



Frameworks and Operating Systems Reference

**Rational Rhapsody
Frameworks and Operating Systems
Reference**



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 IBM® Rational® Rhapsody® 7.5 and to all subsequent releases and modifications until otherwise indicated in new editions.

© Copyright IBM Corporation 1997, 2009.

US Government Users Restricted Rights - Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Frameworks and Operating Systems	1
Real-Time Frameworks	1
Rational Rhapsody Statecharts	2
The Object Execution Framework (OXF)	3
Working with the Object Execution Framework	3
The OXF Library	4
Rational Rhapsody Applications and the RTOS	5
Operating System Abstraction Layer (OSAL)	5
Threads	7
Stack Size	7
Synchronization Services	8
Message Queues	8
Communication Port	9
Timer Service	10
Real-time Operating System (RTOS)	11
AbstractLayer Package (OSAL)	11
Classes	12
OSWrappers Package	12
Adapting Rational Rhapsody for a New RTOS	13
Run-Time Sources	13
Adding the New Adapter	13
Creating the Batch File and Makefiles	14
Sample <env>build.mak File	15
Creating New Makefiles	16
OXF Versions	16
Animation Libraries	16
Implementing the Adapter Classes	18
Modifying rawtypes.h	19
Other Operating System-Related Modifications	19
Building the Framework Libraries	20

Building the C or C++ Framework for Windows Systems	20
Building the Ada Framework	21
Building the Java Framework	22
Building the Framework for Solaris Systems	22
Creating Properties for a New RTOS	24
Modifying the site<lang>.prp Files	24
Setting the Environment	26
Configuring the OXF Properties for the C++ Framework	27
Validating the New Adapter	30
Modifying the Framework	31
Implementing the Abstract Factory	31
Plugging in the Factory	31
OSAL Methods	34
The OSAL Classes	37
Rational Rhapsody Developer for C	38
RiCOSConnectionPort Class	39
RiCOSEventFlag Interface	45
RiCOSMessageQueue Class	50
RiCOSMutex Class	59
RiCOSOXF Class	64
RiCOSSemaphore Class	67
RiCOSSocket Class	73
RiCOSTask Class	80
RiCOSTimer	93
RiCHandleCloser Class	97
Rational Rhapsody Developer for C++	98
OMEventQueue Class	98
OMMessageQueue Class	100
OMOS Class	100
OMOSConnectionPort Class	102
OMOSEventFlag Class	105
OMOSFactory Class	108
OMOSMessageQueue Class	117
OMOSMutex Class	123
OMOSSemaphore Class	126
OMOSSocket Class	129
OMOSThread Class	133
OMOSTimer Class	139
OMTMMessageQueue Class	140

Rebuilding the Rational Rhapsody Framework	145
Borland	145
INTEGRITY	146
Compiling and Building a Rational Rhapsody Sample	147
Downloading the Image and Running the Application	148
Linux	154
Building the Linux Libraries	154
Creating and Running Linux Applications	155
MultiWin32	156
Stepping Through the Generated Application Using MultiWin32	156
Stepping Through the OXF Using MULTI	158
OSE	159
Rebuilding the Framework	159
Using Command-Line Attributes and Flags	159
Editing the Batch Files	160
QNX	161
Using Momentics	162
Using ftp	162
Message Queue Implementation	163
VxWorks	164
Integrated Development Environment (IDE)	165
Defines	165
Structures	165
Makefiles	167
Creating a Make.bat File	167
Running the Batch File	167
Redefining Makefile-Related Properties	168
Redefining the MakeFileContent Property	169
Target Type	170
Compilation Flags	170
Commands Definitions	171
Generated Macros	172
Predefined Macros	173
Generated Dependencies	173
Makefile Linking Instructions	174
Java Users	175
Active Behavior Framework	177

Active and Reactive Classes	177
OMReactive Class	179
OMThread Class	181
OMMainThread Class	181
OMDelay Class	181
OMProtected Class	182
OMGuard Class	182
OMEvent Class	182
OMTimeout Class	183
OMTimerManager Class	183
Customizing Timeout Manager Behavior	184
OMThreadTimer Class	184
OMTimerManagerDefaults Class	184
Services Package	185
MemoryManagement Package	185
Containers Package	185
Event Handling	189
Events	189
Generating and Queuing an Event	190
Dispatching an Event	191
Canceling a Single Event	192
Canceling All Events to a Destination	192
Dispatching a Triggered Operation	193
Timeouts	194
Scheduling a Timeout	195
Dispatching a Timeout	196
Unscheduling a Timeout	197
Delaying a Timeout	197
Analyzing and Customizing	199
Model-level Debugging and Analysis	199
Customizing the Framework	201
The Rational Rhapsody Interrupt-Driven Framework (IDF)	203
Creating a Sample IDF Project	203
Adapting the Framework for a Specific Target	209

Limitations of the IDF	210
OXF Classes and Methods	211
OMAbstractMemoryAllocator Class	212
~OMAbstractMemoryAllocator	212
allocPool	213
callMemoryPoolsEmpty	213
getMemory	214
initiatePool	214
OMSelfLinkedMemoryAllocator	215
returnMemory	215
setAllocator	216
setIncrementNum	216
OMAbstractTickTimerFactory Class	217
createRealTimeTimer	217
createSimulatedTimeTimer	218
TimerManagerCallBack	219
OMAndState Class	219
OMAndState	219
lock	220
unlock	220
OMCollection Class	221
OMCollection	222
~OMCollection	222
add	223
addAt	223
remove	224
removeAll	225
removeByIndex	226
reorganize	226
OMComponentState Class	227
OMComponentState	228
enterState	228
in	228
takeEvent	229
OMDelay Class	230
OMDelay	231
~OMDelay	231
wakeup	232
OMEvent Class	233
Attributes	236

Constants	237
OMEvent	238
~OMEvent	239
Delete	240
getDestination	240
getIId	241
isCancelledTimeout	242
isDeleteAfterConsume	242
isFrameworkEvent	243
isRealEvent	244
isTimeout	244
isTypeOf	245
setDeleteAfterConsume	246
setDestination	247
setFrameworkEvent	247
setIId	248
OMFinalState Class	249
OMFinalState	250
getConcept	251
OMFriendStartBehaviorEvent Class	252
OMFriendStartBehaviorEvent	252
cserialize	253
getEventClass	253
serialize	254
OMFriendTimeout Class	255
OMFriendTimeout	255
cserialize	256
getEventClass	256
serialize	257
OMGuard Class	258
OMGuard	261
~OMGuard	261
getGuard	262
lock	262
unlock	262
OMHeap Class	263
OMHeap	264
~OMHeap	264
add	265
find	265
isEmpty	266
remove	266

top	267
trim	267
update	267
OMInfiniteLoop Class	268
OMIterator Class	268
OMIterator	269
operator *	269
operator ++	270
increment	270
reset	271
value	271
OMLeafState Class	272
OMLeafState	273
entDef	273
enterState	274
exitState	274
in	274
serializeStates	275
OMList Class	276
OMList	279
~OMList	279
operator []	280
add	280
addAt	281
addFirst	282
find	283
getAt	283
getCount	284
getCurrent	284
getFirst	285
getFirstConcept	285
getLast	286
getLastConcept	286
getNext	287
isEmpty	287
_removeFirst	288
remove	288
removeAll	289
removeFirst	289
removeItem	290
removeLast	291
OMListItem Class	292

OMListItem	292
connectTo	293
getNext	293
OMMainThread Class	294
~OMMainThread	295
destroyThread	295
instance	295
start	296
OMMap Class	297
OMMap	301
~OMMap	301
operator []	302
add	303
find	304
getAt	304
getCount	305
getKey	305
isEmpty	306
lookup	306
remove	307
removeAll	308
OMMapItem Class	309
OMMapItem	309
~OMMapItem	310
getConcept	310
OMMemoryManager Class	311
OMMemoryManager	314
~OMMemoryManager	314
getDefaultMemoryManager	315
getMemory	316
getMemoryManager	316
returnMemory	317
OMMemoryManagerSwitchHelper Class	318
OMMemoryManagerSwitchHelper	319
~OMMemoryManagerSwitchHelper	319
cleanup	320
findMemory	320
instance	321
isLogEmpty	321
recordMemoryAllocation	322
recordMemoryDeallocation	323
setUpdateState	324

shouldUpdate	324
OMNotifier Class	325
notifyToError	325
notifyToOutput	326
OMOrState Class	327
OMOrState	328
entDef	328
enterState	328
exitState	329
getSubState	329
in	330
serializeStates	330
setSubState	331
OMProtected Class	332
OMProtected	333
~OMProtected	334
deleteMutex	334
free	334
getGuard	335
initializeMutex	335
lock	336
unlock	336
OMQueue Class	337
OMQueue	341
~OMQueue	341
get	342
getCount	342
getInverseQueue	343
getQueue	343
getSize	344
increaseHead_	344
increaseTail_	344
isEmpty	345
isFull	345
put	346
OMReactive Class	347
OMReactive	357
~OMReactive	357
cancelEvents	358
consumeEvent	358
discarnateTimeout	360
doBusy	361

gen	361
_gen	364
getCurrentEvent	365
getThread	365
handleEventNotConsumed	366
handleTONotConsumed	367
incarnateTimeout	368
inNullConfig	369
isActive	370
isBusy	370
isCurrentEvent	371
isFrameworkInstance	372
isInDtor	373
isValid	373
popNullConfig	374
pushNullConfig	375
registerWithOMReactive	375
rootState_dispatchEvent	376
rootState_entDef	377
rootState_serializeStates	378
runToCompletion	379
serializeStates	379
setCompleteStartBehavior	380
setEventGuard	380
setFrameworkInstance	381
setInDtor	382
setMaxNullSteps	382
setShouldDelete	383
setShouldTerminate	384
setThread	385
setToGuardReactive	386
shouldCompleteRun	387
shouldCompleteStartBehavior	388
shouldDelete	389
shouldTerminate	390
startBehavior	391
takeEvent	392
takeTrigger	393
terminate	394
undoBusy	395
OMStack Class	396
OMStack	396
~OMStack	397
getCount	397

isEmpty.	398
pop.	398
push.	399
top.	399
OMStartBehaviorEvent Class.	400
Animating Start Behavior.	400
OMStartBehaviorEvent.	400
OMState Class.	401
OMState.	403
entDef.	403
entHist.	403
enterState.	404
exitState.	404
getConcept.	404
getHandle.	405
getLastState.	405
getSubState.	406
in.	406
isCompleted.	407
serializeStates.	408
setHandle.	408
setLastState.	409
setSubState.	409
takeEvent.	410
OMStaticArray Class.	411
OMStaticArray.	413
~OMStaticArray.	413
operator [].	414
add.	415
find.	415
getAt.	416
getCount.	416
getSize.	417
isEmpty.	417
removeAll.	418
setAt.	418
OMString Class.	419
OMString.	420
~OMString.	421
Operator[].	421
operator +.	422
operator +=.	423
operator =.	424

operator ==	425
operator >=	426
operator <=	427
operator !=	428
operator >	429
operator <	430
operator <<	431
operator >>	431
operator *	432
CompareNoCase	432
Empty	433
GetBuffer	433
GetLength	434
IsEmpty	434
OMDestructiveString2X	435
resetSize	435
SetAt	436
SetDefaultBlock	436
OMThread Class	437
OMThread	442
~OMThread	444
allowDeleteInThreadsCleanup	445
cancelEvent	445
cancelEvents	446
cleanupAllThreads	447
cleanupThread	447
destroyThread	448
doExecute	448
execute	449
getAOMThread	451
getEventQueue	451
getGuard	451
getOsHandle	452
getOSThreadEndClib	453
getStepper	454
lock	454
omGetEventQueue	454
queueEvent	455
resume	456
schedTm	456
setEndOSThreadInDtor	458
setPriority	459
setToGuardThread	459
shouldGuardThread	460

start	460
stopAllThreads	461
suspend	462
unlock	462
unschedTm	463
OMThreadTimer Class	465
~OMThreadTimer	466
action	466
initInstance	467
OMTimeout Class	469
OMTimeout	472
~OMTimeout	473
operator ==	473
operator >	474
operator <	475
Delete	476
getDelay	477
getDueTime	477
getTimeoutId	478
isNotDelay	479
new	479
setDelay	480
setDueTime	481
setRelativeDueTime	481
setState	482
setTimeoutId	483
OMTimerManager Class	484
OMTimerManager	487
~OMTimerManager	489
action	489
cbkBridge	490
clearInstance	490
consumeTime	491
decNonIdleThreadCounter	491
destroyTimer	492
getElapsedTime	492
goNextAndPost	493
incNonIdleThreadCounter	493
init	494
initInstance	494
instance	495
resume	496
set	497

setElapsedTime	498
softUnschedTm	499
suspend	499
unschedTm	500
OMTimerManagerDefaults Class	502
OMUAbstractContainer Class	503
~OMUAbstractContainer	503
getCurrent	504
getFirst	504
getNext	505
OMUCollection Class	506
OMUCollection	508
~OMUCollection	508
operator []	509
add	510
addAt	511
find	512
getAt	513
getCount	513
getCurrent	514
getFirst	514
getNext	515
getSize	515
isEmpty	516
remove	517
removeAll	518
removeByIndex	519
reorganize	520
setAt	521
OMUIterator Class	522
OMUIterator	523
operator *	523
operator ++	524
reset	524
value	525
OMUList Class	526
OMUList	528
~OMUList	528
operator []	529
add	530
addAt	531
addFirst	532

find	533
getAt	534
getCount	535
getCurrent	535
getFirst	536
getNext	536
isEmpty	537
_removeFirst	537
remove	538
removeAll	539
removeFirst	539
removeItem	540
removeLast	541
OMUListItem Class	542
OMUListItem	542
connectTo	543
getElement	543
getNext	543
setElement	544
OMUMap Class	545
OMUMap	546
~OMUMap	546
operator [].	547
add	548
find	549
getAt	549
getCount	550
getKey	550
isEmpty	551
lookUp	551
remove	552
removeAll	553
removeKey	553
OMUMapItem Class	554
OMUMapItem	554
~OMUMapItem	555
getElement	555
OXF Class	556
animDeregisterForeignThread	557
animRegisterForeignThread	558
delay	559
end	559
getMemoryManager	560

Table of Contents

getTheDefaultActiveClass	560
getTheTickTimerFactory	561
init	562
setMemoryManager	564
setTheDefaultActiveClass	565
setTheTickTimerFactory	566
start	567
Index	569

Frameworks and Operating Systems

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.

