class
- OMLeafState Class
- OMList Class

OMAbstractMemoryAllocator Class

`OMAbstractMemoryAllocator` is the abstract interface for static memory allocation. The abstract class is defined in the header file `AMemAloc.h`; the header file `MemAlloc.h` contains methods for static memory allocation.

Construction Summary

~OMAbstractMemoryAllocator	Destroys the `OMAbstractMemoryAllocator` object

Method Summary

allocPool	Allocates a memory pool big enough to hold the specified number of instances
callMemoryPoolIsEmpty	Controls the overprint of the message displayed when the pool is out of memory
getMemory	Gets the memory for an instance
initiatePool	Initiates the “bookkeeping” for the allocated pool
OMSelfLinkedMemoryAllocator	Constructs the memory allocator
returnMemory	Returns memory from the specified instance
setAllocator	Sets the allocation method
setIncrementNum	Overwrites the increment value

~OMAbstractMemoryAllocator

Visibility

`Public`

Description

The ~OMAbstractMemoryAllocator method is the destructor for the `OMAbstractMemoryAllocator` class.

This method was added to support user-defined memory managers.

Signature

```
virtual ~OMAbstractMemoryAllocator()
```

allocPool

Visibility

Public

Description

The **allocPool** method allocates a memory pool big enough to hold the specified number of instances.

Signature

```
T * allocPool(int numOfInstances);
```

Parameters

`numOfInstances`

The maximum number of instances the pool should be able to contain

callMemoryPoolIsEmpty

Visibility

Public

Description

The **callMemoryPoolIsEmpty** method controls the overprint of the message displayed when the pool is out of memory.

Signature

```
void callMemoryPoolIsEmpty(OMBoolean b)
```

Parameters

`b`

A Boolean value that specifies whether to overprint a message when the pool is out of memory

getMemory

Visibility

Public

Description

The **getMemory** method gets the memory for an instance.

Signature

```
void* getMemory(size_t size)
```

Parameter

```
size
```

Specifies the size of the memory to be allocated

Return

The memory for an instance

See Also

returnMemory

initiatePool

Visibility

```
Public
```

Description

The **initiatePool** method initiates the “bookkeeping” for the allocated pool.

Signature

```
int initiatePool(T * const newBlock, int numOfInstances);
```

Parameters

```
newBlock
```

The default amount of memory to allocate

```
numOfInstances
```

The maximum number of instances that the pool should be able to hold

OMSelfLinkedMemoryAllocator

Visibility

`Public`

Description

The **OMSelfLinkedMemoryAllocator** method constructs the memory allocator, specifies whether it is protected, and how much additional memory should be allocated if the initial pool is exhausted.

Signature

```
OMSelfLinkedMemoryAllocator(int incrementNum,
   OMBoolean isProtected);
```

Parameters

`incrementNum`

Specifies how much additional memory to allocate if the initial pool is exhausted.

`isProtected`

Specifies a Boolean value that determines whether the memory allocator is protected. Set this to `TRUE` to protect the allocator.

returnMemory

Visibility

`Public`

Description

The **returnMemory** method returns the memory from the specified instance.

Signature

```
void returnMemory(void *deadObject, size_t size)
```

Parameters

`deadObject`

A pointer to the memory

`size`

The size of the allocated memory

Return

The memory from the specified instance

See Also

getMemory

setAllocator

Visibility

`Public`

Description

The **setAllocator** method sets the allocation method.

Signature

```
void setAllocator(T * (*newAllocator)(int))
```

Parameters

`newAllocator`

The callback called when the pool runs out of memory

setIncrementNum

Visibility

`Public`

Description

The **setIncrementNum** method overwrites the increment value.

Signature

```
void setIncrementNum(int value)
```

Parameters

`value`

The new increment value

OMAbstractTickTimerFactory Class

The `OMAbstractTickTimerFactory` class is the abstract base class for a user-defined, low-level timer factory.

The class is defined in the header file `timer.h`.

Method Summary

createRealTimeTimer	Creates a real-time timer
createSimulatedTimeTimer	Creates a simulated-time timer
TimerManagerCallBack	Is a callback of the timer manager

createRealTimeTimer

Visibility

`Public`

Description

The **createRealTimeTimer** method creates a real-time timer. Every tick time, the timer should call `TimerManagerCallBack(callBackParams)`.

This method returns a handle to the timer, so it can be deleted when the timer manager is destroyed.

Signature

```
virtual OMOSTimer* createRealTimeTimer(timeUnit tickTime,
   TimerManagerCallBack, void* callBackParams) const =0;
```

Parameters

`tickTime`

Specifies the tick time.

`TimerManagerCallBack`

The call to the callback function. The callback should be called every tick time.

`callBackParams`

Specifies the parameters for the callback function.

Return

The `OMOSTimer`

See Also

TimerManagerCallBack

createSimulatedTimeTimer

Visibility

Public

Description

The createSimulatedTimeTimer method creates a simulated-time timer. Every tick time, the timer should call `TimerManagerCallBack(callBackParams)`.

This method returns a handle to the timer, so it can be deleted when the timer manager is destroyed.

Signature

```
virtual OMOSTimer* createSimulatedTimeTimer(
    TimerManagerCallBack, void* callBackParams) const = 0;
```

Parameters

`TimerManagerCallBack`

The call to the callback function. The callback should be called every tick time.

`callBackParams`

Specifies the parameters for the callback function.

Return

The `OMOSTimer`

See Also

TimerManagerCallBack

TimerManagerCallBack

Visibility

`Public`

Description

The **TimerManagerCallBack** method is a callback of the timer manager. which notifies the manager of the tick.

Signature

```
typedef void (*TimerManagerCallBack)(void*);
```

OMAndState Class

The `OMAndState` class contains functions that affect `And` states in statecharts.

This class is defined in the header file `state.h`.

Construction Summary

OMAndState	Constructs an `OMAndState` object

Method Summary

lock	Locks the mutex of the `OMState` object
unlock	Unlocks the mutex of the `OMState` object

OMAndState

Visibility

`Public`

Description

The **OMAndState** method is the constructor for the `OMAndState` class.

Signature

```
OMAndState(OMState* par, OMState* cmp);
```

Parameters

```
par
```

Specifies the parent

```
cmp
```

Specifies the component

lock

Visibility

`Public`

Description

The lock method locks the mutex of the `OMState` object.

Signature

```
void lock();
```

unlock

Visibility

`Public`

Description

The unlock method unlocks the mutex of the `OMState` object.

Signature

```
void unlock();
```

OMCollection Class

The `OMCollection` class contains basic library functions that enable you to create and manipulate `OMCollections`. An `OMCollection` is an unordered, unbounded container.

This class is defined in the header file `omcollec.h`.

Base Template Class

```
OMStaticArray
```

Construction Summary

OMCollection	Constructs an `OMCollection` object
~OMCollection	Destroys the `OMCollection` object

Method Summary

add	Adds the specified element to the collection
addAt	Adds the specified element to the collection at the given index
remove	Deletes the specified element from the collection
removeAll	Deletes all the elements from the collection
removeByIndex	Deletes the element found at the specified index in the collection
reorganize	Reorganizes the contents of the collection

OMCollection

Visibility

`Public`

Description

The **OMCollection** method is the constructor for the `OMCollection` class.

Signature

```
OMCollection(int theSize=DEFAULT_START_SIZE)
```

Parameters

`theSize`

The initial size of the collection. The initial collection size is `20` elements.

See Also

~OMCollection

~OMCollection

Visibility

`Public`

Description

The **~OMCollection** method is the destructor for the `OMCollection` class.

Signature

```
~OMCollection()
```

See Also

OMCollection

add

Visibility

`Public`

Description

The **add** method adds the specified element to the collection.

Signature

```
void add(Concept p)
```

Parameters

`p`

The element to add

See Also

addAt

remove

removeAll

removeByIndex

addAt

Visibility

`Public`

Description

The **addAt** method adds the specified element to the collection at the given index.

Signature

```
void addAt(int index, Concept p)
```

Parameters

`index`

The index at which to add the new element

`p`

The element to add

See Also

[add]

[remove]

[removeAll]

[removeByIndex]

remove

Visibility

`Public`

Description

The **remove** method deletes the specified element from the collection.

Signature

```
void remove(Concept p);
```

Parameters

`p`

The element to delete

See Also

add

addAt

removeAll

removeByIndex

removeAll

Visibility

`Public`

Description

The **removeAll** method deletes all the elements from the collection.

Signature

```
void removeAll();
```

See Also

add

addAt

remove

removeByIndex

removeByIndex

Visibility

`Public`

Description

The **removeByIndex** method deletes the element found at the specified index in the collection.

Signature

```
void removeByIndex(int i)
```

Parameters

`i`

The index of the element to delete

See Also

add

addAt

remove

removeAll

reorganize

Visibility

`Public`

Description

The **reorganize** method enables you to reorganize the contents of the collection.

Signature

```
void reorganize(int factor = DEFAULT_FACTOR);
```

Parameters

`factor`

Specifies the array size increment factor. For example, if the array size is `20` elements and the factor is `3`, the new array size will be `60` elements. The default factor is `2`.

OMComponentState Class

The `OMComponentState` class defines methods that affect component states in statecharts.

This class is defined in the header file `state.h`.

Flag Summary

active	Marks the component state as active

Construction Summary

OMComponentState	Constructs an `OMComponentState` object

Method Summary

enterState	Specifies the method called on the entry to the state (the entry action)
in	Checks whether the owner class is in this state
takeEvent	Takes the specified event off the queue

Flags

```
active
```

Marks the component state as active. It is defined as follows:

```
OMState* active;
```

OMComponentState

Visibility

`Public`

Description

The **OMComponentState** method is the constructor for the `OMComponentState` class.

Signature

```
OMComponentState(OMState* par = NULL)
```

Parameters

`par`

The parent

enterState

Visibility

`Public`

Description

The **enterState** method specifies the method called on the entry to the state (the entry action).

Signature

```
virtual void enterState();
```

in

Visibility

`Public`

Description

The **in** method checks whether the owner class is in this state. This method is used by the `IS_IN()` macro.

Signature

```
int in();
```

takeEvent

Visibility

Public

Description

The **takeEvent** method takes the specified event off the event queue.

Signature

```
virtual int takeEvent(short lId);
```

Parameters

lId

Specifies the event ID

OMDelay Class

`OMDelay` is used to delay a calling thread. `OMDelay` is essentially another way of issuing a timeout—`OMDelay` calls it on its own.

`OMDelay` is normally used by the application. If a reactive instance creates an `OMDelay`, it will get a timeout after the specified delay time.

This class is defined in the header file `omthread.h`.

Flag Summary

stopDelay	Initiates the delay

Construction Summary

OMDelay	Constructs an `OMDelay` object
~OMDelay	Destroys the `OMDelay` object

Method Summary

wakeup	Resumes processing after the delay time has expired

Flag

stopDelay

Initiates the delay. The syntax is as follows:

```
OMOSEventFlag* stopSignal;
```

The `OMOSEventFlag` class is defined in `os.h`.

OMDelay

Visibility

`Public`

Description

The **OMDelay** method is the constructor for the `OMDelay` class.

Signature

```
OMDelay (timeUnit t);
```

Parameters

`t`

Specifies the delay, in milliseconds

See Also

~OMDelay

~OMDelay

Visibility

`Public`

Description

The **~OMDelay** method is the destructor for the `OMDelay` class.

Signature

```
~OMDelay()
```

See Also

OMDelay

wakeup

Visibility

`Public`

Description

The **wakeup** method resumes processing after the delay time has expired.

Signature

```
void wakeup();
```

OMEvent Class

OMEvent is the base class for all events defined in Rhapsody and from which the code generator implicitly derives all events. OMEvent is an abstract class and is declared in the file event.h.

OMEvent has two important data attributes:

- destination—Every event "knows" which OMReactive started it. When the thread wants to send the event to its destination, it looks to the destination attribute to find the target OMReactive instance.
- lId—Every event has an ID. Rhapsody code generation automatically generates sequential IDs, but you can also specify the ID associated with an event. You might want to do this, for example, to maintain the ID across compilation, add more events, do special things with an event, or use a specific ID because you are sending it out of the application.

You can specify the event ID in the Rhapsody properties at two levels: an individual event ID or a base ID number for every package. Using the base number, Rhapsody assigns every event a sequential ID number.

Every object and event that inherits from OMEvent can add additional data to store event-specific information. For example, if you want to send an event with the current time, you can add an attribute with the relevant type name and the event will have access to the additional data.

Event parameters are mapped by code generation to data members of event classes that inherit from OMEvent.

OMEvent is also the base class for two special kinds of events:

- **timeout event**—In addition to the lId attribute for an event, a timeout has a Timeout attribute. The code generator automatically generates different timeouts. The Timeout attribute specifies how long to wait until the timeout is expired and activated. The Timeout attribute specifies the absolute time when the timeout will be executed (m_Time + Timeout).
- **delay event**—The delay event is used infrequently. Its purpose is to delay a thread. When the thread gets a delay event, it pauses for the delay time.

Events are normally generated in two steps, which are encapsulated within the GEN macro in the framework:

1. An event class is instantiated, resulting in a pointer to the event.
2. The event is queued by adding the new event pointer to the receiver's event queue.

Once the event has been instantiated and added to the event queue of the receiver, the event is ready to be "sent." The success of the send operation relies on the assumption that the memory address space of the sender and receiver are the same. However, this is not always the case.

For example, the following are some examples of scenarios in which the sender and receiver memory address spaces are most likely different:

- The event is sent between different processes in the same host.
- The event is sent between distributed applications.
- The sender and receiver are mapped to different memory partitions.

One common way to solve this problem is to *marshall* the information. Marshalling means to convert the event into raw data, send it using frameworks such as publish/subscribe, and then convert the raw data back to its original form at the receiving end. High-level solutions, such as CORBA®, automatically generate the necessary code, but with low-level solutions, you should take explicit care. Rhapsody allows you to specify how to marshall, and not marshall, events and instances by creating “standard operations” to handle this task.

For low-level solutions, you may use one of these partial animation methods:

- In the same selected component, using properties to enable/disable the animation of specific packages, classes, and so on.
- Mix animated and non-animated components in the same executable.

To support partial animation, C++ code generation has the following characteristics:

- Inheritance of user classes and events from `AOM` elements was canceled.
- For each animated user class (event), a friend class is created in the code. The friend class is responsible for the animation of the user class.
- All the animation-specific methods are now part of the animation `friend` class.

To support partial animation, OXF has the following characteristics:

- Inheritance from `AOM` classes was canceled (`OMEvent` and `OMReactive`).
- Attributes that were protected by `#ifdef _OMINSTRUMENT` are now regular attributes, with default values that can be handled by the non-animated version of the framework.
- Animation friend classes were added for the framework-visible events.

Attribute Summary

deleteAfterConsume	Determines whether an event should be deleted after it is consumed
destination	Specifies an `OMReactive` instance
frameworkEvent	Specifies whether an event is a framework event
lId	Specifies a value for an event ID

Constant Summary

OMEventAnyEventId	Is a reserved event ID that specifies any event
OMCancelledEventId	Is a reserved event ID that specifies a canceled event (an event that should not be sent to its destination)
OMEventNullId	Is a reserved event ID used to consume null transitions
OMEventStartBehaviorId	Is a reserved event ID used for `OMStartBehavior` events
OMEventOXFEndEventId	Is a reserved event ID used to cleanly close the framework when a COM server that uses the framework DLL is deleted
OMEventTimeoutId	Is a reserved event ID used for timeouts

Construction Summary

OMEvent	Constructs an `OMEvent` object
~OMEvent	Destroys the `OMEvent` object

Method Summary

Delete	Deletes an event instance (releases the memory used by an event)
getDestination	Returns the reactive destination of the event
getlId	Returns the event ID
isCancelledTimeout	Determines whether the event is canceled
isDeleteAfterConsume	Returns `TRUE` if the event should be deleted by the event dispatcher (`OMThread`) after its consumption
isFrameworkEvent	Returns `TRUE` if the event is an internal framework event
isRealEvent	Returns `TRUE` if the event is a null-transition event, timeout, or user event
isTimeout	Returns `TRUE` if the event is a timeout
isTypeOf	Returns `TRUE` if the event is from a given type (has the specified ID)
setDeleteAfterConsume	Determines whether the event should be deleted by the event dispatcher (`OMThread`) after it is consumed

setDestination	Sets the event reactive destination
setFrameworkEvent	Sets the event to be considered as a internal framework event
setIId	Sets the event ID

Attributes

deleteAfterConsume

This protected attribute determines whether an event should be deleted after it is consumed. The possible values for this flag are as follows:

- `TRUE`—An event should be deleted after it is consumed. This is the default value.
- `FALSE`—An event should not be deleted after it is consumed.

By default, every event is deleted after it is consumed by the statechart. The thread sends the event, the reactive does what has to be done to consume the event, and when there is nothing left to do, the thread (which maintains the event queue) deletes the event.

`deleteAfterConsume` controls whether to delete the event. You might choose not to delete an event, especially when events are statically allocated. In such cases, you should set `deleteAfterConsume` to `FALSE`.

It is defined as follows:

```
OMBoolean deleteAfterConsume;
```

destination

This protected attribute specifies an `OMReactive` instance.

It is defined as follows:

```
OMReactive* destination;
```

The `OMReactive` class is defined in `omreactive.h`.

frameworkEvent

This protected attribute specifies whether an event is a framework event. The possible values are as follows:

- `TRUE`—The event is a framework event.
- `FALSE`—The event is a user event. This is the default value.

Some events are used internally within the Rhapsody framework; these events require special attention. For example, some internal events should not be instrumented in order to minimize system overhead. If `frameworkEvent` is set to `TRUE`, less information is gathered for the event.

Typically, you will not need to change the default value of `frameworkEvent`.

It is defined as follows:

```
OMBoolean frameworkEvent;
```

lId

This protected attribute specifies a value for an event ID.

Every event has an ID. Code generation automatically generates sequential IDs, but you can also specify the ID associated with an event. You might want to do this, for example, to maintain the ID across compilation, add more events, do special things with an event, or use a specific ID because you are sending it out of the application.

You can specify the event ID in the Rhapsody properties at two levels:

- Specify an individual event ID.
- Specify a base ID number for every package. Using the base number, Rhapsody assigns every event a sequential ID number.

It is defined as follows:

```
short lId;
```

See the **Constants** section for the list of constant values for `lId`.

Constants

OMEventAnyEventId

This is a reserved event ID that specifies any event.

It is defined is as follows:

```
const short OMEventAnyEventId = -4;
```

OMCancelledEventId

This is a reserved event ID that specifies a canceled event (an event that should not be sent to its destination).

It is defined as follows:

```
const short OMEventCancelledEventId = -3;
```

OMEventNullId

This is a reserved event ID used to consume null transitions. It is defined as follows:

```
const short OMEventNullId = -1;
```

OMEventStartBehaviorId

This is a reserved event ID used for `OMStartBehavior` events.

It is defined as follows:

```
const short OMEventStartBehaviorId = -5;
```

OMEventOXFEndEventId

This is a reserved event ID used to cleanly close the framework when a COM server that uses the framework DLL is deleted.

It is defined as follows:

```
const short OMEventOXFEndEventId = -6;
```

OMEventTimeoutId

This is a reserved event ID used for timeouts.

It is defined as follows:

```
const short OMEventTimeoutId = -2;
```

OMEvent

Visibility

`Public`

Description

The **OMEvent** method is the constructor for the `OMEvent` class.

Signature

```
OMEvent (short plId = 0, OMReactive* pdest = NULL);
```

Parameters

`plId`

Specifies the event ID. The default value is `0`.

`pdest`

Specifies the destination `OMReactive` instance. The default value is NULL.

Notes

Events are generated by applying the **gen** method. The **gen** method calls **queueEvent** to queue events to be processed by the thread event loop. The `gen` method is expanded by the **GEN** macro, which also creates the event. See **Macros**for the description of the `GEN` macro.

See Also

gen

~OMEvent

queueEvent

~OMEvent

Description

The **~OMEvent** method is the destructor for the `OMEvent` class.

Signature

```
virtual ~OMEvent()
```

See Also

OMEvent

Delete

Visibility

`Public`

Description

The **Delete** method deletes an event instance (releases the memory used by an event). The `Delete` method is used instead of the standard `delete` operation to support the static memory allocation of events by Rhapsody.

Use only this method to delete events.

Signature

```
virtual void Delete()
```

Notes

If the **deleteAfterConsume** attribute is `TRUE`, the **execute** method calls `Delete` to delete the event.

See Also

execute

getDestination

Visibility

`Public`

Description

The **getDestination** method returns the reactive destination of the event.

Signature

```
OMReactive *getDestination() const
```

Return

The **destination**, which is an `OMReactive` instance

Notes

The `getDestination` method is called by the `OMTimerManager::action` method. It is also called by the `OMThread::`**execute** method to determine the `OMReactive` destination for an event.

See Also

- action
- destination
- execute
- setDestination

getlId

Visibility

`Public`

Description

The getlId method returns the event ID.

Signature

```
short getlId() const
```

Return

`lId`, the value for the event ID

See Also

- lId
- setlId

isCancelledTimeout

Visibility

`Public`

Description

The **isCancelledTimeout** method determines whether the event is canceled.

Signature

```
OMBoolean isCancelledTimeout() const
```

Returns

The method returns one of the following Boolean values:

- `TRUE`—The value of `lId` is **OMCancelledEventId**.
- `FALSE`—The value of `lId` is not **OMCancelledEventId**.

See Also

getlId

lId

setlId

isDeleteAfterConsume

Visibility

`Public`

Description

The **isDeleteAfterConsume** method returns `TRUE` if the event should be deleted by the event dispatcher (`OMThread`) after its consumption.

This method is called by the `OMThread::`**execute** method.

Signature

```
OMBoolean isDeleteAfterConsume() const
```

Returns

The method returns one of the following values:

- TRUE—The event should be deleted after it is consumed.
- FALSE—The event should not be deleted after it is consumed.

See Also

deleteAfterConsume

execute

setDeleteAfterConsume

isFrameworkEvent

Visibility

Public

Description

The isFrameworkEvent method returns TRUE if the event is an internal framework event.

Signature

```
OMBoolean isFrameworkEvent() const
```

Return

The method returns one of the following Boolean values:

- TRUE—The event is a framework event.
- FALSE—The event is not a framework event.

See Also

frameworkEvent

setFrameworkEvent

isRealEvent

Visibility

Public

Description

The **isRealEvent** method returns `TRUE` if the event is a null-transition event, timeout, or user event.

Signature

```
OMBoolean isRealEvent() const
```

Returns

The method returns one of the following Boolean values:

- `TRUE`—The value of `lId` is either **OMEventNullId** or **OMEventTimeoutId**.
- `FALSE`—The value of `lId` is neither **OMEventNullId** nor **OMEventTimeoutId**, or is a user event.

See Also

getlId

lId

setlId

isTimeout

Visibility

Public

Description

The **isTimeout** method returns `TRUE` if the event is a timeout.

Signature

```
OMBoolean isTimeout() const
```

Returns

The method returns one of the following Boolean values:

- `TRUE`—The value of `lId` is **OMEventTimeoutId**.

- FALSE—The value of lId is not OMEventTimeoutId.

See Also

getIId

lId

setIId

isTypeOf

Visibility

Public

Description

The isTypeOf method checks whether the event is from a given type (has the specified ID).

Client events should override this method, as follows:

```
OMBoolean isTypeOf(short id) const {
     if (id == <event>Id) return TRUE;
      return <super event>::isTypeOf(id);
```

Signature

```
virtual OMBoolean isTypeOf(short id) const
```

Parameters

id

Specifies the event ID to check for

Returns

The method returns one of the following Boolean values:

- TRUE—The event has the specified ID.
- FALSE—The event does not have the specified ID.

Note

To handle the consumption of derived events in a generic manner, use the isTypeOf method. With this method, the generated code checks the event type. The isTypeOf method returns TRUE for derived events, as well as for the actual event.

setDeleteAfterConsume

Visibility

Public

Description

The **setDeleteAfterConsume** method determines whether the event should be deleted by the event dispatcher (`OMThread`) after it is consumed.

Signature

```
void setDeleteAfterConsume (OMBoolean doDelete)
```

Parameters

`doDelete`

Specifies the value of the `deleteAfterConsume` attribute. The possible values are as follows:

- `TRUE`—Delete the event after it is consumed.
- `FALSE`—Do not delete the event after it is consumed.

See Also

deleteAfterConsume

isDeleteAfterConsume

setDestination

Visibility

`Public`

Description

The **setDestination** method sets the event reactive destination.

This method is called by the `OMReactive::`**_gen** method when an object is sending an event to an `OMReactive` object.

Signature

```
void setDestination (OMReactive* cb)
```

Parameters

`cb`

Specifies the `OMReactive` instance

See Also

_gen

getDestination

setFrameworkEvent

Visibility

`Public`

Description

The **setFrameworkEvent** method sets the event to be considered as a internal framework event.

Signature

```
void setFrameworkEvent (OMBoolean isFrameworkEvent)
```

Parameters

`isFrameworkEvent`

Specifies the value of the `frameworkEvent` attribute. The possible values are as follows:

- `TRUE`—The event is a framework event.
- `FALSE`—The event is not a framework event.

See Also

[frameworkEvent]

[isFrameworkEvent]

setlId

Visibility

`Public`

Description

The **setlId** method sets the event ID.

Signature

```
void setlId (short pId)
```

Parameters

`pId`

Specifies the new event ID

See Also

getlId

lId

unschedTm

OMFinalState Class

The `OMFinalState` class represents a *final state*—a state that has no exiting transitions and that make its parent state completed (`isCompleted()` returns `true`).

This class is defined in the header file `state.h`.

Construction Summary

OMFinalState	Constructs an `OMFinalState` object

Method Summary

getConcept	Returns the current element

OMFinalState

Visibility

`Public`

Description

The **OMFinalState** method is the constructor for the `OMFinalState` class.

Signature

```
OMFinalState(OMReactive * cpt, OMState * par,
   OMState * cmp, const char * hdl = NULL)
```

```
OMFinalState (OMReactive * cpt, OMState * par,
   OMState * cmp, const char * /* hdl */ = NULL)
```

Parameters

`cpt`

The statechart owner

`par`

The parent

`cmp`

The component

`hdl`

The handle

getConcept

Visibility

Public

Description

The **getConcept** method returns the current element.

Signature

```
virtual AOMInstance * getConcept() const
```

Return

The current element

OMFriendStartBehaviorEvent Class

The `OMFriendStartBehaviorEvent` class was added to animate the start behavior event class in instrumented mode. The friend class declaration is empty for non-instrumented code.

This class is defined in the header file `event.h`.

Construction Summary

OMFriendStartBehaviorEvent	Is the constructor for the `OMStartBehaviorEvent` class

Method Summary

cserialize	Is part of the Rhapsody animation serialization mechanism
getEventClass	Returns the event class
serialize	Is called during animation to send event information

OMFriendStartBehaviorEvent

Visibility

`Public`

Description

The OMFriendStartBehaviorEvent method is the constructor for the `OMFriendStartBehaviorEvent` class.

Signature

```
OMFriendStartBehaviorEvent(OMStartBehaviorEvent*
    userEventPtr);
```

Parameter

`userEventPtr`

A pointer to the event

cserialize

Visibility

Public

Description

The **cserialize** method is part of the animation serialization mechanism. It passes the values of the instance to a string, which is then sent to Rhapsody.

Signature

```
OMSData* cserialize(OMBoolean withParameters) const;
```

Parameter

```
withParameters
```

A Boolean value that specifies whether to include the parameter values

getEventClass

Visibility

Public

Description

The **getEventClass** method returns the event class. This method is used for animation purposes.

Signature

```
AOMEventClass * getEventClass() const
```

serialize

Visibility

`Public`

Description

The **serialize** method is called during animation to send event information.

Signature

```
void serialize (AOMSEvent* e) const;
```

Parameters

`e`

Specifies the event

OMFriendTimeout Class

The `OMFriendTimeout` class animates the timeout class in instrumented mode. The friend class declaration is empty for non-instrumented code.

This class is defined in the header file `event.h`.

Construction Summary

OMFriendTimeout	Is the constructor for the `OMFriendTimeout` class

Method Summary

cserialize	Is part of the Rhapsody animation serialization mechanism
getEventClass	Returns the event class
serialize	Is called during animation to send event information

OMFriendTimeout

Visibility

Public

Description

The OMFriendTimeout method is the constructor for the `OMFriendTimeout` class.

Signature

```
OMFriendTimeout(OMTimeout* userEventPtr)
```

Parameters

userEventPtr

A pointer to the timeout event

cserialize

Visibility

`Public`

Description

The **serialize** method is part of the animation serialization mechanism. It passes the values of the instance to a string, which is then sent to Rhapsody.

Signature

```
OMSData* cserialize(OMBoolean withParameters) const;
```

Parameters

`withParameters`

A Boolean value that specifies whether to include the parameter values

getEventClass

Visibility

`Public`

Description

The **getEventClass** method returns the event class. This method is used for animation purposes.

Signature

```
AOMEventClass * getEventClass() const
```

serialize

Visibility

`Public`

Description

The **serialize** method is called during animation to send event information.

Signature

```
void serialize(AOMSEvent * e) const
```

Parameters

`e`

Specifies the event

OMGuard Class

OMGuard is used to make user operations guarded or locked between entry and exit. It is used in the generated code (in the GUARD_OPERATION macro) to ensure appropriate locking and freeing of the mutex in a guarded operation.

The copy constructor and assignment operator of OMGuard are explicitly disabled to avoid erroneous unlock of the guarded object mutex.

This class is defined in the header file omprotected.h.

Macro Summary

END_REACTIVE_GUARDED_SECTION	Ends protection of a section of code used for a reactive object
END_THREAD_GUARDED_SECTION	Stops protection for an operation of an active user object
GUARD_OPERATION	Guards an operation by an OMGuard class object
START_DTOR_REACTIVE_GUARDED_SECTION	Starts protection of a section of code used for destruction of a reactive instance
START_DTOR_THREAD_GUARDED_SECTION	Starts protection for an active user object destructor
START_REACTIVE_GUARDED_SECTION	Starts protection of a section of code used for a reactive object
START_THREAD_GUARDED_SECTION	Starts protection for an operation of an active user object

Construction Summary

OMGuard	Constructs an OMGuard object
~OMGuard	Destroys the OMGuard object

Method Summary

getGuard	Gets the guard
lock	Locks the mutex of the OMGuard object
unlock	Unlocks the mutex of the OMGuard object

Macros

END_REACTIVE_GUARDED_SECTION

Ends protection of a section of code used for a reactive object. This macro is called in the reactive class event dispatching to prevent a "race" between the event dispatching and a deletion of the reactive class instance. The mechanism is activated when the reactive class `DTOR` is set to be guarded.

END_THREAD_GUARDED_SECTION

Stops protection for an operation of an active user object. The macro is used in `OMThread` event dispatching to guard the event dispatching from deletion of the active object. The mechanism is activated in the code generated for active classes, when the active class `DTOR` is set to be guarded.

The `START_THREAD_GUARDED_SECTION` macro and the `END_THREAD_GUARDED_SECTION` macros are called by the **execute** method if **toGuardThread** is `TRUE`.

GUARD_OPERATION

Guards an operation by an `OMGuard` class object. It is used in the generated code.

This macro supports the aggregation of `OMProtected` in guarded classes as well as inheritance from `OMProtected` by guarded classes.

OMDECLARE_GUARDED

Aggregates `OMProtected` objects inside guarded classes instead of inheritance from `OMProtected`. It is defined as follows:

```
#define OMDECLARE_GUARDED
  public:
     inline void lock() const {m_omGuard.lock();}
     inline void unlock() const
        {m_omGuard.unlock();}
     inline const OMProtected& getGuard()
         const {return m_omGuard;}
```

START_DTOR_REACTIVE_GUARDED_SECTION

Starts protection of a section of code used for destruction of a reactive instance. This macro is called in the `DTOR` of a reactive (not active) class when it is set to guarded. This is done to prevent a "race" (between the deletion and the event dispatching) when deleting a reactive instance.

START_DTOR_THREAD_GUARDED_SECTION

Starts protection for an active user object destructor. This macro is called in the `DTOR` of an active class when it is set to guarded. This is done to prevent a "race" (between the deletion and the event dispatching) when deleting an active instance.

START_REACTIVE_GUARDED_SECTION

Starts protection of a section of code used for a reactive object. This macro is called in the reactive class event dispatching to prevent a "race" between the event dispatching and a deletion of the reactive class instance. The mechanism is activated when the reactive class `DTOR` is set to be guarded.

START_THREAD_GUARDED_SECTION

Starts protection for an operation of an active user object. The macro is used in `OMThread` event dispatching to guard the event dispatching from deletion of the active object. The mechanism is activated in the code generated for active classes when the active class `DTOR` is set to be guarded.

The `START_THREAD_GUARDED_SECTION` macro and the `END_THREAD_GUARDED_SECTION` macros are called by the **execute** method if **toGuardThread** is `TRUE`.

OMGuard

Visibility

`Public`

Description

The `OMGuard` method is the constructor for the `OMGuard` class. It locks the mutex of the user object.

Signature

```
OMGuard (const OMProtected& pObj,
    bool needInstrumentation = true);
```

Parameters

`pObj`

Specifies a guarded user object

`needInstrumentation`

Added for animation support

See Also

~OMGuard

~OMGuard

Visibility

`Public`

Description

The `~OMGuard` method is the destructor for the `OMGuard` class. It frees the mutex of the guarded object.

Signature

```
~OMGuard()
```

See Also

OMGuard

getGuard

Visibility

`Public`

Description

The **getGuard** method gets the guard object.

Signature

```
inline const OMProtected& getGuard() const
```

Return

The guard object

lock

Visibility

`Public`

Description

The **lock** method locks the mutex of the `OMGuard` object.

Signature

```
inline void lock() const
```

unlock

Visibility

`Public`

Description

The **unlock** method unlocks the mutex of the `OMGuard` object.

Signature

```
inline void unlock() const
```

OMHeap Class

The `OMHeap` class contains basic library functions that enable you to create and manipulate `OMHeap` objects. An `OMHeap` is a type-safe, fixed size heap implementation. An `OMHeap` has elements of type `Node*`.

This class is defined in the header file `omheap.h`.

Construction Summary

OMHeap	Constructs an `OMHeap` object
~OMHeap	Destroys the `OMHeap` object

Method Summary

add	Adds the specified element to the heap.
find	Looks for the specified element in the heap.
isEmpty	Determines whether the heap is empty.
remove	Deletes the specified element from the heap.
top	Moves the iterator to the top of the heap.
trim	Deletes the top of the heap.
update	This method is currently unused.

OMHeap

Visibility

`Public`

Description

The **OMHeap** method is the constructor for the `OMHeap` class.

Signature

```
OMHeap(int size=100)
```

Parameters

`size`

The amount of memory to allocate for the heap. The default size is `100` bytes.

See Also

~OMHeap

~OMHeap

Visibility

`Public`

Description

The **~OMHeap** method destroys the `OMHeap` object.

Signature

```
~OMHeap()
```

See Also

OMHeap

add

Visibility

`Public`

Description

The **add** method adds the specified element to the heap.

Signature

```
void add(Node* e);
```

Parameters

`e`

The element to add to the heap

find

Visibility

`Public`

Description

The **find** method looks for the specified element in the heap.

Signature

```
int find(Node* clone) const;
```

Parameters

`clone`

The element to look for

Return

The method returns one of the following values:

- `0`—The element was not found.
- `1`—The element was found.

isEmpty

Visibility

`Public`

Description

The **isEmpty** method determines whether the heap is empty.

Signature

```
int isEmpty() const
```

Return

The method returns one of the following values:

- `0`—The heap is not empty.
- `1`—The heap is empty.

remove

Visibility

`Public`

Description

The **remove** method removes the first occurrence of the specified element from the heap.

Signature

```
Node* remove(Node* clone);
```

Parameters

`clone`

The element to delete

Return

If successful, the method returns the deleted element. Otherwise, it returns `NULL`.

top

Visibility

Public

Description

The **top** method moves the iterator to the top of the heap.

Signature

```
Node* top() const
```

Return

The top-most element

trim

Visibility

Public

Description

The **trim** method deletes the top of the heap.

Signature

```
void trim();
```

update

Visibility

Public

Description

Currently, this method is unused.

Signature

```
void update(Node* e);
```

OMInfiniteLoop Class

`OMInfiniteLoop` is an exception class that should be raised on an infinite loop of null transitions. It is currently not used by the execution framework.

It is declared in the header file `omreactive.h`.

OMIterator Class

The `OMIterator` class contains methods that enable you to use a standard iterator for all the classes derived from `OMAbstractContainer`.

This class is defined in the header file `omabscon.h`.

Construction Summary

OMIterator	Constructs an `OMIterator` object

Method Summary

operator *	Returns the current value of the iterator
operator ++	Increments the iterator
increment	Increments the iterator by 1
reset	Resets the iterator to the beginning or the specified location
value	Returns the value found at the current position

OMIterator

Visibility

`Public`

Description

The **OMIterator** method is the constructor for the `OMIterator` class.

Signature

```
OMIterator();

OMIterator(const OMAbstractContainer<Concept>& l)

OMIterator(const OMAbstractContainer<Concept>* l)
```

Parameters

`l`

The container the iterator will visit

operator *

Visibility

`Public`

Description

The `*` operator returns the current value of the iterator.

Signature

```
Concept& operator*()
```

Return

The current value of the iterator

operator ++

Visibility

`Public`

Description

The ++ operator increments the iterator.

Signature

```
OMIterator<Concept>& operator++()

OMIterator<Concept> operator++(int i)
```

Parameters

`i`

Increments the iterator to the next element in the container

Return

The incremented value of the iterator

increment

Visibility

`Public`

Description

The **increment** method increments the iterator by 1.

Signature

```
OMIterator<Concept>& increment()
```

Return

The new value of the iterator

reset

Visibility

`Public`

Description

The **reset** method resets the iterator to the beginning or the specified location.

Signatures

```
void reset()
```

```
void reset(OMAbstractContainer<Concept>& newLink)
```

Parameters for Signature 2

`newLink`

The new position for the iterator

value

Visibility

`Public`

Description

The **value** method returns the element found at the current position.

Signature

```
Concept& value()
```

Return

The element found at the current position

OMLeafState Class

The `OMLeafState` class sets the active state of the component.

This class is defined in the header file `state.h`.

Construction Summary

OMLeafState	Creates an `OMLeafState` object

Flag Summary

component	Specifies a component

Method Summary

entDef	Specifies the operation called when the state is entered from a default transition
enterState	Specifies the state entry action
exitState	Specifies the state exit action
in	Returns `TRUE` when the owner class is in this state
serializeStates	Is called during animation to send state information

Flags

component

Specifies a component. It is defined as follows:

```
OMComponentState* component;
```

OMLeafState

Visibility

Public

Description

The **OMLeafState** method is the constructor for the `OMLeafState` class.

Signature

```
OMLeafState(OMState* par, OMState* cmp)
```

Parameters

`par`

Specifies the parent

`cmp`

Specifies the component

entDef

Visibility

Public

Description

The **entDef** method specifies the operation called when the state is entered from a default transition.

Signature

```
virtual void entDef();
```

enterState

Visibility

Public

Description

The enterState method specifies the state entry action

Signature

```
virtual void enterState();
```

exitState

Visibility

Public

Description

The exitState method specifies the state exit action.

Signature

```
virtual void exitState();
```

in

Visibility

Public

Description

The in method returns `TRUE` when the owner class is in this state.

Signature

```
int in();
```

Return

The method returns one of the following values:

- `0`—Not in
- `1`—In

serializeStates

Visibility

`Public`

Description

The **serializeStates** method is called during animation to send state information.

Signature

```
virtual void serializeStates (AOMSState* s) const;
```

Parameters

`s`

Specifies the state

OMList Class

The `OMList` class contains basic library functions that enable you to create and manipulate `OMLists`. An `OMList` is a type-safe, linked list.

This class is defined in the header file `omlist.h`.

Base Template Class

`OMStaticArray`

Construction Summary

OMList	Constructs an `OMList` object
~OMList	Destroys the `OMList` object

Flag Summary

first	Specifies the first element in the list
last	Specifies the last element in the list

Method Summary

operator []	Returns the element at the specified position
add	Adds the specified element to the end of the list
addAt	Adds the specified element to the list at the given index
addFirst	Adds an element at the beginning of the list
find	Looks for the specified element in the list
getAt	Returns the element found at the specified index
getCount	Returns the number of elements in the list
getCurrent	Is used by the iterator to get the element at the current position in the list
getFirst	Is used by the iterator to get the first position in the list
getFirstConcept	Returns the first `Concept` element in the list

getLast	Is used by the iterator to get the last position in the list
getLastConcept	Returns the last `Concept` element in the list
getNext	Is used by the iterator to get the next position in the list
isEmpty	Determines whether the list is empty
_removeFirst	Removes the first item from the list.=
remove	Deletes the first occurrence of the specified element from the list
removeAll	Deletes all the elements from the list
removeFirst	Deletes the first element from the list
removeItem	Deletes the specified element from the list
removeLast	Deletes the last element from the list

Flags

first

Specifies the first element in the list. It is defined as follows:

```
OMListItem<Concept>* first;
```

last

Specifies the last element in the list. It is defined as follows:

```
OMListItem<Concept>* last;
```

Example

Consider the following example:

```
OMIterator<Observer*> iter(itsObserver);
    while (*iter)
    {
       (*iter)->notify();
       iter++;
    }
```

OMList

Visibility

`Public`

Description

The **OMList** method is the constructor for the `OMList` class. The method creates an empty list.

Signature

```
OMList()
```

See Also

~OMList

~OMList

Visibility

`Public`

Description

The **~OMList** method empties the list.

Signature

```
virtual ~OMList()
```

See Also

OMList

operator []

Visibility

`Public`

Description

The `[]` operator returns the element at the specified location.

Signature

```
Concept& operator [](int i) const
```

Parameters

`i`

The index of the element to return

add

Visibility

`Public`

Description

The **add** method adds the specified element to the end of the list.

Signature

```
void add(Concept c);
```

Parameter

`c`

The element to add to the end of the list

See Also

addAt

addFirst

remove

removeAll

removeFirst

addAt

Visibility

```
Public
```

Description

The **addAt** method adds the specified element to the list at the given index.

Signature

```
void addAt(int i, Concept c);
```

Parameters

```
i
```

The list index at which to add the element

```
c
```

The element to add

See Also

add

addFirst

remove

removeAll

removeFirst

removeLast

addFirst

Visibility

`Public`

Description

The **addFirst** method adds an element at the beginning of the list.

Signature

```
void addFirst(Concept c);
```

Parameters

`c`

The element to add at the beginning of the list

See Also

- **add**
- **addAt**
- **remove**
- **removeAll**
- **removeFirst**
- **removeLast**

find

Visibility

`Public`

Description

The **find** method looks for he specified element in the list.

Signature

```
int find(Concept c) const;
```

Parameters

`c`

The element to look for

Return

The method returns one of the following values:

- `0`—The element was not found.
- `1`—The element was found.

getAt

Visibility

`Public`

Description

The **getAt** method returns the element found at the specified index.

Signature

```
Concept& getAt (int i) const;
```

Parameters

`i`

The index of the element to retrieve

Return

The element found at the specified index

See Also

[getCount]

[getCurrent]

[getFirst]

[getLast]

[getNext]

getCount

Visibility

`Public`

Description

The **getCount** method returns the number of elements in the list.

Signature

```
int getCount() const;
```

Return

The number of elements in the list

getCurrent

Visibility

`Public`

Description

The **getCurrent** method is used by the iterator to get the element at the current position in the list.

Signature

```
virtual Concept& getCurrent(void* pos) const
```

Parameters

`pos`

The position

Return

The element (`Concept`) at the current position in the list

getFirst

Visibility

`Public`

Description

The **getFirst** method is used by the iterator to get the first position in the list.

Signature

```
virtual void getFirst(void*& pos) const
```

Parameters

`pos`

The first position in the list

See Also

getLast

getNext

getFirstConcept

Visibility

`Public`

Description

The **getFirstConcept** method returns the first `Concept` element in the list.

Signature

```
Concept& getFirstConcept() const
```

Return

The first `Concept` element in the list

See Also

getLastConcept

getLast

Visibility

Public

Description

The **getLast** method is used by the iterator to get the last position in the list.

Signature

```
virtual void getLast(void*& pos) const
```

Parameters

pos

The last position in the list

See Also

getFirst

getNext

getLastConcept

Visibility

Public

Description

The **getLastConcept** method returns the last `Concept` element in the list.

Signature

```
Concept& getLastConcept() const
```

Return

The last `Concept` element in the list

See Also

getFirstConcept

getNext

Visibility

`Public`

Description

The **getNext** method is used by the iterator to get the next position in the list.

Signature

```
virtual void getNext(void*& pos) const
```

Parameters

`pos`

The next position in the list

See Also

getFirst

getLast

isEmpty

Visibility

`Public`

Description

The **isEmpty** method determines whether the list is empty.

Signature

```
int isEmpty() const
```

Return

The method returns one of the following values:

- `0`—The list is not empty.
- `1`—The list is empty.

_removeFirst

Visibility

`Public`

Description

The **_removeFirst** method removes the first item from the list.

Note

It is safer to use the method **removeFirst** because that method has more checks than **_removeFirst**.

Signature

```
inline void _removeFirst()
```

See Also

removeFirst

remove

Visibility

`Public`

Description

The **remove** method deletes the first occurrence of the specified element from the list.

Signature

```
void remove(Concept c);
```

Parameters

`c`

The element to delete

See Also

add

addAt

removeAll

removeFirst

removeLast

removeAll

Visibility

Public

Description

The removeAll method deletes all the elements from the list.

Signature

```
void removeAll()
```

See Also

add

addAt

remove

removeFirst

removeLast

removeFirst

Visibility

Public

Description

The removeFirst method deletes the first element from the list.

Signature

```
void removeFirst()
```

See Also

add

addAt

remove

removeAll

removeLast

removeItem

Visibility

Public

Description

The removeItem method deletes the specified element from the list.

Signature

```
void removeItem(OMListItem<Concept> *item);
```

Parameters

item

The item to delete

See Also

add

addAt

remove

removeAll

removeFirst

removeLast

removeLast

Visibility

`Public`

Description

The **removeLast** method deletes the last element from the list.

Note

This method is not efficient because the Rhapsody framework does not keep backward pointers. It is recommended that you use one of the other `remove` functions to delete elements from the list.

Signature

```
void removeLast()
```

See Also

add

addAt

remove

removeAll

removeFirst

removeItem

OMListItem Class

The `OMListItem` class is a helper class for `OMList` that contains functions that enable you to manipulate list elements.

This class is defined in the header file `omlist.h`.

Construction Summary

OMListItem	Constructs an `OMListItem` object

Method Summary

connectTo	Connects the list item to the list
getNext	Gets the next item in the list

OMListItem

Visibility

`Public`

Description

The OMListItem method is the constructor for the `OMListItem` class.

Signature

```
OMListItem(const Concept& theConcept)
```

Parameters

`theConcept`

The new list element

connectTo

Visibility

`Public`

Description

The **connectTo** method connects the specified list item to the list.

Signature

```
void connectTo(OMListItem *item)
```

Parameters

`item`

The list item

getNext

Visibility

`Public`

Description

The **getNext** method gets the next item in the list.

Signature

```
OMListItem<Concept>* getNext() const
```

Return

The next item in the list

OMMainThread Class

OMMainThread is a special case of OMThread that defines the default, active class of the application. By default, this class takes control over the application's main thread (see the **start** method for detailed information). The OMMainThread class is a singleton—only one instance is created.

This class is declared in omthread.h.

Base Class

OMThread

Construction Summary

~OMMainThread	Destroys the OMMainThread object

Method Summary

destroyThread	Cleans up the singleton instance of OMMainThread
instance	Creates and retrieves the singleton instance of OMMainThread
start	Starts the singleton event loop (OMThread::execute) of the main thread singleton

~OMMainThread

Visibility

`Public`

Description

The `~OMMainThread` method is the destructor for the `OMMainThread` class.

Signature

```
virtual ~OMMainThread()
```

destroyThread

Visibility

`Public`

Description

The **destroyThread** method cleans up the singleton instance of `OMMainThread`. This method overrides the method `OMThread::destroyThread`.

Signature

```
virtual void destroyThread()
```

instance

Visibility

`Public`

Description

The `instance` method creates and retrieves the singleton instance of `OMMainThread`.

Signature

```
static OMThread* instance (int create = 1);
```

Parameters

`create`

Specifies whether an instance should be created. If this is set to 1, an `OMMainThread` instance is created.

If `create` is set to `0`, the `instance` method returns one of the following values:

- The singleton instance, if it already exists
- NULL, if the instance does not exist

Return

`OMThread*`

Notes

If a main thread does not exist, `OMMainThread` creates one and returns `OMMainThread`. If a main thread already exists, `OMMainThread` returns the `OMMainThread`.

start

Visibility

`Public`

Description

The `start` method starts the singleton event loop (`OMThread::execute`).

Signature

```
virtual void start (int doFork = 0);
```

Parameters

`doFork`

Specifies whether the `OMMainThread` singleton event loop should run on the application main thread (`doFork == 0`) or in a separate thread (`doFork == 1`).

Sample Use

For example, many applications require a GUI with its own library. The Rhapsody library has an event queue and a main thread, and the GUI usually has its own event queue. In order for both event queues to work together, you can start the main thread with `doFork = 1`. This starts the main thread of the GUI and forks a new thread for the Rhapsody library.

OMMap Class

The `OMMap` class contains basic library functions that enable you to create and manipulate `OMMaps`. An `OMMap` is a type-safe map, based on a balanced binary tree (`log(n)` search time).

This class is defined in the header file `ommap.h`.

Construction Summary

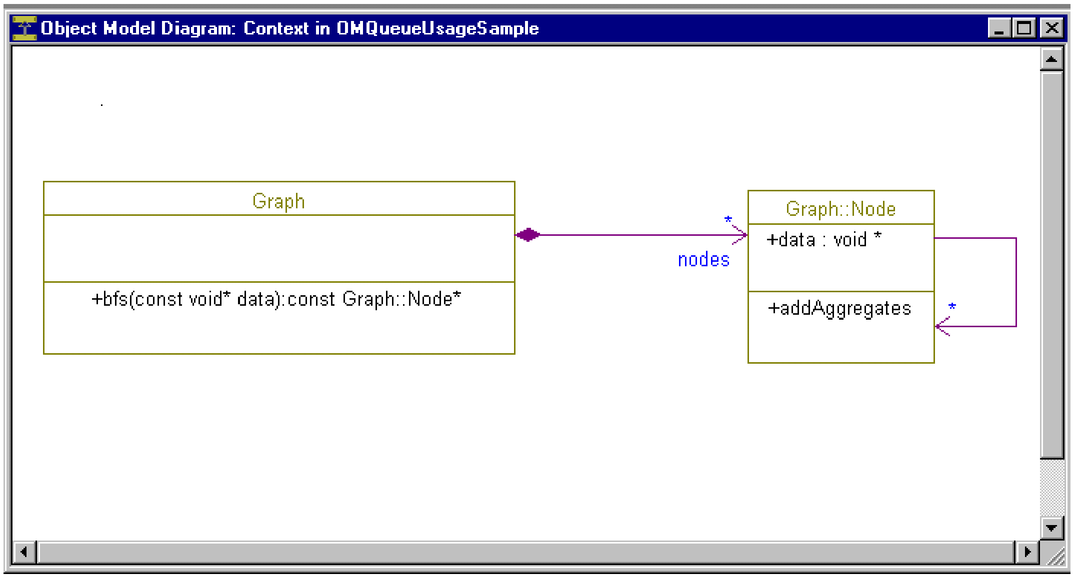
OMMap	Constructs an `OMMap` object
~OMMap	Destroys the `OMMap` object

Method Summary

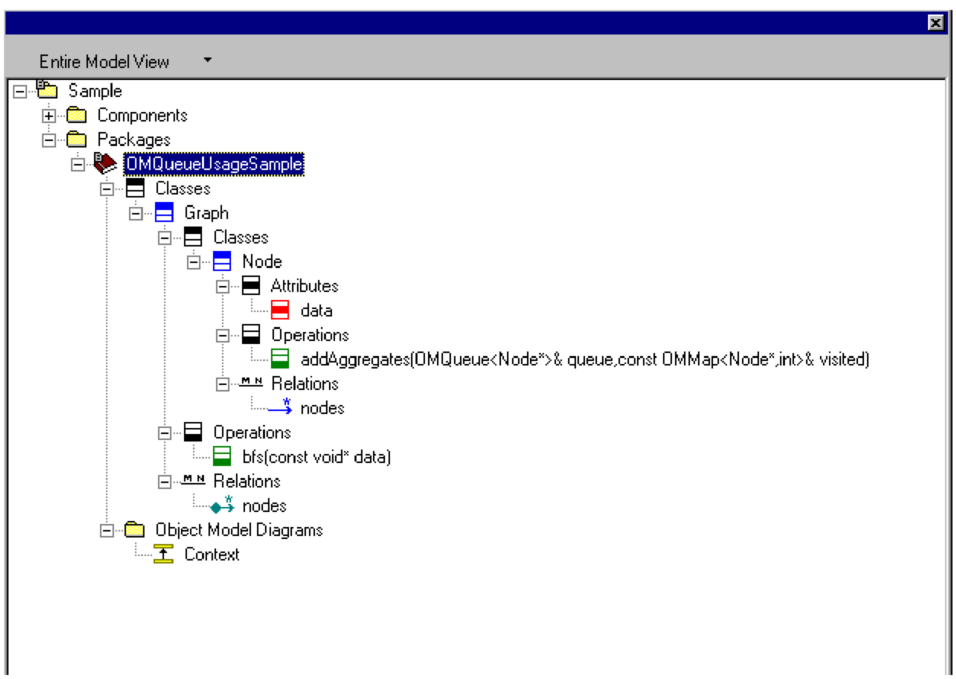
operator []	Returns the element found for the specified key
add	Adds an element to the map
find	Looks for the specified element is in the map
getAt	Returns the element for the specified key
getCount	Returns the number of elements in the map
getKey	Gets the element for the specified key
isEmpty	Determines whether the map is empty
lookUp	Looks up the specified element in the map
remove	Deletes the specified element from the map
removeAll	Deletes all the elements from the map

Example

Consider a class, `Graph`, that has a `bfs()` operation that performs BFS search on the graph nodes to find a node with the specified data. The following figure shows the OMD of the `Graph` class.

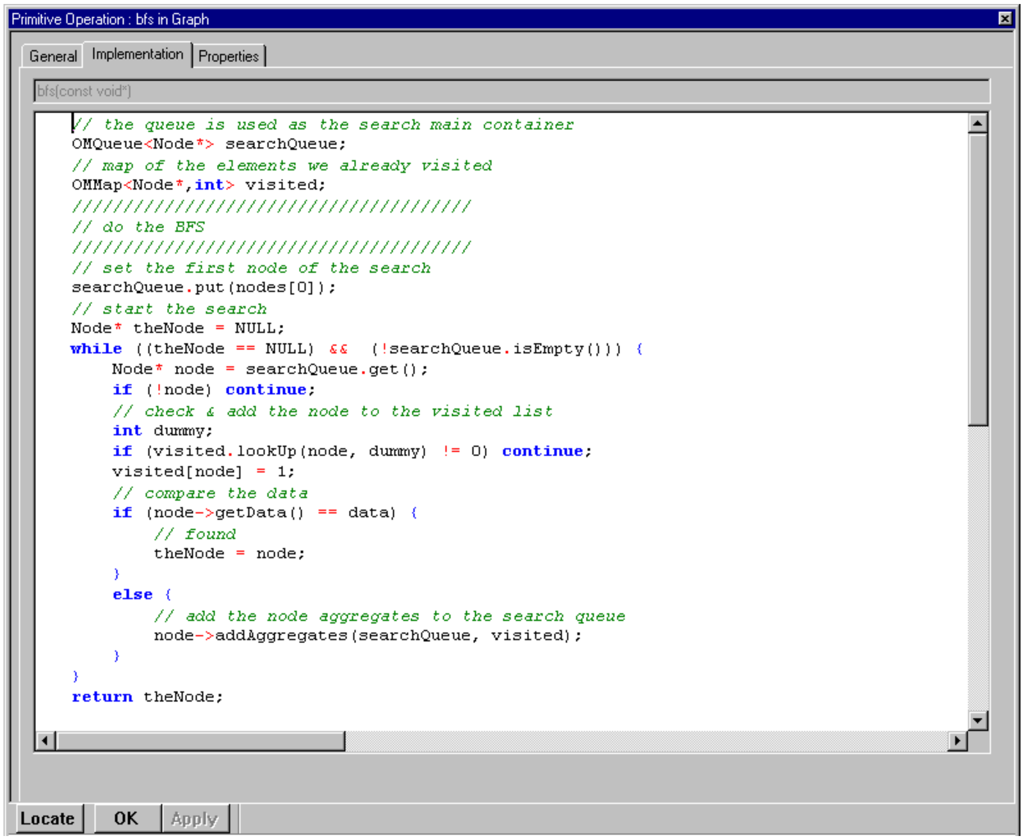


The following figure shows the browser view of the `Graph` class.

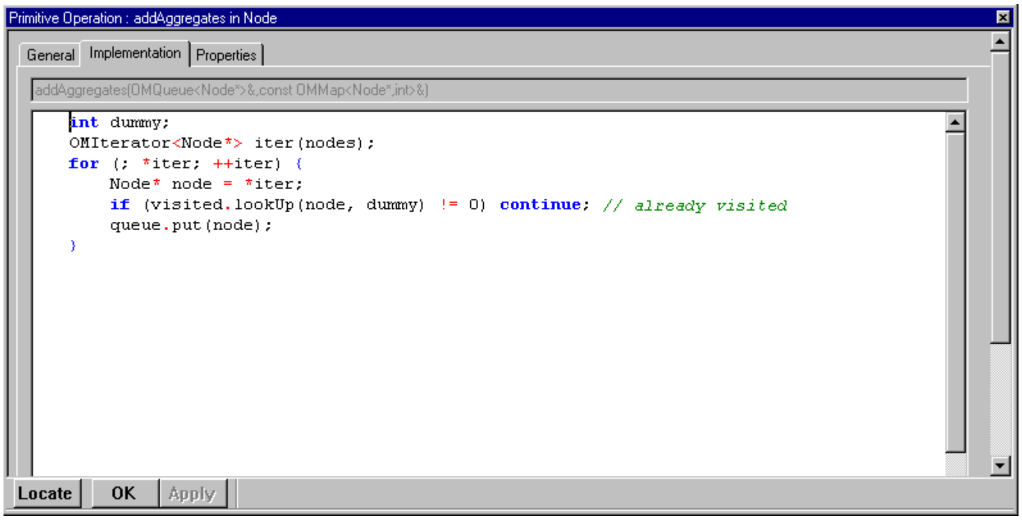


The `bfs()` implementation uses `OMQueue` as the search container and `OMMap` as a record of the visited elements.

The following figure shows the implementation of `Graph::bfs()`.



The following figure shows the implementation of `Graph::Node::addAggregates()`.



OMMap

Visibility

`Public`

Description

The **OMMap** method is the constructor for the `OMMap` class.

Signature

```
OMMap()
```

See Also

~OMMap

~OMMap

Visibility

Public

Description

The ~OMMap method destroys the OMMap object.

Signature

```
~OMMap()
```

See Also

OMMap

operator []

Visibility

Public

Description

The [] operator returns the element for the specified key.

Signature

```
Concept& operator [](const Key& k)
```

Parameters

k

The key of the element to get

Return

The element at the specified key

add

Visibility

`Public`

Description

The **add** method adds the specified element to the given key.

Signature

```
void add(Key k, Concept p);
```

Parameters

`k`

The map key to which to add the element

`p`

The element to add

See Also

remove

removeAll

find

Visibility

`Public`

Description

The **find** method looks for the specified element in the map.

Signature

```
int find(Concept p) const
```

Return

The method returns one of the following values:

- `0`—The element was not found in the map.
- `1`—The element was found.

getAt

Visibility

`Public`

Description

The **getAt** method returns the element found at the specified location.

Signature

```
Concept& getAt(int i) const;
```

Parameters

`i`

The location of the element to get

Return

The element found at the specified location

getCount

Visibility

`Public`

Description

The **getCount** method returns the number of elements in the map.

Signature

```
int getCount() const
```

Return

The number of elements in the map

getKey

Visibility

`Public`

Description

The **getKey** method gets the element for the specified key.

Signature

```
Concept& getKey(const Key& k) const
```

Parameters

`k`

The map key

Return

The element for the specified key

isEmpty

Visibility

`Public`

Description

The **isEmpty** method determines whether the map is empty.

Signature

```
int isEmpty() const
```

Return

The method returns one of the following values:

- `0`—The map is not empty.
- `1`—The map is empty.

lookUp

Visibility

`Public`

Description

The **lookUp** method determines whether the specified element is in the map. If it is, it places the contents of the concept referenced by the key in the `c` parameter, and returns the value 1.

Signature

```
int lookUp(const Key k, Concept& c) const
```

Parameters

`k`

The map key

`c`

The element to look up

Return

The method returns one of the following values:

- `0`—The element was not found in the map.

- `1`—The element was found.

remove

Visibility

`Public`

Description

The **remove** method deletes the specified element.

Signature

```
void remove(Key k)

void remove(Concept p)
```

Parameters for Signature 1

`k`

The map key of the element to delete

Parameters for Signature 2

`p`

The element to delete. The method deletes the first occurrence of the object.

See Also

add

removeAll

removeAll

Visibility

`Public`

Description

The **removeAll** method deletes all the elements from the map.

Signature

```
void removeAll()
```

See Also

add

remove

OMMapItem Class

The `OMMapItem` class is a helper class for `OMMap` that contains functions that enable you to manipulate map elements.

This class is defined in the header file `ommap.h`.

Construction Summary

OMMapItem	Constructs an `OMMapItem` object
~OMMapItem	Destroys the `OMMapItem` object

Method Summary

getConcept	Returns the current map item

OMMapItem

Visibility

Public

Description

The OMMapItem method is the constructor for the `OMMapItem` class.

Signature

```
OMMapItem(Key theKey, Concept theConcept);
```

Parameters

`theKey`

The map key

`theConcept`

The new map element

See Also

~OMMapItem

~OMMapItem

Visibility

`Public`

Description

The **~OMMapItem** method destroys the `OMMapItem` object.

Signature

```
virtual ~OMMapItem()
```

See Also

OMMapItem

getConcept

Visibility

`Public`

Description

The **getConcept** method returns the current element.

Signature

```
Concept& getConcept()
```

Return

The current element

OMMemoryManager Class

OMMemoryManager is the default memory manager for the framework. It is part of the mechanism that enables you to use custom memory managers.

The OXF had built-in memory control support for the following elements:

- All generic types except for states. There is no full support for reusable state machines.
- OS adapter support for VxWorks. To add support to other OS adapters, add OM_DECLARE_FRAMEWORK_MEMORY_ALLOCATION_OPERATORS in the adapter classes' declaration, and use the OMNEW and OMDELETE macros for buffer allocation and deletion.

The OMMemoryManager class supports user control over memory allocation.