Real-Time Frameworks

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.

Rational Rhapsody Statecharts

Rational Rhapsody supports UML state machines as Rational Rhapsody *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 Rational 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 Rational 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.

The Object Execution Framework (OXF)

Rational 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 Rational Rhapsody is framework-based: it includes a fixed, predefined framework called the OXF (**O**bject **eX**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 multi-threaded, 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 OXF Library

Rational 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 Rational Rhapsody.

Note

For a list of the most relevant files in the directory `<install_dir>/Share/LangCpp/oxf`, see [Configuring the OXF Properties for the C++ Framework](#) 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, multi-threaded system. For more information, see [Active Behavior Framework](#).
- ◆ **Operating system package** (`OSLayer`)—Provides a thin abstraction layer through which the framework and generated code access operating system services. For more information, see [Operating System Abstraction Layer \(OSAL\)](#).
- ◆ **Services package** (`Services`)—Consists of two subpackages: `MemoryManagement` and `Containers`. For more information, see [Services Package](#).

Rational Rhapsody Applications and the RTOS

The deployment environment is the set of tools and third-party software required to develop and deploy a Rational Rhapsody-generated application in a particular hardware environment. The major components of the deployment environment are as follows:

- ◆ Real-time operating system (RTOS)
- ◆ Compiler
- ◆ Make facility

Rational Rhapsody generates implementation source code, in several high-level languages, that is RTOS-independent. This is achieved using a set of adapter classes known as the *Operating System abstraction layer* (OSAL), which is part of the Rational Rhapsody *object execution framework* (OXF). The OXF itself is operating system-independent, except for the OSAL, which serves as the only interface to the operating system and is the only operating system-dependent package within the OXF.

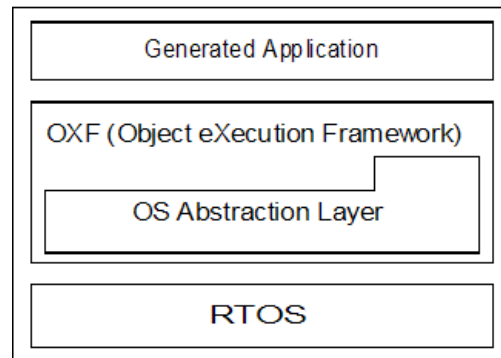
Each target environment requires a special OXF version. Preparing the OXF is primarily the process of providing an implementation for the OSAL. Each implementation of the OSAL for a particular target is known as an *adapter*.

Operating System Abstraction Layer (OSAL)

The *OSAL* consists of a set of interfaces (abstract classes) that provide all the required operating system services for the application, including:

- ◆ Tasking services
- ◆ Synchronization services
- ◆ Message queues
- ◆ Communication port
- ◆ Timer service

The OSAL separates the OXF from the underlying RTOS using the layered approach.



The OSAL supports each of these services by implementing thin wrappers around real operating system entities, adding minimal overhead.

These abstract interfaces need an *implementation*, which is a set of concrete classes that inherit from the abstract interfaces and provide an implementation for the pure, virtual operations defined in the interface. The OSAL enables you to encapsulate any RTOS by changing the implementation of the relevant framework classes (but not their interface) to meet the requirements of the given RTOS.

Mediation between the concrete classes, which are RTOS-dependent, and the neutral interfaces is accomplished using an abstract factory class, which returns to the application the concrete class that implements a particular interface. This singleton class acts as a broker that constructs the proper adapter class once requested by the application. [The OSAL Classes](#) describes the abstract factory in greater detail.

Most of the adapter classes have direct counterparts in the targeted RTOS and their implementation is straightforward. However, sometimes a certain operating system does not provide a certain object, such as a message queue. In this case, you must implement the object from primitive constructs.

Threads

Rational Rhapsody supports multitasking via *threads*. Also known as *lightweight processes*, threads are basic units of CPU utilization. Each thread consists of a program counter, register set, and stack space. It shares its code section, data section, and operating system resources, such as open files and signals, with peer threads. If an RTOS does not support multitasking via threads, the operating system adapter written for that environment must provide it.

The factory has two create thread operations that create two different kinds of threads:

- ◆ `createOMOSThread`—Creates a simple thread. This is the most common case. Simple threads are constructed in suspended mode by default. This means that the thread does not start execution until you call `start`. Otherwise, it might start execution immediately and try to access variables or data that are not yet valid.
- ◆ `createOMOSWrapperThread`—Creates a wrapper thread. A *wrapper thread* is used to wrap an external thread so it can be treated as one of the application threads on the call stack. A wrapper thread can be suspended, resumed, have its priority set, and participate in animation. Wrapper threads are used only for instrumentation. They represent user-defined threads (threads defined outside the Rational Rhapsody framework).

Stack Size

The stack size is determined by the implementation of the wrapper thread object `<env>Thread`, derived from the `OMOSThread` interface. Specifically, the stack size is defined in the constructor body, which is executed upon the thread creation call. For example, in the constructor for a `VxThread` object in `VxWorks`, the stack size is set to the default value of `OMOSThread::DefaultStackSize` in `VxOS.h`, as follows:

```
VxThread(void tfunc(void *), void *param,
         const char* const name = NULL,
         const long stackSize =
         OMOSThread::DefaultStackSize);
```

`DefaultStackSize` in `OMOSThread` is set to `DEFAULT_STACK` (defined as 20000 for `VxWorks`) in the `VxOS.cpp` file, as follows:

```
const long OMOSThread::DefaultStackSize = DEFAULT_STACK;
```

To change the size of the stack for all new threads, change the definition of `DEFAULT_STACK` in the `<env>OS.h` file. Alternatively, you can change the size of the stack for a particular thread by passing a different value as the fourth parameter to the thread constructor.

Synchronization Services

The OSAL provides synchronization services by using event flags for signaling between threads and by protecting access to shared resources through the use of mutexes and semaphores. A *mutex* provides binary mutual exclusion, whereas a *semaphore* provides access by a limited number of threads. For more information, see the sections [OMOSMutex Class](#) and [OMOSSemaphore Class](#).

Message Queues

A *message queue* is an interprocess communication (IPC) mechanism that allows independent but cooperating tasks (that is, active classes) within a single CPU to communicate with one another. An active class is considered a task in Rational Rhapsody.

The message queue is a buffer that is used in non-shared memory environments, where tasks communicate by passing messages to each other rather than by accessing shared variables. Tasks share a common buffer pool, with `OMOSMessageQueue` implementing the buffer. The message queue is an unbounded FIFO queue that is protected from concurrent access by different threads.

Events are asynchronous. When a class sends an event to another class, rather than sending it directly to the target reactive class, it passes the event to the operating system message queue and the target class retrieves the event from the head of the message queue when it is ready to process it. Synchronous events can be passed using triggered operations instead.

Many tasks can write messages into the queue, but only one can read messages from the queue at a time. The reader waits on the message queue until there is a message to process. Messages can be of any size.

Processes that want to communicate with each other must be linked somehow. A communication link consists of a relation, as in the form of an association line drawn between classes in an object model diagram. The link can be either unidirectional or bidirectional (symmetric). In the case of a unidirectional link from class A to class B, class A can send messages to class B, but class B cannot send messages to class A. With bidirectional links, both classes can send messages to each other. The message queue is attached to the link, and allows the sender and receiver of the message to continue on with their own processing activities independently of each other.

In operating systems with memory protection, one active class can call an operation of another active class, given an association relation between them, if the operating system itself supports such direct calls. For operating systems with shared memory, Rational Rhapsody knows how to pass events using the operating system messaging. Whether direct function calls are supported with memory protection depends on the operating system itself, not the Rational Rhapsody framework.

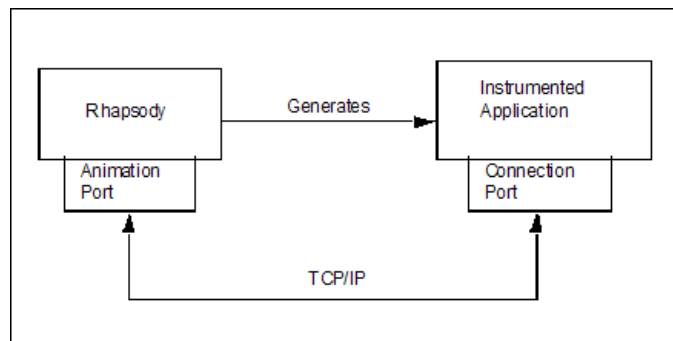
In Rational Rhapsody applications, the `BaseNumberOfInstances` property (under `CG::Event`) specifies the initial size of the memory pool that is allocated for events. This pool is dynamically allocated at program initialization. The `AdditionalNumberOfInstances` property (under `CG::Event`) specifies the size of any additional memory that should be allocated during run time if the initial pool becomes full. Additional memory allocation is done on the heap and includes rearranging of the initial memory pool.

Communication Port

A *communication port* provides interprocess communication between Rational Rhapsody and instrumented applications. Unlike a regular message queue, which is used for communication between tasks on the same processor, a connection port has some unique identification, generally a socket address and number, that allows Rational Rhapsody to communicate with processes running on either the same machine or different machines. This allows Rational Rhapsody to communicate, for example, with an animated application running on a remote target board.

Rational Rhapsody requires the TCP/IP protocol to be installed on the host machine. Processes connect to the animation server via the connection port using the TCP/IP protocol. The port number is included at the start of message packets that are addressed to the animation server.

The following figure illustrates the interprocess communication.



Note the following:

- ◆ Rational Rhapsody listens to the port number defined in the `rhapsody.ini` file.
- ◆ The framework inserts the same port number into the connection port.

The instrumented application can be running on either the same machine as Rational Rhapsody (the host machine) or on a remote target.

For more information, see [OMOSConnectionPort Class](#) and [OMOSSocket Class](#).

Timer Service

The operating system factory provides two different kinds of timers:

- ◆ **Tick timer**—Used for real-time modeling. The tick timer is compiled into the `<env>oxf` and `<env>oxfinst` libraries.

The factory's `createOMOSTickTimer` method creates a constant-interval application timer. The timer calls a callback function at a set interval.

- ◆ **Idle timer**—Used for simulated-time modeling.

Both timers are implementations of `OMOSTimer`. For more information, see [OMOSTimer Class](#).

Real-time Operating System (RTOS)

The typical embedded software application created in Rational Rhapsody is designed to work with a real-time operating system (RTOS). Rational Rhapsody includes a number of adapters that cover the more common RTOSes, as described in [Rebuilding the Rational Rhapsody Framework](#). In addition, you can customize the Rational 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 \(OSAL\)](#)
- ◆ [OSWrappers Package](#)

AbstractLayer Package (OSAL)

The operating system `AbstractLayer` package (OSAL) allows you to develop and test the application and algorithmic code of embedded, real-time systems in the environment that best suits your needs. You can implement and test the actual concurrent behavior and interactions, including interleave and stress testing, in an implementation environment. Then, when ready, you can adapt the code to an embedded target where debug facilities are often extremely limited. The interface provided by the operating system adapter remains the same.

The 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. 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 Rational Rhapsody and an instrumented application.
- ◆ `OMOSConnectionPort` used for interprocess communication between instrumented applications and Rational 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. See the [OSAL Methods](#) for a list of all methods and their definitions.

OSWrappers Package

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

Adapting Rational Rhapsody for a New RTOS

To adapt Rational Rhapsody for a new RTOS, first follow these installation steps:

1. Launch the Rational Rhapsody installation and select the Development Edition.
2. Select the development language or languages and the **Check for Real Time OS Settings** check box.
3. Select the new development environment that is the same or as close as possible to the desired environment to use as starting point.
4. Click **Next** and make any necessary path selections or changes on the next two wizard screens.
5. Select the **Typical** installation and complete the installation.

Run-Time Sources

During the Rational Rhapsody installation, the run-time source files for your language (C or C++) are added to the Rational Rhapsody \Share\Lang<Language>\oxf directory. For example, if you installed the runtime source files for C++, the directory \Share\LangC++\oxf contains both .h and .cpp files.

Adding the New Adapter

After the new environment has been installed, these additional general steps are required to add the new adapter:

1. Create new makefiles for building the framework libraries for the new environment. See [Creating the Batch File and Makefiles](#).
2. Build the framework libraries for the new environment. See [Building the Framework Libraries](#).
3. Create a set of code generation properties for the new environment and a batch file that sets its compiler environment. You may use the properties and batch file for the closest installed compiler and linker combination as a starting point. See [Creating Properties for a New RTOS](#).
4. Validate the new adapter. See [Validating the New Adapter](#).
5. Create a new configuration and select the new RTOS as its target environment.
6. Generate and make code in the new environment.

Creating the Batch File and Makefiles

Each adapter must provide a set of makefiles and a batch file for building the new OXF libraries (including the OSAL), using its provided cross-compiler. The following table lists the makefile for each library.

Makefile	Description	Built With
oxf	Run-time libraries	<env>oxf.mak
aom	Instrumentation libraries that support both tracing and animation	<env>aom.mak
tom	Instrumentation library that supports tracing	<env>tom.mak
omcom	Communication libraries that support communication between Rational Rhapsody and an instrumented application	<env>omcom.mak

The compiled framework libraries are linked to the application generated from the Rational Rhapsody model, which has its own makefile. The application makefile is specified via the `MakeFileContent` property, which you modify in the `site<lang>.prp` file. See [Makefiles](#) for more details.

1. Create a batch file to set the environment named `<env>make.bat`, call the makefile, and save it to `$OMROOT\etc`. This file can be used to build the framework as well as a Rational Rhapsody model (see also [Building the C or C++ Framework in One Step](#)).
2. Create the following makefiles and save them to the specified locations.

File	Location	Description
<env>build.mak	<code>\$OMROOT\Lang<lang></code>	Calls the other makefiles to build the Rational Rhapsody framework libraries (see Sample <env>build.mak File).
<env>aom.mak	<code>\$OMROOT\Lang<lang>\aom</code>	Builds the instrumentation libraries: <ul style="list-style-type: none"> • <code><env>aomtrace</code> • <code><env>aoanim</code>
<env>omcom.mak	<code>\$OMROOT\Lang<lang>\omcom</code>	Builds the communication library for instrumentation (<code><env>omcomappl</code>)
<env>oxf.mak	<code>\$OMROOT\Lang<lang>\oxf</code>	Builds the OXF libraries: <ul style="list-style-type: none"> • <code><env>oxf</code> • <code><env>oxfinst</code> See OXF Versions for descriptions of the different OXF libraries.

File	Location	Description
<env>tom.mak	\$OMROOT\tom	Builds the tracing libraries: <ul style="list-style-type: none"> • <env>tomtrace • <env>tomtraceRiC (for Rational Rhapsody Developer for C)

You might also need to copy any RTOS-specific configuration files required to build the libraries to \$OMROOT\MakeTempl. For example, pSOSystem™ requires drv_conf.c and sys_conf.h. In addition, you might need to copy the root.cpp file. Replace these files with any board-specific versions, if necessary.

Sample <env>build.mak File

The following is an example of the vxbuild.mak file, which is used to build the framework for the VxWorks environment.

```

MAKE=make

CPU=I80486

ifeq ($(PATH_SEP),)
all :
    @echo PATH_SEP is not defined. Please define it as \\
or /
else
all :
    $(MAKE) all -C oxf -f vxoxf.mak CFG=vxoxf CPU=$(CPU)
        PATH_SEP=$(PATH_SEP)
    $(MAKE) all -C oxf -f vxoxf.mak CFG=vxoxfsim
CPU=$(CPU)
    PATH_SEP=$(PATH_SEP)
    $(MAKE) all -C oxf -f vxoxf.mak CFG=vxoxfinst
        CPU=$(CPU) PATH_SEP=$(PATH_SEP)
    $(MAKE) all -C oxf -f vxoxf.mak CFG=vxoxfsiminst
        CPU=$(CPU) PATH_SEP=$(PATH_SEP)
    $(MAKE) all -C omcom -f vxomcom.mak CFG=vxomcomapplCPU=$(CPU)
PATH_SEP=$(PATH_SEP)

    $(MAKE) all -C tom -f vxtom.mak CFG=vxtomtrace
        CPU=$(CPU) PATH_SEP=$(PATH_SEP)

    $(MAKE) all -C tom -f vxtom.mak CFG=vxtomtraceRiC
        CPU=$(CPU) PATH_SEP=$(PATH_SEP)

    $(MAKE) all -C aom -f vxaom.mak CFG=vxaomtrace
        CPU=$(CPU) PATH_SEP=$(PATH_SEP)

    $(MAKE) all -C aom -f vxaom.mak CFG=vxaomanim
        CPU=$(CPU) PATH_SEP=$(PATH_SEP)

endif

```

This makefile:

- ◆ Sets the make command for the VxWorks environment (`make`).
- ◆ Sets the CPU being targeted (`I80486` = Intel 80486).
- ◆ Checks whether the path separator (`PATH_SEP`) character was properly set. If not, it generates an error and cancels the build.
- ◆ Sets the `all:` command to build the framework libraries for the various configurations (with and without animation, real-time or simulated time, and so on).

Creating New Makefiles

You should use the existing makefile for the environment that most closely resembles the new RTOS as a template. The GNU version of the Solaris makefile (`sol2buildGNU.mak`) is the most neutral makefile because it is based on general GNU make capabilities, as opposed to the more target-specific makefiles (such as `msoxf.mak`), which are specific to a particular environment.

OXF Versions

In the current implementation, the Rational Rhapsody OXF is compiled in the following versions:

- ◆ OXF—Production, real-time OXF
- ◆ OXFINST—Instrumented OXF (for animation)

Animation Libraries

To support instrumentation (animation or tracing), Rational Rhapsody requires other libraries besides the OXF libraries to be linked to the generated application. These libraries are specific to the target operating system. The `aom` and `omcom` libraries have corresponding makefiles that are similar to the OXF.

C++ Libraries

The compiled C++ libraries are located in the `$OMROOT\LangCPP\lib` directory:

- ◆ For C++ animation, you need `<env>aomanip.lib` (for example, `vxaomanip.lib`) and `<env>omComAppl.lib`.
- ◆ For C++ trace, you need `<env>aomtrace.lib`, `<env>omComAppl.lib`, and `<env>tomtrace.lib`.

The C++ libraries require support for C++ I/O streams. For operating systems without I/O streams (such as Windows CE®), set the `_OM_NO_Iostream` flag in the makefile used to compile the libraries to the `RHAP_FLAGS` command, as follows:

```
RHAP_FLAGS=-D _OM_NO_Iostream
```

Note

Windows CE does not support tracing because it does not have I/O streams. There is no tracer library that does not require I/O streams.

C Libraries

The compiled C libraries are located in the `$OMROOT\LangC\lib` directory:

- ◆ For C animation, you need `<env>aomanim.lib` and `<OS>omComAppl.lib`.
- ◆ For C trace, you need `<env>aomtrace.lib`, `<env>omComAppl.lib`, `<env>tomtraceRiC.lib`, and `<env>oxfinst.lib`.

Of the instrumentation libraries for C, five were written natively in C. However, `<env>tomtraceRiC` is a C++ library that is located in `$OMROOT\LangCpp\lib`. It provides C tracing services, although the library itself was written in C++. Because the library is precompiled, you need only link to it. Therefore, the language in which it was written should be of no concern.

Java Libraries

The compiled Java libraries are supplied as `jar` files in the `$OMROOT\LangJava\lib` directory. For Java animation, you need the files `anim.jar` and `animcom.jar`.

Implementing the Adapter Classes

To implement the adapter classes, you inherit from the OXF classes defined in the `os.h` file and provide an implementation for each of these classes. You must implement the following classes:

- ◆ `OMOSConnectionPort`
- ◆ `OMOSEventFlag`
- ◆ `OMOSMessageQueue`
- ◆ `OMOSMutex`
- ◆ `OMOSSemaphore`
- ◆ `OMOSSocket`
- ◆ `OMOSThread`
- ◆ `OMOSTimer`

It is common practice to add the `<env>` prefix to each implemented class.

For example, you would implement the `OMOSMutex` class for VxWorks as follows:

1. The OXF class for a mutex is `OMOSMutex`, so the VxWorks adapter class that inherits from `OMOSMutex` is named `VxMutex`.
2. Implement each of the interface operations defined for the class. The `OMOSMutex` class is defined in `os.h` as follows:

```
class RP_FRAMEWORK_DLL OSMutex {
    OM_DECLARE_FRAMEWORK_MEMORY_ALLOCATION_OPERATORS
public:
    virtual ~OSMutex(){};
    virtual void lock() = 0;
    virtual void unlock() = 0;
    virtual void* getOsHandle() const = 0;
#ifdef OSE_DELTA
    // backward compatibility support for non-OSE
    // applications
    void free() {unlock();}
#endif
};
```

3. Place the specification of the new adapter class in the `VxOS.h`:

```
class VxMutex: public OSMutex {
private:
    SEM_ID    hMutex;
public:
    void lock() {semTake(hMutex, WAIT_FOREVER);}
    void unlock() {semGive(hMutex);}
};
```

```

VxMutex() {
    // hMutex = semBCreate(SEM_Q_FIFO, SEM_FULL);
    hMutex = semMCreate(SEM_Q_FIFO);
}

~VxMutex() {semDelete(hMutex);}

void* getHandle() {return (void *)hMutex;}
virtual void* getOsHandle() const {return (void*)
    hMutex;}
};

```

Modifying rawtypes.h

The `rawtypes.h` file contains the basic types supplied by the RTOS to be used by the OXF. If you are creating a new RTOS, you must add the include file for that environment.

For example, the VxWorks section of the `rawtypes.h` file is as follows:

```

// Basic os definitions

#ifdef VxWorks
#include <vxWorks.h>
#endif

```

Other Operating System-Related Modifications

You might need to modify the `setInput` method of the `TOMUI` class to support tracing in a new operating system. When creating input streams for the stepper, there might be compilation errors if the call to create a new `ifstream` in the `setInput` method uses `ios::nocreate`. Because `ios::nocreate` is not part of the C++ standard, some compilers (such as Green Hills) do not support it. Currently, the implementation of `setInput` in the `tom\tomstep.cpp` file has options to create `ifstream`s for UNIX and the STL without using `ios::nocreate`. The implementation is as follows:

```

#ifdef unix
    // unix : Actually Solaris 2 cannot open for READ if
    // the ios::nocreate is placed here
    ifstream* file = new ifstream(filename);
#else
#ifdef OM_USE_STL
    ifstream* file = new ifstream(filename);
#else
    ifstream* file = new ifstream(filename,ios::nocreate);
#endif
#endif

```

In addition, you might need to add another `#ifdef` clause if the new environment does not support `ios::nocreate`. For example, add the following lines of code before the last `#else` for the Green Hills compiler:

```
#else
#ifdef green
    ifstream* file = new ifstream(filename);
```

Building the Framework Libraries

The following sections describe how to rebuild the framework libraries, according to language and platform. The topics are as follows:

- ◆ [Building the C or C++ Framework for Windows Systems](#)
- ◆ [Building the Ada Framework](#)
- ◆ [Building the Java Framework](#)
- ◆ [Building the Framework for Solaris Systems](#)

Note

Some environments require you to set additional macros in the invocation command. Typically, there are also optional switches to control compilation.

Building the C or C++ Framework for Windows Systems

You can build the framework libraries for C or C++ on Windows systems in either one or two steps.

Building the C or C++ Framework in Two Steps

To build the framework, follow these steps:

1. If necessary, set any environment variables required by the target cross-compiler. For example, running `$OMROOT\etc\vcvars32.bat` sets the environment for the Microsoft® compiler.

Note: The `<env>build.mak` makefile builds all run-time libraries and saves them to the `$OMROOT\lib` directory.

2. Change directory to the `$OMROOT\Lang<lang>` directory and issue the appropriate `make` command for the target environment with the `<env>build.mak` file as an argument. For example:

```
> make -f vxbuild.mak PATH_SEP=<path separator>
```

Note that the path separator for VxWorks can be defined as either `\\` or `/`.