In addition, protection against early destruction on application exit is provided. This protection ensures that the internal memory manager singleton is valid throughout the termination of the application. To accomplish this, the following members are supplied in the class:

- OMMemoryManager—A constructor
- ~OMMemoryManager–A destructor
- static bool _singletonDestroyed—A destruction indicator flag

Base Class

OMAbstractMemoryAllocator

Construction Summary

OMMemoryManager	Constructs an OMMemoryManager object
~OMMemoryManager	Destroys the OMMemoryManager object

Macro and Operator Summary

OM_DECLARE_FRAMEWORK_ MEMORY_ALLOCATION_ OPERATORS	Defines the memory allocation operators
OMDELETE	Deletes the specified memory using either the memory manager or the global delete operator (when the framework and application are compiled with OM_NO_FRAMEWORK_MEMORY_MANAGER)
OMGET_MEMORY	Allocates memory using either the memory manager or the global new operator (when the framework and application are compiled with OM_NO_FRAMEWORK_MEMORY_MANAGER)
OMNEW	Allocates memory using either the memory manager or the global new operator (when the framework and application are compiled with OM_NO_FRAMEWORK_MEMORY_MANAGER)

Method Summary

getDefaultMemoryManager	Returns the default memory manager
getMemory	Records the memory allocated by the default manager
getMemoryManager	Returns the current memory manager
returnMemory	Returns the memory from an instance

Operators and Macros

OM_DECLARE_FRAMEWORK_MEMORY_ALLOCATION_OPERATORS

The macros and operators support user control over memory allocation. The new parameter NEW_DUMMY_PARAM is set to "size_t=0" for every compiler except for Diablo, where it is set to nothing.

The updated definition is as follows:

```
define OM_DECLARE_FRAMEWORK_MEMORY_ALLOCATION_OPERATORS

public:
    static void* operator new (size_t size
        NEW_DUMMY_PARAM)
    static void* operator new[] (size_t size
        NEW_DUMMY_PARAM
    static void operator delete (void * object,
        size_t size)
    static void operator delete[] (void * object,
        size_t size)
```

OMGET_MEMORY

Allocates memory using either the memory manager or the global `new` operator (when the framework and application are compiled with `OM_NO_FRAMEWORK_MEMORY_MANAGER`).

It is defined as follows:

```
#define OMGET_MEMORY(size)
```

OMNEW

Allocates memory using either the memory manager or the global `new` operator (when the framework and application are compiled with `OM_NO_FRAMEWORK_MEMORY_MANAGER`).

It is defined as follows:

```
#define OMNEW(type, size)
```

OMDELETE

Deletes the specified memory using either the memory manager or the global delete operator (when the framework and application are compiled with the `OM_NO_FRAMEWORK_MEMORY_MANAGER` switch).

It is defined as follows:

```
#define OMDELETE(object,size)
```

OMMemoryManager

Visibility

`Public`

Description

The **OMMemoryManager** method is the constructor for the `OMMemoryManager` class.

Signature

```
OMMemoryManager(bool theFrameworkSingleton = false);
```

Parameter

`theFrameworkSingleton`

A Boolean value that specifies that this is not the memory manager singleton

~OMMemoryManager

Visibility

`Public`

Description

The **~OMMemoryManager** method is the destructor for the `OMMemoryManager` class.

Signature

```
virtual ~OMMemoryManager();
```

See Also

OMMemoryManager

getDefaultMemoryManager

Visibility

Public

Description

The **getDefaultMemoryManager** method returns the default memory manager for the framework, regardless of the manager currently being used.

Signature

```
static OMAbstractMemoryAllocator*
    getDefaultMemoryManager();
```

Return

The default memory manager for the framework

See Also

getMemory

getMemoryManager

getMemory

Visibility

Public

Description

The **getMemory** method provides the memory requested. This method is optional, and is available if you compiled the framework with the OM_ENABLE_MEMORY_MANAGER_SWITCH compiler switch.

This method is called from the framework object's new operator.

Signature

```
virtual void * getMemory (size_t size);
```

Parameter

size

Specifies the size of the memory to be allocated by the default manager

See Also

returnMemory

getMemoryManager

Visibility

Public

Description

The **getMemoryManager** method returns the current memory manager.

Signature

```
static OMAbstractMemoryAllocator* getMemoryManager();
```

Return

The current memory manager

See Also

getDefaultMemoryManager

returnMemory

Visibility

Public

Description

The **returnMemory** method returns the allocated memory.

This method is called from framework object's `delete` operator.

Signature

```
virtual void returnMemory (void * object, size_t size);
```

Parameters

`object`

A pointer to the reclaimed memory

`size`

The size of the allocated memory

See Also

getMemory

OMMemoryManagerSwitchHelper Class

`OMMemoryManagerSwitchHelper` is a singleton of the `OMMemoryManagerSwitchHelper` class. It is responsible for logging memory allocations, and enables client objects to check whether a specific memory allocation is registered.

By default, the switch helper logic is disabled. To enable it, compile the framework using the `OM_ENABLE_MEMORY_MANAGER_SWITCH` compiler switch.

Construction Summary

OMMemoryManagerSwitchHelper	Creates an `OMMemoryManagerSwitchHelper` object
~OMMemoryManagerSwitchHelper	Destroys an `OMMemoryManagerSwitchHelper` object

Method Summary

cleanup	Cleans up the allocated memory list
findMemory	Searches for a recorded memory allocation
instance	Returns the singleton instance of the `OMMemoryManagerSwitchHelper`
isLogEmpty	Determines whether the memory log is empty
recordMemoryAllocation	Records a single memory allocation
recordMemoryDeallocation	Records a single memory deallocation
setUpdateState	Specifies whether the singleton should be updated
shouldUpdate	Determines whether the singleton should be updated (and have new memory allocations recorded)

OMMemoryManagerSwitchHelper

Visibility

`Public`

Description

The **OMMemoryManagerSwitchHelper** method is the constructor for the `OMMemoryManagerSwitchHelper` class.

Signature

```
OMMemoryManagerSwitchHelper()
```

See Also

~OMMemoryManagerSwitchHelper

~OMMemoryManagerSwitchHelper

Visibility

`Public`

Description

The **~OMMemoryManagerSwitchHelper** method is the destructor for the `OMMemoryManagerSwitchHelper` class.

Signature

```
~OMMemoryManagerSwitchHelper()
```

See Also

OMMemoryManagerSwitchHelper

cleanup

Visibility

`Public`

Description

The **cleanup** method cleans up the allocated memory log.

Signature

```
void cleanup();
```

findMemory

Visibility

`Public`

Description

The **findMemory** method searches for a recorded memory allocation.

Signature

```
bool findMemory (const void*) const;
```

Return

The method returns one of the following Boolean values:

- `true`—The memory was found in the recorded memory.
- `false`—The memory was not found.

instance

Visibility

Public

Description

The **instance** method returns the singleton instance of the `OMMemoryManagerSwitchHelper`.

Signature

```
static OMMemoryManagerSwitchHelper* instance();
```

Return

The singleton instance of `OMMemoryManagerSwitchHelper`

isLogEmpty

Visibility

Public

Description

The **isLogEmpty** method determines whether the memory log is empty.

Signature

```
inline bool isLogEmpty() const
```

Return

The method returns one of the following Boolean values:

- `true`—The memory log is empty.
- `false`—The memory log is not empty.

recordMemoryAllocation

Visibility

`Public`

Description

The **recordMemoryAllocation** method records a single memory allocation. It is called by the default memory manager when the framework is compiled using the `OM_ENABLE_MEMORY_MANAGER_SWITCH` compiler switch.

Signature

```
bool recordMemoryAllocation (const void* memory);
```

Parameters

`memory`

Specifies the memory allocation to record

Return

The method returns `true` if successful; `false` otherwise.

See Also

recordMemoryDeallocation

recordMemoryDeallocation

Visibility

`Public`

Description

The **recordMemoryDeallocation** method records a single memory deallocation. It is called by the default memory manager when the framework is compiled using the `OM_ENABLE_MEMORY_MANAGER_SWITCH` compiler switch.

Signature

```
bool recordMemoryDeallocation (const void* memory);
```

Parameters

`memory`

Specifies the memory allocation to record

Return

The method returns `true` if the memory record was found and removed successfully. Otherwise, it returns `false`.

See Also

recordMemoryAllocation

setUpdateState

Visibility

Public

Description

The **setUpdateState** method specifies whether the memory log should be updated. It is called by the `OXF::`**init** method.

Signature

```
void setUpdateState (bool);
```

Parameters

```
bool
```

Set this to `true` to have the memory log updated (and have new memory allocations recorded). Otherwise, set this to `false`.

See Also

shouldUpdate

shouldUpdate

Visibility

Public

Description

The **shouldUpdate** method determines whether the memory log should be updated (and have new memory allocations recorded).

Signature

```
bool shouldUpdate() const;
```

Return

The method returns `true` if the singleton should be updated. Otherwise, it returns `false`.

See Also

setUpdateState

OMNotifier Class

The `OMNotifier` class defines methods that write messages to either the error log or to standard output.

This class is defined in the header file `oxf.h`.

Method Summary

notifyToError	Writes messages to the error log
notifyToOutput	Writes messages to standard output

notifyToError

Visibility

`Public`

Description

The notifyToError method writes messages to the error log.

Signature

```
static void notifyToError(const char *msg);
```

Parameters

`msg`

The message to display on the screen

notifyToOutput

Visibility

`Public`

Description

The **notifyToOutput** method writes messages to standard output.

Signature

```
static void notifyToOutput(const char *msg);
```

Parameters

`msg`

The message to display on the screen

OMOrState Class

The `OMOrState` class defines methods that affect Or states in statecharts.

This class is defined in the header file `state.h`.

Construction Summary

OMOrState	Constructs an `OMOrState` object

Flag Summary

subState	Specifies a substate

Method Summary

entDef	Specifies the operation called when the state is entered from a default transition
enterState	Specifies the state entry action
exitState	Specifies the state exit action
getSubState	Gets the substate
in	Returns TRUE when the owner class is in this state
serializeStates	Is called during animation to send state information
setSubState	Sets the substate

Flags

subState

Specifies a substate. It is defined as follows:

```
OMState* subState;
```

OMOrState

Visibility

`Public`

Description

The **OMOrState** method is the constructor for the `OMOrState` class.

Signature

```
OMOrState(OMState* par = NULL)
```

Parameters

`par`

Specifies the parent

entDef

Visibility

`Public`

Description

The **entDef** method specifies the operation called when the state is entered from a default transition.

Signature

```
virtual void entDef();
```

enterState

Visibility

`Public`

Description

The **enterState** method specifies the state entry action.

Signature

```
virtual void enterState();
```

exitState

Visibility

Public

Description

The **exitState** method specifies the state exit action.

Signature

```
virtual void exitState();
```

getSubState

Visibility

Public

Description

The **getSubState** method returns the substate.

Signature

```
virtual OMState* getSubState();
```

Return

The substate

in

Visibility

`Public`

Description

The **in** method returns `TRUE` when the owner class is in this state.

Signature

```
int in()
```

Return

The method returns one of the following values:

- `0`—The owner class is not in this state.
- `1`—The owner class is in this state.

serializeStates

Visibility

`Public`

Description

The **serializeStates** method is called during animation to send state information.

Signature

```
virtual void serializeStates (AOMSState* s) const;
```

Parameters

`s`

Specifies the state

setSubState

Visibility

`Public`

Description

The **setSubState** method sets the specified substate.

Signature

```
virtual void setSubState(OMState* s);
```

Parameters

`s`

Specifies the substate

OMProtected Class

`OMProtected` is the base class for protected objects. It embodies a mutex and `lock` and `unlock` methods that are automatically embedded within a concrete public method defined for the object.

This class is declared in the file `omprotected.h`.

Construction Summary

OMProtected	Constructs an `OMProtected` object
~OMProtected	Destroys the `OMProtected` object

Macro Summary

OMDECLARE_GUARDED	Aggregates `OMProtected` objects inside guarded classes instead of inheriting from `OMProtected`.

Method Summary

deleteMutex	Deletes the mutex and sets its value to NULL.
free	Is provided for backward compatibility. It calls the `unlock` method.
getGuard	Gets the guard object.
initializeMutex	Creates an RTOS mutex, if it has not been created already.
lock	Locks the mutex of the `OMProtected` object.
unlock	Unlocks the mutex of the `OMProtected` object.

Macros

OMDECLARE_GUARDED

Aggregates `OMProtected` objects inside guarded classes instead of inheriting from `OMProtected`. It is defined as follows:

```
#define OMDECLARE_GUARDED

public:
    inline void lock() const {m_omGuard.lock();}
    inline void unlock() const {m_omGuard.unlock();}
    inline const OMProtected& getGuard() const
        {return m_omGuard;}
private:
    OmProtected m_omGuard;
```

OMProtected

Visibility

`Public`

Description

The **OMProtected** method is the constructor for the `OMProtected` object.

Signatures

```
OMProtected()

OMProtected(OMBoolean createMutex)
```

Parameters

`createMutex`

A Boolean value that specifies whether to create the RTOS mutex later in the lifetime of the protected object. If you specify `TRUE`, the framework creates the mutex by calling the **initializeMutex** operation.

Notes

- `OMProtected` uses the `createOMOSMutex` method to create an `OMOSMutex` object. Initially, the mutex is free.
- `createOMOSMutex` is defined in *`xxos.cpp`*.

See Also

~OMProtected

initializeMutex

~OMProtected

Visibility

`Public`

Description

The ~**~OMProtected** method is the destructor for the `OMProtected` object. The method deletes (destroys) the operating system entity that the instance wraps.

Signature

```
~OMProtected()
```

See Also

OMProtected

deleteMutex

Visibility

`Public`

Description

The **deleteMutex** method deletes the mutex and sets its value to NULL.

Signature

```
inline void deleteMutex()
```

free

Visibility

`Public`

Description

The **free** method is provided for backward compatibility. It calls the `unlock` method.

Note

This method is not defined for OSE RTOSes.

Signature

```
void free()
```

getGuard

Visibility

`Public`

Description

The **getGuard** method gets the guard object. This allows uniform handling of guarded classes and classes the inherit from `OMProtected`.

Signature

```
inline const OMProtected& getGuard() const
```

Return

The guard object

initializeMutex

Visibility

`Public`

Description

The **initializeMutex** method creates an RTOS mutex, if it has not been created already.

Signature

```
void initializeMutex()
```

lock

Visibility

`Public`

Description

The **lock** method locks the mutex of the `OMProtected` object.

Signature

```
inline void lock() const
```

Notes

The same thread can nest `lock` and `free` calls of the same mutex without blocking itself indefinitely. This means that `OMOSMutex` can implement a recursive mutex (that is, the same thread can `lock` twice and `free` twice, but only the outer `lock` and `free` count).

See Also

unlock

unlock

Visibility

`Public`

Description

The **unlock** method unlocks the mutex of the `OMProtected` object.

Signature

```
inline void unlock() const
```

Notes

The same thread can nest `lock` and `free` calls of the same mutex without blocking itself indefinitely. This means that `OMOSMutex` can implement a recursive mutex (that is, the same thread can `lock` twice and `free` twice, but only the outer `lock` and `free` count).

See Also

lock

OMQueue Class

The `OMQueue` class contains basic library functions that enable you to create and manipulate `OMQueues`. An `OMQueue` is a type-safe, dynamically sized queue. It is implemented on a cyclic array, and implements a FIFO (first in, first out) algorithm. An `OMQueue` is implemented with `OMCollection`.

This class is defined in the header file `omqueue.h`.

Attributes and Collections

m_grow	Specifies whether the queue size can be enlarged
m_head	Specifies the head of the queue
m_myQueue	Specifies the queue implementation
m_tail	Specifies the tail of the queue

Construction Summary

OMQueue	Constructs an `OMQueue` object
~OMQueue	Destroys the `OMQueue` object

Method Summary

get	Gets the current element in the queue
getCount	Gets the number of elements in the queue
getInverseQueue	Returns the element that will be returned by `get()` in the tail of the queue
getQueue	Returns the element that will be returned by `get()` in the head of the queue
getSize	Returns the size of the memory allocated for the queue
increaseHead_	Increases the size of the queue head
increaseTail_	Increases the size of the queue tail
isEmpty	Determines whether the queue is empty
isFull	Determines whether the queue is full
put	Adds an element to the queue

Attributes and Collections

m_grow

This Boolean attribute specifies whether the queue size can be enlarged. It is defined as follows:

```
OMBoolean m_grow;
```

m_head

This attribute specifies the head of the queue. It is defined as follows:

```
int m_head;
```

m_myQueue

This collection specifies the queue implementation. `OMQueue` is implemented as a cyclic array.

It is defined as follows:

```
OMCollection<Concept> m_myQueue;
```

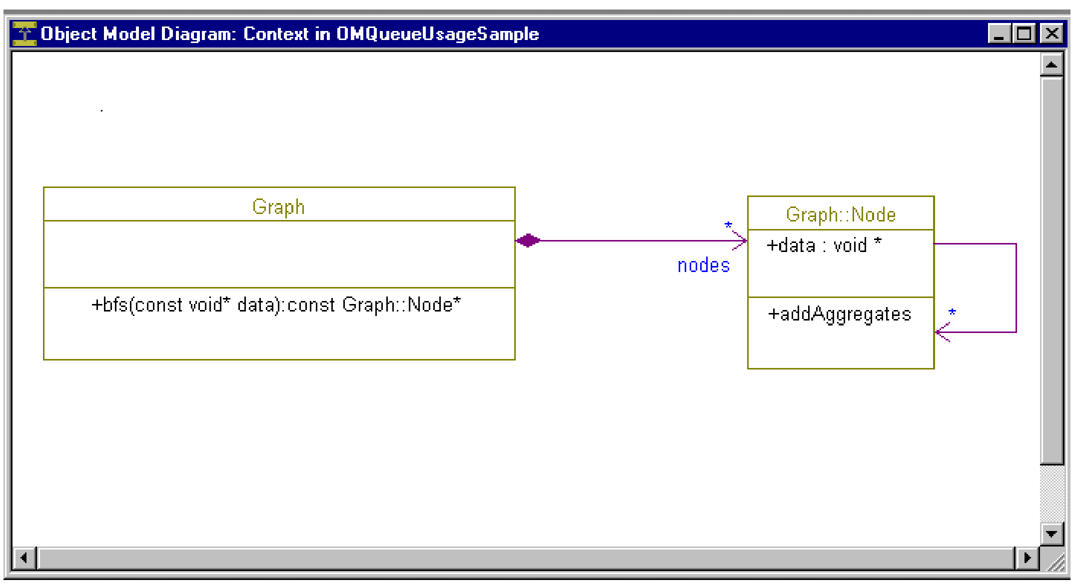
m_tail

This attribute specifies the tail of the queue. It is defined as follows:

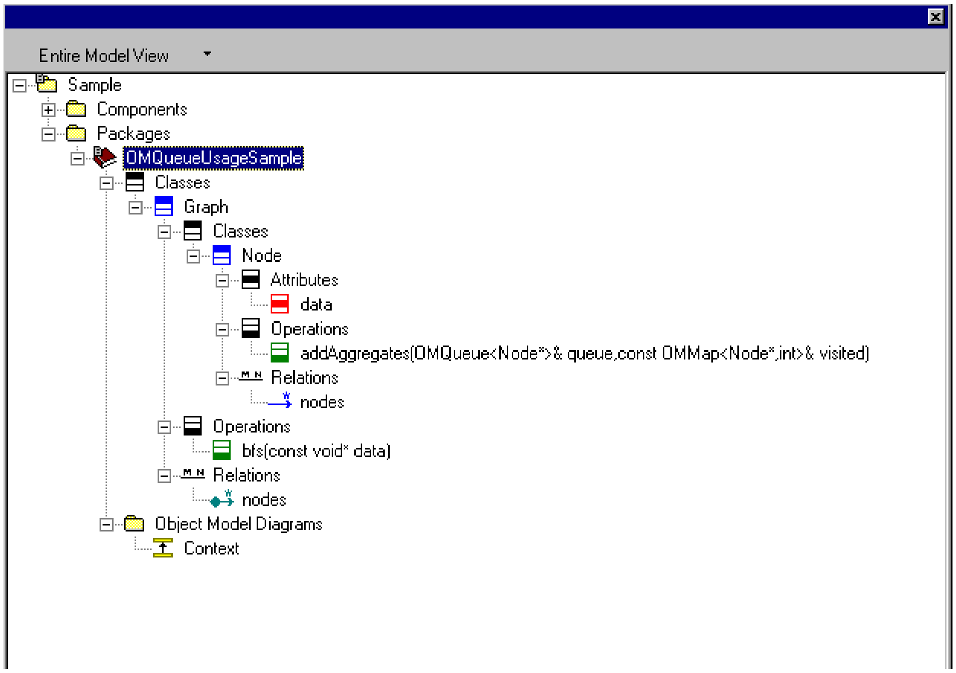
```
int m_tail;
```

Example

Consider a class, `Graph`, that has a `bfs()` operation that performs BFS search on the graph nodes to find a node with the specified data. The following figure shows the OMD of the `Graph` class.

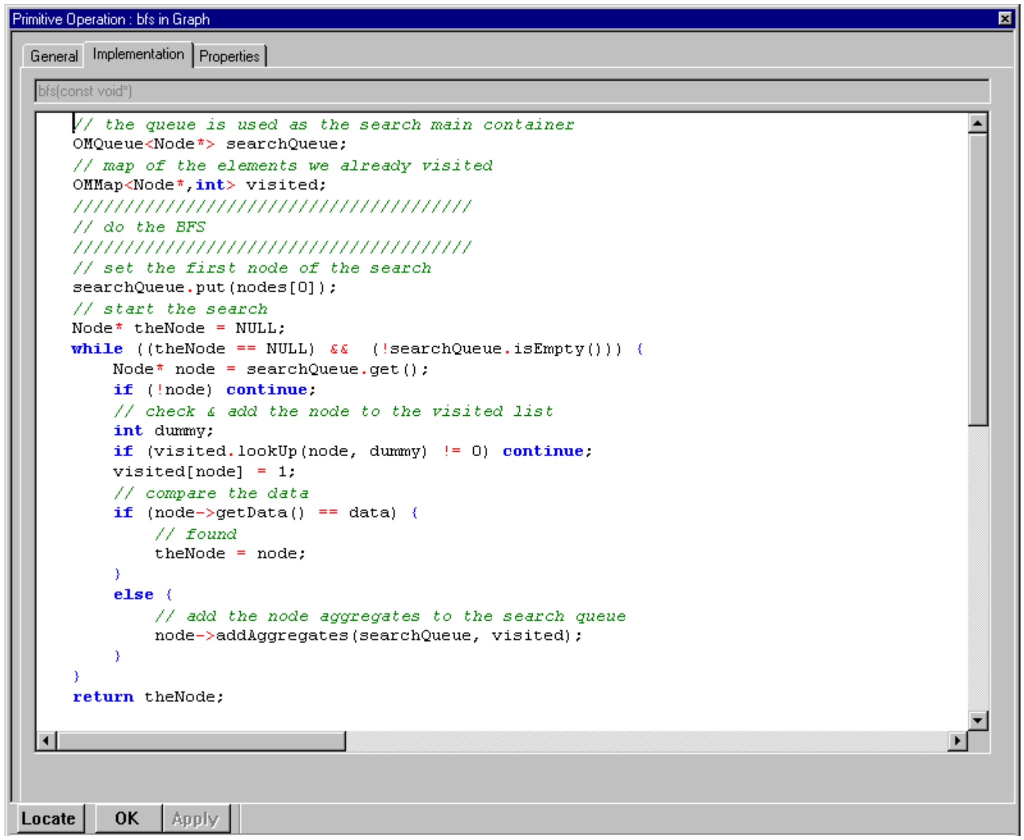


The following figure shows the browser view of the Graph class.

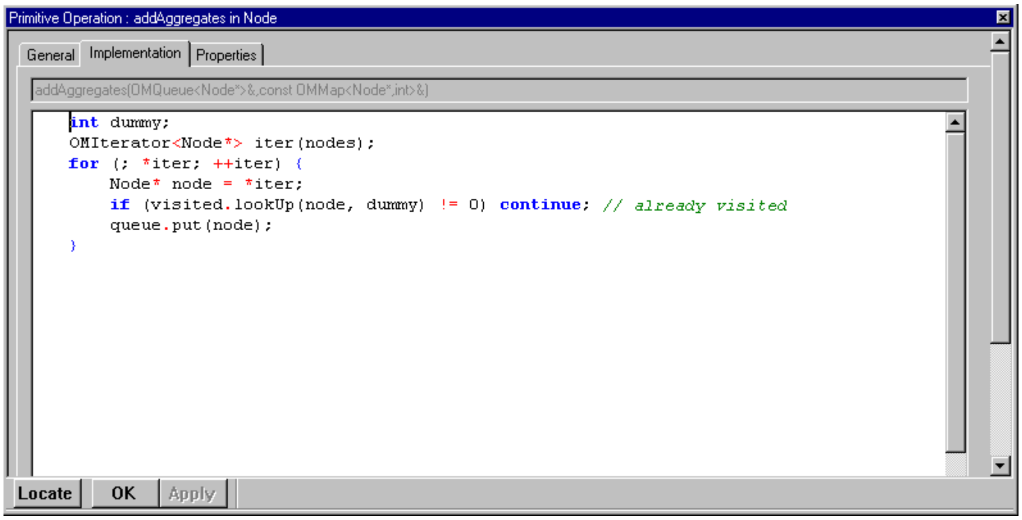


The bfs() implementation uses OMQueue as the search container and OMMap as a record of the visited elements.

The following figure shows the implementation of `Graph::bfs()`.



The following figure shows the implementation of `Graph::Node::addAggregates()`.



OMQueue

Visibility

`Public`

Description

The **OMQueue** method is the constructor for the `OMQueue` class.

Signature

```
OMQueue(OMBoolean shouldGrow = TRUE, int initSize = 100);
```

Parameters

`shouldGrow`

The value `TRUE` specifies that you should be able to enlarge the queue as necessary.

`initSize`

Specifies the initial size of the queue.

See Also

~OMQueue

~OMQueue

Visibility

Public

Description

The ~OMQueue method destroys the OMQueue object.

Signature

```
virtual ~OMQueue() {};
```

See Also

OMQueue

get

Visibility

Public

Description

The get method gets the current element in the queue.

Signature

```
virtual Concept get();
```

Return

The current element in the queue

getCount

Visibility

Public

Description

The getCount method gets the number of elements in the queue.

Signature

```
int getCount() const
```

Return

The number of elements in the queue

getInverseQueue

Visibility

`Public`

Description

The **getInverseQueue** method returns the element that will be returned by `get()` in the tail of the queue.

Signature

```
virtual void getInverseQueue(OMList<Concept>& list)
    const;
```

Parameters

`list`

The element that will be returned by `get()` in the tail of the queue

getQueue

Visibility

`Public`

Description

The **getQueue** method returns the element that will be returned by `get()` in the head of the queue.

Signature

```
virtual void getQueue(OMList<Concept>& list) const;
```

Parameters

`list`

The element returned by a `get()` in the head of the queue

getSize

Visibility

`Public`

Description

The **getSize** method returns the size of the memory allocated for the queue.

Signature

```
virtual int getSize() const
```

Return

The size of the allocated memory

increaseHead_

Visibility

`Public`

Description

The **increaseHead_** method increases the size of the queue head.

Signature

```
void increaseHead_();
```

increaseTail_

Visibility

`Public`

Description

The **increaseTail_** method increases the size of the queue tail.

Signature

```
void increaseTail_();
```

isEmpty

Visibility

`Public`

Description

The **isEmpty** method determines whether the queue is empty.

Signature

```
OMBoolean isEmpty() const
```

Return

The method returns one of the following Boolean values:

- `TRUE`—The queue is empty.
- `FALSE`—The queue is not empty.

isFull

Visibility

`Public`

Description

The **isFull** method determines whether the queue is full.

Signature

```
OMBoolean isFull() const;
```

Return

The method returns one of the following Boolean values:

- `TRUE`—The queue is full.
- `FALSE`—The queue is not full.

put

Visibility

`Public`

Description

The **put** method adds an element to the queue.

Signature

```
virtual OMBoolean put(Concept c);
```

Parameters

```
c
```

The element to add to the queue

Return

The method returns one of the following Boolean values:

- `TRUE`—The method was successful.
- `FALSE`—The method failed.

OMReactive Class

The `OMReactive` class is the framework base class for all reactive objects and implements basic event handling functionality. It is declared in the file `omreactive.h`.

Reactive objects process events, typically via statecharts or activity diagrams. The primary interfaces for reactive objects are the **gen** and **takeTrigger** methods.

Triggered operations are synchronous events that affect the reactive class state. The generated code creates an event, then passes it to the reactive class by calling the **takeTrigger** method. For additional information on triggered operations, see **Dispatching a Triggered Operation**.

Sender objects apply the **gen** method to send an event to a receiver, which inherits from `OMReactive`. The event is then queued inside a thread. See **Generating and Queuing an Event**.

The `execute` method waits on the thread's event queue. When an event is present on the queue, it dispatches it to the appropriate `OMReactive` object using the **takeTrigger** method. For more information, see **Generating and Queuing an Event**.

Attribute Summary

active	Specifies whether the reactive object (the concrete object derived from `OMReactive`) is also an active object
frameworkInstance	Specifies whether the reactive object is used by the framework itself (it is not a user-defined object)
myStartBehaviorEvent	Activates an object that has null transitions as part of the default transition
omrStatus	Defines the internal state (as opposed to the user-class state in the statechart) of the reactive object
toGuardReactive	Specifies that the consumption of an event should be guarded with a mutex (a binary semaphore)

Constant Summary

eventConsumed	Specifies that the event has been consumed.
eventNotConsumed	Specifies that the event was completed, but was not consumed.
OMRDefaultStatus	Specifies the default value for the `omrStatus` attribute
OMDefaultThread	Defines the default thread for an `OMReactive` object
OMRInDtor	Stops event dispatching
OMRNullConfig	Determines whether null transitions (transitions with no trigger) need to be taken in the generated code
OMRNullConfigMask	Determines whether an `OMReactive` instance should take null transitions in the state machine
OMRShouldCompleteStartBehavior	Determines whether the entry to the state machine on the call to startBehavior was completed, and, if not, whether there are additional null transitions to take
OMRShouldDelete	Determines whether a reactive object should be deleted by its active object when it reaches a termination connector in its state machine
OMRShouldTerminate	Allows the safe destruction of a reactive instance by its active instance

Macro Summary

GEN	Generates a new event
GEN_BY_GUI	Generates an event from a GUI
GEN_BY_X	Generates a new event from a sender object to a receiver object
GEN_ISR	Generates an event from an interrupt service request (ISR)

Relation Summary

event	Specifies the active or current event (the one that is now being processed) for the `OMReactive` instance

m_eventGuard	Used, in collaboration with the generated code, to protect the event consumption from mutual exclusion between events and triggered operations
myThread	Specifies the active class that queues events and dispatches events (so they are consumed on the active class's thread) for a reactive object
rootState	Defines the root state of the `OMReactive` statechart (when the system is using a reusable statechart implementation)

Construction Summary

OMReactive	Constructs an `OMReactive` object
~OMReactive	Destroys the `OMReactive` object

Method Summary

cancelEvents	Cancels all the queued events for the reactive object.
consumeEvent	Is the main event consumption method.
discarnateTimeout	Destroys a timeout object for the reactive object.
doBusy	Sets the value of **omrStatus** to 1 or `TRUE.`
gen	Is used by a sender object to send an event to a receiver object.
_gen	Queues events sent to the reactive object.
getCurrentEvent	Gets the currently processed event.
getThread	Retrieves the thread associated with a reactive object.
handleEventNotConsumed	Is called when an event is not consumed by the reactive class.
handleTONotConsumed	Is called when a triggered operation is not consumed by the reactive class.
incarnateTimeout	Creates a timeout object to be invoked on the reactive object.
inNullConfig	Determines whether an `OMReactive` instance should take null transitions (transitions without triggers) in the state machine.
isActive	Determines whether a reactive object is also an active object.

isBusy	Returns the current value of the omrStatus attribute.
isCurrentEvent	Determines whether the specified ID is the currently processed event.
isFrameworkInstance	Determines the current value of the frameworkInstance attribute.
isInDtor	Determines whether event dispatching should be stopped.
isValid	Makes sure the reactive class is not deleted.
popNullConfig	Decrements the omrStatus attribute after a null transition is taken.
pushNullConfig	Counts null transitions and increments the `omrStatus` attribute after a state is exited.
registerWithOMReactive	Registers a user instance as a reactive class in the animation framework
rootState_dispatchEvent	Consumes an event inside a real statechart.
rootState_entDef	Initializes the statechart by taking the default transitions.
rootState_serializeStates	Is a virtual method that performs the actual event consumption.
runToCompletion	Takes all the null transitions (if any) that can be taken after an event has been consumed.
serializeStates	Is called during animation to send state information.
setCompleteStartBehavior	Sets the value of the OMRShouldCompleteStartBehaviorattribute.
setEventGuard	Is used to set the event guard flag (m_eventGuard).
setFrameworkInstance	Changes the value of the frameworkInstance attribute.
setInDtor	Specifies that event dispatching should be stopped.
setMaxNullSteps	Sets the maximum number of null transitions (those without a trigger) that can be taken sequentially in the statechart.
setShouldDelete	Specifies whether a reactive object should be deleted by its active object when it reaches a termination connector in its state machine.
setShouldTerminate	Specifies that a reactive instance can be safely destroyed by its active instance.
setThread	Sets the thread of a reactive object.
setToGuardReactive	Specifies the value of the toGuardReactive attribute.
shouldCompleteRun	Checks the value of omrStatus to determine whether there are null transitions to take.

shouldCompleteStartBehavior	Checks the start behavior state.
shouldDelete	Determines whether a reactive object should be deleted by its active object when it reaches a termination connector in its state machine.
shouldTerminate	Determines whether a reactive instance can be safely destroyed by its active instance.
startBehavior	Initializes the behavioral mechanism and takes the initial (default) transitions in the statechart before any events are processed.
takeEvent	Is used by the event loop (within the thread) to make the reactive object process an event.
takeTrigger	Consumes a triggered operation event (synchronous event).
terminate	Sets the `OMReactive` instance to the terminate state (the statechart is entering a termination connector).
undoBusy	Sets the value of the `sm_busy` attribute to 0 or FALSE.