Building the C or C++ Framework in One Step

You can combine the two steps into one by using the `<env>make.bat` file with `<env>build.mak` as its argument. The batch file sets the environment before invoking the makefile. For example, the following is the `msmake.bat` file used to set the environment and then build files for the Microsoft environment:

```
@echo off
if "%2"==" " set target=all
if "%2"=="build" set target=all
if "%2"=="rebuild" set target=clean all
if "%2"=="clean" set target=clean
call "D:\Rhapsody\Share\etc\Vcvars32.bat" x86
echo ``nmake.exe
nmake /nologo /I /S /F %1 %target%
```

The `<lang>_CG::<Environment>InvokeMake` property uses the `<env>make.bat` batch file to build a Rational Rhapsody model for a specific target environment. You can use the same batch file to build the framework libraries for that environment. Thus, the command to build the C or C++ framework libraries (from the `$OMROOT\Lang<lang>` directory) for most environments becomes:

```
> ..\etc\<<env>make.bat <env>build.mak
```

This is the preferred method for building the framework libraries for all environments and operating systems except Solaris (see [Building the Framework for Solaris Systems](#)) and the JDK.

Building the Ada Framework

To use animation with Rational Rhapsody Developer for Ada, you must have version 3.13p of the GNAT compiler. Otherwise, you must recompile the framework.

To recompile the framework, follow these steps:

1. Install the Rational Rhapsody Developer for C framework source code. The Rational Rhapsody Developer for C framework is used to enable Rational Rhapsody Developer for Ada animation.
2. Build the Ada behavioral libraries as follows:
 - a. Open the model `<Rhapsody>\Share\LangAda83\model\RiAServices.rpy`.
 - b. Generate and build the code.
 - c. Build the animation C libraries using the makefile included in the directory `<Rhapsody>\Share\LangC`. For example:

```
make -f AdaWinbuild.mak GNAT_HOME=e:/gnat/*
```

Note

In GNAT 3.15p, the directory layout was modified. If you are using 3.15p and higher, update the C makefiles by replacing the string "mingw32" with the string "GNAT_WIN32_LIBS=pentium-mingw32msv" to the makefile invocation command.

If you have several compilers installed on your machine, make sure that you invoke the make utility supplied by GNAT (verify that the GNAT\bin directory is added to your path before any other compiler).

Note

To compile the C framework with GNAT, you must install the Windows API support package as well as the Ada common package.

Building the Java Framework

Rational Rhapsody Developer for Java[®] provides a real-time framework for Java[™] in the form of a Rational Rhapsody model (oxf.rpy) under the \$OMROOT\LangJava\model\oxf directory. The best way to build the Java framework is to open this model and build it in Rational Rhapsody (by selecting **Code > Generate/Make**).

You can also build the Java framework outside of Rational Rhapsody. However, you must first generate code for the oxf.rpy model inside of Rational Rhapsody to create the OXFLib.bat file (using **Code > Generate > NonInstrumented**). Build the Java framework using the following steps:

1. Open a command prompt window.
2. Change directory to the OMROOT\LangJava\src directory.
3. Run OXFLib.bat.

Building the Framework for Solaris Systems

Because there is no cross-compiler that can build Solaris code on the PC, the framework libraries that are linked into Solaris applications must be built on Solaris. In addition to the framework source files, you need a script that removes carriage returns from framework source files to be built on Solaris. These are provided in the Solaris library tar file, which is installed when you select the **Solaris 2.x Libraries** option during the Rational Rhapsody installation.

To build the framework, follow these steps:

1. When installing Rational Rhapsody on the PC, select the **Solaris 2.x Libraries** option. This installs the sol2shr.tar file, which contains the files needed to build the framework for Solaris.

2. On the Solaris machine, create a `rhapsody` directory. For example:

```
$ mkdir /usr/rhapsody
```

3. Copy the `sol2shr.tar` file from the PC to the `rhapsody` directory on the Solaris machine.
4. On the Solaris machine, unzip the `sol2shr.tar` file in the `rhapsody` directory using the following command:

```
$ tar xvf sol2shr.tar
```

This creates a `Share` directory under `rhapsody` and extracts the framework source files to the appropriate subdirectories. It also extracts the GNU `make` executable and the `removeCR.sh` script to the `Share/etc` directory. The script removes carriage returns from UNIX files.

5. On the Solaris machine, set the `OMROOT` environment variable to point to the new `Share` directory. For example, if you created the `Share` directory as `/usr/rhapsody/Share`, use the following command to set `OMROOT`:

```
$ setenv OMROOT /usr/rhapsody/Share
```

6. Ensure that the path to the compiler is set in the `PATH` variable.
7. Change directory to `$OMROOT/Lang<lang>`.
8. Run the `removeCR.sh` script to remove carriage returns from the `sol2build.mak` and `sol2buildGNU.mak` files using the following command:

```
$ ../etc/removeCR.sh sol2build*.mak
```

9. Change directory to `$OMROOT/Lang<lang>/aom` and run the `removeCR.sh` script to remove carriage returns from all the makefiles and source files in the directory using the following command:

```
$ ../../etc/removeCR.sh *.mak *.h *.cpp
```

10. Repeat go to step 9 for each of the `omcom`, `oxf`, and `tom` subdirectories of `$OMROOT/Lang<lang>`.

11. Change directory to `$OMROOT/Lang<lang>`.

12. If you are using the Forte compiler, build the framework libraries using the following command:

```
$ ../etc/make -f sol2build.mak
```

If you are using the GNU compiler, use the following command:

```
$ ../etc/make -f sol2buildGNU.mak
```

Creating Properties for a New RTOS

To complete the addition of a new environment, identify the following information about the development tools it uses for Rational Rhapsody:

- ◆ Compiler
- ◆ Linker
- ◆ Make utility
- ◆ Libraries

To accomplish this, customize all of the language-specific code generation properties for the new environment by creating `site<lang>.prp` files in the `$OMROOT\Properties` directory for each language you intend to support in the new environment. The language-independent `site.prp` file is required for any build; any language-specific `site<lang>.prp` files are used only if they are present.

Modifying the `site<lang>.prp` Files

The search path in Rational Rhapsody for site properties is as follows:

```
site<lang>.prp -> site.prp ->
```

As you move from left to right in this search path, properties defined in the files on the left override the same properties defined in files on the right.

Note

Do not modify any of the original `factory.prp` or language-specific `factory<lang>.prp` files. Otherwise, you will not be able to return to the factory defaults. See [Implementing the Abstract Factory](#).

To add the new environment as a possible selection for a configuration, follow these steps:

1. Open the `site<lang>.prp` properties file for each language that the new environment supports.
2. From the existing `site.prp` file, create language-specific `site<lang>.prp` files for each language that the new environment supports. For example, if the environment supports Java, save the file as `siteJava.prp`.
3. In the new `site<lang>.prp` file, insert the following line above the line that contains the end keyword:

```
Subject <lang>_CG
```

Replace <lang> with CPP for C++, C for C, or JAVA for Java (case sensitive).
Repeat for each language.

4. In the new site<lang>.prp file, add the following lines between the Subject <lang>_CG and end lines, with this indentation:

```
MetaClass Configuration
end
```

5. From the new .prp file, copy the Property Environment line from the MetaClass Configuration and paste it into the corresponding location in the new site<lang>.prp file.

6. Add the new environment to the end of the enumerated values in the Environment property. For example, change the line Property Environment Enum "Microsoft,Vxworks,..." to the following:

```
Property Environment Enum "Microsoft,Vxworks,...,<env>OS"
```

7. If the new operating system will be the default environment for the respective language, replace the last string in the Environment line with the name of the new environment. For example, change the line Property Environment Enum "Microsoft,VxWorks,...,envOS" "Microsoft" to the following:

```
Property Environment Enum "Microsoft,VxWorks,...,
<env>OS" "<env>OS"
```

For example, if you are creating C++ code generation properties, your siteC++.prp file would now look like this:

```
Subject CPP_CG
  MetaClass Configuration
    Property Environment Enum "Microsoft,VxWorks,
      Solaris2, Borland, MSStandardLibrary, PsosPPC,
      MicrosoftWinCE,OseSfk,<env>OS" "<env>OS"
  end
end
```

8. In the factory<lang>.prp file, find the metaclass for the environment that most closely resembles the new target environment.
9. Copy the entire metaclass, including its closing end line, into the new site<lang>.prp file, between the closing end statement for the Configuration metaclass and that for the <lang>_CG subject.
10. Save the new site<lang>.prp file.
11. Repeat the process for each language.

12. In the new `site<lang>.prp` file, rename the copied metaclass to the name of the new operating system:

```
Metaclass <env>OS
  Property InvokeExecutable String ...
end
```

13. Modify the `InvokeMake` property (under `<lang>_CG::<Environment>`) to use the correct `<env>make.bat` batch file for the new environment.
14. Modify each of the code generation properties, especially `MakeFileContent` and its related properties (described in [Makefiles](#)) as appropriate for the new environment, replacing any occurrences of the operating system-specific prefix with the corresponding prefix for the new operating system.

Note: The most important properties for a new environment are those that interact with the makefile.

15. Save the `site<lang>.prp` file. Repeat for each language.
16. Restart Rational Rhapsody to load the new `site<lang>.prp` files.

Setting the Environment

You can set the new environment as the default or you can select it from the list of available environments for a configuration in the Rational Rhapsody browser.

In the browser, you can set the `Environment` property as follows:

- ◆ **For a project**—All new components (and their configurations) will use the environment by default.
- ◆ **For a component**—All new configurations within the component will use the environment by default.
- ◆ **For a particular configuration**—Only that configuration will use the environment by default.

To set the `Environment` property, follow these steps:

1. Decide the scope of the setting:
 - a. To set the environment for the entire project, select **File > Project Properties**.
 - b. To set the environment for a component or a configuration, right-click the component or configuration, then select **Properties** from the popup-menu.
2. In the Features dialog box, under the `<lang>_CG` subject, select the `Configuration` metaclass.

3. Select the `Environment` property and change it to the name of the new environment. For example, `<env>OS`.

Configuring the OXF Properties for the C++ Framework

To configure the OXF properties for your C++ framework, use either of these methods:

- ◆ Open your C++ project and select **File > Project Properties**. In the Features dialog box, filter the properties for those containing “OXF” and make the appropriate changes in the `CPP_CG:Framework` category.
- ◆ You may directly edit specific framework files in the OXF directory using the table below.

The Rational Rhapsody Developer for C++ framework files are located in the directory `<install_dir>/LangCpp/oxf`.

Important OXF Files

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

Important OXF Files

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

Important OXF Files

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

Validating the New Adapter

To test the new adapter, follow these steps:

1. Try building a simple “Hello World” using Rational Rhapsody and your new adapter. In Rational Rhapsody, create a class that prints the string “Hello World” when the class is instantiated. When you generate code, be sure to select your new environment in the configuration settings.
2. Try building the application. This will immediately find problems in your adapter, because building the application requires the use of the generated makefile. To see the generated makefile, right-click on the configuration in Rational Rhapsody and select **Edit Makefile**. At this point, you might need to adjust the properties to get the correct generated makefile for your application.
3. When you have successfully built the Hello World application, make your application more complex by adding more classes, putting in include paths, and specifying some libraries to link in. This continues to test the properties you defined in [Creating Properties for a New RTOS](#).
4. You must test the framework part of the adapter (see [Modifying the Framework](#)) by running the Hello World example. If it does not run correctly, you might not have implemented the framework classes correctly.

For example, Rational Rhapsody creates a main thread for all applications. Check to make sure that this thread was created correctly for your particular environment.

Note: Note that for this step, it is best to use your native compiler.

5. When the Hello World application runs successfully, make your application more complex. For example:
 - a. Create some active objects.
 - b. Create statecharts for some objects.
 - c. Use timeouts in the statecharts.
 - d. Send messages and events between objects and active objects.
 - e. Use protection by guarding operations and attributes.
 - f. Change the instrumentation to tracing.
 - g. Change the instrumentation to animation.

By implementing an application that tests for this functionality, you validate a major portion of the adapter. To complete the validation, request a copy of the RTOS Adapter Test Suite from IBM

Rational Rhapsody Support. This test suite consists of several models that cover most of the scenarios needed to test an RTOS adapter.

Modifying the Framework

The adapter interfaces and the abstract factory interface are declared in the following header files:

- ◆ `oxf.h`—Object execution framework (OXF) classes
- ◆ `os.h`—Abstract operating system classes
- ◆ `rawtypes.h`—Data types used by the OXF

These header files are located in the `$OMROOT\Lang<lang>\oxf` subdirectory of the Rational Rhapsody installation. In this path, `$OMROOT` is an environment variable that points to the `Rhapsody\Share` directory.

Implementing the Abstract Factory

Each RTOS adapter consists of a concrete operating system factory, which implements the abstract operating system factory. To create the concrete factory for a new target, follow these steps:

1. Create a specification file and an implementation file, each prefixed by the operating system (environment) name using the convention `<env>OS`, where `<env>` is an abbreviation for the environment name. For example, the adapter source files for VxWorks are named `VxOS.h` and `VxOS.cpp`. The concrete factory for the VxWorks environment is implemented in these files.