Attributes and Defines

active

This protected attribute specifies whether the reactive object (the concrete object derived from `OMReactive`) is also an active object. An active object creates its own thread and also inherits from an `OMThread` object.

The default value is `0` or `FALSE`.

If the reactive object is an active object, the user application will call the thread `start`; otherwise, it will not.

It is defined as follows:

```
OMBoolean active;
```

frameworkInstance

This protected attribute specifies whether the reactive object is used by the framework itself (it is not a user-defined object).

The default value is `0` or `FALSE`, and is specified by **OMReactive**, the constructor for a reactive object.

The `frameworkInstance` attribute can be used to model the Rhapsody framework in terms of itself. The default value is `FALSE`; you would not normally want to change the default.

It is defined as follows:

```
OMBoolean frameworkInstance;
```

myStartBehaviorEvent

This protected attribute activates an object that has null transitions as part of the default transition.

It is defined as follows:

```
OMStartBehaviorEvent myStartBehaviorEvent;
```

omrStatus

This protected attribute defines the internal state (as opposed to the user-class state in the statechart) of the reactive object.

The default value is **OMRDefaultStatus**, and is specified by **OMReactive**, the constructor for a reactive object.

It is defined as follows:

```
long omrStatus;
```

toGuardReactive

This protected attribute specifies that the consumption of an event should be guarded with a mutex (a binary semaphore).

The default value is `0` or `FALSE`, and is specified by **OMReactive**, the constructor for a reactive object. `toGuardReactive` is set to `TRUE` automatically by code generation, based on user modeling.

It is defined as follows:

```
OMBoolean toGuardReactive;
```

Constants

eventConsumed

Specifies that the event was consumed. It is defined as follows:

```
#define eventConsumed
    OMReactive::OMTakeEventCompleted
```

eventNotConsumed

Specifies that the event was completed, but was not consumed. It is defined as follows:

```
#define eventNotConsumed
    OMReactive::OMTakeEventCompletedEventNotConsumed
```

OMRDefaultStatus

Specifies the default value for the `omrStatus` attribute. This is used by `OMReactive`.

It is defined as follows:

```
const long OMRDefaultStatus = 0x00000000L;
```

OMDefaultThread

Defines the default thread for an `OMReactive` object. The default value is `0` or NULL, which tells the `OMReactive` object to process its events on the system default active class.

It is defined as follows:

```
#define OMDefaultThread 0
```

OMRInDtor

Used to set and get the `OMReactive` internal state stored in **omrStatus**. It is used in conjunction with **omrStatus** to stop event dispatching.

`OMRInDtor` does not provide protection from mutual exclusion (an attempt to dispatch an event to a class deleted on another thread). If you want to provide mutual exclusion protection, refer to the Rhapsody code generation documentation.

It is defined as follows:

```
const long OMRInDtor = 0x00020000L;
```

OMRNullConfig

Used to get and set the `OMReactive` internal state stored in `omrStatus`. It is used in conjunction with **omrStatus** to determine whether null transitions (transitions with no trigger) need to be taken in the generated code.

It is defined as follows:

```
const long OMRNullConfig = 0x00000001L;
```

OMRNullConfigMask

Used to get and set the `OMReactive` internal state stored in `omrStatus`. It is used in conjunction with **omrStatus** to determine whether an `OMReactive` instance should take null transitions in the state machine.

It is defined as follows:

```
const long OMRNullConfigMask = 0x0000FFFFL;
```

OMRShouldCompleteStartBehavior

Used to get and set the `OMReactive` internal state stored in `omrStatus`. It is used in conjunction with `omrStatus` to determine whether the entry to the state machine on the call to **startBehavior** was completed, and, if not, whether there are additional null transitions to take.

This bit is set by the **startBehavior** method if the **shouldCompleteRun** method returns an **omrStatus** of `TRUE`.

This bit is reset by the **consumeEvent** method on the first event.

It is defined as follows:

```
const long OMRShouldCompleteStartBehavior =
    0x00080000L;
```

OMRShouldDelete

Used to get and set the `OMReactive` state stored in `omrStatus`. It is used in conjunction with **omrStatus** to determine whether a reactive object should be deleted by its active object when it reaches a termination connector in its state machine. This permits statically allocated objects to have a termination connector in their state machine.

It is defined as follows:

```
const long OMRShouldDelete = 0x00040000L;
```

OMRShouldTerminate

Used to get and set the `OMReactive` internal state stored in `omrStatus`. It is used in conjunction with `omrStatus` to allow the safe destruction of a reactive instance by its active instance.

It is defined as follows:

```
const long OMRShouldTerminate = 0x00010000L;
```

Macros

GEN

Generates a new event. The `GEN` macro uses the `gen` method, then calls the `new` operator to create a new event.

The macro is defined as follows:

```
#define GEN (event) gen (new event)
```

GEN_BY_GUI

Generates an event from a GUI. The `GEN_BY_GUI` macro uses the **gen** method, then calls the `new` operator to create a new event. `OMGui` specifies the GUI thread.

The macro is defined as follows:

```
#define GEN_BY_GUI (event) gen ((OMEvent*)
    (new event), OMGui)
```

`OMGui` is defined in `aoxf.h`.

GEN_BY_X

Generates a new event from a sender object to a receiver object. It specifies a sender and is typically used to generate events from external elements, such as a GUI. The `GEN_BY_X` macro uses the **gen** method, then calls the `new` operator (with the sender as a parameter) to create a new event.

The macro is defined as follows

```
#define GEN_BY_X (event, sender) gen (new event,
    sender)
```

GEN_ISR

Generates an event from an interrupt service request (ISR). The `GEN_ISR` macro uses the **gen** method with the `genFromISR` parameter specified as `TRUE` to create a new event from an ISR.

It is the user's responsibility to allocate the event; `GEN_ISR` itself does not allocate the event.

The macros is defined as follows:

```
#define GEN_ISR (event) gen (event, TRUE)
```

For VxWorks, `GEN_ISR` generates an event with urgent priority that is placed at the head of the event queue. If another event from `GEN_ISR` occurs before the first one has been processed, it will be placed in front of the previous event. The implementation of `GEN_ISR` for VxWorks was aimed to address a use case where a reactive object has a flow of "plain" events, and from time to time it gets a single, high-priority event that is placed at the front of the queue for immediate consumption.

If a burst of `GEN_ISR` events are being injected into the system, you can comment out the setting of the priority in the framework to treat events from interrupts with equal priority. In `OMBoolean VxOSMessageQueue::put(void* m, OMBoolean fromISR)`, comment out the line `priority` = `MSG_PRI_URGENT`.

Relations

event

This public relation specifies the active or current event (the one that is now being processed) for the `OMReactive` instance. The relation is assigned only when an event is taken from the event queue.

The default value is NULL, and is specified by **OMReactive**, the constructor for a reactive object.

The relation is defined as follows:

```
OMEvent *event;
```

m_eventGuard

Used, in collaboration with the generated code, to protect the event consumption from mutual exclusion between events and triggered operations.

If a user reactive class has a guarded triggered operation, this relation will be set to the `OMProtected` part of the reactive class, and the `takeEvent` method will lock the guard before calling `consumeEvent`.

It is defined as follows:

```
const OMProtected * m_eventGuard;
```

myThread

This protected relation specifies the active class that queues events and dispatches events (so they are consumed on the active class's thread) for a reactive object.

There is a one-way relationship between a thread and a reactive class. The thread does not know its reactive class—it might have many. However, the reactive class has a relation to its thread, specified by `myThread`.

The relation is defined as follows:

```
OMThread *myThread;
```

rootState

This relation defines the root state of the `OMReactive` statechart (when the system is using a reusable statechart implementation).

The default value is NULL, and is specified by `OMReactive`, the constructor for a reactive object.

It is defined as follows:

```
OMComponentState* rootState;
```

The `OMComponentState` class is defined in `state.h`.

OMReactive

Visibility

`Public`

Description

The `OMReactive` method is the constructor for the `OMReactive` class.

Signature

```
OMReactive(OMThread *pthread = OMDefaultThread);
```

Parameters

`pthread`

Defines the thread on which events for the `OMReactive` instance are processed. The default value is **OMDefaultThread**, which is set to the system default active class.

Composite classes use this parameter to inherit threads to components.

See Also

OMDefaultThread

~OMReactive

~OMReactive

Visibility

`Public`

Description

The `~OMReactive` method is the destructor for the `OMReactive` class.

Signature

```
virtual ~OMReactive();
```

See Also

OMReactive

cancelEvents

Visibility

Public

Description

The `cancelEvents` method cancels all the queued events for the reactive object. This method is called upon destruction of the reactive object to prevent the thread from sending additional events to a destroyed object.

Signature

```
void cancelEvents();
```

Notes

- If there are several events in the event queue targeted for an `OMReactive` instance, but the instance has already been destroyed because it reached a termination connector in the statechart, the framework uses the `cancelEvents` method to cancel the events.
- `cancelEvents` calls the `OMThread::`cancelEvents method.

See Also

cancelEvents

consumeEvent

Visibility

Public

Description

The `consumeEvent` method is the main event consumption method. It handles the passing of events and triggered operations from the framework to the user-defined statechart, which then consumes them. This method is called by the `takeEvent` and `takeTrigger` methods.

You can override `consumeEvent` to specialize different event consumption behaviors:

- Create a reactive class that consumes events without a statechart.
- Add functionality to a class's event consumption.

Signature

```
virtual TakeEventStatus consumeEvent (OMEvent* ev);
```

Parameters

`ev`

Specifies the event to be consumed

Return

The method returns one of the values defined in the `TakeEventStatus` enumerated type. You can use these values to determine whether and how to continue with event processing on the reactive object.

The possible values are as follows:

- `OMTakeEventCompletedEventNotConsumed (0)`—The event was completed, but not consumed.
- `OMTakeEventCompleted (1)`—The event was completed. This is the normal status.
- `OMTakeEventInDtor (2)`—The event was not completed because the `OMReactive` instance is in destruction.
- `OMTakeEventReachTerminate (3)`—The event was not completed because the statechart has reached a termination connector and the reactive object should be destroyed.

Note

The **consumeEvent** method includes the ability to handle events and triggered operations that were not consumed. This is conceptually a callback method that you must override to define the actual handling of unconsumed events. To support this modification, the method signature was changed.

See Also

takeEvent

takeTrigger

discarnateTimeout

Visibility

`Public`

Description

The `discarnateTimeout` method is used by the framework to destroy a timeout object for the reactive object.

Signature

```
virtual void discarnateTimeout(OMTimeout * tm);
```

Parameters

`tm`

Specifies the timeout to be destroyed

See Also

undoBusy

doBusy

Visibility

`Public`

Description

The `doBusy` method sets the value of **omrStatus** to 1 or `TRUE`. It is called by the `rootState_dispatchEvent` method.

Signature

```
void doBusy()
```

Notes

The **undoBusy** method returns the current value of **omrStatus** and sets the value of `sm_busy` to `0` or `FALSE`.

See Also

isBusy

omrStatus

rootState_dispatchEvent

undoBusy

gen

Visibility

`Public`

Description

The `gen` method is an overloaded public method used by a sender object to send an event to a receiver object. `gen` first checks to see whether the receiver object is under destruction.

In uninstrumented code, the call `gen(OMEvent)` is always sufficient. The call is also sufficient in instrumented code when you include the `notifyContextSwitch` method.

Multithread instrumented applications should use the call `gen(OMEvent* event, void* sender)`. If the sender is a GUI element, use the syntax `gen(theEvent, OMGUI)`. `OMGui` is defined in the file `aoxf.h`.

Signatures

```
virtual OMBoolean gen (OMEvent *event,
   OMBoolean genFromISR = FALSE);
```

```
virtual OMBoolean gen (OMEvent *event, void * sender);
```

```
void gen (AOMEvent *theEvent, void * sender)
```

Parameters for Signature 1

`event`

Specifies a pointer to the event to be sent to the reactive object.

`genFromISR`

Indicates whether the event is from an operating system interrupt service request (ISR). If it is, it requires special treatment.

Parameters for Signature 2

`event`

Specifies the event to send

`sender`

Specifies the object sending the event

Parameters for Signature 3

`theEvent`

Specifies the event to send

`sender`

Specifies the object sending the event

Return

The method returns one of the following Boolean values:

- `TRUE`—The event was successfully queued.
- `FALSE`—The event was not queued.

Notes

- The `gen` method is typically used within actions and methods that you write.
- Note the following distinctions between the different method calls:

- The first method syntax does not specify a sender. `gen` first checks to see whether the receiver object is under destruction.

  This version of the method is expanded by the following macros:

  GUARD_OPERATION—Creates the event

  GEN_BY_GUI—Generates an event requested by a GUI

  GEN_ISR—Generates an event from an ISR

- The second version of the method is used to send events from external elements, such as a GUI. It registers the “top” of the call stack as its sender.

  This version of the method is expanded by the START_THREAD_GUARDED_SECTION macro, which also creates the event.

  - The `genFromISR` flag supports RTOSes (for example, VxWorks) that have restrictions on resource usage (for example, no memory allocation or waiting on semaphores) during an ISR.
  - To extend framework customization, the `gen` method was set to virtual in Version 3.0.

See Also

_gen

GEN_BY_GUI

GEN_ISR

GUARD_OPERATION

START_THREAD_GUARDED_SECTION

_gen

Visibility

Public

Description

The _gen method queues events sent to the reactive object.

_gen works in the following way:

- First, it sets the destination for the event by calling the **setDestination** method.
- Next, it calls the **queueEvent** method to queue the event in the OMThread event queue assigned to this OMReactive instance.

Signature

```
virtual OMBoolean _gen (OMEvent *event,
   OMBoolean genFromISR = FALSE);
```

Parameters

event

Specifies a pointer to the event to be sent to the reactive object.

genFromISR

Indicates whether the event is from an operating system interrupt service request (ISR). If it is, it requires special treatment.

Return

The method returns one of the following Boolean values:

- TRUE—The event was successfully queued.
- FALSE—The event was not queued.

Notes

- The event consumption is asynchronous. _gen causes the event to be inserted into an OMThread event queue—the reactive object does not have to respond to the event immediately.
- The genFromISR flag supports RTOSes (for example, VxWorks) that have restrictions on resource usage (for example no memory allocation or waiting on semaphores) during an ISR.
- To extend framework customization, the _gen method was set to virtual.

getCurrentEvent

Visibility

Public

Description

The getCurrentEvent method gets the currently processed event.

Signature

```
inline const OMEvent* getCurrentEvent() const
```

Return

The ID of the current event

See Also

isCurrentEvent

getThread

Visibility

`Public`

Description

The **getThread** method is an accessor function used to retrieve the thread associated with a reactive object. This method is called by the `action` method.

Signature

```
OMThread *getThread()
```

Return

The thread associated with the reactive object

See Also

action

setThread

handleEventNotConsumed

Visibility

`Public`

Description

The **handleEventNotConsumed** method is a virtual method called when an event is not consumed by the reactive class. To handle an unconsumed event, you must override this method.

This method is part of the framework for handling unconsumed events.

Signature

```
virtual void handleEventNotConsumed (OMEvent* event);
```

Parameters

`event`

Specifies the event

See Also

handleTONotConsumed

handleTONotConsumed

Visibility

Public

Description

The **handleTONotConsumed** method is a virtual method called when a triggered operation is not consumed by the reactive class. To handle an unconsumed triggered operation, you must override this method.

This method is part of the framework for handling unconsumed triggered operations.

Signature

```
virtual void handleTONotConsumed (OMEvent* event);
```

Parameters

`event`

Specifies the triggered operation

See Also

handleEventNotConsumed

incarnateTimeout

Visibility

Public

Description

The **incarnateTimeout** method is used by the framework to create a timeout object to be invoked on the reactive object. It is called by the **schedTm** method.

Signature

```
virtual OMTimeout *incarnateTimeout (short id,
   timeUnit delay, const OMHandle* theState);
```

Parameters

`id`

Identifies the timeout, either at delivery or for canceling. Every timeout has a specific `id` so it can be distinguished from other timeouts.

`delay`

Specifies the delay time, in milliseconds, before the timeout is triggered.

`theState`

Is used by the Rhapsody animation to designate the state name upon which the timeout is scheduled. There is no default value.

See Also

discarnateTimeout

schedTm

inNullConfig

Visibility

`Public`

Description

The **inNullConfig** method determines whether an `OMReactive` instance should take null transitions (transitions without triggers) in the state machine.

Signature

```
long inNullConfig() const
```

Return

The method returns **omrStatus** & **OMCancelledEventId**. If this value is `0`, there are no null transitions. If this value is greater than `0`, the value specifies the number of null transitions to take.

Notes

The **omrStatus** attribute specifies the maximum number of null transitions that are allowed. The default value is `100`.

See Also

popNullConfig

pushNullConfig

isActive

Visibility

`Public`

Description

The **isActive** method determines whether a reactive object is also an active object.

Signature

```
OMBoolean isActive()
```

Return

The method returns one of the following Boolean values:

- `TRUE`—The reactive object is also an active object.
- `FALSE`—The reactive object is not an active object.

isBusy

Visibility

`Public`

Description

The **isBusy** method returns the current value of the **omrStatus** attribute. It is called by the `rootState_dispatchEvent` method.

Signature

```
int isBusy() const
```

Return

The method returns one of the following integers:

- `1`—The object is currently consuming an event.
- `0`—The object is idle.

Notes

The **doBusy** method sets the value of `sm_busy` to 1 or `TRUE`; the **undoBusy** method sets the value of `sm_busy` to `0` or `FALSE`.

Rhapsody applies a safety mechanism to the flat statechart implementation that prevents self-directed trigger operations. If Rhapsody finds this condition, it simply ignores the invocation.

To omit the safety, you can override `OMReactive::`**consumeEvent**`()` in the user class code (this omits the check of `isBusy()` but does not modify the framework code. However, this can make the behavior unpredictable. The **handleEventNotConsumed** or **handleTONotConsumed** operations provide more predictable results.

See Also

doBusy

handleEventNotConsumed

handleTONotConsumed

omrStatus

rootState_dispatchEvent

undoBusy

isCurrentEvent

Visibility

`Public`

Description

The **isCurrentEvent** method determines whether the specified event ID matches the currently processed event.

Signature

```
OMBoolean IsCurrentEvent(short eventId) const;
```

Parameters

`eventId`

The event ID to check

Return

The method returns one of the following Boolean values:

- `TRUE`—The specified event is the current event.
- `FALSE`—The specified event is not the current event.

See Also

[getCurrentEvent]

isFrameworkInstance

Visibility

`Public`

Description

The **isFrameworkInstance** method determines the current value of the **frameworkInstance** attribute.

Signature

```
OMBoolean isFrameworkInstance() const
```

Return

The method returns one of the following Boolean values:

- `TRUE`—The reactive object is used by the framework itself.
- `FALSE`—The reactive object is not used by the framework; it is a user-defined object. This is the default value.

Notes

The `frameworkInstance` attribute can be used to model the Rhapsody framework in terms of itself. The default value is `FALSE`; you would not normally want to change the default.

See Also

[setFrameworkInstance]

isInDtor

Visibility

Public

Description

The **isInDtor** method determines whether event dispatching should be stopped. It is called by the `consumeEvent` and `rootState_dispatchEvent` methods.

Signature

```
unsigned char isInDtor() const
```

Return

If the return value is `0`, the object is not under destruction. If the value is greater than `0`, the object is under destruction.

See Also

consumeEvent

rootState_dispatchEvent

setInDtor

isValid

Visibility

Public

Description

The **isValid** method makes sure the reactive class is not deleted. This method is used by animation.

Signature

```
static OMBoolean isValid (const OMReactive*
    const p_reactive);
```

Parameters

`p_reactive`

Specifies the reactive class

Return

The method returns `TRUE` if the reactive class is valid; `FALSE` if the class has been deleted.

Note

The method **isValid** supersedes the method `isValidOMReactive`.

popNullConfig

Visibility

`Public`

Description

The **popNullConfig** method decrements the **omrStatus** attribute after a null transition is taken.

Signature

```
void popNullConfig();
```

Notes

The **omrStatus** attribute specifies the maximum number of null transitions that are allowed. The default value is `100`.

See Also

inNullConfig

omrStatus

pushNullConfig

pushNullConfig

Visibility

`Public`

Description

The **pushNullConfig** method counts null transitions. After a state is exited on a null transition, `pushNullConfig` increments the **omrStatus** attribute.

Signature

```
void pushNullConfig();
```

Notes

The **omrStatus** attribute specifies the maximum number of null transitions that are allowed. The default value is `100`.

See Also

inNullConfig

omrStatus

popNullConfig

registerWithOMReactive

Visibility

`Public`

Description

The **registerWithOMReactive** method registers a user instance as a reactive class in the animation framework. This method is used for animation support.

Signature

```
void registerWithOMReactive(void* myReal,
   AOMInstance *theAOMInstance)
```

Parameters

`myReal`

The user instance

`theAOMInstance`

The animation instance that reflects the user instance

rootState_dispatchEvent

Visibility

`Public`

Description

The **rootState_dispatchEvent** method is responsible for consuming an event inside a real statechart. It is called by the **consumeEvent** method.

Signature

```
virtual int rootState_dispatchEvent (short id);
```

Parameters

`id`

Specifies the ID of the event being consumed

Return

The method returns one of the following values:

- `0`—The method did not consume the event.
- `1`—The method consumed the event.

Notes

`OMReactive` has an implementation for the `rootState_dispatchEvent` and **undoBusy** methods. For flat statechart implementation, every class that inherits from `OMReactive` overwrites these methods according to its specific statechart implementation. For reusable statechart implementation, these methods are used as-is.

The Rhapsody framework "knows" nothing about the real statechart; it knows about the `rootState_entDef` and `rootState_dispatchEvent` methods only. Every concrete class knows how to react to every event because it has generated code for itself. Therefore, for flat statechart implementation, the concrete class overwrites these two virtual methods with its own customized implementation.

Flat statecharts are constructed using `switch` and `if` statements.They are more efficient in both time and space, and offer a customized implementation. Reusable statecharts are constructed using objects, and provide typical object-oriented features (for example, inheritance, encapsulation, and polymorphism). They offer a generic implementation. The Rhapsody default is flat statecharts.

In a reusable statechart implementation, `rootstate_dispatchEvent` invokes the root state **takeTrigger** operation.

See Also

consumeEvent

rootState_dispatchEvent

rootState_entDef

rootState_entDef

Visibility

`Public`

Description

The **rootState_entDef** method initializes the statechart by taking the default transitions.

Signature

```
virtual void rootState_entDef();
```

Notes

`OMReactive` has an implementation for the `rootState_entDef` and **undoBusy** methods. For flat statechart implementation, every class that inherits from `OMReactive` overwrites these methods according to its specific statechart implementation. For reusable statechart implementation, these methods are used as-is.

The Rhapsody framework “knows” nothing about the real statechart; it knows only about the `rootState_dispatchEvent` and `rootState_entDef` methods. Every concrete class knows how to react to every event because it has generated code for itself. Therefore, for flat statechart implementation, the concrete class overwrites these two virtual methods with its own customized implementation.

Flat statecharts are constructed using `switch` and `if` statements.They are more efficient in both time and space, and offer a customized implementation. Reusable statecharts are constructed using objects, and provide typical object-oriented features (for example, inheritance, encapsulation, and polymorphism). They offer a generic implementation. The Rhapsody default is flat statecharts.

See Also

rootState_dispatchEvent

rootState_entDef

rootState_serializeStates

Visibility

Public

Description

The **rootState_serializeStates** method is a virtual method that performs the actual event consumption.

In a flat statechart implementation, this method is not called, and the user class override is called instead.

In a reusable statechart implementation, this method calls the root state's `takeEvent` method to consume the event. The root state is a user class derived from `State`.

Signature

```
void rootState_serializeStates (AOMSState* aomsState)
   const;
```

Parameters

```
aomsState
```

Specifies the root state

runToCompletion

Visibility

`Public`

Description

The **runToCompletion** method takes all the null transitions (if any) that can be taken after an event has been consumed. In normal designs, this should not take more than several steps, so there is a safety limit that protects against infinite loops (considered to be design errors).

The `consumeEvent` method calls `runToCompletion`.

For more information, see `omreactive.cpp`.

Signature

```
void runToCompletion();
```

See Also

consumeEvent

shouldCompleteRun

serializeStates

Visibility

`Public`

Description

The **serializeStates** method is called during animation to send state information.

Signature

```
void serializeStates (AOMSState* s) const;
```

Parameters

`s`

Specifies the state

setCompleteStartBehavior

Visibility

Public

Description

The **setCompleteStartBehavior** method sets the value of the **OMRShouldCompleteStartBehavior** attribute.

Signature

```
void setCompleteStartBehavior (OMBoolean b)
```

Parameters

b

Specifies whether the entry to the state machine on the call to **startBehavior** was completed, and, if not, if there are additional null transitions to take

See Also

OMRShouldCompleteStartBehavior

omrStatus

setEventGuard

Visibility

Public

Description

The **setEventGuard** method is used to set the event guard flag (**m_eventGuard**).

Signatures

```
inline void setEventGuard (const OMProtected* eventGuard)
inline void setEventGuard (const OMProtected& eventGuard)
```

Parameters

eventGuard

Specifies the protected part of the reactive instance used to guard the event loop from mutual exclusion between events and triggered operation consumption

setFrameworkInstance

Visibility

`Public`

Description

The `setFrameworkInstance` method changes the value of the frameworkInstance attribute.

Signature

```
void setFrameworkInstance(OMBoolean is)
```

Parameters

`is`

Specifies the value for the `frameworkInstance` attribute. The possible values are as follows:

- `TRUE`—The framework uses the instance.
- `FALSE`—The framework does not use the instance.

Note

The `frameworkInstance` attribute can be used to model the Rhapsody framework in terms of itself. The default value is `FALSE`; you would not normally want to change the default.

See Also

frameworkInstance

isFrameworkInstance

setInDtor

Visibility

`Public`

Description

The **setInDtor** method is called by the `OMReactive` instance to specify that event dispatching should be stopped.

Signature

```
void setInDtor()
```

See Also

isInDtor

OMRInDtor

omrStatus

setMaxNullSteps

Visibility

`Public`

Description

The **setMaxNullSteps** method sets the maximum number of null transitions (those without a trigger) that can be taken sequentially in the statechart. If **omrStatus** is exceeded, event consumption is aborted.

The default value is defined in `omreactive.cpp` as follows:

```
#define OMDEFAULT_MAX_NULL_STEPS 100
```

Signature

```
static void setMaxNullSteps (int newMax)
```

Parameters

`newMax`

Specifies the new value for `maxNullSteps`

Notes

- The **pushNullConfig** method increments the **omrStatus** attribute after a state that has a null transition state is exited.
- The **popNullConfig** method decrements the **omrStatus** attribute after a null transition is taken.

See Also

omrStatus

popNullConfig

pushNullConfig

setShouldDelete

Visibility

`Public`

Description

The **setShouldDelete** method specifies whether a reactive object should be deleted by its active object when it reaches a termination connector in its state machine. This permits statically allocated objects to have a termination connector in their state machine.

This method is called by `OMReactive`, the constructor for a reactive object.

Signature

```
void setShouldDelete (OMBoolean b)
```

Parameters

`b`

If this is `TRUE`, the `OMReactive` instance is deleted. Otherwise, it is not deleted.

By default, this value is `TRUE`. To statically allocate a reactive object with a termination connector, you must explicitly call `setShouldDelete(FALSE)`.

See Also

OMRShouldDelete

omrStatus

shouldDelete

setShouldTerminate

Visibility

`Public`

Description

The **setShouldTerminate** method specifies that a reactive instance can be safely destroyed by its active instance.

Signature

```
void setShouldTerminate (OMBoolean b)
```

Parameters

`b`

Set this to `TRUE` to terminate the `OMReactive` instance. Otherwise, set this to `FALSE`.

See Also

OMRShouldTerminate

omrStatus

shouldTerminate

terminate

setThread

Visibility

Public

Description

The **setThread** method is a mutator function that sets the thread of a reactive object. It is an alternate way to set the thread instead of providing it in the reactive object's constructor.

This method is called by `OMReactive`, the constructor for a reactive object.

Note

Calling `setThread` out of the object `CTOR` is dangerous on systems where reactive objects can be deleted, because the events in the queue of the old thread will not be canceled upon the destruction of the reactive object.

Signature

```
virtual void setThread (OMThread *t,
   OMBoolean active = FALSE);
```

Parameters

`t`

Specifies the thread to be set

`active`

Signals the reactive instance that it is also active (the user object also inherits from `OMThread`)

See Also

getThread

OMReactive

setToGuardReactive

Visibility

`Public`

Description

The **setToGuardReactive** method specifies the value of the **toGuardReactive** attribute. If **toGuardReactive** is set to `TRUE`, event consumption is guarded.

Note

You need to guard event consumption in order to protect the reactive object from being deleted by another thread while it is consuming an event.

Signature

```
void setToGuardReactive(OMBoolean flag);
```

Parameters

`flag`

Specifies the value of the reactive event consumption flag. The possible values are as follows:

- `TRUE`—The reactive event consumption should be guarded.
- `FALSE`—The reactive event consumption should not be guarded.

See Also

toGuardReactive

shouldCompleteRun

Visibility

`Public`

Description

The **shouldCompleteRun** method checks the value of **omrStatus** to determine whether there are null transitions to take. It is called by the `consumeEvent` method.

Signature

```
long shouldCompleteRun() const
```

Return

A `long` that represents the value of **omrStatus**

Notes

The **runToCompletion** method is used to take all the null transitions (if any) that can be taken after an event has been consumed.

See Also

consumeEvent

omrStatus

runToCompletion

setEventGuard

shouldCompleteStartBehavior

Visibility

`Public`

Description

The **shouldCompleteStartBehavior** method checks the start behavior state.

When the user code calls the `startBehavior` method of a reactive class, the class takes the default transition of the statechart. If there are null transitions immediately after the default transition, the reactive class sends a special event (**OMStartBehaviorEvent**) to itself, and changes its state accordingly. The **shouldCompleteStartBehavior** method checks the value of this state.

Signature

```
long shouldCompleteStartBehavior() const
```

Return

A `long` that represents the state

shouldDelete

Visibility

`Public`

Description

The **shouldDelete** method determines whether a reactive object should be deleted by its active object when it reaches a termination connector in its state machine. This method is called by the `consumeEvent` and `takeTrigger` methods.

Signature

```
OMBoolean shouldDelete() const
```

Return

The method returns one of the following Boolean values:

- `TRUE`—The framework should delete the object after it reaches a termination connector.
- `FALSE`—The framework should not attempt to delete the object.

See Also

consumeEvent

setShouldDelete

takeTrigger

shouldTerminate

Visibility

`Public`

Description

The **shouldTerminate** method determines whether a reactive instance can be safely destroyed by its active instance. This method is called by the `consumeEvent` and `takeTrigger` methods.

Signature

```
long shouldTerminate() const
```

Return

The method returns `omrStatus & OMRShouldTerminate`. If this value is `0`, the object should not terminate. If the value is greater than `0`, the object should terminate.

See Also

consumeEvent

setShouldTerminate

takeTrigger

terminate

startBehavior

Visibility

`Public`

Description

The startBehavior method initializes the behavioral mechanism and takes the initial (default) transitions in the statechart before any events are processed. After this call is completed, the statechart is set to the initial configuration.

Note that `startBehavior` is called on the thread that creates the reactive object; default transitions are taken on the creator thread.

Note

Do not call `startBehavior` within the class `CTOR`.

Signature

```
virtual OMBoolean startBehavior();
```

Return

The method returns one of the following values:

- `TRUE`—The behavior initialization succeeded.
- `FALSE`—The behavior initialization failed.

Notes

- If you manually declare an instance (in user code), it is your responsibility to explicitly invoke `startBehavior;` otherwise, the object will not respond to events.
- The `startBehavior` method executes on the thread that invoked it (if the class is an active class, this is *not* the class's thread).
- The `startBehavior` method involves execution of actions, and in esoteric cases might result in the destruction of an instance.

takeEvent

Visibility

`Public`

Description

The takeEvent method is used by the event loop (within the thread) to make the reactive object process an event. After some preliminary processing, the `takeEvent` method calls consumeEvent to consume the event. This is a virtual function and can be overridden.

Signature

```
virtual TakeEventStatus takeEvent(OMEvent* ev);
```

Parameters

`ev`

Specifies the event to be processed

Return

The method returns one of the values defined in the `TakeEventStatus` enumerated type. You can use these values to determine whether and how to continue with event processing on the reactive object. The possible values are as follows:

- `OMTakeEventCompletedEventNotConsumed (0)`—The event was completed, but not consumed.
- `OMTakeEventCompleted (1)`—The event was completed. This is the normal status.
- `OMTakeEventInDtor (2)`—The event was not completed because the `OMReactive` instance is in destruction.
- `OMTakeEventReachTerminate (3)`—The event was not completed because the statechart has reached a termination connector and the reactive object should be destroyed.

Notes

- This method is used by the framework. Typically, you do not use it unless you want to rewrite the event consumption.
- `takeEvent` is called by the execute method to request that the reactive object process an event.

See Also

consumeEvent

execute

takeTrigger

Visibility

Public

Description

The takeTrigger method consumes a triggered operation event (synchronous event). This is a virtual function and can be overridden. The `takeTrigger` method works in the following way:

1. First, it calls the consumeEvent method to consume the event.
2. Next, it calls the shouldTerminate and setShouldDelete methods. If `(shouldTerminate() && shouldDelete())` is 1 (or `TRUE`), `takeTrigger` deletes the event.

Signature

```
virtual void takeTrigger (OMEvent* ev);
```

Parameters

ev

Specifies the triggered event

Notes

A triggered operation is a synchronous event—the event is sent to the `OMReactive` instance and consumed immediately. Most statechart events are asynchronous—the event is sent to the `OMReactive` instance, but is not necessarily consumed immediately.

See Also

consumeEvent

setShouldDelete

shouldDelete

shouldTerminate

terminate

Visibility

`Public`

Description

The **terminate** method sets the `OMReactive` instance to the terminate state (the statechart is entering a termination connector).

Signature

```
void terminate (const char* c = "");
```

Parameters

`c`

Set to an empty string (""). This parameter is used for animation purposes.

See Also

setShouldTerminate

shouldTerminate

undoBusy

Visibility

`Public`

Description

The **undoBusy** method sets the value of the `sm_busy` attribute to `0` or `FALSE`. It is called by the `rootState_dispatchEvent` method.

Signature

```
void undoBusy()
```

Notes

- The **undoBusy** method returns the current value of **omrStatus**.
- The **undoBusy** method sets the value of `sm_busy` to 1 or `TRUE`.

See Also

doBusy

isBusy

omrStatus

rootState_dispatchEvent

OMStack Class

The `OMStack` class contains basic library functions that enable you to create and manipulate `OMStacks`. An `OMStack` is a type-safe stack that implements a LIFO (last in, first out) algorithm.

This class is defined in the header file `omstack.h`.

Construction Summary

OMStack	Constructs an `OMStack` object
~OMStack	Destroys the `OMStack` object

Method Summary

getCount	Gets the number of items on the stack
isEmpty	Determines whether the stack is empty
pop	Pops an item off the stack
push	Pushes an item onto the stack
top	Moves the iterator to the first item in the stack

OMStack

Visibility

`Public`

Description

The **OMStack** method is the constructor for the `OMStack` class.

Signature

```
OMStack()
```

See Also

~OMStack

~OMStack

Visibility

`Public`

Description

The **~OMStack** method destroys the `OMStack` object.

Signature

```
~OMStack()
```

See Also

OMStack

getCount

Visibility

`Public`

Description

The **getCount** method gets the number of items in the stack.

Signature

```
int getCount() const
```

Return

The number of items in the stack

isEmpty

Visibility

`Public`

Description

The **isEmpty** method determines whether the stack is empty.

Signature

```
int isEmpty() const
```

Return

The method returns one of the following values:

- `0`—The stack is not empty.
- `1`—The stack is empty.

pop

Visibility

`Public`

Description

The **pop** method pops the next item off the stack.

Signature

```
Concept pop()
```

Return

The item popped off the stack

push

Visibility

`Public`

Description

The **push** method pushes an item onto the stack.

Signature

```
void push(Concept p)
```

Parameters

`p`

The item to add to the stack

top

Visibility

`Public`

Description

The **top** method moves the iterator to the first item in the stack.

Signature

```
Concept& top()
```

Return

The first item on the stack

OMStartBehaviorEvent Class

The `OMStartBehaviorEvent` class is used to handle the special case when a reactive class injects events to itself, and the **startBehavior** method has null transitions that should be taken after the default transition.

Using this class, you can execute the null transitions in the context of the reactive thread, instead of in the context of the thread that called **startBehavior**.

Animating Start Behavior

The friend class, `OMFriendStartBehaviorEvent`, animates the start behavior event class in instrumented mode. The friend class declaration is empty except for non-instrumented mode.

These classes are defined in the header file `event.h`.

Construction Summary

OMStartBehaviorEvent	Is the constructor for the `OMStartBehaviorEvent` class

OMStartBehaviorEvent

Visibility

`Public`

Description

The **OMStartBehaviorEvent** method is the constructor for the `OMStartBehaviorEvent` class.

Signature

```
OMStartBehaviorEvent();
```

OMState Class

The `OMState` class defines methods that affect statecharts.

This class is defined in the header file `state.h`.

Attribute Summary

`parent`	Specifies the parent

Construction Summary

OMState	Constructs an `OMState` object

Macro Summary

IS_EVENT_TYPE_OF(id)	Supports generic derived event handling
OM_DECLARE_FRAMEWORK_MEMORY_ALLOCATION_OPERATORS	Supports enhanced user control over framework memory allocation

Method Summary

entDef	Specifies the operation called when the state is entered from a default transition
entHist	Enters a history connector
enterState	Specifies the state entry action
exitState	Specifies the state exit action
getConcept	Gets the statechart owner
getHandle	Gets the handle
getLastState	Gets the last state
isCompleted	Gets the substate
in	Returns TRUE when the owner class is in this state
isCompleted	Determines whether the `OR` state reached a final state, and therefore can be exited on a null transition
serializeStates	Is called during animation to send state information
setHandle	Sets the handle

setLastState	Sets the last state
setSubState	Sets the substate
takeEvent	Takes the specified event off the event queue

Attributes

parent

This attribute specifies the parent state of this state (the state this state is contained in). It is defined as follows:

```
OMState* parent;
```

Macros

IS_EVENT_TYPE_OF(id)

This macro helps support generic derived event handling.

Rhapsody provides a generic way to handle the consumption of derived events. The support in generic handling of derived events was done by adding a new method, `isTypeOf()`, for every event, and modifying the generated code to check the event using this method. The `isTypeOf()` method returns `True` for derived events, as well as for the actual event.

OM_DECLARE_FRAMEWORK_MEMORY_ALLOCATION_OPERATORS

This macro helps support user control over framework memory allocation.

Rhapsody supports application control over memory allocated in the framework in two ways:

- Complete the memory management coverage, so every memory allocation in the generic framework as well as all the RTOS adaptors is using the memory management mechanism.
- Complete the usage of the `returnMemory()` interface, so the memory size returned is passed.

OMState

Visibility

Public

Description

The **OMState** method is the constructor for the `OMState` class.

Signature

```
OMState(OMState* par = NULL);
```

Parameters

`par`

Specifies the parent

entDef

Visibility

Public

Description

The **entDef** method specifies the operation called when the state is entered from a default transition.

Signature

```
virtual void entDef()=0;
```

entHist

Visibility

`Public`

Description

The **entHist** method enters a history connector.

Signature

```
virtual void entHist();
```

enterState

Visibility

`Public`

Description

The **enterState** method specifies the state entry action.

Signature

```
virtual void enterState();
```

exitState

Visibility

`Public`

Description

The **exitState** method specifies the state exit action.

Signature

```
virtual void exitState()=0;
```

getConcept

Visibility

Public

Description

The **getConcept** method gets the current concept. This method should be overridden by the concrete classes.

Signature

```
virtual AOMInstance * getConcept() const // animation

virtual void * getConcept() const        //no animation
```

Return

The concept

getHandle

Visibility

Public

Description

The **getHandle** method gets the handle. This method is used for animation purposes.

Signature

```
const char * getHandle() const
```

Return

The handle

getLastState

Visibility

`Public`

Description

The **getLastState** method returns the last state.

Signature

```
virtual OMState* getLastState();
```

Return

The last state

getSubState

Visibility

`Public`

Description

The **isCompleted** method returns the substate.

Signature

```
virtual OMState* getSubState();
```

Return

The substate

in

Visibility

`Public`

Description

The **in** method returns `TRUE` when the owner class is in this state.

Signature

```
virtual int in()=0;
```

isCompleted

Visibility

`Public`

Description

The **isCompleted** method determines whether the `OR` state reached a final state, and therefore can be exited on a null transition.

Signature

```
virtual OMBoolean isCompleted()
```

Return

The method returns one of the following Boolean values:

- `TRUE`—The operation is complete.
- `FALSE`—The operation is not complete.

serializeStates

Visibility

`Public`

Description

The **serializeStates** method is called during animation to send state information.

Signature

```
virtual void serializeStates (AOMSState* s) const = 0;

virtual void serializeStates(void*) //no animation
```

Parameters

`s`

Specifies the state

setHandle

Visibility

`Public`

Description

The **setHandle** method sets the handle. This method is used for animation purposes.

Signature

```
void setHandle(const char * hdl)
```

Parameters

`hdl`

Specifies the handle

setLastState

Visibility

`Public`

Description

The **setLastState** method sets the last state.

Signature

```
virtual void setLastState(OMState* s);
```

Parameters

`s`

Specifies the last state

setSubState

Visibility

`Public`

Description

The **setSubState** method sets the specified substate.

Signature

```
virtual void setSubState(OMState* s);
```

Parameters

`s`

Specifies the substate

takeEvent

Visibility

`Public`

Description

The **takeEvent** method takes the specified event off the event queue.

Signature

```
virtual int takeEvent(short lId);
```

Parameters

`lId`

Specifies the event ID

OMStaticArray Class

The `OMStaticArray` class contains basic library functions that enable you to create and manipulate `OMStaticArray` objects. An `OMStaticArray` is a type-safe, fixed-size array.

This class is defined in the header file `omstatic.h`.

Attribute Summary

count	Specifies the number of elements in the static array
theLink	Specifies the link to an element in the static array
size	Specifies the amount of memory allocated for the static array

Construction Summary

OMStaticArray	Constructs an `OMStaticArray` object
~OMStaticArray	Destroys the `OMStaticArray` object

Method Summary

operator []	Returns the element at the specified position
add	Adds the specified element to the array
find	Looks for the specified element in the array
getAt	Returns the element found at the specified index
getCount	Determines how many elements are in the array
getSize	Returns the amount of memory allocated for the array
isEmpty	Determines whether the array is empty
removeAll	Deletes all the elements from the array
setAt	Inserts the specified element at the given index in the array

Attributes

count

This attribute specifies the number of elements in the static array. It is defined as follows:

```
int count;
```

theLink

This attribute specifies the link to an element in the static array. It is defined as follows:

```
void** theLink;
```

size

This attribute specifies the amount of memory allocated for the static array. It is defined as follows:

```
int size;
```

Example

To use a static array, the multiplicity must be bounded (for example, `MAX_OBSERVERS`).

```
Consider the following example:

Observer* itsObserver[MAX_OBSERVERS];
for (int iter=0; iter<MAX_OBSERVERS; iter++)
{
    if (itsObserver[iter] != NULL)
    itsObserver[iter]->notify();
}
```

OMStaticArray

Visibility

`Public`

Description

The **OMStaticArray** method is the constructor for the `OMStaticArray` class.

Signature

```
OMStaticArray(int theSize)
```

Parameters

`theSize`

Specifies the amount of memory to allocate for the static array

See Also

~OMStaticArray

~OMStaticArray

Visibility

`Public`

Description

The **~OMStaticArray** method destroys the `OMStaticArray` object.

Signature

```
~OMStaticArray()
```

See Also

OMStaticArray

operator []

Visibility

`Public`

Description

The `[]` operator returns the element at the specified position.

Note

This is not the preferred method because it does not include a check of the index range.

Signature

```
Concept& operator [](int i)
```

Parameters

`i`

The index of the element to return

Return

The element at the specified position

add

Visibility

`Public`

Description

The **add** method adds the specified element to the array.

Signature

```
void add(Concept c)
```

Parameters

`c`

The element to add

See Also

removeAll

find

Visibility

`Public`

Description

The **find** method looks for the specified element in the array.

Signature

```
int find(Concept c) const;
```

Parameters

`c`

The element you want to find

Return

An integer that represents the index of the element in the array

getAt

Visibility

`Public`

Description

The **getAt** method returns the element found at the specified index.

Signature

```
Concept& getAt (int i) const
```

Parameters

`i`

The index of the element to retrieve

Return

The element found at the specified index

getCount

Visibility

`Public`

Description

The **getCount** method returns the number of elements in the static array.

Signature

```
int getCount() const
```

Return

The number of elements in the array

getSize

Visibility

`Public`

Description

The **getSize** method gets the size of the memory allocated for the static array.

Signature

```
int getSize() const
```

Return

The size

isEmpty

Visibility

`Public`

Description

The **isEmpty** method determines whether the static array is empty.

Signature

```
int isEmpty() const
```

Return

The method returns one of the following values:

- `0`—The static array is not empty.
- `1`—The static array is empty.

removeAll

Visibility

`Public`

Description

The **removeAll** method deletes all the elements from the array.

Signature

```
void removeAll()
```

See Also

add

setAt

Visibility

`Public`

Description

The **setAt** method inserts the specified element at the given index in the array.

Signature

```
void setAt(int index, const Concept& c)
```

Parameters

`index`

The index at which to add the new element

`c`

The element to add

OMString Class

The `OMString` class contains basic library functions that enable you to create and manipulate `OMStrings`. An `OMString` is a basic string class.

This class is defined in the header file `omstring.h`.

Construction Summary

OMString	Constructs an `OMCollection` object
~OMString	Destroys the `OMCollection` object

Method and Operator Summary

operator []	Returns the character at the specified position
operator +	Adds a string
operator +=	Adds to the existing string
operator =	Sets a string
operator ==	Determines whether two objects are equal
operator >=	Determines whether the first object is greater than or equal to the second
operator <=	Determines whether the first object is less than or equal to the second
operator !=	Determines whether the first object is not equal to the second object
operator >	Determines whether the first object is greater than the second
operator <	Determines whether the first object is less than the second
operator <<	Compares an output stream and a string
operator >>	Compares an input stream and a string
operator *	Is a customizable operator
CompareNoCase	Performs a case-insensitive comparison of two strings.
Empty	Empties the string
GetBuffer	Returns the string buffer
GetLength	Returns the length of the string
IsEmpty	Determines whether the string is empty
OMDestructiveString2X	Is used to support animation
resetSize	Makes the string larger

SetAt	Sets a character at the specified position in the string
SetDefaultBlock	Sets the default string size

OMString

Visibility

`Public`

Description

The OMString method is the constructor for the `OMString` class.

Signatures

```
OMString();
```

```
OMString(const char c);
```

```
OMString(const char* c);
```

```
OMString(const OMString& s);
```

Parameters for Signatures 2 and 3

`c`

The character to add to the newly created string

Parameters for Signature 4

`s`

The string of characters to add to the newly created string

See Also

~OMString

~OMString

Visibility

`Public`

Description

The ~OMString method destroys the `OMString` object.

Signature

```
~OMString()
```

See Also

OMString

operator []

Visibility

`Public`

Description

The `[]` operator returns the character at the specified position.

Signature

```
char operator [](int i) const
```

Parameters

`i`

The index of the character to return

Return

The character at the specified position

operator +

Visibility

```
Public
```

Description

The + operator adds a string.

Signatures

```
OMString operator+(const OMString& s);
```

```
OMString operator+(const char s);
```

```
OMString operator+(const char * s)
```

```
inline OMString operator+(const OMString& s1,
    const OMString& s2)
```

```
inline OMString operator+(const OMString& s1,
    const char * s2)
```

```
inline OMString operator+(const char* s1,
    const OMString& s2)
```

Parameters for Signatures 1, 2, and 3

`s`

The string to add

Parameters for Signature 4, 5, and 6

`s1`

The string to which to add string 2

`s2`

The string to add to string 1

Return

The new string

operator +=

Visibility

`Public`

Description

The += operator adds to the existing string.

Signatures

```
const OMString& operator+=(const OMString& s);
```

```
const OMString& operator+=(const char s);
```

```
const OMString& operator+=(const char * s);
```

Parameters

`s`

The characters to add to the string

Return

The updated string

operator =

Visibility

`Public`

Description

The = operator sets the string.

Signatures

```
const OMString& operator=(const OMString& s);

const OMString& operator=(const char s);

const OMString& operator=(const char * s);
```

Parameters

`s`

The string to set

Return

The string

operator ==

Visibility

```
Public
```

Description

The == operator is a comparison function used by `OMString` to determine whether two objects are equal.

Signatures

```
int operator==(const OMString& s2) const
```

```
int operator==(const char * c2) const
```

```
inline int operator==(const char * c1,
    const OMString& s2)
```

Parameters for Signature 1

`s2`

The string to compare to the current string

Parameters for Signature 2

`c2`

The character to compare to the current character

Parameters for Signature 3

`c1`

The character to compare to the specified string

`s2`

The string to compare to the specified character

Return

The method returns one of the following values:

- `1`—The objects are equal.
- `0`—The objects are not equal.

operator >=

Visibility

Public

Description

The >= operator determines whether the first object is greater than or equal to the second.

Signatures

```
int operator>=(const OMString& s2) const
```

```
int operator>=(const char * c2) const
```

```
inline int operator>=(const char * c1,
    const OMString& s2)
```

Parameters for Signature 1

`s2`

The string to compare to the current string

Parameters for Signature 2

`c2`

The character to compare to the current character

Parameters for Signature 3

`c1`

The character to compare to the specified string

`s2`

The string to compare to the specified character

Return

The method returns one of the following values:

- `1`—The first object is greater than or equal to the second object.
- `0`—The first object is less than the second object.

operator <=

Visibility

```
Public
```

Description

The <= operator determines whether the first object is less than or equal to the second.

Signatures

```
int operator<=(const OMString& s2) const
```

```
int operator<=(const char * c2) const
```

```
inline int operator<=(const char * c,
    const OMString& s)
```

Parameters for Signature 1

`s2`

The string to compare to the current string

Parameters for Signature 2

`c2`

The character to compare to the current character

Parameters for Signature 3

`c`

The character to compare to the specified string

`s`

The string to compare to the specified character

Return

The method returns one of the following values:

- `1`—The first object is less than or equal to the second.
- `0`—The first object is greater than the second.

operator !=

Visibility

`Public`

Description

The `!=` operator determines whether the first object is not equal to the second.

Signatures

```
int operator!=(const OMString& s2) const

int operator!=(const char * c2) const

inline int operator!=(const char * c, const OMString& s)
```

Parameters for Signature 1

`s2`

The string to compare to the current string

Parameters for Signature 2

`c2`

The character to compare to the current character

Parameters for Signature 3

`c`

The character to compare to the specified string

`s`

The string to compare to the specified character

Return

The method returns one of the following values:

- `1`—The two objects are not equal.
- `0`—The two objects are equal.

operator >

Visibility

Public

Description

The > operator determines whether the first object is greater than the second.

Signatures

```
int operator>(const OMString& s2) const

int operator>(const char * c2) const

inline int operator>(const char * c, const OMString& s)
```

Parameters for Signature 1

`s2`

The string to compare to the current string

Parameters for Signature 2

`c2`

The character to compare to the current character

Parameters for Signature 3

`c`

The character to compare to the specified string

`s`

The string to compare to the specified character

Return

The method returns one of the following values:

- `1`—The first object is greater than the second.
- `0`—The first object is not greater than the second.

operator <

Visibility

`Public`

Description

The < operator determines whether the first object is less than the second.

Signatures

```
int operator<(const OMString& s) const

int operator<(const char * c2) const

inline int operator<(const char * c, const OMString& s)
```

Parameters for Signature 1

`s`

The string to compare to the current string

Parameters for Signature 2

`c2`

The character to compare to the current character

Parameters for Signature 3

`c`

The character to compare to the specified string

`s`

The string to compare to the specified character

Return

The method returns one of the following values:

- `1`—The first object is less than the specified second.
- `0`—The first object is not less than the second.

operator <<

Visibility

`Public`

Description

The << operator is used to compare an iostream and a string.

Signature

```
inline omostream& operator<<(omosteam& os,
    const OMString& s)
```

Parameters

`os`

The output stream to compare to the string

`s`

The string to compare to the output stream

operator >>

Visibility

`Public`

Description

The >> operator is used to compare an iostream and a string.

Signature

```
omistream& operator>>(omisteam& is, OMString& s)
```

Parameters

`os`

The input stream to compare to the string

`s`

The string to compare to the input stream

operator *

Visibility

Public

Description

The `*` operator is a customizable operator.

Signature

```
operator const char *()
```

CompareNoCase

Visibility

Public

Description

The **CompareNoCase** method performs a case-insensitive comparison of two strings.

Signatures

```
int CompareNoCase(const OMString& s) const

int CompareNoCase(char * s) const
```

Parameters

`s`

The string to compare to the current string

Return

The method returns one of the following values:

- `0`—The two strings are not the same.
- `1`—The two strings are the same (regardless of case).

Empty

Visibility

`Public`

Description

The **Empty** method empties the string.

Signature

```
void Empty()
```

GetBuffer

Visibility

`Public`

Description

The **GetBuffer** method gets the string buffer.

Signature

```
char * GetBuffer(int buffer) const
```

Parameters

`buffer`

A pointer to the resized string buffer

Return

The buffer contents

GetLength

Visibility

Public

Description

The GetLength method returns the length of the string.

Signature

```
int GetLength() const;
```

Returns

The string length

IsEmpty

Visibility

Public

Description

The IsEmpty method determines whether the string is empty.

Signature

```
int IsEmpty() const
```

Return

The method returns one of the following values:

- `0`—The string is not empty.
- `1`—The string is empty.

OMDestructiveString2X

Visibility

Public

Description

The **OMDestructiveString2X** method is provided to support animation. It converts a `char*` string to `OMString` as part of the Rhapsody deserialization mechanism.

Signature

```
inline OMString OMDestructiveString2X(char * c,
    OMString& s)
```

Parameters

`c`

The input string

`s`

A dummy parameter (used for overloading)

Return

An `OMString`

resetSize

Visibility

Public

Description

The **resetSize** method enlarges the string and copies the contents into the larger string.

Signature

```
void resetSize(int newSize);
```

Parameters

`newSize`

The new size for the string

SetAt

Visibility

`Public`

Description

The **SetAt** method sets a character at the specified position in the string.

Signature

```
void SetAt(int i, char c)
```

Parameters

`i`

The position at which to add the character

`c`

The character to add

SetDefaultBlock

Visibility

`Public`

Description

The **SetDefaultBlock** method sets the default string size.

Signature

```
static void setDefaultBlock(int blkSize)
```

Parameters

`blkSize`

The new, default string size

OMThread Class

`OMThread` is a framework base active class. Its responsibilities are as follows:

- Manage an event queue of events sent to reactive classes.
- Dispatch the events in the queue to their reactive destinations on a separate RTOS thread.
- Allow the client application to control the RTOS thread.

This class is defined in the header file `omthread.h`.

`OMThread` is a base class for every class that is active. An object of an active class:

- Has its own operating system thread for execution
- Has an event queue and manages it

Therefore, every active object has an `OMThread` instance, which is composed of two things:

- An operating system thread
- An event (message) queue

By default, there are at least two threads in an application: the timer thread and the main thread. In this simple case, all events are queued in the main thread event queue.

Every operating system has a different implementation of a native thread.

The thread is responsible for providing event services to all instances running on it. Every event that is assigned to an object is sent to its relevant thread. The thread stores the events in an event queue. `OMThread` uses a `while` loop to consume events as they appear at the front of the queue.

An active object can also serve a nonactive object. For example, your application might have a class `a` that has a statechart but is also active, so it inherits from `OMThread` and `OMReactive`. Your application might also have a class `p` that has a statechart, but is not active. Class `p` inherits from `OMReactive`.

Suppose that `p` is running under `a`'s thread. Every event that is targeted for `p` must be stored somewhere, and `p` does not have an event queue. Therefore, `p` delegates events destined for it to `a`'s event queue, because `p` is running on `a`'s operating system thread and `a` has an event queue.

If you have the following line of code, generating an event `e` to class `p`, `e` is stored inside `a`'s `OMThread` event queue:

```
p -> GEN(e)
```

In `OMThread`, the **execute** method cycles through the event queue looking for more events. When it finds one or more events, it pops the first event (for example, `e`) from the event queue. The event has a field specifying the destination (`p`, in this example). `p` is then notified that it should react to event `e`. The event is not necessarily consumed immediately—it waits in the event queue. When the time arrives for the event to be consumed, it is popped from the event queue and injected into `p`'s `OMReactive` using the **takeEvent** method.

In Version 4.0, the inheritance from `OMProtected` was replaced with aggregation. As a result, the following were added to the `OMThread` interface:

- `void lock() const`—Puts a lock on the thread mutex
- `void unlock() const`—Unlocks the thread mutex
- `const OMProtected& getGuard() const`—Gets the reference to the `OMProtected` part
- `OMProtected m_omGuard`—Is a private `OMProtected` part

Attribute Summary

aomthread	Specifies the “instrumented” part of the thread
endOfProcess	Specifies whether the application is at the end of a process
eventQueue	Specifies the thread’s event queue
thread	Specifies the “os” part of the thread
toGuardThread	Determines whether a section of thread code will be protected

Construction Summary

OMThread	Constructs an `OMThread` object
~OMThread	Destroys the `OMThread` object

Method Summary

allowDeleteInThreadsCleanup	Postpones the destruction of a framework thread until the application terminates and all user threads are deleted
cancelEvent	Marks a single event as canceled (that is, it changes the event’s ID to **OMCancelledEventId**)

cancelEvents	Marks all events targeted for the specified `OMReactive` instance as canceled (that is, it changes the events' IDs to **OMCancelledEventId**)
cleanupAllThreads	"Kills" all threads in an application except for the main thread and the thread running the `cleanupAllThreads` method
cleanupThread	Provides a "hook" to allow a thread to be cleaned up without a call to the `DTOR`
destroyThread	Destroys the default active class or object for the framework
doExecute	Is the entry point to the thread main loop function
execute	Is the thread main loop function
getAOMThread	Is used by the framework for animation purposes
getEventQueue	Is used by the framework for animation purposes
getGuard	Gets the reference to the `OMProtected` part
getOsHandle	Returns the thread's operating system ID
getOSThreadEndClb	Requests a callback to end the current operating system thread
getStepper	Is used by the framework for animation purposes
lock	Puts a lock on the thread mutex
omGetEventQueue	Returns the event queue
queueEvent	Queues events to be processed by the thread event loop (**execute**)
resume	Resumes a thread suspended by the **suspend** method
schedTm	Creates a timeout request and delegates the request to `OMTimerManager`
setEndOSThreadInDtor	Specifies whether an operating system thread in destruction should be deleted
setPriority	Sets the priority of the thread being executed
setToGuardThread	Sets the **toGuardThread** flag
shouldGuardThread	Determines whether the thread should be guarded
start	Activates the thread to start its event-processing loop
stopAllThreads	Is used to support the DLL version of the Rhapsody in C++ execution framework (COM)

suspend	Suspends the thread
unlock	Unlocks the thread mutex
unschedTm	Cancels a timeout request

Attributes and Flags

aomthread

This protected attribute specifies the “instrumented” part of the thread.

It is defined as follows:

```
AOMThread *aomthread;
```

The `AOMThread` class is defined in the animation framework in the instrumented application, and set to an empty class in non-instrumented mode.

endOfProcess

This public attribute specifies whether the application is at the end of a process. If it is, the last thread in the process must “clean up.”

The possible values for this flag are as follows:

- `0`—Not at the end of a process
- `1`—At the end of a process

It is defined as follows:

```
static int endOfProcess;
```

eventQueue

This protected attribute specifies the thread’s event queue.

It is defined as follows:

```
OMEventQueue *eventQueue;
```

The class `OMEventQueue` is defined in `os.h`.

thread

This protected attribute specifies the "os" part of the thread.

It is defined as follows:

```
OMOSThread *thread;
```

The `OMOSThread` class is defined in `os.h`.

toGuardThread

This protected attribute determines whether a section of thread code will be protected. If it is set to `TRUE`, the code is protected. Otherwise, the code is not protected.

It is defined as follows:

```
OMBoolean toGuardThread;
```

`OMBoolean` is defined in `rawtypes.h`.

`toGuardThread` is checked by the **execute** method before it starts its event loop iteration. If `toGuardThread` is `TRUE`, `execute` calls the **START_THREAD_GUARDED_SECTION** and the **END_THREAD_GUARDED_SECTION** macros.