Note: You should use an existing implementation as a starting point for the adapter. For example, if VxWorks is the closest existing environment to the new target, copy and rename the `VxOS.h` and `VxOS.cpp` files to use as a template. Make sure that all the adapter implementation classes in these files are prefixed in a consistent manner. For example, the concrete factory for VxWorks is named `VxOSFactory`.

2. Rename all environment-specific prefixes in the copied files from the old to the new environment name. Note that using the operating system as a prefix for operating system wrapper classes is a Rational Rhapsody convention; you can create your own naming scheme.

Plugging in the Factory

The factory mediates between the application and the concrete, operating system-dependent adapter classes.

To plug in the concrete factory, you must create a specific `<env>OSFactory` that inherits from the `OMOSFactory` in the OXF. This class is declared in the `<env>OS.h` file.

For example, in the `VxOS.h` file, the `VxOSFactory` class inherits from the `OMOSFactory` in the OXF, as follows:

```
////////////////////////////////////  
class VxOSFactory : public OMOFactory {  
    // OSFactory hides the RTOS mechanisms for tasking and  
    // synchronization
```

Defining the Virtual Operations

Within the `<env>OSFactory` class declaration, you must define a set of virtual operations that will create the operating system services needed by the application. These services include tasking, synchronization, connection ports, message queues, and timing services.

In the `VxOS.h` file, the declaration of virtual operations is as follows:

```
public:  
    virtual OMOMessageQueue *createOMOSMessageQueue(  
        OMBBoolean /* shouldGrow */ = TRUE,  
        const long messageQueueSize =  
            OMOThread::DefaultMessageQueueSize)  
    { return (OMOSMessageQueue*)new  
        VxOSMessageQueue(messageQueueSize); }  
    virtual OMOConnectionPort *createOMOSConnectionPort()  
    {  
#ifdef _OMINSTRUMENT  
        return (OMOSConnectionPort*)new VxConnectionPort();  
#else  
        return NULL;  
#endif  
    }  
    virtual OMOEventFlag* createOMOEventFlag() {  
        return (OMOEventFlag *)new VxOSEventFlag(); }  
    virtual OMOThread *createOMOSThread(void tfunc(  
        void*), void *param,  
        const char* const threadName = NULL,  
        const long stackSize=OMOThread::DefaultStackSize)  
    {return (OMOSThread*)new VxThread(tfunc, param,  
        threadName, stackSize);};  
    virtual OMOThread* createOMOSWrapperThread(  
        void* osHandle) {  
        if (NULL == osHandle)  
            osHandle = getCurrentThreadHandle();  
        return (OMOSThread*)new VxThread(osHandle);  
    }  
    virtual OMOSEM *createOMOSMutex() {return  
        (OMOSMutex*)new VxMutex();}  
    virtual OMOSTimer *createOMOSTickTimer(timeUnit tim,  
        void cbkfunc(void*), void *param) {  
        return (OMOSTimer*)new VxTimer(tim, cbkfunc,  
            param); // TickTimer for real time  
    }  
    virtual OMOSTimer *createOMOSIdleTimer(  
        void cbkfunc(void*), void *param) {  
        return (OMOSTimer*)new VxTimer(cbkfunc, param);  
    }  
    // Idle timer for simulated time  
    }  
    virtual OMOSSemaphore* createOMOSSemaphore(  
        void* osHandle, void cbkfunc(void*), void *param)
```

```
        unsigned long semFlags = 0,  
        unsigned long initialCount = 1,  
        unsigned long /* maxCount */ = 1,  
        const char * const /* name */ = NULL)  
    {  
        return (OMOSemaphore*) new VxSemaphore(  
            semFlags, initialCount);  
    }  
  
    virtual void* getCurrentThreadHandle();  
    virtual void delayCurrentThread(timeUnit ms);  
    virtual OMBoolean waitOnThread(void* osHandle,  
        timeUnit ms) {return FALSE;  
    }  
};
```

The instance Function

To finish plugging in the concrete factory, you must create the instance function, defined in `<env>OS.cpp`, which returns a pointer to the concrete operating system factory. The instance method creates a single instance of the `OMOSFactory`. It is defined as follows:

```
static OMOFactory* instance();
```

For example, in `VxWorks`, the declaration is as follows:

```
OMOSFactory* OMOFactory::instance()  
{  
    static VxOSFactory theFactory;  
    return &theFactory;  
}
```

OSAL Methods

The following table briefly describes each OSAL method. For ease of use, the methods are listed in alphabetical order.

OSAL Method	Description
~OMOSConnectionPort	Destroys the OMOSConnectionPort object.
~OMOSEventFlag	Destroys the OMOSEventFlag object.
~OMOSMessageQueue	Destroys the OMOSMessageQueue object.
~OMOSMutex	Destroys the OMOSMutex object.
~OMOSSemaphore	Destroys the OMOSSemaphore object.
~OMOSSocket	Destroys the OMOSSocket object.
~OMOSThread	Destroys the OMOSThread object.
~OMOSTimer	Destroys the OMOSTimer object.
~OMTMMessageQueue	Destroys the OMTMMessageQueue object.
cleanup	Cleans up the memory after an object is deleted.
Close	Closes the socket.
Connect	Connects a process to the instrumentation server at a given socket address and port.
create	Creates a new object.
Create	Creates a new socket.
createOMOSConnectionPort	Creates a connection port.
createOMOSEventFlag	Creates an event flag.
createOMOSIdleTimer	Creates an idle timer.
createOMOSMessageQueue	Creates a message queue.
createOMOSMutex	Creates a mutex.
createOMOSSemaphore	Creates a semaphore.
createOMOSThread	Creates a thread.
createOMOSTickTimer	Creates a tick timer.
createOMOSWrapperThread	Creates a wrapper thread.
createSocket	Creates a new socket.
delayCurrentThread	Delays the current thread for the specified length of time.
destroy	Destroys the object.
endApplication	Ends a running application.
endMyTask	Terminates the current task.
endOtherTask	Terminates a task other than the current task.
endProlog	Ends the prolog.

OSAL Method	Description
<u>exeOnMyTask</u>	Determines whether the method was invoked from the same operating system task as the one on which the object is running.
<u>exeOnMyThread</u>	Determines whether the method was invoked from the same operating system thread as the one on which the object is running.
<u>free</u>	Releases the lock, possibly causing the underlying operating system to reschedule threads.
<u>get</u>	Retrieves the message at the beginning of the queue.
<u>getCurrentTaskHandle</u>	Returns the native operating system handle to the task.
<u>getCurrentThreadHandle</u>	Returns the native operating system handle to the thread.
<u>getMessageList</u>	Retrieves the list of messages.
<u>getOSHandle</u>	Retrieves the task's operating system ID.
<u>getOsHandle</u>	Retrieves the thread's operating system ID.
<u>getOsQueue</u>	Retrieves the event queue.
<u>getTaskEndCbkb</u>	Is a callback function that ends the current operating system task.
<u>getThreadEndCbkb</u>	Is a callback function that ends the current operating system thread.
<u>init</u>	Initializes the new object.
<u>initEpilog</u>	Executes operating system-specific actions to be taken at the end of <code>OXF::init</code> after the environment has been set (that is, the main thread and the timer have been started) and before it returns.
<u>instance</u>	Creates a single instance of <code>OMOSFactory</code> .
<u>isEmpty</u>	Determines whether the message queue is empty.
<u>isFull</u>	Determines whether the queue is full.
<u>lock</u>	Determines whether the mutex is free and reacts accordingly.
<u>OMEventQueue</u>	Constructs an <code>OMEventQueue</code> object.
<u>OMTMMessageQueue</u>	Constructs an <code>OMTMMessageQueue</code> object.
<u>pend</u>	Blocks the thread making the call until there is a message in the queue.
<u>put</u>	Adds a message to the end of the message queue.
<u>receive</u>	Waits on the socket to receive the data.
<u>Receive</u>	Receives data through the socket.
<u>reset</u>	Forces the event flag into a known state.
<u>resume</u>	Resumes a suspended thread.
<u>RiCOSEndApplication</u>	Ends a running application.

OSAL Method	Description
RiCOSOXFInitEpilog	Initializes the epilog.
send	Sends data from the socket.
Send	Sends data out from the connection port. or Sends data out from the socket.
SetDispatcher	Sets the dispatcher function, which is called whenever there is an input on the connection port (input from the socket).
setEndOSTaskInCleanup	Determines whether destruction of the <code>RiCOSTask</code> class should kill the operating system task associated with the class.
setEndOSThreadInDtor	Determines whether destruction of the <code>OMOSThread</code> class should kill the operating system thread associated with the class.
setOwnerProcess	Sets the thread that owns the message queue.
setPriority	Sets the operating system priority of the task or thread.
signal	Releases a blocked thread.
start	Starts the task or thread processing.
suspend	Suspends the task or thread.
unlock	Releases the lock, possibly causing the underlying operating system to reschedule threads.
wait	Blocks the thread making the call until some other thread releases it by calling <code>signal</code> on the same event flag instance.
waitOnThread	Waits for a thread to terminate.

See [The OSAL Classes](#) for detailed information.

The OSAL Classes

The operating system adapter is an implementation of an abstract factory pattern. For example, in Rational Rhapsody Developer for C++, the abstract operating system interface consists of the `OMOSFactory` class, whose abstract products are classes that represent operating services such as `OMOSThread`, `OMOSMutex`, and so on. Each target operating system has its own concrete factory and concrete products that are similarly named, but with the `OMOS` prefix replaced with an operating system-dependent prefix. For example, the prefix for VxWorks is `VxOS`, the prefix for pSOSystem is `PsosOS`, and so on.

The abstract operating system interfaces are defined in `RiCOSWrap.h` (under `$OMROOT\LangC\oxf`) and `*os.h` (under `$OMROOT\LangCpp\oxf`). Code that uses an operating system adapter directly should include the appropriate file for the class definitions and link with the compiled `<env>oxf` library or a variant of it.

The operating system interface provides abstract methods to create each type of operating system entity. Because the created classes are abstract, the interface hides the concrete class and returns its abstract representation.

This section contains reference pages for the classes and methods that comprise the abstract interface. For ease-of-use, the classes are presented in alphabetical order under each programming language for C and C++.

Rational Rhapsody Developer for C

The single file `RiCOSWrap.h` defines the abstract classes and methods used for multiple environment definitions (`RiCOSNT.c`, `RiCVxWorks.c`, and so on). Each adapter defines the specific data (for example, `struct`) in its own `.h` file (`RiCOSNT.h`, `RiCVxWorks.h`, and so on).

The C methods described in this section include the corresponding VxWorks implementations (defined in the file `RiCOSVxWorks.c`). Note that the VxWorks-specific methods are not included in this section; see the appropriate files for details.

The C classes for the abstract interface are as follows:

- ◆ [RiCOSConnectionPort Class](#)
- ◆ [RiCOSEventFlag Interface](#)
- ◆ [RiCOSMessageQueue Class](#)
- ◆ [RiCOSMutex Class](#)
- ◆ [RiCOSOXF Class](#)
- ◆ [RiCOSSemaphore Class](#)
- ◆ [RiCOSSocket Class](#)
- ◆ [RiCOSTask Class](#)
- ◆ [RiCOSTimer](#)
- ◆ [RiCHandleCloser Class](#)

RiCOSConnectionPort Class

The `RiCOSConnectionPort` class is used for interprocess communication between instrumented applications and Rational Rhapsody.

Creation Summary

create	Creates an <code>RiCOSConnectionPort</code> object
destroy	Destroys the <code>RiCOSConnectionPort</code> object
cleanup	Cleans up after an <code>RiCOSConnectionPort</code> object
init	Initializes an <code>RiCOSConnectionPort</code> object

Method Summary

Connect	Connects a process to the instrumentation server at the specified socket address and port
Send	Sends data out from the connection port
SetDispatcher	Sets the connection dispatcher function, which is called whenever there is an input on the connection port (input from the socket)

create

Description

The `create` method creates an `RiCOSConnectionPort` object.

Signature

```
RiCOSConnectionPort *RiCOSConnectionPort_create();
```

Returns

The newly created connection port

Example

```
RiCOSConnectionPort * RiCOSConnectionPort_create()
{
    RiCOSConnectionPort * me =
        malloc(sizeof(RiCOSConnectionPort));
    RiCOSConnectionPort_init(me);
    return me;
}
```

destroy

The destroy method destroys the connection port.

Signature

```
void RiCOSConnectionPort_destroy(  
    RiCOSConnectionPort * const me);
```

Parameters

me

The RiCOSConnectionPort object to delete

Example

```
void RiCOSConnectionPort_destroy(  
    RiCOSConnectionPort * const me)  
{  
    if (me == NULL) return;  
    RiCOSConnectionPort_cleanup(me);  
    free(me);  
}
```

cleanup

Description

The cleanup method cleans up after an RiCOSConnectionPort object is destroyed.