OMThread

Visibility

Public

Description

The **OMThread** method is the constructor for the `OMThread` class. See the section *Notes* for detailed information.

Signatures

```
OMThread (int wrapThread);


OMThread(const char* const name = NULL, const long
   priority = OMOSThread::DefaultThreadPriority,
   const long stackSize = OMOSThread::DefaultStackSize,
   const long messageQueueSize =
      OMOSThread::DefaultMessageQueueSize,
   OMBoolean dynamicMessageQueue = TRUE);
```

Parameters for Signature 1

`wrapThread`

Specifies whether a new operating system thread is constructed (the default, `wrapThread = 0`), or is a wrapper on the current thread.

A wrapper thread might be used, for example, in GUI applications where Rhapsody creates its own thread to attach to an existing GUI thread.

Parameters for Signature 2

`name`

Specifies a name for the thread. The default value is NULL.

`priority`

Specifies the thread priority.

`DefaultThreadPriority` is defined in `os.h` as follows:

```
static const long DefaultThreadPriority;
```

The default value is specified in `xxos.cpp`. For example, `ntos.cpp` specifies the following value:

```
const long OMOSThread::DefaultThreadPriority =
    THREAD_PRIORITY_NORMAL;
```

`stackSize`

Specifies the size of the stack.

`DefaultStackSize` is defined in `os.h` as follows:

```
static const long DefaultStackSize;
```

The default value is specified in `xxos.cpp`. For example, `ntos.cpp` specifies the following value:

```
const long OMOSThread::DefaultStackSize = 0;
```

`messageQueueSize`

Specifies the size of the message queue.

`DefaultMessageQueueSize` is defined in `os.h` as follows:

```
static const long DefaultMessageQueueSize;
```

The default value is specified in `xxos.cpp`. For example, `ntos.cpp` specifies the following value:

```
const long OMOSThread::DefaultMessageQueueSize =
    100;
```

`dynamicMessageQueue`

Specifies whether the message queue is dynamic. The default value is `TRUE`.

Notes

- `OMThread` inherits from the `OMProtected` class, a neutral implementation of a mutex. Every `OMThread` instance has a mutex because, in a multithreaded environment, your application must protect critical sections of code.
- `OMThread` aggregates `OMOSThread` to get the basic threading features.
- Initially, the message queue is empty. The maximum length of the message queue is operating system- and implementation-dependent, and is usually set in the file implementing the adapter for a specific operating system.

The message queue is an important building block for `OMThread`. It is used for intertask communication between Rhapsody tasks (active classes). `OMOSThread` provides a thread-safe, unbounded message queue (FIFO) for multiple writers and one reader. The reader pends the message queue until there is a message to process.

- Message queues are protected against concurrent operations from different threads.
- Initially, the thread is suspended until the **start** method is called. The **resume** and **suspend** methods provide a way of stopping and starting the thread. Because threads usually block when waiting for a resource like a mutex or event flag, these methods are rarely used.

Note the following distinctions between the different method calls:

- The first version of the method is the constructor for the `OMThread` class when a new thread is constructed as a wrapper on the current thread.
- `OMThread` creates a thread that is a wrapper on either the current thread or the thread whose ID it is passed. Wrapper threads are used only for instrumentation to represent user-defined threads (those defined outside the Rhapsody framework).
- The second version of the method is the constructor for the `OMThread` class when a new thread is constructed (as opposed to a wrapper on the current thread).
- The constructor works in the following way:
  - First, it calls the `init` method and passes to it the `name`, `stackSize`, `messageQueueSize`, and `dynamicMessageQueue` parameters that it was given. In addition, it passes `0` for the `wrapThread` parameter. Refer to the alternate constructor **OMThread** (defined in `omthread.h`).
  - Next, it calls the **setPriority** method and passes to it the `priority` parameter that it was given.

See Also

init

~OMThread

resume

start

suspend

~OMThread

Visibility

`Public`

Description

The ~OMThread method is the destructor for the `OMThread` class. It is called by the doExecute method.

`~OMThread` deletes (destroys) the thread if it is not the current thread. If the thread to be deleted is the current thread, it cannot be destroyed (because the system will halt). In this case, the thread is marked for destruction after it is no longer the current thread.

Signature

```
virtual ~OMThread()
```

See Also

doExecute

allowDeleteInThreadsCleanup

Visibility

`Public`

Description

The allowDeleteInThreadsCleanup method postpones the destruction of a framework thread until the application terminates and all user threads are deleted.

Do not override this method in user active classes.

Signature

```
virtual OMBoolean allowDeleteInThreadsCleanup()
```

cancelEvent

Visibility

`Public`

Description

The **cancelEvent** method marks a single event as canceled (that is, it changes the event’s ID to **OMCancelledEventId**).

Signature

```
virtual void cancelEvent(OMEvent* ev);
```

Parameters

`ev`

Specifies the event to be canceled

Notes

In the framework, `cancelEvent` is virtual to support enhanced framework customization. It can also support several event queues per task.

See Also

cancelEvents

cancelEvents

Visibility

`Public`

Description

The **cancelEvents** method marks all events targeted for the specified `OMReactive` instance as canceled (that is, it changes the events' IDs to **OMCancelledEventId**).

You might want to use the `cancelEvents` method if, for example, there are several events in the event queue targeted for a specific `OMReactive` instance, but the instance has already been destroyed because it reached a termination connector in the statechart.

The `cancelEvents` method works in the following way:

- It calls **unschedTm** and asks `OMThreadTimer::instance()` to cancel all timeouts (events) targeted to the specified `destination`.
- It gets a list of events in the event queue and iterates through the event queue. If the method finds an event targeted for destination, it sets its ID to **OMCancelledEventId**. The event still remains in the event queue; after it is eventually removed from the event queue, it is discarded.

Signature

```
virtual void cancelEvents(OMReactive* destination);
```

Parameters

`destination`

Specifies an `OMReactive` instance

Notes

In the framework, `cancelEvents` is virtual to support enhanced framework customization. It can also support several event queues per task.

See Also

cancelEvent

destination

unschedTm

cleanupAllThreads

Visibility

Public

Description

The **cleanupAllThreads** method "kills" all threads in an application except for the main thread and the thread running the `cleanupAllThreads` method.

The method supports static instances of active classes (particularly the static instance of `OMMainThread`).

Signature

```
static OMThread* cleanupAllThreads();
```

Notes

The `cleanupAllThreads` method is only called in RTOSes where the process cannot be "exited" in a simple manner.

cleanupThread

Visibility

Public

Description

The **cleanupThread** method provides a "hook" to allow a thread to be cleaned up without a call to the `DTOR`. This method enables you to clean up a thread without destroying the virtual function table.

Signature

```
virtual void cleanupThread()
```

destroyThread

Visibility

Public

Description

The **destroyThread** method destroys the default active class or object for the framework. It supports static instances of active classes (particularly the static instance of `OMMainThread`).

If you have a custom RTOS adaptor that deletes threads in `OSEndApplication`, modify the adapter to call `destroyThread` instead of the `delete` operator.

If you create by-value instances of an active class, you should override the `destroyThread` method to prevent the system from attempting to delete the static instances.

Signature

```
virtual void destroyThread()
```

doExecute

Visibility

Public

Description

The **doExecute** method is the entry point to the thread main loop function. `doExecute` handles "bookkeeping" issues and calls the **execute** method to do the actual event loop processing.

`doExecute` handles situations where the event loop is stopped for some reason. For example, if there is a single active object running on its own thread, and the object reaches a termination connector, it must "kill" itself and its thread. However, it cannot kill the thread until after it exits the event loop.

Signature

```
static void doExecute (void* me);
```

Parameters

`me`

Specifies a pointer to the `OMThread` instance to activate

Notes

The doExecute method calls ~OMThread, the destructor for the OMThread class, to delete a thread.

See Also

execute

~OMThread

execute

Visibility

Public

Description

The execute method is the thread main loop function. By default, this protected function processes the events in the thread's queue.

You can overwrite execute in order to implement customized thread behaviors.

The execute method works in the following way:

1. First, it sets the destination to NULL and the determinate attribute (defined in omreactive.cpp) to FALSE. The method continues iterating through the event queue in an almost infinite loop until toTerminate = TRUE.

2. execute enters a while loop to process events. First, it checks the toGuardThread attribute. If toGuardThread is TRUE, execute calls the START_THREAD_GUARDED_SECTION macro. toGuardThread should be set to TRUE by your application, if necessary.

3. execute gets the first event from the event queue. If the event is not a NULL event, execute calls the getDestination method to determine the OMReactive destination for the event.

4. If the event is not a canceled event, execute calls the takeEvent method to request that the reactive object process the event.

5. Finally, execute calls the isDeleteAfterConsume method to determine whether the deleteAfterConsume attribute is TRUE. If it is, execute calls the Delete method to delete the event.

Signature

```
virtual OMReactive* execute();
```

Return

This method returns `OMReactive`, which specifies the reactive class that "owns" the thread (active).

Note

The Rhapsody framework does not provide any default exception handler. One reason for this is that you can configure BSPs to exclude exception handling, which impacts footprint and performance. However, this does not prevent you from using your own C++ exception handler.

You may prefer to put a general fallback handler in the main loop of `OMThread` in the `execute` method. You can also add exception handling as a conditional code segment that should be disabled by default.

You can override `execute` to specialize different thread behaviors. For example, you can create an active class that is not reactive (see **Active and Reactive Classes**).

See Also

Delete

doExecute

getDestination

isDeleteAfterConsume

start

START_THREAD_GUARDED_SECTION

takeEvent

toGuardThread

getAOMThread

Visibility

Public

Description

The **getAOMThread** method is used by the framework for animation purposes.

Signature

```
AOMThread* getAOMThread() const
```

getEventQueue

Visibility

Public

Description

The **getEventQueue** method is used by the framework for animation purposes.

Signature

```
AOMEventQueue* getEventQueue() const;
```

getGuard

Visibility

Public

Description

The **getGuard** method gets the reference to the `OMProtected` part.

Signature

```
inline const OMProtected& getGuard() const
```

Return

The reference to the `OMProtected` part

getOsHandle

Visibility

`Public`

Description

The getOsHandle method returns the thread's operating system ID. This method is operating system-dependent.

Signatures

```
void* getOsHandle();
```

```
void* getOsHandle(void*& osHandle);
```

Parameters for Signature 2

`osHandle`

Specifies the operating system handle

Return

The thread's operating system ID

Notes

- The second version of the method supports the DLL version of the framework (COM).
- A real-time operating system (RTOS) usually provides a pointer to an ID or handle for the active thread. This is useful if you need to know the ID of the real thread that is running, because the object itself only "knows" that it is running on `OMThread`.

See Also

getOsHandle

getOSThreadEndClb

Visibility

`Public`

Description

The getOSThreadEndClb method requests a callback to end the current operating system thread. There are two callbacks, depending on whether you are "sitting" on your own thread, or you are an object belonging to another thread.

Signature

```
void getOSThreadEndClb (
   OMOSThread::OMOSThreadEndCallBack *clb_p,
   void **arg1_p, OMBoolean onExecuteThread = TRUE)
   const;
```

Parameters

`clb_p`

Is a pointer to the callback function.

`arg1_p`

Specifies the argument for the callback function.

`onExecuteThread`

Specifies how the current thread will be "killed." If this is `TRUE`, the current thread kills itself. If it is `FALSE`, another thread will kill the current thread

Note

The `getOSThreadEndClb` method is typically used in conjunction with the setEndOSThreadInDtor method.

See Also

setEndOSThreadInDtor

getStepper

Visibility

Public

Description

The **getStepper** method is used by the framework for animation purposes.

Signature

```
AOMStepper* getStepper() const;
```

lock

Visibility

Public

Description

The **lock** method puts a lock on the thread mutex.

Signature

```
inline void lock() const
```

omGetEventQueue

Visibility

Public

Description

The **omGetEventQueue** method returns the event queue. This method is not used by the framework.

Signature

```
virtual const OMEventQueue* omGetEventQueue() const
```

Return

The event queue

queueEvent

Visibility

Public

Description

The queueEvent method queues events to be processed by the thread event loop (execute).

Signature

```
virtual OMBoolean queueEvent(OMEvent* ev,
   OMBoolean fromISR = FALSE);
```

Parameters

`ev`

Specifies the event to be queued

`fromISR`

Specifies whether the event has been generated by an interrupt service request (ISR)

Return

The method returns one of the following Boolean values:

- `TRUE`—The method successfully queued the event.
- `FALSE`—The method was unable to queue the event.

Notes

In the framework, `queueEvent` is virtual to support enhanced framework customization. It can also support several event queues per task.

See Also

action

execute

gen

resume

Visibility

`Public`

Description

The **resume** method resumes a thread suspended by the **suspend** method.

Threads usually block when waiting for a resource like a mutex or event flag, so `resume` is rarely used by the generated code. You can use `resume` for advanced scheduling.

Signature

```
void resume();
```

See Also

suspend

schedTm

Visibility

`Public`

Description

The **schedTm** method creates a timeout request and delegates the request to `OMTimerManager`.

Signature

```
virtual void schedTm (timeUnit delteTime, short id,
   OMReactive *instance, const OMHandle * state = NULL);
```

Parameters

`delteTime`

Specifies the delay time, in milliseconds, before the timeout request is triggered.

`id`

Identifies the timeout, either at delivery or for canceling. Every timeout has a specific ID to distinguish it from other timeouts.

`instance`

Specifies a pointer to the `OMReactive` instance requestor. After a timeout has matured, this parameter points to the instance that should be notified.

`state`

Specifies an optional parameter used by the Rhapsody instrumentation to designate a pointer to the state name upon which the timeout is scheduled. The default value is NULL, for the noninstrumented case.

Notes

- In the framework, `schedTm` is virtual to support enhanced framework customization. It can also support several timer managers in the system (for example, one per active class).
- `schedTm` creates the timeout using the **incarnateTimeout** method defined in `omreactive.h`.
- `schedTm` delegates the timeout to `OMTimerManager` using the **set** method defined in `timer.h`.
- The code generator generates a call to `schedTm` when it encounters timeout transitions.
- You can use `schedTm` if the statechart implementation is overridden.

See Also

incarnateTimeout

set

setEndOSThreadInDtor

Visibility

`Public`

Description

The **setEndOSThreadInDtor** method specifies whether an operating system thread in destruction should be deleted.

Signature

```
void setEndOSThreadInDtor (OMBoolean val)
```

Parameters

`val`

Specifies one of the following Boolean values:

- `TRUE`—Delete the object representing the operating system thread (and release the resources).
- `FALSE`—Do not delete the object representing the operating system thread. For example, the application is executing on this thread and, if it is deleted, the system will “leak” resources.

Notes

- **~OMThread** calls `setEndOSThreadInDtor` with a value of `TRUE` prior to destroying the thread.
- `deregisterThread` (private) calls `setEndOSThreadInDtor` with a value of `TRUE` prior to destroying the thread.
- `setEndOSThreadInDtor` is typically used in conjunction with the **isNotDelay** method.

See Also

~OMThread

isNotDelay

setPriority

Visibility

`Public`

Description

The **setPriority** method sets the priority of the thread being executed.

This method is operating system-dependent.

Signature

```
void setPriority (int pr);
```

Parameters

`pr`

Specifies the thread's priority

See Also

OMThread

setToGuardThread

Visibility

`Public`

Description

The **setToGuardThread** method sets the **toGuardThread** flag.

Signature

```
inline void setToGuardThread (OMBoolean flag)
```

Parameters

`flag`

Specifies the value for the **toGuardThread** attribute

See Also

toGuardThread

shouldGuardThread

Visibility

`Public`

Description

The **shouldGuardThread** method determines whether the thread should be guarded.

Signature

```
inline OMBoolean shouldGuardthread() const
```

Return

The method returns one of the following Boolean values:

- `TRUE`—Guard the thread.
- `FALSE`—Do not guard the thread.

start

Visibility

`Public`

Description

The **start** method activates the thread to start its event-processing loop.

If an object has its own thread, when the object is created, the thread is suspended. The `start` method is used to start event processing. This enables an active class to initialize itself by calling the **startBehavior** method, then to call the `start` method to start event processing.

The `start` method works in the following way:

- If the value of the `doFork` attribute is `FALSE`, `start` calls the **execute** method and the main thread simply grabs control from the system.
- If the value of the `doFork` attribute is `TRUE`, `start` issues the following calls:

```
OMOSThread * oldWrapperThread = thread;
thread = theOSFactory()->createOMOSThread(
    doExecute, this);
```

In this situation, the thread is registered, but does not take control. Another thread (for example, a GUI thread) will be responsible for event loop processing.

Signature

```
virtual void start(int = 0);
```

Notes

- The constructor of the composite object starts preexisting instances.
- The creator should start any dynamically created instances of `OMThread`.

See Also

execute

resume

suspend

stopAllThreads

Visibility

Public

Description

The stopAllThreads method is used to support the DLL version of the Rhapsody in C++ execution framework (COM).

Note

The method is used in the COM environment only, as part of the implementation of `OXF::`end.

Signature

```
static OMThread* stopAllThreads(OMThread* skipme);
```

Parameters

`skipme`

The framework uses this parameter to avoid killing the `NTHandleCloser` in the Microsoft environment.

suspend

Visibility

`Public`

Description

The **suspend** method suspends the thread.

Threads usually block when waiting for a resource like a mutex or event flag, so `suspend` is rarely used by the generated code. You can use `suspend` for advanced scheduling.

Signature

```
void suspend();
```

See Also

resume

unlock

Visibility

`Public`

Description

The **unlock** method unlocks the thread mutex.

Signature

```
inline void unlock() const
```

unschedTm

Visibility

`Public`

Description

The **unschedTm** method cancels a timeout request.

This method is used when:

- **Exiting a state**—The timeout is no longer relevant.
- **An object has been destroyed**—In this case, all timers associated with the object are destroyed.

Signature

```
virtual void unschedTm (short id, OMReactive *c);
```

Parameters

`id`

Specifies the ID tag of the timeout request. If this is **OMEventAnyEventId**, `unschedTm` cancels all events whose destination is this specific instance of `OMReactive`. If this is set to a specific event ID, `unschedTm` cancels only that event.

`c`

Specifies a pointer to the `OMReactive` instance requestor. After a timeout has been canceled, this parameter points to the instance that should be notified.

Notes

- In the framework, `unschedTm` is virtual to support enhanced framework customization. It can also support several timer managers in the system (for example, one per active class).
- The code generator generates a call to `unschedTm` when the state upon which the timeout was scheduled has been exited.
- `unschedTm` calls the **unschedTm** method defined in `timer.h`.
- Canceling a timeout requires one of two actions:
  - Deleting the timeout from the heap
  - Canceling it inside the event queue (if it was already dispatched) by iterating the event queue
- You can use `unschedTm` in cases where the statechart implementation is overridden.

See Also

OMEventAnyEventId,

cancelEvents

OMThreadTimer Class

OMThreadTimer inherits from OMTimerManager and performs the actual timing services for the framework and your application. This class is declared in the file timer.h.

Thread timing is delegated to OMThreadTimer by OMTimerManager so OMTimerManager can be a general purpose timer, and other timers can be created to perform specific timing tasks. For example, OMThreadTimer is a *periodic* timer—every tick time it starts working, then suspends itself for the tick time period (so as not to consume CPU time). Another possible type of timer would be an *asynchronous* timer—one activated by an interrupt from the operating system.

Currently, OMThreadTimer is the only specific timer in the Rhapsody framework.

Note

The OMThreadTimer method is part of the base class, OMTimerManager.

Construction Summary

~OMThreadTimer	Destroys the OMThreadTimer object

Method Summary

action	Sends a matured timeout request to the relevant thread, where it is then inserted into the thread's event queue
initInstance	Creates an instance of OMThreadTimer

~OMThreadTimer

Visibility

Public

Description

The **~OMThreadTimer** method is the destructor for the `OMThreadTimer` class.

Signature

```
RP_FRAMEWORK_DLL virtual ~OMThreadTimer
```

action

Visibility

Public

Description

The **action** method sends a matured timeout request to the relevant thread, where it is then inserted into the thread's event queue.

The `action` method checks the value of `isNotDelay` to see whether the timeout is a delay. If the timeout is not a delay (`isNotDelay = TRUE`), `action` determines the thread of the receiver. First, `action` calls **getDestination** to determine the `OMReactive` instance to which the timeout is delegated.

If the `OMReactive` instance exists, `action` calls **getThread** to determine the `OMThread` to which the timeout is delegated. If the `OMThread` instance exists, `action` calls **queueEvent** to insert the timeout in the thread's event queue.

If the timeout is a delay (`isNotDelay = False`), the thread is the receiver. `action` calls `getDestination`, then calls `wakeup`.

Signature

```
RP_FRAMEWORK_DLL virtual void action (Timeout *timeout);
```

Parameters

`timeout`

Specifies the timeout request to be sent to the thread

Note

The `action` method overrides the private `action` method defined in the `OMTTimerManager` class.

See Also

getDestination

getThread

isNotDelay

OMDelay

OMTimerManager

queueEvent

wakeup

initInstance

Visibility

Public

Description

The **initInstance** method creates an instance of `OMThreadTimer`. `OMThreadTimer` is a singleton.

Signature

```
RP_FRAMEWORK_DLL static OMThreadTimer* initInstance(
    int ticktime =
        OMTimerManagerDefaults::defaultTicktime,
    unsigned maxTM = OMTimerManagerDefaults::defaultMaxTM,
    OMBoolean isRealTimeModel = TRUE);
```

Parameters

ticktime

Specifies the basic system tick, in milliseconds. Every ticktime, the framework and user application are notified that the time was advanced.

defaultTicktime is defined in `timer.h` as follows:

```
static const unsigned defaultTicktime;
```

The default value is specified in `oxf.cpp` as follows:

```
const unsigned
     OMTimerManagerDefaults::defaultTicktime = 100;
```

`maxTM`

Specifies the maximum number of timeouts that can exist simultaneously in the system. The value for `maxTM` is used to construct the heap and matured list for storing timeouts.

defaultMaxTM is defined in `timer.h` as follows:

```
static const unsigned defaultMaxTM;
```

The default value is specified in `oxf.cpp` as follows:

```
const unsigned
     OMTimerManagerDefaults::defaultMaxTM = 100;
```

See Also

OMTimerManager

OMTimeout Class

A *timeout* is an event used for notification that a specified time interval has expired (that is, it implements a UML time event).

Timeouts are either created by instances entering states with timeout transitions, or delay requests from user code. In the latter case, the `timeoutDelayId` of this event is as follows:

```
    const short timeoutDelayId = -1;
```

The `OMTimeout` class is declared in the header file `event.h`.

`OMTimeout` uses the following comparison functions to manipulate its heap structure:

```
int operator==(OMTimeout& tn)
    {OMBoolean matchDest = getDestination() ==
        tn.getDestination();
    OMBoolean matchId = ((getTimeoutId() ==
        tn.getTimeoutId()) || (getTimeoutId() ==
        OMEventAnyEventId) ||
        (OMEventAnyEventId == tn.getTimeoutId()));
    return (matchDest && matchId);
}
int operator>(OMTimeout& tn) {return dueTime >
    tn.dueTime;}
int operator<(OMTimeout& tn) {return dueTime <
    tn.dueTime;}
```

Attribute Summary

timeoutDelayId	Identifies a delay request from user code

Macro Summary

DECLARE_MEMORY_ALLOCATOR	Specifies a set of methods that declare the memory pool for timeouts

Construction Summary

OMTimeout	Constructs an `OMTimeout` object
~OMTimeout	Destroys the `OMTimeout` object

Method Summary

operator ==	Determines whether the current values of `destination` and `Timeout` are the same as those of the specified timeout
operator >	Determines whether the current value of `Timeout` is greater than the due time of the specified timeout
operator <	Determines whether the current value of `Timeout` is less than the due time of the specified timeout
Delete	Deletes a timeout from the heap
getDelay	Returns the current value of `delayTime`
getDueTime	Returns the due time of a timeout request stored in the heap
getTimeoutId	Returns the current value for `timeoutId`
isNotDelay	Determines whether a timeout event is a timeout delay
new	Allocates additional memory
setDelay	Sets the value of `Timeout`
setDueTime	Specifies the value for the `Timeout` attribute
setRelativeDueTime	Calculates and sets the due time for a timeout based on the current system time and the requested delay time
setState	Used by the framework to set the current state
setTimeoutId	Specifies the value for `timeoutId`

Attribute

timeoutDelayId

This global attribute identifies a delay request from user code. It is defined as follows:

```
const short timeoutDelayId = -1;
```

Macro

DECLARE_MEMORY_ALLOCATOR

This public macro specifies a set of methods that declare the memory pool for timeouts. The default number of timeouts is `100`.

The `DECLARE_MEMORY_ALLOCATOR` macro is defined in `MemAlloc.h` as follows:

```
#define DECLARE_MEMORY_ALLOCATOR (CLASSNAME)

   public:

   CLASSNAME * OMMemoryPoolNextChunk;
   DECLARE_ALLOCATION_OPERATORS
      static void OMMemoryPoolIsEmpty();
      static void OMMemoryPoolSetIncrement(int value);
      static void OMCallMemoryPoolIsEmpty(
         OMBoolean flagValue);
      static void OMSetMemoryAllocator(
         CLASSNAME*(*newAllocator)(int));
```

OMTimeout

Visibility

`Public`

Description

The **OMTimeout** method is the constructor for the `OMTimeout` class.

Signatures

```
OMTimeout();


OMTimeout (short id, OMReactive* pdest, timeUnit delay,
   const OMHandle* theState);
```

Parameters

`id`

Specifies the timeout ID

`pdest`

Specifies the destination `OMReactive` instance

`delay`

Specifies the requested delay, in milliseconds

`theState`

Specifies an optional state handle used for Rhapsody instrumentation purposes

See Also

~OMTimeout

~OMTimeout

Visibility

Public

Description

The ~OMTimeout method is the destructor for the `OMTimeout` class.

Signature

```
~OMTimeout();
```

See Also

OMTimeout

operator ==

Visibility

Public

Description

The == operator is a comparison function used by `OMTimerManager` to manipulate its heap structure. It determines whether the current values of `destination` and `Timeout` are the same as those of the specified timeout.

The comparison yields one of the following values:

- `1`—The current values of `destination` and `Timeout` are the same as those of the specified timeout.
- `0`—The current values of `destination` and `Timeout` are not the same as those of the specified timeout.

Signature

```
int operator == (OMTimeout& tn) {
   OMBoolean matchDest = getDestination() ==
      tn.getDestination();
   OMBoolean matchId = ((getTimeoutId() ==
```

```
        tn.getTimeoutId()) ||
        (getTimeoutId() == OMEventAnyEventId) ||
        (OMEventAnyEventId == tn.getTimeoutId()));
    return (matchDest && matchId);}
```

Parameters

`tn`

Specifies the address of the timeout

See Also

[operator >](#)

[operator <](#)

operator >

Visibility

`Public`

Description

The > operator is a comparison function used by `OMTimerManager` to manipulate its heap structure. It determines whether the current value of `Timeout` is greater than the due time of the specified timeout.

The comparison yields one of the following values:

- `1`—The current value of `Timeout` is greater than the due time for the specified timeout.
- `0`—The current value of `Timeout` is not greater than the due time for the specified timeout.

Signature

```
int operator > (OMTimeout& tn)
```

Parameters

`tn`

Specifies the address of the timeout

See Also

[operator ==](#)

[operator <](#)

operator <

Visibility

`Public`

Description

The < operator is a comparison function used by `OMTimerManager` to manipulate its heap structure. It determines whether the current value of `Timeout` is less than the due time of the specified timeout.

The comparison yields one of the following values:

- `1`—The current value of `Timeout` is less than the due time for the specified timeout.
- `0`—The current value of `Timeout` is not less than the due time for the specified timeout.

Signature

```
int operator < (OMTimeout& tn)
```

Parameters

`tn`

Specifies the address of the timeout

See Also

operator ==, page 305

operator >, page 306

Delete

Visibility

Public

Description

The `Delete` method deletes a timeout from the heap. This is the only method that should be used to delete timeouts.

Signature

```
void Delete();
```

Notes

- The **unschedTm** method iterates through the heap, and calls the `Delete` method to delete one or more timeouts.
- The **DECLARE_MEMORY_ALLOCATOR** macro creates the memory pool for timeouts. The `Delete` operator returns memory to the memory pool. The `new` operation gets memory from the memory pool.

See Also

DECLARE_MEMORY_ALLOCATOR

new

unschedTm

getDelay

Visibility

`Public`

Description

The `getDelay` method returns the current value of `delayTime`.

Signature

```
timeUnit getDelay() const
```

Return

The value for timeout delays, in milliseconds

See Also

setDelay

getDueTime

Visibility

`Public`

Description

The `getDueTime` method returns the due time of a timeout request stored in the heap.

Signature

```
timeUnit getDueTime() const
```

Return

The time at which the timeout request becomes due (ready to be sent to the relevant thread as an event)

getTimeoutId

Visibility

`Public`

Description

The `getTimeoutId` method returns the current value for `timeoutId`.

Signature

```
short getTimeoutId() const
```

Return

The timeout ID

Notes

Rhapsody defines several special ID values, as follows:

Rhapsody ID	Value	Description
`OMEventNullId`	-1	The null event ID
`OMEventTimeoutId`	-2	The timeout event ID
`OMEventCancelledEventId`	-3	The canceled event ID
`OMEventAnyEventId`	-4	The ID for all events delegated to a specific `OMReactive` instance
`OMEventStartBehaviorId`	-5	The ID reserved for the `OMReactive startBehavior` event
`OMEventOXFEndEventId`	-6	Used for COM support in terminating the framework when it is used by multiple COM servers in different DLLs

See Also

[setTimeoutId]

isNotDelay

Visibility

`Public`

Description

The **isNotDelay** method determines whether a timeout event is a timeout delay.

Signature

```
OMBoolean isNotDelay() const
```

Return

The method returns one of the following Boolean values:

- `TRUE`—The timeout is not a delay.
- `FALSE`—The timeout is a delay.

new

Visibility

`Public`

Description

The **new** operator allocates additional memory.

The following macros call this method:

- GEN
- GEN_BY_GUI
- GEN_BY_X

Signature

```
void * operator new (size_t size, void * p);
```

Parameters

`size`

Specifies the memory required

`p`

Specifies a pointer to the memory location

Notes

- Rhapsody overwrites the standard `new` operator to support its static architecture during run time.
- Rhapsody uses `malloc` and dynamic memory allocation (DMA) during initialization.
- The DECLARE_MEMORY_ALLOCATOR macro creates the memory pool for timeouts. The `new` operator gets memory from the memory pool. The `Delete` operation returns memory to the memory pool.

See Also

DECLARE_MEMORY_ALLOCATOR

Delete

GEN

GEN_BY_GUI

GEN_BY_X

setDelay

Visibility

`Public`

Description

The setDelay method sets the value of `Timeout`.

Signature

```
void setDelay(timeUnit delay)
```

Parameters

`delay`

Specifies the timeout delay, in milliseconds

See Also

getDelay

setDueTime

Visibility

`Public`

Description

The **setDueTime** method specifies the value for the `Timeout` attribute.

Signature

```
void setDueTime(timeUnit newDueTime)
```

Parameters

`newDueTime`

Specifies the new value for `Timeout`

See Also

getDueTime

setRelativeDueTime

Visibility

`Public`

Description

The **setRelativeDueTime** method calculates and sets the due time for a timeout based on the current system time and the requested delay time. This method is called by the `set` method.

Signature

```
void setRelativeDueTime(timeUnit now)
```

Parameters

`now`

Specifies the current system time

See Also

set

setState

Visibility

```
Public
```

Description

The **setState** method is used by the framework to set the current state. This method is used for animation purposes.

Signature

```
void setState(const OMHandle * s)
```

See Also

getTimeoutId

setTimeoutId

Visibility

`Public`

Description

The **setTimeoutId** method specifies the value for `timeoutId`.

Signature

```
void setTimeoutId (short id)
```

Parameters

`id`

Specifies the identifier to assign to `timeoutId`

Notes

Rhapsody defines several special ID values, as follows:

Rhapsody ID	Value	Description
`OMEventNullId`	-1	The null event ID
`OMEventTimeoutId`	-2	The timeout event ID
`OMEventCancelledEventId`	-3	The canceled event ID
`OMEventAnyEventId`	-4	The ID for all events delegated to a specific `OMReactive` instance
`OMEventStartBehaviorId`	-5	The ID reserved for the `OMReactive startBehavior` event
`OMEventOXFEndEventId`	-6	Used for COM support in terminating the framework when it is used by multiple COM servers

OMTimerManager Class

`OMTimerManager` provides timer services for all threads using a single timer task. The class is declared in the header file `timer.h`.

`OMTimerManager` manages timeout requests and issues timeout events to the system objects. `OMTimerManager` is a singleton active object. During framework initialization, the singleton is created and a single new thread is created for managing the timeout requests.

Note

In every Rhapsody-generated application, a separate thread provides timer support for the application. If your application is single-threaded, the Rhapsody-generated application will have two threads—one thread for the application and one thread for timer support.

`OMThreadTimer` inherits from `OMTimerManager` and performs the actual timing services for the framework and your application. For more information on `OMThreadTimer`, see **OMThreadTimer Class**.

`OMTimerManager` can implement two time models:

- **real time**—Time advances according to the actual underlying operating system clock.
- **simulated time**—Time advances *explicitly*, by calling the **consumeTime** method, or *implicitly*, when all reactive objects are idle (they do not have an event in their event queue) and there is at least one pending timeout.

Simulated time is useful for debugging and algorithm validation.

The simulated time support is in run-time (a parameter is provided to the framework in the application initialization). However, in order to switch between real and simulated time, you need to regenerate and build the code.

In the current version, simulated time is handled at initialization time, via the `isRealTime` parameter in `OXF::`**init**.

The following methods are used with simulated time mode: **init**, the `OMTimerManagerDefaults` class, `goNext` (private), and **goNextAndPost**.

Attribute Summary

overflowMark	Specifies the value used to determine whether the current system time has "overflowed"

Construction Summary

OMTimerManager	Constructs an `OMTimerManager` object
~OMTimerManager	Destroys the `OMTimerManager` object

Method Summary

action	Sends a matured timeout request to the relevant thread, where it is then inserted into the thread's event queue
cbkBridge	Is a bridge to get an interrupt from the operating system via the `timeTickCbk` (private) method
clearInstance	Cleans up the singleton instance of the timer manager
consumeTime	Is used in simulated time mode to simulate time consumption
destroyTimer	Cleans up the timer manager singleton instance
getElapsedTime	Returns the value of `m_Time`, the current system time.
goNextAndPost	Is used in simulated time mode
init	Starts the timer ticking
initInstance	Initializes the singleton instance
instance	Creates the singleton instance of the timer manager
resume	Is used by the framework to resume the timer during animation
set	Delegates a timeout request to `OMTimerManager`
setElapsedTime	Sets the value of `m_Time`, the current system time
softUnschedTm	Removes a specific timeout from the matured list
suspend	Is used by the framework to suspend the timer during animation

unschedTm	Cancels a timeout request

Attributes

overflowMark

This protected attribute specifies the value used to determine whether the current system time (`m_Time`) has “overflowed.” `m_Time` is implemented as an unsigned long integer; its maximum value is implementation-dependent.

It is defined as follows:

```
RP_FRAMEWORK_DLL static const timeUnit
        overflowMark;
```

The `timeUnit` method is defined in `rawtypes.h` as follows:

```
typedef unsigned long timeUnit;
```

The value for `overflowMark` is specified in `timer.cpp` as follows:

```
const timeUnit OMTimerManager::overflowMark =
        0x80000000;
```

The `post` method compares `m_Time` to `overflowMark` after it gets a pointer to the current timeout request in the heap. If `m_Time >= overflowMark`, the `post` method iterates over the heap to adjust the `dueTime` of each timeout request, and resets `m_Time` as follows:

```
m_Time &= ~overflowMark;
```

Updating `dueTime` and `m_Time` uses system resources. You should monitor `m_Time` carefully for your application.

OMTimerManager

Visibility

`Public`

Description

The **OMTimerManager** method is the constructor for the `OMTimerManager` class.

Signature

```
RP_FRAMEWORK_DLL OMTimerManager (int ticktime =
   OMTimerManagerDefaults::defaultTicktime,
   unsigned int maxTM =
      OMTimerManagerDefaults::defaultMaxTM,
   OMBoolean isRealTimeModel = TRUE);
```

Parameters

`ticktime`

Specifies the basic system tick, in milliseconds. At every tick, the Rhapsody framework and user application are notified that the time was advanced.

The `defaultTicktime` specifies the default tick time, defined in `timer.h` as follows:

```
static const unsigned defaultTicktime;
```

The default value is specified in `oxf.cpp` as follows:

```
const unsigned OMTimerManagerDefaults::
   defaultTicktime = 100;
```

`maxTM`

Specifies the maximum number of timeouts that can exist simultaneously in the system. The value for `maxTM` is used to construct the heap and the matured list for storing timeouts.

The `defaultMaxTM` is defined in `timer.h` as follows:

```
static const unsigned defaultMaxTM;
```

The default value is specified in `oxf.cpp` as follows:

```
const unsigned OMTimerManagerDefaults::
   defaultMaxTM = 100;
```

`isRealTimeModel`

Specifies whether the time model is real (`TRUE`) or simulated (`FALSE`).

Notes

- The defaultTicktime is 100 milliseconds. As you decrease ticktime (for example, to 50 ms) you get a "finer" timer accuracy, but the thread consumes more CPU time (because it's a separate thread). In addition, the actions that your application performs every ticktime also take time. If you specify a very small ticktime, the system might get into conflicts. You should use 100 milliseconds for this value.
- You can change the default clock tick of 100 milliseconds by editing the value assigned to defaultTicktime in the constructor and then recompiling the OXF libraries.
- You can override the default tick time by setting the TimerResolution property (under <lang>_CG::Framework).
- The framework uses maxTM to construct a heap and a matured list of timeouts. The defaultMaxTM is 100. maxTM enables the dynamic framework to provide a static architecture, thereby avoid dynamic memory allocation during run time. In addition, a static run-time architecture enables you to easily analyze the system. Rhapsody static events facilitate real-time and safety-critical systems that do not require (or allow) dynamic memory management during run time. Note, however, that Rhapsody requires malloc during initialization and your application must support dynamic memory management.
- The DECLARE_MEMORY_ALLOCATOR macro creates the memory pool for timeouts. The new operator gets memory from the memory pool. The Delete operation returns memory to the memory pool.
- To change the value of maxTM for your application, change the defaultMaxTM attribute. You can also override the default maximum number of timeouts by setting the TimerMaxTimeouts property (under <lang>_CG::Framework).
- If your application exceeds maxTM and tries to create additional timeouts, the return value will be NULL. You must specify, in advance, the maximum number of timeouts that can exist together in the system.

See Also

DECLARE_MEMORY_ALLOCATOR

defaultMaxTM

defaultTicktime

~OMTimerManager

Visibility

Public

Description

The ~OMTimerManager method is the destructor for the `OMTimerManager` class. It deletes (destroys) the operating system entity that the instance wraps.

Signature

```
RP_FRAMEWORK_DLL virtual ~OMTimerManager();
```

See Also

OMTimerManager

action

Visibility

Public

Description

The action method sends a matured timeout request to the relevant thread, where it is then inserted into the thread's event queue.

This method is overridden by the `OMThreadTimer::action` method.

Signature

```
RP_FRAMEWORK_DLL virtual void action (
    OMTimeout *timeout);
```

Parameters

timeout

Specifies the timeout request to be sent to the thread

See Also

action

cbkBridge

Visibility

`Public`

Description

The **cbkBridge** method is a bridge to get an interrupt from the operating system via the `timeTickCbk` (private) method.

This method is defined because the API of most RTOSes expects a C function to handle an interrupt.

Signature

```
RP_FRAMEWORK_DLL static void cbkBridge (void *me)
```

Parameters

`me`

Gets the interrupt from the `timeTickCbk` method

clearInstance

Visibility

`Public`

Description

The **clearInstance** method cleans up the singleton instance of the timer manager.

Signature

```
RP_FRAMEWORK_DLL static void clearInstance()
```

consumeTime

Visibility

`Public`

Description

The **consumeTime** method is used in simulated time mode to simulate time consumption. It increases time incrementally so it can be preempted by other tasks.

Signature

```
RP_FRAMEWORK_DLL void consumeTime (timeUnit interval,
   timeUnit step = 1);
```

Parameters

`interval`

Defines the time interval used for clock updates.

`step`

Defines how many intervals to change at each clock update. The default value is 1.

decNonIdleThreadCounter

Visibility

`Public`

Description

The **decNonIdleThreadCounter** method decreases the `nonIdleThreadCounter` private attribute.

Signature

```
RP_FRAMEWORK_DLL void decNonIdleThreadCounter()
```

See Also

incNonIdleThreadCounter

destroyTimer

Visibility

`Public`

Description

The **destroyTimer** method cleans up the timer manager singleton instance.

Signature

```
RP_FRAMEWORK_DLL void destroyTimer()
```

getElapsedTime

Visibility

`Public`

Description

The **getElapsedTime** method returns the value of `m_Time`, the current system time.

This method is useful for debugging purposes. Using it, you can determine when a state was entered, when an event was put in the event queue, and so on.

Signature

```
RP_FRAMEWORK_DLL timeUnit getElapsedTime() const
```

Return

`m_Time`, the current system time

See Also

setElapsedTime

goNextAndPost

Visibility

`Public`

Description

The **goNextAndPost** method is used in simulated time mode. It creates a mutex, then calls the `goNext` method, followed by the `post` method. Note that `goNext` and `post` are private methods.

Signature

```
RP_FRAMEWORK_DLL void goNextAndPost();
```

incNonIdleThreadCounter

Visibility

`Public`

Description

The **incNonIdleThreadCounter** method increases the `nonIdleThreadCounter` private attribute.

Signature

```
RP_FRAMEWORK_DLL void incNonIdleThreadCounter()
```

See Also

decNonIdleThreadCounter

init

Visibility

Public

Description

The **init** method starts the timer ticking. It is used by the framework initialization.

In real-time mode, `init` creates an `OMOSTickTimer`, as follows:

```
osTimer = theOSFactory() ->
    createOMOSTickTimer(tick, cbkBridge, this);
```

In simulated time mode, `init` creates an `OMOSIdleTimer`, as follows:

```
osTimer = theOSFactory() ->
    createOMOSIdleTimer (cbkBridge, this);
```

Signature

```
RP_FRAMEWORK_DLL virtual void init();
```

initInstance

Visibility

Public

Description

The **initInstance** method initializes the singleton instance.

Signature

```
RP_FRAMEWORK_DLL static OMTimerManager* initInstance(
    int tickTime =
    OMTimerManagerDefaults::defaultTicktime,
    unsigned int maxTM =
    OMTimerManagerDefaults::defaultMaxTM,
    OMBoolean isRealTimeModel=TRUE);
```

Parameters

`ticktime`

Specifies the basic system tick, in milliseconds. At every tick, the Rhapsody framework and user application are notified that the time was advanced.

The `defaultTicktime` specifies the default tick time, defined in `timer.h` as follows:

```
static const unsigned defaultTicktime;
```

The default value is specified in `oxf.cpp` as follows:

```
const unsigned OMTimerManagerDefaults::
    defaultTicktime = 100;
```

`maxTM`

Specifies the maximum number of timeouts that can exist simultaneously in the system. The value for `maxTM` is used to construct the heap and the matured list for storing timeouts.

The `defaultMaxTM` is defined in `timer.h` as follows:

```
static const unsigned defaultMaxTM;
```

The default value is specified in `oxf.cpp` as follows:

```
const unsigned OMTimerManagerDefaults::
    defaultMaxTM = 100;
```

`isRealTimeModel`

Specifies whether the time model is real (`TRUE`) or simulated (`FALSE`).

instance

Visibility

Public

Description

The **instance** method creates the singleton instance of the timer manager.

Signature

```
RP_FRAMEWORK_DLL static OMTimerManager* instance()
```

resume

Visibility

`Public`

Description

The **resume** method is used by the framework to resume the timer during animation.

Signature

```
RP_FRAMEWORK_DLL void resume()
```

See Also

suspend

set

Visibility

`Public`

Description

The set method delegates a timeout request to `OMTimerManager`.

Signature

```
RP_FRAMEWORK_DLL void set(OMTimeout* timeout);
```

Parameters

`timeout`

Specifies the timeout event to be delegated to `OMTimerManager`

Notes

- The `set` method is called by the schedTm method, defined in `omthread.h`.
- The `set` method first locks a mutex, calls setRelativeDueTime to set the due time for the timeout based on the current value of `m_Time`, then adds the timeout to the timeout heap.
- After the `set` operation is completed, the heap contains a list of requested timeouts, with the first timeout request in the heap scheduled to occur next.

See Also

schedTm

setRelativeDueTime

setElapsedTime

Visibility

Public

Description

The **setElapsedTime** method sets the value of m_Time, the current system time.

Note

The setElapsedTime method is used for debugging purposes to start the timer at a specific time. This method should be used only with great care.

Signature

```
RP_FRAMEWORK_DLL void setElapsedTime (timeUnit newTime);
```

Parameters

newTime

Specifies the new system time

See Also

getElapsedTime

softUnschedTm

Visibility

`Public`

Description

The **softUnschedTm** method removes a specific timeout from the matured list.

This method is called only from `~OMTimeout`, the timeout destructor.

Signature

```
RP_FRAMEWORK_DLL void softUnschedTm (OMTimeout* Timeout);
```

Parameters

`Timeout`

Specifies the timeout to remove from the matured list

See Also

~OMTimeout

suspend

Visibility

`Public`

Description

The **suspend** method is used by the framework to suspend the timer during animation.

Signature

```
RP_FRAMEWORK_DLL void suspend()
```

See Also

resume

unschedTm

Visibility

`Public`

Description

The **unschedTm** method cancels a timeout request.

This method is used when:

- **Exiting a state**—The timeout is no longer relevant.
- **An object has been destroyed**—In this case, all timers associated with the object are destroyed.

The `unschedTm` method works in the following way:

1. If the `OMReactive` instance does not exist, `unschedTm` returns; otherwise, it invokes a mutex to protect the following operations:
   - If `id ==` **OMEventAnyEventId**, `unschedTm` cancels all events whose destination is this specific instance of `OMReactive`.
   - `unschedTm` calls the `isCurrentEvent` method to determine whether the current event is delegated to this `OMReactive`. If it is, `unschedTm` calls the `findInList` method (private) to locate the timeout in the matured list, then removes it from the matured list.
2. Next, `unschedTm` creates three clones for the following items:
   - The timeout ID, using the **setTimeoutId** method
   - The timeout destination, using the **setDestination** method
   - The timeout delay, using the **setDelay** method
3. The `unschedTm` method iterates through the heap and calls the **Delete** method to delete those timeouts whose destination is the specific `OMReactive`.
4. Finally, the method looks for matching timeouts in the matured list. It calls the `findInList` method to iterate over the matured list to find matching timeouts. When it finds one, it calls the **setId** method to set the timeout's ID to **OMCancelledEventId**, then removes it from the matured list.
5. If `id ==` **OMEventTimeoutId**, `unschedTm` cancels only that event.

Signature

```
RP_FRAMEWORK_DLL void unschedTm (short id,
   OMReactive *c);
```

Parameters

`id`

Specifies the ID tag of the timeout request.

If **OMEventAnyEventId** is specified, `unschedTm` cancels all events whose destination is this specific instance of `OMReactive`. If **OMEventTimeoutId** is specified, `unschedTm` cancels only that timeout.

`c`

Specifies a pointer to the `OMReactive` instance requestor. After a timeout has been canceled, this parameter points to the instance that should be notified.

Notes

- Canceling a timeout requires one of two actions:
  - Deleting the timeout from the heap.
  - Canceling it inside the event queue, if it is already dispatched. This in done by iterating the event queue.
- You can use `unschedTm` in cases where the statechart implementation is overridden.
- `unschedTm` is called by **unschedTm** (defined in `omthread.h`).

See Also

OMEventAnyEventId

Delete

setDelay

setDestination

setlId

setTimeoutId

OMEventTimeoutId

OMTimerManagerDefaults Class

`OMTimerManagerDefaults` defines default values for the tick interval (**defaultTicktime**) and the maximum number of time ticks before restarting the time tick count (**defaultMaxTM**).

This class is declared in the header file `oxf.h`.

Constant Summary

defaultMaxTM	Specifies the default for the maximum number of time ticks before restarting the time tick count
defaultTicktime	Specifies the default for the basic system tick interval, in milliseconds

Constants

defaultMaxTM

Specifies the default for the maximum number of time ticks before restarting the time tick count. It is used by the `maxTM` parameter in **OMTimerManager**, the constructor for the `OMTimerManager` class.

The default value is specified in `oxf.cpp` as follows:

```
 const unsigned OMTimerManagerDefaults::
    defaultMaxTM = 100;
static const unsigned defaultMaxTM;
```

defaultTicktime

Specifies the default for the basic system tick interval, in milliseconds. It is used by the `ticktime` parameter in **OMTimerManager**, the constructor for the `OMTimerManager` class.

The default value is specified in `oxf.cpp` as follows:

```
 const unsigned OMTimerManagerDefaults::
   defaultTicktime = 100;
static const unsigned defaultTicktime;
```

OMUAbstractContainer Class

The `OMAbstractContainer` class is the base class for abstract, typeless containers, based on the template (typed) classes. It includes the friend class `OMUIterator`, which provides a standard iterator for classes derived from `OMUAbstractContainer`. See **OMIterator Class** for more information on iteration methods.

This class is defined in the header file `omuabscon.h`.

Construction Summary

~OMUAbstractContainer	Destroys the `OMAbstractContainer` object

Method Summary

getCurrent	Gets the current element
getFirst	Gets the first element in the container
getNext	Gets the next element in the container

~OMUAbstractContainer

Visibility

`Public`

Description

The **~OMUAbstractContainer** destroys the `OMUAbstructContainer` object.

Signature

```
virtual ~OMUAbstractContainer()
```

getCurrent

Visibility

Public

Description

The getCurrent method gets the current element in the container.

Signature

```
virtual void* getCurrent(void* pos) const=0;
```

Parameters

pos

Specifies the current position

getFirst

Visibility

Public

Description

The getFirst method gets the first element in the container.

Signature

```
virtual void getFirst(void*& pos) const=0;
```

Parameters

pos

Specifies the first position in the container

getNext

Visibility

Public

Description

The getNext method gets the next element in the container.

Signature

```
virtual void getNext(void*& pos) const=0;
```

Parameters

pos

Specifies the next position in the container

OMUCollection Class

In Rhapsody, `omu*` containers are containers that are not implemented with templates. The use of template-free containers reduces the size of the generated code considerably. An `OMUCollection` is a typeless, dynamically sized array.

This class is defined in the header file `omucollec.h`.

Attribute Summary

count	Specifies the number of elements in the collection
theLink	Specifies the link to the element in the collection
size	Specifies the amount of memory allocated for the collection

Construction Summary

OMUCollection	Constructs an `OMUCollection` object
~OMUCollection	Destroys the `OMUCollection` object

Method Summary

operator []	Returns the element at the specified position
add	Adds the specified element to the collection
addAt	Adds the specified element to the collection at the given index
find	Looks for the specified element in the collection
getAt	Returns the element found at the specified index
getCount	Returns the number of elements in the collection
getCurrent	Is used by the iterator to get the element at the current position in the collection
getFirst	Is used by the iterator to get the first position in the collection
getNext	Is used by the iterator to get the next position in the collection
getSize	Gets the size of the memory allocated for the collection
isEmpty	Determines whether the collection is empty

remove	Deletes the specified element from the collection
removeAll	Deletes all the elements from the collection
removeByIndex	Deletes the element found at the specified index in the collection
reorganize	Reorganizes the contents of the collection
setAt	Inserts the specified element at the given index in the collection

Attributes

count

This attribute specifies the number of elements in the collection. It is defined as follows:

```
int count;
```

theLink

This attribute specifies the link to an element in the collection. It is defined as follows:

```
void** theLink;
```

size

This attribute specifies the amount of memory allocated for the collection. It is defined as follows

```
int size;
```

OMUCollection

Visibility

Public

Description

The OMUCollection method is the constructor for the `OMUCollection` class.

Signature

```
OMUCollection(int theSize=DefaultStartSize)
```

Parameters

`theSize`

The starting size. The default collection size is `20` elements.

See Also

~OMUCollection

~OMUCollection

Visibility

Public

Description

The ~OMUCollection method is the destructor for the `OMUCollection` class.

Signature

```
~OMUCollection()
```

See Also

OMUCollection

operator []

Visibility

`Public`

Description

The `[]` operator returns the element at the specified position.

Signatures

```
void * operator[](int i)
const void * operator[](int i) const
```

Parameters

`i`

The index of the element to return

Return

The element at the specified index, or NULL if you selected an out-of-range value

add

Visibility

```
Public
```

Description

The **add** method adds the specified element to the collection.

Signature

```
void add(void* p)
```

Parameters

```
p
```

The element to add

See Also

addAt

remove

removeAll

removeByIndex

addAt

Visibility

`Public`

Description

The **addAt** method adds the specified element to the collection at the given index.

Signature

```
int addAt(int index, void* p)
```

Parameters

`index`

The index at which to add the new element

`p`

The element to add

See Also

add

remove

removeAll

removeByIndex

find

Visibility

Public

Description

The **find** method looks for the specified element in the collection.

Signature

```
int find(void* p) const
```

Parameters

p

The element you want to find

Return

The method returns one of the following values:

- 0—The element was not found in the collection.
- 1—The element was found in the collection.

getAt

Visibility

Public

Description

The **getAt** method returns the element found at the specified index.

Signature

```
void* getAt (int i) const
```

Parameters

`i`

The index of the element to retrieve

Return

The element found at the specified location

getCount

Visibility

Public

Description

The **getCount** method returns the number of elements in the collection.

Signature

```
int getCount() const
```

Return

The number of elements in the collection

getCurrent

Visibility

`Public`

Description

The **getCurrent** method is used by the iterator to get the element at the current position in the collection.

Signature

```
void* getCurrent(void* pos) const
```

Parameters

`pos`

The position of the element to retrieve

Return

The element at the current position in the collection

getFirst

Visibility

`Public`

Description

The **getFirst** method is used by the iterator to get the first position in the collection.

Signature

```
void getFirst(void*& pos) const
```

Parameters

`pos`

The position of the element to retrieve

See Also

getNext

getNext

Visibility

`Public`

Description

The **getNext** method is used by the iterator to get the next position in the collection.

Signature

```
void getNext(void*& pos) const
```

Parameters

`pos`

The position of the element to retrieve

See Also

getFirst

getSize

Visibility

`Public`

Description

The **getSize** method gets the size of the memory allocated for the collection.

Signature

```
int getSize() const
```

Return

The size

isEmpty

Visibility

`Public`

Description

The **isEmpty** method determines whether the collection is empty.

Signature

```
int isEmpty() const
```

Return

The method returns one of the following values:

- `0`—The collection is not empty.
- `1`—The collection is empty.

remove

Visibility

`Public`

Description

The **remove** method deletes the specified element from the collection.

Signature

```
void remove(void* p);
```

Parameters

`p`

The element to delete

See Also

add

addAt

removeAll

removeByIndex

removeAll

Visibility

`Public`

Description

The **removeAll** method deletes all the elements from the collection.

Signature

```
void removeAll()
```

See Also

add

addAt

remove

removeByIndex

removeByIndex

Visibility

`Public`

Description

The **removeByIndex** method deletes the element found at the specified index in the collection.

Signature

```
void removeByIndex(int i)
```

Parameters

`i`

The index of the element to delete

See Also

add

addAt

remove

removeAll

reorganize

Visibility

`Public`

Description

The reorganize method enables you to reorganize the contents of the collection, and enlarge it if necessary.

Signature

```
void reorganize(int factor = DefaultFactor)
```

Parameters

`factor`

The growth factor. The default value is 2.

setAt

Visibility

`Public`

Description

The **setAt** method inserts the specified element at the given index in the collection.

Signature

```
int setAt(int index, const void* p)
```

Parameters

`index`

The index at which to add the new element

`p`

The element to add

Return

The method returns one of the following values:

- `0`—The method failed.
- `1`—The method was successful.

OMUIterator Class

The `OMUIterator` class provides a standard iterator for containers derived from `OMUAbstructContainer`.

This class is defined in the header file `omuabscon.h`.

Construction Summary

OMUIterator	Constructs an `OMUIterator` object

Method Summary

operator *	Returns the current value of the iterator
operator ++	Increments the iterator
reset	Resets the iterator to the first position in the container
value	Returns the current value of the iterator

OMUIterator

Visibility

Public

Description

The OMUIterator method is the constructor for the `OMUIterator` class.

Signatures

```
OMUIterator();

OMUIterator(const OMUAbstractContainer& l)

OMUIterator(const OMUAbstractContainer* l)
```

Parameters

`l`

The container the iterator will visit

operator *

Visibility

Public

Description

The `*` operator returns the current value of the iterator.

Signature

```
void* operator*()
```

Return

The current value of the iterator

operator ++

Visibility

`Public`

Description

The ++ operator increments the iterator.

The first signature defines the ++ operator used for "`++i`" usage; the second signature is used for "`i++`".

Signatures

```
OMUIterator& operator++()        //prefix

OMUIterator operator++(int i)    //postfix
```

Parameters

`i`

Dummy parameter

reset

Visibility

`Public`

Description

The **reset** method resets the iterator tp the first position in the container.

Signatures

```
void reset()
void reset(OMUAbstractContainer& newLink)
```

Parameters

`newLink`

The new position

value

Visibility

Public

Description

The **value** method returns the current value of the iterator.

Signature

```
void* value()
```

OMUList Class

In Rhapsody, omu* containers are containers that are not implemented with templates. The use of template-free containers reduces the size of the generated code considerably. An OMUList is a typeless, linked list.

This class is defined in the header file omulist.h.

Construction Summary

OMUList	Constructs an OMUList object
~OMUList	Destroys the OMUList object

Flag Summary

first	Specifies the first element in the list
last	Specifies the last element in the list

Method Summary

operator []	Returns the element at the specified position.
add	Adds the specified element to the end of the list
addAt	Adds the specified element to the list at the given index
addFirst	Adds an element to the beginning of the list
find	Looks for the specified element in the list
getAt	Returns the element found at the specified index
getCount	Returns the number of elements in the list
getCurrent	Is used by the iterator to get the element at the current position in the list
getFirst	Is used by the iterator to get the first position in the list
getNext	Is used by the iterator to get the next position in the list
isEmpty	Determines whether the list is empty
_removeFirst	Removes the first item from the list

remove	Deletes the first occurrence of the specified element from the list
removeAll	Deletes all the elements from the list
removeFirst	Deletes the first element from the list
removeItem	Deletes the specified element from the list
removeLast	Deletes the last element from the list

Flags

first

Specifies the first element in the list. It is defined as follows:

```
OMUListItem* first;
```

last

Specifies the last element in the list. It is defined as follows:

```
OMUListItem* last;
```

Example

Consider the following example:

```
OMUIterator iter(itsObserver);
while (*iter)
{
     (static_cast<Observer*>(*iter))->notify();
     iter++;
}
```

OMUList

Visibility

`Public`

Description

The **OMUList** method is the constructor for the `OMUList` class. The method creates an empty list.

Signature

```
OMUList()
```

See Also

~OMUList

~OMUList

Visibility

`Public`

Description

The **~OMUList** method empties the list.

Signature

```
virtual ~OMUList()
```

See Also

OMUList

operator []

Visibility

`Public`

Description

The `[]` operator returns the element at the specified position.

Signature

```
void * operator[](int i) const
```

Parameters

`i`

The index of the element to return

add

Visibility

`Public`

Description

The **add** method adds the specified element to the end of the list.

Signature

```
void add(void *p)
```

Parameters

`p`

The element to add to the list

See Also

addAt

addFirst

remove

removeAll

removeFirst

removeLast

addAt

Visibility

`Public`

Description

The **addAt** method adds the specified element to the list at the given index.

Signature

```
void addAt(int i, void* p)
```

Parameters

`i`

The list index at which to add the element

`p`

The element to add

See Also

add

addFirst

remove

removeAll

removeFirst

removeLast

addFirst

Visibility

Public

Description

The **addFirst** method adds an element to the beginning of the list.

Signature

```
void addFirst(void *p)
```

Parameters

p

The element to add to the beginning of the list

See Also

add

addAt

remove

removeAll

removeFirst

removeLast

find

Visibility

`Public`

Description

The find method looks for the specified element in the list.

Signature

```
int find(const void* p) const
```

Parameters

`p`

The element you want to find

Return

The method returns one of the following values:

- `0`—The element was not found in the list.
- `1`—The element was found in the list.

getAt

Visibility

Public

Description

The **getAt** method returns the element found at the specified index.

Signature

```
void* getAt (int i) const
```

Parameters

`i`

The index of the element to retrieve

See Also

getCount

getCurrent

getFirst

getNext

getCount

Visibility

`Public`

Description

The **getCount** method returns the number of elements in the list.

Signature

```
int getCount() const
```

Return

The number of elements in the list

getCurrent

Visibility

`Public`

Description

The **getCurrent** method is used by the iterator to get the element at the current position in the list.

Signature

```
virtual void* getCurrent(void* pos) const
```

Parameters

`pos`

The position of the element you want to retrieve

getFirst

Visibility

Public

Description

The getFirst method is used by the iterator to get the first position in the list.

Signature

```
virtual void getFirst(void*& pos) const
```

Parameters

pos

The position

See Also

getNext

getNext

Visibility

Public

Description

The getNext method is used by the iterator to get the next position in the list.

Signature

```
virtual void getNext(void*& pos) const
```

Parameters

pos

The position

See Also

getFirst

isEmpty

Visibility

`Public`

Description

The **isEmpty** method determines whether the list is empty.

Signature

```
int isEmpty() const
```

Return

The method returns one of the following values:

- `0`—The list is not empty.
- `1`—The list is empty.

_removeFirst

Visibility

`Public`

Description

The **_removeFirst** method removes the first item from the list.

Note

It is safer to use the method **removeFirst** because that method has more checks than **_removeFirst**.

Signature

```
inline void _removeFirst()
```

See Also

removeFirst

remove

Visibility

`Public`

Description

The **remove** method deletes the first occurrence of the specified element from the list.

Signature

```
void remove(const void* p)
```

Parameters

`p`

The element to delete

See Also

add

addAt

removeAll

removeFirst

removeLast

removeAll

Visibility

`Public`

Description

The **removeAll** method deletes all the elements from the list.

Signature

```
void removeAll()
```

See Also

add

addAt

remove

removeFirst

removeLast

removeFirst

Visibility

`Public`

Description

The **removeFirst** method deletes the first element from the list.

Signature

```
void removeFirst()
```

See Also

add

addAt

remove

removeAll

removeLast

removeItem

Visibility

Public

Description

The **removeItem** method deletes the specified element from the list.

Signature

```
void removeItem(const OMUListItem* item)
```

Parameters

`item`

The element to delete

See Also

add

addAt

remove

removeAll

removeLast

removeLast

Visibility

`Public`

Description

The **removeLast** method deletes the last element from the list.

Note

This method is not efficient because the Rhapsody framework does not keep backward pointers. It is preferable to use one of the other `remove` functions to delete elements from the list.

Signature

```
void removeLast()
```

See Also

add

addAt

remove

removeAll

removeItem

OMUListItem Class

The `OMUListItem` class is a helper class for `OMUList` that contains functions that enable you to manipulate list elements.