Signature

```
void RiCOSConnectionPort_cleanup(  
    RiCOSConnectionPort * const me);
```

Parameters

me

The object to clean up after

Example

```
void RiCOSConnectionPort_cleanup(  
    RiCOSConnectionPort * const me)  
{  
    if (me==NULL) return;  
    RiCOSSocket_cleanup(&me->m_Socket);  
  
    /* Assumes you will have only one connection port  
     so the data for m_Buf can be freed; if it is not  
     the case, the readFromSockLoop will allocate it. */  
  
    if (me->m_Buf) {  
        free(me->m_Buf);  
    }  
}
```

```
    me->m_BufSize = 0;
}
```

init

The init method initializes the connection port.

Signature

```
RiCBoolean RiCOSConnectionPort_init(
    RiCOSConnectionPort * const me);
```

Parameters

me

The RiCOSConnectionPort object

Returns

The method returns RiCTRUE if successful.

Example

```
RiCBoolean RiCOSConnectionPort_init(
    RiCOSConnectionPort * const me)
{
    RiCBoolean b;

    if (me==NULL) return RiCFALSE;
    me->m_Buf = NULL;
    b = RiCOSMutex_init(&me->m_SendMutex);
    b &= RiCOSEventFlag_init(&me->m_AckEventFlag);
    me->m_BufSize = 0;
    me->m_Connected = 0;
    me->m_dispatchfunc = NULL;
    me->m_ConnectionThread = NULL;
    me->m_ShouldWaitForAck = 1;
    me->m_NumberOfMessagesBetweenAck = 0;
    RiCOSEventFlag_reset(&me->m_AckEventFlag);
    return b;
}
```

Connect

The Connect method connects a process to the instrumentation server at the specified socket address and port.

Signature

```
int RiCOSConnectionPort_Connect(
    RiCOSConnectionPort *const me,
    const char* const SocketAddress,
    unsigned int nSocketPort);
```

Parameters

me

The RiCOSConnectionPort object.

SocketAddress

The socket address. The default value is NULL.

nSocketPort

The port number of the socket. The default value is 0.

Returns

The connection status. The possible values are as follows:

- ◆ 1—Success
- ◆ 0—Failure

Example

```
RiCOSResult RiCOSConnectionPort_Connect(
    RiCOSConnectionPort * const me,
    const char* const SocketAddress,
    unsigned int nSocketPort)
{
    if (me==NULL) return 0;

    if (NULL == me->m_dispatchfunc) {
        fprintf(stderr, "RiCOSConnectionPort_SetDispatcher
        should be called before
        RiCOSConnectionPort_Connect()\n");
        return 0;
    }

    if ( 0 == me->m_Connected ) {
        (void)RiCOSSocket_init(&me->m_Socket);
        me->m_Connected = RiCOSSocket_createSocket(
            &me->m_Socket, SocketAddress, nSocketPort);
    }

    if (0 == me->m_Connected)
        return 0;
}
```

```

    /* Connection established invoking thread to
       receive messages from the socket */

    me->m_ConnectionThread = RiCOSTask_create((
        void (*)(void *))readFromSockLoop,
        (void *)me, "tRhpSock", RiCOSDefaultStackSize);
    RiCOSTask_start(me->m_ConnectionThread);
    return me->m_Connected;
}

```

Send

The Send method sends data out from the connection port. This operation should be thread-protected.

Signature

```

int RiCOSConnectionPort_Send(
    RiCOSConnectionPort *const me, struct RiCSData *m);

```

Parameters

me

The RiCOSConnectionPort object from which to send the data

m

The data to be sent from the port

Returns

An integer that represents the number of bytes sent through the socket

Example

```

RiCOSResult RiCOSConnectionPort_Send(
    RiCOSConnectionPort * const me, struct RiCSData *m)
{
    int rv = 0, m_NumberOfMessagesBetweenAck = 0;
    RiCOSMutex_lock(&me->m_SendMutex);

    if (me->m_Connected) {
        char lenStr[MAX_LEN_STR+1];
        (void)sprintf(lenStr, "%d", RiCSData_getLength(m));
        rv = RiCOSSocket_send(&me->m_Socket,
            lenStr, MAX_LEN_STR);
        if (rv > 0) {
            rv = RiCOSSocket_send(&me->m_Socket,
                RiCSData_getRawData(m), RiCSData_getLength(m));
        }
        if (me->m_ShouldWaitForAck) {
            const int maxNumOfMessagesBetweenAck = 127;
            /* This MUST match the number in Rhapsody. */
            if (maxNumOfMessagesBetweenAck > 0) {
                m_NumberOfMessagesBetweenAck++;
                if (m_NumberOfMessagesBetweenAck >=

```

```
        maxNumOfMessagesBetweenAck) {
            m_NumberOfMessagesBetweenAck = 0;
            RiCOSEventFlag_wait(
                &me->m_AckEventFlag, -1);
            RiCOSEventFlag_reset(
                &me->m_AckEventFlag);
        }
    }
    RiCOSMutex_free(&me->m_SendMutex);
    /* cleanup */
    RiCSData_cleanup(m);
    return rv;
}
```

SetDispatcher

The SetDispatcher method sets the connection dispatcher function, which is called whenever there is an input on the connection port (input from the socket).

Signature

```
RiCBoolean RiCOSConnectionPort_SetDispatcher(
    RiCOSConnectionPort *const me,
    RiCOS_dispatchfunc dispfunc);
```

Parameters

me

The RiCOSConnectionPort object

dispfunc

The dispatcher function

Returns

The method returns RiCTRUE if successful.

Example

```
RiCBoolean RiCOSConnectionPort_SetDispatcher(
    RiCOSConnectionPort * const me,
    RiCOS_dispatchfunc dispfunc)
{
    if (me==NULL) return RiCFALSE;
    me->m_dispatchfunc = dispfunc;
    return RiCTRUE;
}
```


RiCOSEventFlag Interface

An *event flag* is a synchronization object used for signaling between 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. The event flag is initially in the unsignaled (reset) state.

With the Rational Rhapsody implementation of event flags, at least one of the waiting threads is released when an event flag is signaled. This is in contrast to the regular semantics in some operating systems, in which all waiting threads are released when an event flag is signaled.

Creation Summary

create	Creates an RiCOSEventFlag object
destroy	Destroys the RiCOSEventFlag object
cleanup	Cleans up after an RiCOSEventFlag object
init	Initializes an RiCOSEventFlag object

Method Summary

reset	Forces the event flag into a known state
signal	Releases a blocked task
wait	Blocks the task making the call until some other task releases it by calling <code>signal</code> on the same event flag instance

create

The `create` method creates an RiCOSEventFlag object.

Signature

```
RiCOSEventFlag *RiCOSEventFlag_create();
```

Returns

The newly created RiCOSEventFlag

Example

```
RiCOSEventFlag * RiCOSEventFlag_create()
{
    RiCOSEventFlag * me = malloc(sizeof(RiCOSEventFlag));
    if (me != NULL) RiCOSEventFlag_init(me);
    return me;
}
```

destroy

The destroy method destroys the `RiCOSEventFlag` object.

Signature

```
void RiCOSEventFlag_destroy (RiCOSEventFlag *const me);
```

Parameters

me

The `RiCOSEventFlag` object to delete

Example

```
void RiCOSEventFlag_destroy(RiCOSEventFlag * const me)
{
    if (me != NULL) {
        RiCOSEventFlag_cleanup(me);
        free(me);
    }
}
```

cleanup

The cleanup method cleans up the memory after an `RiCEventFlag` object is destroyed.

Signature

```
void RiCOSEventFlag_cleanup (RiCOSEventFlag *const me);
```

Parameters

me

The object to clean up after

Example

```
void RiCOSEventFlag_cleanup(RiCOSEventFlag * const me)
{
    if (me != NULL && me->hEventFlag != NULL) {
        semDelete(me->hEventFlag);
        me->hEventFlag = NULL;
    }
}
```

init

The init method initializes the `RiCEventFlag` object.

Signature

```
RiCBoolean RiCOSEventFlag_init (
    RiCOSEventFlag *const me);
```

Parameters

me

The `RiCOSEventFlag` object to initialize

Returns

The method returns `RiCTRUE` if successful.

Example

```
RiCBoolean RiCOSEventFlag_init(RiCOSEventFlag * const me)
{
    if (me == NULL) return RiCFALSE;
    me->hEventFlag = semBCreate(SEM_Q_FIFO, SEM_EMPTY);
    return (me->hEventFlag != NULL);
}
```

reset

The reset method forces the event flag into a known state. This method is called almost immediately prior to a [wait](#).

Signature

```
RiCOSResult RiCOSEventFlag_reset(
    RiCOSEventFlag *const me);
```

Parameters

me

The `RiCOSEventFlag` object

Returns

The `RiCOSResult` object, as defined in the `RiCOS*.h` files

Example

```
RiCOSResult RiCOSEventFlag_reset(
    RiCOSEventFlag * const me)
{
    if (me == NULL) {return 0;}
    semTake(me->hEventFlag, NO_WAIT);
    return (RiCOSResult)1;
}
```

signal

The signal method releases a blocked task. If more than one task is waiting for an event flag, a call to this method release sat least one of them.

Signature

```
RiCOSResult RiCOSEventFlag_signal(  
    RiCOSEventFlag *const me);
```

Parameters

me

The RiCOSEventFlag object

Returns

The RiCOSResult object, as defined in the RiCOS*.h files

Example

```
RiCOSResult RiCOSEventFlag_signal(  
    RiCOSEventFlag * const me)  
{  
    if (me == NULL) {return 0;}  
    semGive(me->hEventFlag);  
    return (RiCOSResult)1;  
}
```

See Also

[wait](#)

wait

The wait method blocks the task making the call until some other task releases it by calling signal on the same event flag instance.

Signature

```
RiCOSResult RiCOSEventFlag_wait(  
    RiCOSEventFlag *const me, int tminms);
```

Parameters

me

The RiCOSEventFlag object.

tmins

Specifies the length of time, in milliseconds, that the thread should remain blocked. A value of -1 means to wait indefinitely.

Returns

The RiCOSResult object, as defined in the RiCOS*.h files

Example

```
RiCOSResult RiCOSEventFlag_wait(  
    RiCOSEventFlag * const me, int tminms)  
{  
    if (me == NULL) {return 0 /*WAIT_FAILED*/;}  
  
    if (-1 == tminms) {  
        semTake(me->hEventFlag, WAIT_FOREVER);  
    }  
    else {  
        int ticks = cvrtTmInMStoTicks(tminms);  
        semTake(me->hEventFlag, ticks);  
    }  
    return (RiCOSResult)1;  
}
```

See Also

[signal](#)

RiCOSMessageQueue Class

The RiCOSMessageQueue class represents a list of messages (events).

Creation Summary

<u>create</u>	Creates an RiCOSMessageQueue object
<u>destroy</u>	Destroys RiCOSMessageQueue object
<u>cleanup</u>	Cleans up after an RiCOSMessageQueue object
<u>init</u>	Initializes an RiCOSMessageQueue object

Method Summary

<u>get</u>	Retrieves the message at the beginning of the message queue
<u>getMessageList</u>	Retrieves a list of messages
<u>isEmpty</u>	Determines whether the message queue is empty
<u>isFull</u>	Determines whether the message queue is full
<u>pend</u>	Locks the thread making the call until there is a message in the queue
<u>put</u>	Adds a message to the end of the message queue

create

The create method creates an RiCOSMessageQueue object.

Signature

```
RiCOSMessageQueue * RiCOSMessageQueue_create(  
    RiCBoolean shouldGrow, int initSize);
```

Parameters

shouldGrow

Determines whether the queue should be of fixed size (RiCFALSE) or able to expand as needed (RiCTRUE).

initSize

Specifies the initial size of the queue. The default message queue size is set by the variable RiCOSDefaultMessageQueueSize.

The maximum length of the message queue is operating system- and implementation-dependent. It is usually set in the adapter for a particular operating system.

Returns

The newly created `RiCOSMessageQueue`

Example

```
RiCOSMessageQueue * RiCOSMessageQueue_create(
    RiCBoolean shouldGrow, int initSize)
{
    RiCOSMessageQueue * me = malloc(
        sizeof(RiCOSMessageQueue));
    RiCOSMessageQueue_init(me, shouldGrow, initSize);
    return me;
}
```

destroy

The destroy method destroys the `RiCOSMessageQueue` object.

Signature

```
void RiCOSMessageQueue_destroy(
    RiCOSMessageQueue *const me);
```

Parameters

me

The `RiCOSMessageQueue` object to destroy

Example

```
void RiCOSMessageQueue_destroy(
    RiCOSMessageQueue * const me)
{
    if (me == NULL) return;
    RiCOSMessageQueue_cleanup(me);
    free(me);
}
```

cleanup

The cleanup method cleans up after the `RiCOSMessageQueue` object.

Signature

```
void RiCOSMessageQueue_cleanup(  
    RiCOSMessageQueue * const me);
```

Parameters

me

The object to clean up after

Example

```
void RiCOSMessageQueue_cleanup(  
    RiCOSMessageQueue * const me)  
{  
    if (me == NULL) return;  
    if (me->hVxMQ) {  
        (void)msgQDelete(me->hVxMQ);  
        me->hVxMQ = 0;  
    }  
}
```

init

The init method initializes the `RiCOSMessageQueue` object.