This class is defined in the header file `omulist.h`.

Construction Summary

OMUListItem	Constructs an `OMUListItem` object

Method Summary

connectTo	Connects to the specified item in the list
getElement	Gets the list element
getNext	Gets the next item in the list
setElement	Sets the specified list element

OMUListItem

Visibility

`Public`

Description

The **OMUListItem** method is the constructor for the `OMUListItem` class.

Signature

```
OMUListItem(void* theElement)
```

Parameters

`theElement`

The new list element

connectTo

Visibility

Public

Description

The **connectTo** method connects to the specified item in the list.

Signature

```
void connectTo(OMUListItem* item)
```

Parameters

`item`

The item to connect to

getElement

Visibility

Public

Description

The **getElement** method gets the list element.

Signature

```
void* getElement() const
```

getNext

Visibility

Public

Description

The **getNext** method gets the next item in the list.

Signature

```
OMUListItem* getNext() const
```

Return

The next item in the list

setElement

Visibility

`Public`

Description

The **setElement** method sets the specified list element.

Signature

```
void setElement(void* p)
```

Parameters

`p`

The list element to set

OMUMap Class

In Rhapsody, `omu*` containers are containers that are not implemented with templates. The use of template-free containers reduces the size of the generated code considerably. An `OMUMap` is a typeless map.

This class is defined in the header file `omumap.h`.

Construction Summary

OMUMap	Constructs an `OMUMap` object
~OMUMap	Destroys the `OMUMap` object

Method Summary

operator []	Returns the element found at the specified location
add	Adds an element to the map
find	Determines whether the specified element is in the map
getAt	Returns the element for the specified key
getCount	Returns the number of elements in the map
getKey	Gets the element for the specified key
isEmpty	Determines whether the map is empty
lookUp	Looks for the specified element in the map
remove	Deletes the specified element from the map
removeAll	Deletes all the elements from the map
removeKey	Deletes the element from the map, given its key

OMUMap

Visibility

`Public`

Description

The **OMUMap** method is the constructor for the `OMUMap` class.

Signature

```
OMUMap()
```

See Also

~OMUMap

~OMUMap

Visibility

`Public`

Description

The **~OMUMap** method destroys the `OMUMap` object.

Signature

```
~OMUMap()
```

See Also

OMMap

operator []

Visibility

`Public`

Description

The `[]` operator returns the element at the specified key.

Signature

```
void* operator[](void* theKey) const
```

Parameters

`theKey`

The key of the element to get

Return

The element at the specified key

add

Visibility

`Public`

Description

The **add** method adds the specified element to the given key.

Signature

```
void add(void* theKey, void* p);
```

Parameters

`theKey`

The map key to which to add the element

`p`

The element to add to the key

See Also

remove

removeAll

removeKey

find

Visibility

`Public`

Description

The **find** method determines whether the specified element is in the map.

Signature

```
int find(void* p) const
```

Parameters

`p`

The element to look for

Return

The method returns one of the following values:

- `0`—The element was not found in the map.
- `1`—The element was found.

getAt

Visibility

`Public`

Description

The **getAt** method returns the element for the specified key.

Signature

```
void* getAt(const void* theKey) const
```

Parameters

`theKey`

The key for the element to get

getCount

Visibility

`Public`

Description

The **getCount** method returns the number of elements in the map.

Signature

```
int getCount() const
```

Return

The number of elements in the map

getKey

Visibility

`Public`

Description

The **getKey** method gets the element for the specified key.

Signature

```
void* getKey(const void* theKey) const
```

Parameters

`theKey`

The map key whose element you want

isEmpty

Visibility

`Public`

Description

The **isEmpty** method determines whether the map is empty.

Signature

```
int isEmpty() const
```

Return

The method returns one of the following values:

- `0`—The map is not empty.
- `1`—The map is empty.

lookUp

Visibility

`Public`

Description

The **lookUp** method finds the specified element in the map, given its key. If the element is found, the method places the contents of the element referenced by the key in the `element` parameter, and returns the value 1.

Signature

```
int lookUp(const void* theKey, void*& element) const
```

Parameters

`theKey`

The map key

`element`

The element to look up

Return

The method returns one of the following values:

- `0`—The element was not found in the map.
- `1`—The element was found.

remove

Visibility

`Public`

Description

The **remove** method deletes the specified element from the map.

Signature

```
void remove(void* p)
```

Parameters

`p`

The element to delete

See Also

add

removeAll

removeKey

removeAll

Visibility

Public

Description

The **removeAll** method deletes all the elements from the map.

Signature

```
void removeAll()
```

See Also

add

remove

removeKey

removeKey

Visibility

Public

Description

The **removeKey** method deletes the element from the map, given its key.

Signature

```
void removeKey(void* theKey)
```

Parameters

theKey

The key for the element to delete

See Also

add

remove

removeAll

OMUMapItem Class

The `OMUMapItem` class is a helper class for `OMUMap` that contains functions that enable you to manipulate map elements.

This class is defined in the header file `omumap.h`.

Construction Summary

OMUMapItem	Constructs an `OMUMapItem` object
~OMUMapItem	Destroys the `OMUMapItem` object

Method Summary

getElement	Returns the current element

OMUMapItem

Visibility

Public

Description

The OMUMapItem method is the constructor for the `OMUMapItem` class.

Signature

```
OMUMapItem(void* theKey, void* theElement)
```

Parameters

`theKey`

The map key

`theElement`

The new map element

See Also

~OMUMapItem

~OMUMapItem

Visibility

`Public`

Description

The ~OMUMapItem method destroys the `OMUMapItem` object.

Signature

```
virtual ~OMUMapItem()
```

See Also

OMMapItem

getElement

Visibility

`Public`

Description

The getElement method returns the current element.

Signature

```
void* getElement()
```

Return

The current element

OXF Class

The `oxf.h` file defines general API classes used by the execution framework.

Method Summary

animDeregisterForeignThread	Unregisters the external thread
animRegisterForeignThread	Registers an external thread (not an `OMThread`) in the animation framework
delay	Delays the calling thread for the specified length of time
end	Ends the event processing of the default event dispatching thread
getMemoryManager	Returns the current framework memory manager
getTheDefaultActiveClass	Returns the default active class
getTheTickTimerFactory	Returns the low-level timer factory
init	Initializes the timer, creates the default event dispatching thread, and initializes the framework
setMemoryManager	Specifies the current framework memory manager
setTheDefaultActiveClass	Registers an alternate default active object on the framework
setTheTickTimerFactory	Registers a timer factory on the framework, causing the framework to use the user-defined timers instead of the predefined timers
start	Starts the event processing of the default event dispatching thread

animDeregisterForeignThread

Visibility

`Public`

Description

The animDeregisterForeignThread method unregisters the external thread.

Signature

```
static void animDeregisterForeignThread(void* theHandle);
```

Parameters

`theHandle`

Specifies the handle to the external thread to unregister

See Also

animRegisterForeignThread

animRegisterForeignThread

Visibility

`Public`

Description

The **animRegisterForeignThread** method registers an external thread (not an `OMThread`) in the animation framework.

Signature

```
static void animRegisterForeignThread(char * name,
   void* theHandle);
```

Parameters

`name`

Specifies the name of the external thread

`theHandle`

Specifies the handle to the thread

See Also

animDeregisterForeignThread

delay

Visibility

`Public`

Description

The delay method delays the calling thread for the specified length of time.

Signature

```
static void delay (timeUnit t);
```

Parameters

`t`

Specifies the delay, in milliseconds

end

Visibility

`Public`

Description

The end method closes the framework-dependent parts in the application, without closing the application.

This method was added to support Microsoft COM technology, and is fully implemented for Microsoft adapters only.

Signature

```
static void end();
```

See Also

init

start

getMemoryManager

Visibility

`Public`

Description

The **getMemoryManager** method returns the current framework memory manager.

Signature

```
static OMAbstractMemoryAllocator* getMemoryManager()
```

Return

The framework memory manager

See Also

setMemoryManager

getTheDefaultActiveClass

Visibility

`Public`

Description

The **getTheDefaultActiveClass** method returns the default active class.

Signature

```
static OMThread* getTheDefaultActiveClass()
```

Return

The default active class

See Also

setTheDefaultActiveClass

getTheTickTimerFactory

Visibility

Public

Description

The **getTheTickTimerFactory** method returns the low-level timer factory.

Signature

```
static const OMAbstractTickTimerFactory*
   getTheTickTimerFactory()
```

Return

theTickTimerFactory

See Also

setTheTickTimerFactory

init

Visibility

Public

Description

In instrumented code, init initializes the framework instances that need to be available for the application built on top of the framework.

This method must be called before any other framework-related code is executed.

Note

You must call `OXF::init()` in a DLL even if the application loading the DLL has called `OXF::init()`; otherwise, there will be a leak in the state machine thread handle.

Signature

```
static int init (
   int numProgArgs = 0,
   char **progArgs = NULL,
   unsigned int defaultPort = 0,
   const char* defaultHost = NULL,
   unsigned ticktime =
      OMTimerManagerDefaults::defaultTicktime,
   unsigned maxTM =
      OMTimerManagerDefaults::defaultMaxTM,
   OMBoolean isRealTimeModel = TRUE);
```

Parameters

`numProgArgs`

Specifies the number of program arguments.

`progArgs`

Specifies the list of program arguments.

`defaultPort`

Is an animation-specific parameter that specifies the port used for communicating with the animation server.

If you are using an animation port other than 6423 (the default value), this number must match that assigned to the `AnimationPortNumber` variable in your `rhapsody.ini` file.

`defaultHost`

Is an animation-specific parameter that specifies the default host name of the machine on which Rhapsody is running.

`tickTime`

Specifies the basic system tick in milliseconds. Every ticktime, the framework timeout manager checks for expired timeouts. The default ticktime is every `100` milliseconds.

You can override the default tick time by setting the `<lang>_CG::Framework::TimerResolution` property.

`maxTM`

Specifies the maximum number of timeouts (set or matured) that can coexist in the application. The default value is `100` timeouts.

You can override the default maximum number of timeouts by setting the `<lang>_CG::Framework::TimerMaxTimeouts` property.

`isRealTimeModel`

Specifies whether the model runs in real time (the default) or simulated time. The default value is real time.

`OMTimerManager` can implement two time models:

- **real time**—Time advances according to the actual underlying operating system clock.
- **simulated time**—Time advances either explicitly, by calling the **consumeTime** method or implicitly, when all reactive objects are idle (that is, they do not have an event in their event queue) and there is at least one pending timeout.

Simulated time is useful for debugging and algorithm validation.

setMemoryManager

Visibility

`Public`

Description

The **setMemoryManager** method specifies the current framework memory manager. It controls memory allocated in the framework at the application level (for example, when adding an object to a relation implemented as `OMList`). If you do not register a memory manager, the framework uses the global `new` and `delete` operators.

To have an effect, call this method before making any memory allocation requests, or compile the framework with the `OM_ENABLE_MEMORY_MANAGER_SWITCH` compiler flag.

Signature

```
static OMBoolean setMemoryManager(
   OMAbstractMemoryAllocator* const memoryManager);
```

Parameters

`memoryManager`

Specifies the new framework memory manager

Return

The method returns `TRUE` if the memory manager was set successfully. Otherwise, it returns `FALSE`.

See Also

getMemoryManager

setTheDefaultActiveClass

Visibility

Public

Description

The **setTheDefaultActiveClass** method registers an alternate default active object instead of the `OMMainThread` singleton. This is useful when you customize the behavior of application active classes.

To have an effect, the user factory must be registered before the framework initialization (`OXF::`**init**) and before any request of the default active class is made.

Signature

```
static OMBoolean setTheDefaultActiveClass (OMThread* t);
```

Parameters

`t`

Specifies the new default active class

Return

The method returns `TRUE` if the active object was set successfully. Otherwise, it returns `FALSE`.

See Also

getTheDefaultActiveClass

init

setTheTickTimerFactory

Visibility

```
Public
```

Description

The **setTheTickTimerFactory** registers a timer factory on the framework, causing the framework to use the user-defined timers instead of the predefined timers. You can register a timer factory that does not create any timers, causing the timing mechanisms of the framework to be disabled. For example:

```
disable tm()
```

To have an effect, the user factory must be registered before the framework initialization (`OXF::`**init**).

Note

You can set the low-level timer factory only once for the entire lifetime of the application.

Signature

```
static OMBoolean setTheTickTimerFactory(
   const OMAbstractTickTimerFactory* factory);
```

Parameters

```
factory
```

Specifies the new low-level timer factory

Return

The method returns `TRUE` if the active object was set successfully. Otherwise, it returns `FALSE`.

See Also

getTheTickTimerFactory

init

start

Visibility

`Public`

Description

The **start** method starts the event processing of the active class (by default, the `OMMainThread` singleton). The `doFork` parameter determines whether the current thread (the caller of **init**) is the default event dispatching thread or a new, separate thread. If `doFork` is `FALSE`, `OXF::`**start** will not return, unless the default active class is destroyed.

`OXF::`**start** does not return in the generated application (this can be controlled via a Rhapsody property). Even if all statecharts terminate, it still runs. This is because the framework was specifically written for embedded applications, which generally do not end. Use Ctrl+C to kill the application.

Signature

```
static void start(int doFork = FALSE);
```

Parameters

`doFork`

Determines whether the current thread (the caller of **init**) is the default event dispatching thread or a separate thread. If `doFork` is `TRUE`, the control returns to the caller; otherwise, control remains in `OXF::`**start** for the lifetime of the application.

The syntax is as follows:

```
int doFork = FALSE
```

This parameter is useful in environments such as MS Windows, where the root thread has its own “agenda” (for example, GUI processing).

Quick Reference

This section lists the framework methods, macros, and operators, and provides a brief description of each. For ease of use, the methods are presented in alphabetical order.

Method Name	Description
operator *	Returns the current value of the iterator. *or* Is a customizable operator.
operator ++	Increments the iterator.
operator []	Returns the element at the specified location.
`operator +`	Adds a string.
`operator +=`	Adds to the existing string.
`operator =`	Sets a string.
`operator ==`	Determines whether two objects are equal.
`operator >=`	Determines whether the first object is greater than or equal to the second.
`operator <=`	Determines whether the first object is less than or equal to the second.
`operator !=`	Determines whether the first object is not equal to the second.
`operator >`	Determines whether the first object is greater than the second.
`operator <`	Determines whether the first object is less than the second.
`operator <<`	Compares an iostream and a string.
`operator >>`	Compares an iostream and a string.
_gen	Queues events sent to the reactive object.
_removeFirst	Removes the first item from the list.
~OMAbstractMemoryAllocator	Is the destructor for the `OMAbstractMemoryAllocator` class.
~OMCollection	Is the destructor for the `OMCollection` class.
~OMDelay	Is the destructor for the `OMDelay` class.
~OMEvent	Is the destructor for the `OMEvent` class.
~OMGuard	Is the destructor for the `OMGuard` class.

Method Name	Description
~OMHeap	Is the destructor for the `OMHeap` class.
~OMList	Is the destructor for the `OMList` class.
~OMMainThread	Is the destructor for the `OMMainThread` class.
~OMMap	Is the destructor for the `OMMap` class.
~OMMapItem	Is the destructor for the `OMMapItem` class.
~OMMemoryManager	Is the destructor for the `OMMemoryManager` class.
~OMMemoryManagerSwitchHelper	Is the destructor for the `OMMemoryManagerSwitchHelper` class.
~OMProtected	Is the destructor for the `OMProtected` class.
~OMQueue	Is the destructor for the `OMQueue` class.
~OMReactive	Is the destructor for the `OMReactive` class.
~OMStack	Is the destructor for the `OMStack` class.
~OMStaticArray	Is the destructor for the `OMStaticArray` class.
~OMString	Is the destructor for the `OMString` class.
~OMThread	Is the destructor for the `OMThread` class.
~OMThreadTimer	Is the destructor for the `OMThreadTimer` class.
~OMTimerManager	Is the destructor for the `OMTimerManager` class.
~OMTimeout	Is the destructor for the `OMTimeout` class.
~OMUAbstractContainer	Is the destructor for the `OMUAbstractContainer` class.
~OMUCollection	Is the destructor for the `OMUCollection` class.
~OMUList	Is the destructor for the `OMUList` class.
~OMUMap	Is the destructor for the `OMUMap` class.
~OMUMapItem	Is the destructor for the `OMUMapItem` class.
action	Sends a matured timeout request to the relevant thread, where it is then inserted into the thread's event queue.
add	Adds the specified element to the container.
addAt	Adds the specified element to the collection at the given index.
addFirst	Adds an element at the beginning of the list.
allocPool	Allocates a memory pool big enough to hold the specified number of instances.
allowDeleteInThreadsCleanup	Postpones the destruction of a framework thread until the application terminates and all user threads are deleted.
animDeregisterForeignThread	Unregisters the external thread.
animRegisterForeignThread	Registers an external thread (not an `OMThread`) in the animation framework.
callMemoryPoolIsEmpty	Controls the overprint of the message displayed when the pool is out of memory.

Method Name	Description
cancelEvent	Marks a single event as canceled (that is, it changes the event's ID to OMCancelledEventId).
cancelEvents	Cancels all the queued events for the reactive object.
cbkBridge	Is a bridge to get an interrupt from the operating system via the `timeTickCbk` (private) method.
cleanup	Cleans up the allocated memory list.
cleanupAllThreads	"Kills" all threads in an application except for the main thread and the thread running the `cleanupAllThreads` method.
cleanupThread	Provides a "hook" to allow a thread to be cleaned up without a call to the `DTOR`.
clearInstance	Cleans up the singleton instance of the timer manager.
CompareNoCase	Performs a case-insensitive comparison of two strings.
connectTo	Connects the list item to the list.
consumeEvent	Is the main event consumption method.
consumeTime	Is used in simulated time mode to simulate time consumption.
createRealTimeTimer	Creates a real-time timer.
createSimulatedTimeTimer	Creates a simulated-time timer.
cserialize	Passes the values of the instance to a string, which is then sent to Rhapsody. It is part of the animation mechanism.
decNonIdleThreadCounter	Decreases the `nonIdleThreadCounter` private attribute.
delay	Delays the calling thread for the specified length of time.
Delete	Deletes an event instance (releases the memory used by an event), or deletes a timeout from the heap.
deleteMutex	Deletes the mutex and sets its value to `NULL`.
destroyThread	Destroys the thread, or destroys the default active class or object for the framework.
destroyTimer	Cleans up the timer manager singleton instance.
discarnateTimeout	Destroys a timeout object for the reactive object.
doBusy	Sets the value of omrStatus to 1 or `TRUE`.
doExecute	Is the entry point to the thread main loop function.
Empty	Empties the string.
end	Ends the event processing of the default event dispatching thread.
entDef	Specifies the operation called when the state is entered from a default transition.
`enterState`	Specifies the state entry action.
`entHist`	Enters a history connector.
execute	Is the thread main loop function.

Method Name	Description
exitState	Specifies the state exit action.
find	Looks for the specified element in the container.
findMemory	Searches for a recorded memory allocation.
free	Is provided for backward compatibility. It calls the `unlock` method.
gen	Is used by a sender object to send an event to a receiver object.
get	Gets the current element in the queue.
getAOMThread	Is used by the framework for animation purposes.
getAt	Returns the element found at the specified index.
GetBuffer	Returns the string buffer.
getConcept	Gets the current concept.
getCount	Returns the number of elements in the container.
getCurrent	Is used by the iterator to get the element at the current position in the list.
getCurrentEvent	Gets the currently processed event.
getDefaultMemoryManager	Returns the default memory manager.
getDelay	Returns the current value of `delayTime`.
getDestination	Returns the reactive destination of the event.
getDueTime	Returns the due time of a timeout request stored in the heap.
getElapsedTime	Returns the value of `m_Time`, the current system time.
getElement	Gets the list element.
getEventClass	Returns the event class.
getEventQueue	Is used by the framework for animation purposes.
getFirst	Is used by the iterator to get the first position in the container.
getFirstConcept	Returns the first `Concept` element in the list.
getGuard	Gets the reference to the `OMProtected` part.
getHandle	Gets the handle.
getInverseQueue	Returns the element that will be returned by `get()` in the tail of the queue.
getKey	Gets the element for the specified key.
getLast	Is used by the iterator to get the last position in the list.
getLastConcept	Returns the last `Concept` element in the list.
getLastState	Gets the last state.
GetLength	Returns the length of the string.
getlId	Returns the event ID.
getMemory	Records the memory allocated by the default manager.

Method Name	Description
getMemoryManager	Returns the current memory manager.
getNext	Gets the next item in the container.
getOsHandle	Returns the thread's operating system ID.
getOSThreadEndClb	Requests a callback to end the current operating system thread.
getQueue	Returns the element that will be returned by `get()` in the head of the queue.
getSize	Returns the size of the memory allocated for the container.
getStepper	Is used by the framework for animation purposes.
getSubState	Returns the substate.
getTheDefaultActiveClass	Returns the default active class.
getTheTickTimerFactory	Returns the low-level timer factory.
getThread	Retrieves the thread associated with a reactive object.
getTimeoutId	Returns the current value for `timeoutId.`
goNextAndPost	Is used in simulated time mode.
handleEventNotConsumed	Is called when an event is not consumed by the reactive class.
handleEventNotConsumed	Is called when a triggered operation is not consumed by the reactive class.
`in`	Returns `TRUE` when the owner class is in this state.
incarnateTimeout	Creates a timeout object to be invoked on the reactive object.
incNonIdleThreadCounter	Increases the `nonIdleThreadCounter` private attribute.
increaseHead_	Increases the size of the queue head.
increaseTail_	Increases the size of the queue tail.
increment	Increments the iterator by 1.
`init`	Starts the timer ticking. Initializes the timer, creates the default event dispatching thread, and initializes the framework.
initializeMutex	Creates an RTOS mutex, if it has not been created already.
initiatePool	Initiates the "bookkeeping" for the allocated pool.
initInstance	Creates an instance of `OMThreadTimer`.
inNullConfig	Determines whether an `OMReactive` instance should take null transitions (transitions without triggers) in the state machine.

Method Name	Description
instance	Creates and retrieves the singleton instance of `OMMainThread`. *or* Returns the singleton instance of the `OMMemoryManagerSwitchHelper`.
isActive	Determines whether a reactive object is also an active object.
isBusy	Returns the current value of the **omrStatus** attribute.
isCancelledTimeout	Determines whether the event is canceled.
isCompleted	Determines whether the `OR` state reached a final state, and therefore can be exited on a null transition.
isCurrentEvent	Determines whether the specified ID is the currently processed event.
isDeleteAfterConsume	Returns `TRUE` if the event should be deleted by the event dispatcher (`OMThread`) after its consumption.
IsEmpty	Determines whether the string is empty.
isEmpty	Determines whether the container is empty.
isFrameworkEvent	Returns `TRUE` if the event is an internal framework event.
isFrameworkInstance	Determines the current value of the **frameworkInstance** attribute.
isFull	Determines whether the queue is full.
isInDtor	Determines whether event dispatching should be stopped.
isLogEmpty	Determines whether the memory log is empty.
isNotDelay	Determines whether a timeout event is a timeout delay.
isRealEvent	Returns `TRUE` if the event is a null-transition event, a timeout, or a user event.
isTimeout	Returns `TRUE` if the event is a timeout.
isTypeOf	Returns `TRUE` if the event is from a given type (has the specified ID).
isValid	Makes sure the reactive class is not deleted.
lock	Locks the mutex of the `OMProtected` object.
`lookUp`	Looks for the specified element in the map.
new	Allocates additional memory.
notifyToError	Writes messages to the error log.
notifyToOutput	Writes messages to standard output.
OMAndState	Is the constructor for the `OMAndState` class.
OMCollection	Is the constructor for the `OMCollection` class.
OMComponentState	Is the constructor for the `OMComponentState` class.
OMDelay	Is the constructor for the `OMDelay` class.

Method Name	Description
OMDestructiveString2X	Is used to support animation.
OMEvent	Is the constructor for the `OMEvent` class.
OMFinalState	Is the constructor for the `OMFinalState` class.
OMFriendStartBehaviorEvent	Is the constructor for the `OMFriendStartBehaviorEvent` class.
OMFriendTimeout	Is the constructor for the `OMFriendTimeout` class.
omGetEventQueue	Returns the event queue.
OMGuard	Is the constructor for the `OMGuard` class. It locks the mutex of the user object.
OMHeap	Is the constructor for the `OMHeap` class.
OMIterator	Is the constructor for the `OMIterator` class.
OMLeafState	Is the constructor for the `OMLeafState` class.
OMList	Is the constructor for the `OMList` class.
OMListItem	Is the constructor for the `OMListItem` class.
OMMap	Is the constructor for the `OMMap` class.
OMMapItem	Is the constructor for the `OMMapItem` class.
OMMemoryManager	Is the constructor for the `OMMemoryManager` class.
OMMemoryManagerSwitchHelper	Is the constructor for the `OMMemoryManagerSwitchHelper` class.
OMOrState	Is the constructor for the `OMOrState`.
OMProtected	Is the constructor for the `OMProtected` class.
OMQueue	Is the constructor for the `OMQueue` class.
OMReactive	Is the constructor for the `OMReactive` class.
OMSelfLinkedMemoryAllocator	Constructs the memory allocator, specifies whether it is protected, and how much additional memory should be allocated if the initial pool is exhausted.
OMStack	Is the constructor for the `OMStack` class.
OMState	Is the constructor for the `OMState` class.
OMStartBehaviorEvent	Is the constructor for the `OMStartBehavior` class.
OMStaticArray	Is the constructor for the `OMStatic` class.
OMString	Is the constructor for the `OMString` class.
OMThread	Is the constructor for the `OMThread` class.
OMTimeout	Is the constructor for the `Timeout` class.
OMTimerManager	Is the constructor for the `OMTimerManager` class.
OMUCollection	Is the constructor for the `OMUCollection` class.
OMUIterator	Is the constructor for the `OMUIterator` class.
OMUList	Is the constructor for the `OMUList` class.
OMUListItem	Is the constructor for the `OMUListItem` class.

Method Name	Description
OMUMap	Is the constructor for the `OMUMap` class.
OMUMapItem	Is the constructor for the `OMUMapItem` class.
pop	Pops an item off the stack.
popNullConfig	Decrements the **omrStatus** attribute after a null transition is taken.
push	Pushes an item onto the stack.
pushNullConfig	Counts null transitions. After a state is exited on a null transition, `pushNullConfig` increments the `omrStatus` attribute.
put	Adds an element to the queue.
queueEvent	Queues events to be processed by the thread event loop (**execute**).
recordMemoryAllocation	Records a single memory allocation.
recordMemoryDeallocation	Records a single memory deallocation.
registerWithOMReactive	Registers a user instance as a reactive class in the animation framework.
remove	Deletes the specified element from the container.
removeAll	Deletes all the elements from the container.
removeByIndex	Deletes the element found at the specified index in the container.
removeFirst	Deletes the first element from the list.
removeItem	Deletes the specified element from the list.
removeKey	Deletes the element from the map, given its key.
removeLast	Deletes the last element from the list.
reorganize	Enables you to reorganize the contents of the collection.
reset	Resets the iterator to the first position in the container.
resetSize	Makes the string larger.
resume	Resumes a thread or timer suspended by the `suspend` method.
returnMemory	Returns the memory from an instance.
rootState_dispatchEvent	Consumes an event inside a real statechart.
rootState_entDef	Initializes the statechart by taking the default transitions.
rootState_serializeStates	Is a virtual method that performs the actual event consumption.
runToCompletion	Takes all the null transitions (if any) that can be taken after an event has been consumed.
schedTm	Creates a timeout request and delegates the request to `OMTimerManager`.
serialize	Is called during animation to send event information.

Method Name	Description
serializeStates	Is called during animation to send state information.
set	Delegates a timeout request to OMTimerManager.
setAllocator	Sets the allocation method.
SetAt	Sets a character at the specified position in the container.
setAt	Inserts the specified element at the given index in the array.
setCompleteStartBehavior	Sets the value of the OMRShouldCompleteStartBehavior attribute.
SetDefaultBlock	Sets the default string size.
setDelay	Sets the value of Timeout.
setDeleteAfterConsume	Determines whether the event should be deleted by the event dispatcher (OMThread) after it is consumed.
setDestination	Sets the event reactive destination.
setDueTime	Specifies the value for the Timeout attribute.
setElapsedTime	Sets the value of m_Time, the current system time.
setElement	Sets the specified list element.
setEndOSThreadInDtor	Specifies whether an operating system thread in destruction should be deleted.
setEventGuard	Is used to set the event guard flag (m_eventGuard).
setFrameworkEvent	Sets the event to be considered as an internal framework event.
setFrameworkInstance	Changes the value of the frameworkInstance attribute.
setHandle	Sets the handle.
setIncrementNum	Overwrites the increment value.
setInDtor	Specifies that event dispatching should be stopped.
setlId	Sets the event ID.
setLastState	Sets the last state..
setMaxNullSteps	Sets the maximum number of null transitions (those without a trigger) that can be taken sequentially in the statechart.
setMemoryManager	Specifies the current framework memory manager.
setPriority	Sets the priority of the thread being executed.
setRelativeDueTime	Calculates and sets the due time for a timeout based on the current system time and the requested delay time.
setShouldDelete	Specifies whether a reactive object should be deleted by its active object when it reaches a termination connector in its state machine.
setShouldTerminate	Specifies that a reactive instance can be safely destroyed by its active instance.
setState	Is used by the framework to set the current state.
setSubState	Sets the substate.

Method Name	Description
setTheDefaultActiveClass	Registers an alternate default active object on the framework.
setTheTickTimerFactory	Registers a timer factory on the framework, causing the framework to use the user-defined timers instead of the predefined timers.
setThread	Is a mutator function that sets the thread of a reactive object.
setTimeoutId	Specifies the value for `timeoutId`.
setToGuardReactive	Specifies the value of the toGuardReactive attribute.
setToGuardThread	Sets the toGuardThread flag.
setUpdateState	Specifies whether the singleton should be updated.
shouldCompleteRun	Checks the value of omrStatus to determine whether there are null transitions to take.
shouldCompleteStartBehavior	Checks the start behavior state.
shouldDelete	Determines whether a reactive object should be deleted by its active object when it reaches a termination connector in its state machine.
shouldGuardThread	Determines whether the thread should be guarded.
shouldTerminate	Determines whether a reactive instance can be safely destroyed by its active instance.
shouldUpdate	Determines whether the singleton should be updated (and have new memory allocations recorded).
softUnschedTm	Removes a specific timeout from the matured list.
`start`	Starts the singleton event loop (`OMThread::execute`) of the main thread singleton. Starts the event processing of the default event dispatching thread.
startBehavior	Initializes the behavioral mechanism and takes the initial (default) transitions in the statechart before any events are processed.
stopAllThreads	Is used to support the DLL version of the Rhapsody in C++ execution framework (COM).
suspend	Suspends the thread or timer.
takeEvent	Takes the specified event off the event queue for processing. Is used by the event loop (within the thread) to make the reactive object process an event.
takeTrigger	Consumes a triggered operation event (synchronous event).
terminate	Sets the `OMReactive` instance to the terminate state (the statechart is entering a termination connector).
TimerManagerCallBack	Is a callback of the timer manager. which notifies the manager of the tick.

Method Name	Description
top	Deletes the top of the heap. *or* Moves the iterator to the first item in the stack.
trim	This method is currently unused.
undoBusy	Sets the value of the `sm_busy` attribute to 0 or `FALSE.`
unlock	Unlocks the mutex of the `OMProtected` object.
unschedTm	Cancels a timeout request.
update	Currently, this method is unused.
value	Returns the current value of the iterator.
wakeup	Resumes processing after the delay time has expired.

Index

Symbols

A

B

C

D

G

H

I

K

L

M

N

O

S

T

U

V

W