Signature

```
RiCBoolean RiCOSMessageQueue_init(  
    RiCOSMessageQueue *const me, RiCBoolean shouldGrow,  
    int initSize);
```

Parameters

me

Specifies the `RiCOSMessageQueue` object to initialize.

shouldGrow

Determines whether the queue should be of fixed size (`RiCFALSE`) or able to expand as needed (`RiCTRUE`).

initSize

Specifies the initial size of the queue. The default message queue size is set by the variable `RiCOSDefaultMessageQueueSize`. You can override the default value by passing a different value when you create the message queue.

The maximum length of the message queue is operating system- and implementation-dependent. It is usually set in the adapter for a particular operating system.

Returns

The method returns `RiCTRUE` if successful.

Example

```
RiCBoolean RiCOSMessageQueue_init(  
    RiCOSMessageQueue * const me, RiCBoolean shouldGrow,  
    int initSize)  
{  
    if (me == NULL) return RiCFALSE;  
  
    if (initSize < 0) initSize =  
        RiCOSDefaultMessageQueueSize;  
    me->m_State = noData;  
    me->hVxMQ = msgQCreate(initSize, sizeof(void*),  
        MSG_Q_FIFO);  
    return RiCTRUE;  
}
```

get

The get method retrieves the message at the beginning of the message queue.

Signature

```
gen_ptr RiCOSMessageQueue_get(  
    RiCOSMessageQueue * const me);
```

Parameters

me

The `RiCOSMessageQueue` from which to retrieve the message

Returns

The message

Example

```
gen_ptr RiCOSMessageQueue_get(  
    RiCOSMessageQueue * const me)  
{  
    gen_ptr m = NULL;  
  
    if (me == NULL) return NULL;  
  
    if (me->m_State == dataReady) {  
        m = me->pmessage;  
        me->m_State = noData;  
    }  
  
    else { /* function returns NULL if there are
```

```
        no messages in me->hVxMQ queue */
        if (msgQReceive(me->hVxMQ, (char*)&m, sizeof(m),
            NO_WAIT) <= 0)/* nonblocking semantics */
            return NULL;
    }
    return m;
}
```

See Also

[getMessageListput](#)

getMessageList

The `getMessageList` method retrieves a list of messages. It is used for two reasons:

- ◆ To cancel events

When a reactive class is destroyed, it notifies its thread to cancel all events in the queue that are triggered for that reactive class. The thread iterates over the queue, using [getMessageList](#) to retrieve the data, and marks as canceled all events whose target is the reactive class.

- ◆ To show the data in the event queue during animation

Signature

```
RiCOSResult RiCOSMessageQueue_getMessageList(
    RiCOSMessageQueue *const me, RiCList *l);
```

Parameters

`me`

The `RiCOSMessageQueue`

`l`

The list of messages in the queue

Returns

The `RiCOSResult` object, as defined in the `RiCOS*.h` files

Example

```
RiCOSResult RiCOSMessageQueue_getMessageList(
    RiCOSMessageQueue * const me, RiCList * l)
{
    RiCList_removeAll(l);

    if (me == NULL) return 0;

    if (!RiCOSMessageQueue_isEmpty(me)) {
        MSG_Q_INFO msgQInfo;

        if (noData != me->m_State) {
```

```

        RiCList_addTail(l,me->pmessage);
    }

    msgQInfo.taskIdListMax = 0;
    msgQInfo.taskIdList = NULL;

    /* do not care which tasks are waiting */

    msgQInfo.msgListMax = 0;
    msgQInfo.msgPtrList = NULL;
    msgQInfo.msgLenList = NULL;

    /* Do not care about message length. The
    first call will retrieve the numMsgs data
    member. */

    if (OK == msgQInfoGet(me->hVxMQ, &msgQInfo)) {
        if (msgQInfo.numMsgs > 0) {
            int numMsgs = msgQInfo.numMsgs;
            msgQInfo.msgListMax = numMsgs;
            msgQInfo.msgPtrList = malloc(
                (numMsgs+1)*sizeof(void*));
            if (OK == msgQInfoGet(me->hVxMQ, &msgQInfo)) {
                void *m;
                int i;
                for (i = 0; i < numMsgs; i++) {
                    m = *(void **)msgQInfo.msgPtrList[i];
                    RiCList_addTail(l,m);
                }
                free(msgQInfo.msgPtrList);
            }
        }
    }
    return 1;
}

```

See Also[getput](#)

isEmpty

The isEmpty method determines whether the message queue is empty.

Signature

```
RiCBoolean RiCOSMessageQueue_isEmpty(  
    RiCOSMessageQueue *const me);
```

Parameters

me

The RiCOSMessageQueue to check

Returns

The method returns one of the following values:

- ◆ RiCTRUE—The queue is empty.
- ◆ RiCFALSE—The queue is not empty.

See Also

[isFull](#)

isFull

The isFull method determines whether the message queue is full.

Signature

```
RiCBoolean RiCOSMessageQueue_isFull(  
    RiCOSMessageQueue * const me);
```

Parameters

me

The RiCOSMessageQueue to check

Returns

The method returns one of the following values:

- ◆ RiCTRUE—The queue is full.
- ◆ RiCFALSE—The queue is not full.

Example

```
RiCBoolean RiCOSMessageQueue_isFull(  
    RiCOSMessageQueue * const me)
```

```

{
    MSG_Q_INFO msgQInfo;

    if (RiCOSMessageQueue_isEmpty(me)) return FALSE;

    if (OK != msgQInfoGet(me->hVxMQ, &msgQInfo))
        return TRUE; /* Assume the worst case. */

    if (msgQInfo.numMsgs < msgQInfo.maxMsgs) return FALSE;

    return TRUE;
}

```

pend

The pend method blocks the task making the call until there is a message in the queue. A reader generally waits until the queue contains a message that it can read.

Signature

```

RiCOSResult RiCOSMessageQueue_pend(
    RiCOSMessageQueue *const me);

```

Parameters

me

The RiCOSMessageQueue

Returns

The RiCOSResult object, as defined in the RiCOS*.h files

Example

```

RiCOSResult RiCOSMessageQueue_pend(
    RiCOSMessageQueue * const me)
{
    if (me == NULL) return 0;

    if (me->m_State == noData) {
        gen_ptr m = NULL;
        if (msgQReceive(me->hVxMQ, (char*)&m, sizeof(m),
            NO_WAIT) <= 0) /* if the queue is empty */
            (void)msgQReceive(me->hVxMQ, (char*)&m,
                sizeof(m), WAIT_FOREVER); /* wait for message */
        me->m_State = dataReady;
        me->pmessage = m;
    }
    return 1;
}

```

put

The put method adds a message to the end of the message queue.

Signature

```
RiCOSResult RiCOSMessageQueue_put(  
    RiCOSMessageQueue *const me, gen_ptr message,  
    RiCBoolean fromISR);
```

Parameters

me

The RiCOSMessageQueue to which to add the message

message

The message to be added to the queue

fromISR

A Boolean value that determines whether the message being added was generated from an interrupt service routine (ISR)

Returns

The RiCOSResult object, as defined in the RiCOS*.h files

Example

```
RiCOSResult RiCOSMessageQueue_put(  
    RiCOSMessageQueue * const me, gen_ptr message,  
    RiCBoolean fromISR)  
{  
    static gen_ptr NULL_VAL = NULL;  
    int timeout = WAIT_FOREVER;  
    int priority = MSG_PRI_NORMAL;  
  
    if (message == NULL) message = NULL_VAL;  
  
    if (fromISR) {  
        timeout = NO_WAIT;  
        priority = MSG_PRI_URGENT;  
    }  
    return (msgQSend(me->hVxMQ, (char*)&message,  
        sizeof(message), timeout, priority) == OK);  
}
```

See Also

[get](#)

[getMessageList](#)

RiCOSMutex Class

A *mutex* is the basic synchronization mechanism used to protect critical sections within a thread. Mutexes are used to implement protected objects. The mutex allows one thread mutually exclusive access to a resource. Mutexes are useful when only one thread at a time can be allowed to modify data or some other controlled resource. For example, adding nodes to a linked list is a process that should only be allowed by one thread at a time. By using a mutex to control the linked list, only one thread at a time can gain access to the list.

The Rational Rhapsody implementation of a mutex is as a recursive lock mutex. This means that the same thread can lock the mutex several times without blocking itself. In other words, the mutex is actually a counted semaphore. When implementing `OMOSMutex` for the target environment, you should implement it as a recursive lock mutex.

Mutexes can be either free or locked (they are initially free). When a task executes a `lock` operation and finds a mutex locked, it must wait. The task is placed on the waiting queue associated with the mutex, along with other blocked tasks, and the CPU scheduler selects another task to execute. If the `lock` operation finds the mutex free, the task places a lock on the mutex and enters its critical section. When any task releases the mutex by calling `free`, the first blocked task in the waiting queue is moved to the ready queue, where it can be selected to run according to the CPU scheduling algorithm.

The same thread can nest `lock` and `free` calls of the same mutex without indefinitely blocking itself. Nested locking by the same thread does not block the locking thread. However, the nested locks are counted so the proper `free` actually releases the mutex.

Creation Summary

<u>create</u>	Creates an <code>RiCOSMutex</code> object
<u>destroy</u>	Destroys the <code>RiCOSMutex</code> object
<u>cleanup</u>	Cleans up after an <code>RiCOSMutex</code> object
<u>init</u>	Initializes an <code>RiCOSMutex</code> object

Method Summary

<u>free</u>	Frees the lock, possibly causing the underlying operating system to reschedule tasks
<u>lock</u>	Determines whether the mutex is locked

create

The create method creates an `RiCOSMutex` object.

Signature

```
RiCOSMutex * RiCOSMutex_create();
```

Returns

The newly created `RiCOSMutex`

Example

```
RiCOSMutex * RiCOSMutex_create()
{
    RiCOSMutex * me = malloc(sizeof(RiCOSMutex));
    RiCOSMutex_init(me);
    return me;
}
```

destroy

The destroy method destroys the `RiCOSMutex` object.

Signature

```
void RiCOSMutex_destroy (RiCOSMutex * const me);
```

Parameters

`me`

The `RiCOSMutex` object to destroy

Example

```
void RiCOSMutex_destroy(RiCOSMutex * const me)
{
    if (me != NULL) {
        RiCOSMutex_cleanup(me);
        free(me);
    }
}
```


cleanup

The cleanup method cleans up the memory after an `RiCOSMutex` object is destroyed.

Signature

```
void RiCOSMutex_cleanup (RiCOSMutex * const me);
```

Parameters

me

The deleted `RiCOSMutex` object to clean up after

Example

```
void RiCOSMutex_cleanup(RiCOSMutex * const me)
{
    if (me != NULL && me->hMutex !=NULL) {
        semDelete(me->hMutex);
        me->hMutex = NULL;
    }
}
```

init

The init method initializes the `RiCOSMutex` object.

Signature

```
RiCBoolean RiCOSMutex_init (RiCOSMutex * const me);
```

Parameters

me

The `RiCOSMutex` object to initialize

Returns

The method returns `RiCTRUE` if successful.

Example

```
RiCBoolean RiCOSMutex_init(RiCOSMutex * const me)
{
    if (me == NULL) return 0;

    me->hMutex = semMCreate(SEM_Q_FIFO);
    return (me->hMutex != NULL);
}
```

free

The free method frees the lock, possibly causing the underlying operating system to reschedule tasks.

In environments other than pSOSystem, this is a macro that implements the same interface.

Signature

```
RiCOSResult RiCOSMutex_free (RiCOSMutex *const me);
```

Parameters

me

The RiCOSMutex object to free

Returns

The RiCOSResult object, as defined in the RiCOS*.h files

Example

```
RiCOSResult RiCOSMutex_free(RiCOSMutex * const me)
{
    if (me == NULL) { return 0; }
    if (semGive(me->hMutex)==OK)
        return 1;
    else
        return 0;
}
```

See Also

[lock](#)

lock

The lock method determines whether the mutex is free and reacts accordingly:

- ◆ If the mutex is free, this operation locks it and allows the calling task to enter its critical section.
- ◆ If the mutex is already locked, this operation places the calling task on a waiting queue with other blocked tasks.

In environments other than pSOSystem, this is a macro that implements the same interface.

Signature

```
RiCOSResult RiCOSMutex_lock (RiCOSMutex *const me);
```

Parameters

me

The RiCOSMutex object to lock

Returns

The RiCOSResult object, as defined in the RiCOS*.h files

Example

```
RiCOSResult RiCOSMutex_lock(RiCOSMutex * const me)
{
    if (me == NULL) {return 0;}

    if (semTake(me->hMutex, WAIT_FOREVER)==OK) {
        return 1;
    }
    else
        return 0;
}
```

See Also

[free](#)

RiCOSOXF Class

The `RiCOSOXF` class defines the operating system-specific actions to take at the end of `RiCOXFInit` after the environment is set (such as the main thread, timer, and so on) and before the return from the function.

Method Summary

RiCOSEndApplication	Ends a running application
RiCOSOXFInitEpilog	Initializes the epilog

Constants

The type definitions depend on the deployment environment. For example, if the type is “long,” the type definitions would be as follows:

```
extern const long RiCOSDefaultStackSize;
extern const long RiCOSDefaultMessageQueueSize;
extern const long RiCOSDefaultThreadPriority;
```

However, if the OXF source file is `RiCOSWrap.h` and you replace `PUBLIC` with `extern`, then the type definitions would be as follows:

```
extern const RiC_StackSizeType RiCOSDefaultStackSize;
extern const RiC_MessageQueueSizeType RiCOSDefaultMessageQueueSize;
extern const RiC_ThreadPriorityType RiCOSDefaultThreadPriority;
```

RiCOSEndApplication

This method ends a running application. The operation should be implemented in the concrete adapter for the target operating system.

Signature

```
extern void RiCOSEndApplication (int errorCode);
```

Parameters

`errorCode`

Specifies the error code to be passed to the operating system, if required

Example

```
void RiCOSEndApplication(int errorCode)
{
    RiCTask* currentThread, *maint;
    RiCOSTask_endOfProcess = 1;

    #ifdef _OMINSTRUMENT
```

```

        ARCSD_instance();
        ARCSD_closeConnection();
    #endif

    currentThread = RiCTask_cleanupAllTasks();

    #ifdef _OMINSTRUMENT
        ARCSD_Destroy();
    #endif

    RiCTimerManager_cleanup(&RiCSystemTimer);
    maint = RicMainTask();

    if (maint) {

        RiCOSHandle maintHandle = RiCOSTask_getOSHandle(
            RiCTask_getOSTask(maint));
        char * maintName = taskName(maintHandle);
        int killmainthread = 1;

        if (maintName && *maintName) {
            if (!strcmp(maintName, "tShell"))
                taskRestart(maintHandle);
            else
                taskDeleteForce(maintHandle);
            killmainthread = 0;
        }

        if (killmainthread) {
            RiCTask_destroy(maint);
        }
    }

    if (currentThread) {
        RiCOSTaskEndCallBack theOSThreadEnderClb;
        void * arg1;

        /* Get a callback to end the thread. */
        (void)RiCTask_getTaskEndClbk(
            currentThread, &theOSThreadEnderClb,
            &arg1, RiCTRUE);
        RiCOSTask_setEndOSTaskInCleanup(
            RiCTask_getOSTask(currentThread), FALSE);
        /* Do not really end the os thread because you
           are executing on this thread and if you do,
           there will be a resource leak. */
        RiCTask_destroy(currentThread);
        /* Delete the whole object through a virtual
           destructor. */
        if (theOSThreadEnderClb != NULL) {
            (*theOSThreadEnderClb)(arg1);
            /* Now end the os thread. */
        }
    }
    /* Make sure that the execution thread is being
       ended. */
    RiCOSTask_endMyTask((void *) taskIdSelf());
}

```

RiCOSOXFInitEpilog

This method initializes the epilog.

Signature

```
extern void RiCOSOXFInitEpilog();
```

Example

```
void RiCOSOXFInitEpilog()
{
    taskDelay(2);
}
```

RiCOSSemaphore Class

A *semaphore* is a synchronization device that 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.

Semaphores are useful in controlling access to a shared resource that can support only a limited number of users. The current count of the semaphore is the number of additional users allowed. When the count reaches zero, all attempts to use the resource controlled by the semaphore are inserted into a system queue and wait until they either time out or the count again rises above zero. The maximum number of users who can access the controlled resource at one time is specified at construction time.

The Rational Rhapsody framework itself does not use semaphores. However, the `RiCOSSemaphore` primitive is provided as a service for environments that need it (such as Windows NT and pSOSystem).

Creation Summary

create	Creates an <code>RiCOSSemaphore</code> object
destroy	Destroys the <code>RiCOSSemaphore</code> object
cleanup	Cleans up after an <code>RiCOSSemaphore</code> object
init	Initializes an <code>RiCOSSemaphore</code> object

Method Summary

signal	Releases the semaphore token
wait	Waits for a semaphore token

create

This method creates an `RiCOSSemaphore` object.

Signature

```
RiCOSSemaphore *RiCOSSemaphore_create(  
    unsigned long semFlags, unsigned long initialCount,  
    unsigned long maxCount, const char *const name);
```

Parameters

`semFlags`

The adapter-specific creation flags

`initialCount`

The initial number of tokens available in the semaphore

`maxCount`

The maximum number of tokens available in the semaphore

`name`

The unique name of the semaphore

Returns

The newly created `RiCOSSemaphore` object

Example

```
RiCOSSemaphore * RiCOSSemaphore_create(  
    unsigned long semFlags, unsigned long initialCount,  
    unsigned long maxCount, const char * const name)  
{  
    RiCOSSemaphore * me = malloc(sizeof(RiCOSSemaphore));  
    RiCOSSemaphore_init(me, semFlags, initialCount,  
        maxCount, name);  
    return me;  
}
```


destroy

This method destroys the `RiCOSSemaphore` object.

Signature

```
void RiCOSSemaphore_destroy (RiCOSSemaphore *const me);
```

Parameters

`me`

The `RiCOSSemaphore` object to destroy

Example

```
void RiCOSSemaphore_destroy(RiCOSSemaphore * const me)
{
    if (me == NULL) return;

    RiCOSSemaphore_cleanup(me);
    free(me);
}
```

cleanup

This method cleans up after the `RiCOSSemaphore` object.

Signature

```
void RiCOSSemaphore_cleanup (RiCOSSemaphore *const me);
```

Parameters

`me`

The object to clean up after

Example

```
void RiCOSSemaphore_cleanup(RiCOSSemaphore * const me)
{
    if (me == NULL) return;

    if (me->m_semId) {
        semFlush(me->m_semId);
        semDelete(me->m_semId);
        me->m_semId = NULL;
    }
}
```

init

This method initializes the `RiCOSSemaphore`.

Signature

```
RiCBoolean RiCOSSemaphore_init (  
    RiCOSSemaphore *const me, unsigned long semFlags,  
    unsigned long initialCount, unsigned long maxCount,  
    const char *const name);
```

Parameters

me

The RiCOSSemaphore object to initialize

semFlags

The adapter-specific creation flags

initialCount

The initial number of tokens available in the semaphore

maxCount

The maximum number of tokens available in the semaphore

name

The unique name of the semaphore

Returns

The method returns RiCTRUE if successful.

Example

```
RiCBoolean RiCOSSemaphore_init(RiCOSSemaphore * const me,  
    unsigned long semFlags, unsigned long initialCount,  
    unsigned long maxCount, const char * const name)  
{  
    if (me == NULL) return RiCFALSE;  
  
    me->m_semId = NULL;  
    me->m_semId = semCCreate((int)semFlags,  
        (int)initialCount);  
    return (me->m_semId != NULL);  
}
```

signal

This method releases the semaphore token.

Signature

```
RiCOSResult RiCOSSemaphore_signal(  
    RiCOSSemaphore *const me);
```

Parameters

me

The `RiCOSSemaphore` object

Returns

The `RiCOSResult` object, as defined in the `RiCOS*.h` files

Example

```
RiCOSResult RiCOSSemaphore_signal(  
    RiCOSSemaphore * const me)  
{  
    if (!(me && me->m_semId)) return 0;  
    return (semGive(me->m_semId) == OK);  
}
```

See Also

[wait](#)

wait

This method waits for a semaphore token.

Signature

```
RiCOSResult RiCOSSemaphore_wait(  
    RiCOSSemaphore *const me, long timeout);
```

Parameters

me

The `RiCOSSemaphore` object.

timeout

The number of ticks to lock on a semaphore before timing out. The possible values are < 0 (wait indefinitely); 0 (do not wait); and > 0 (the number of ticks to wait). For Solaris systems, a value of > 0 means to wait indefinitely.

Returns

The `RiCOSResult` object, as defined in the `RiCOS*.h` files

Example

```
RiCOSResult RiCOSSemaphore_wait(  
    RiCOSSemaphore * const me, long timeout)  
{  
    if (!(me && me->m_semId)) return FALSE;  
}
```

```
        if (timeout < 0) timeout = WAIT_FOREVER;
        return (semTake(me->m_semId, timeout) == OK);
    }
```

See Also

[signal](#)

RiCOSSocket Class

The `RiCOSSocket` class represents the socket through which data is passed between Rational Rhapsody and an instrumented application. `RiCOSSocket` is generally used for animation, but it can also be used for other connections, as long as you provide a host name and port number. `RiCOSSocket` represents the client side of the connection, and assumes that somewhere over the network there is a server listening to the connection.

Creation Summary

create	Creates an <code>RiCOSSocket</code> object
destroy	Destroys the <code>RiCOSSocket</code> object
cleanup	Cleans up after an <code>RiCOSSocket</code> object
init	Initializes an <code>RiCOSSocket</code> object

Method Summary

createSocket	Creates a new socket
receive	Waits on the socket to receive the data
send	Sends data through the socket

create

This method creates an `RiCOSSocket` object.

Signature

```
RiCOSSocket *RiCOSSocket_create();
```

Returns

The newly created `RiCOSSocket`

Example

```
RiCOSSocket *RiCOSSocket_create()
{
    RiCOSSocket * me = (RiCOSSocket*)malloc(sizeof(
        RiCOSSocket));
    if (me != NULL) RiCOSSocket_init(me);
    return me;
}
```

destroy

This method destroys the `RiCOSSocket` object.

Signature

```
void RiCOSSocket_destroy (RiCOSSocket *const me);
```

Parameters

me

The `RiCOSSocket` object to destroy

Example

```
void RiCOSSocket_destroy(RiCOSSocket * const me)
{
    if (me != NULL) {
        RiCOSSocket_cleanup(me);
        free(me);
    }
}
```

cleanup

This method cleans up after the `RiCOSSemaphore` object

Signature

```
void RiCOSSocket_cleanup (RiCOSSocket *const me);
```

Parameters

me

The `RiCOSSocket` object to clean up after

Example

```
void RiCOSSocket_cleanup(RiCOSSocket * const me)
{
    if (me == NULL) return;
    if (me->theSock != 0) {
        (void) shutdown(me->theSock, 2);
        (void) close(me->theSock);
        me->theSock = 0;
    }
}
```

init

This method initializes the `RiCOSSocket` object.

Signature

```
RiCBoolean RiCOSSocket_init (RiCOSSocket *const me);
```

Parameters

`me`

The `RiCOSSocket` object to initialize

Returns

The method returns `RiCTRUE` if successful.

Example

```
RiCBoolean RiCOSSocket_init(RiCOSSocket * const me)
{
    if (me == NULL) return 0;

    me->theSock = 0;
    return 1;
}
```

createSocket

This method creates a new socket.

Signature

```
int RiCOSSocket_createSocket (RiCOSSocket * const me,
    const char *SocketAddress, unsigned int nSocketPort);
```

Parameters

`me`

The `RiCOSSocket` object.

`SocketAddress`

The socket address. This can be set to a host name that is a character string. The default value is `NULL`.

`nSocketPort`

The socket port number. The default value is 0.

Returns

The socket creation status. The possible values are as follows:

- ◆ 1—Success
- ◆ 0—Failure

Example

```
int RiCOSSocket_createSocket(RiCOSSocket * const me,
    const char * SocketAddress, unsigned int nSocketPort)
{
    static struct sockaddr_in addr;
    int proto;
    char hostName[128];
    int rvStat;

    if (me == NULL) {return 0;}

    if (nSocketPort == 0) {
        nSocketPort = 6423;
    }

    addr.sin_family = AF_INET;
    proto = IPPROTO_TCP;
    (void)gethostname(hostName, sizeof(hostName)-1);

    if (NULL != SocketAddress && strlen(SocketAddress)
        != 0) {
        if (!strcmp(hostName, SocketAddress)) {
            SocketAddress = NULL;}
        else {
            (void)strcpy(hostName, SocketAddress);
            addr.sin_addr.s_addr = inet_addr(hostName);
            if (((unsigned long)ERROR) ==
                addr.sin_addr.s_addr) {
                addr.sin_addr.s_addr =
                    hostGetByName(hostName);
            }
            if (((unsigned long)ERROR) ==
                addr.sin_addr.s_addr) {
                fprintf(stderr, "Could not get the address
                    of host '%s'\n", hostName);
                return 0;
            }
        }
    }

    if (NULL == SocketAddress || strlen(SocketAddress)
        == 0) {
        addr.sin_addr.s_addr = htonl(INADDR_LOOPBACK);
    }

#ifdef unix
    endprotoent();
#endif /* unix */

    addr.sin_port = htons((u_short)nSocketPort);
    if ((me->theSock = socket(AF_INET, SOCK_STREAM,
        proto)) == -1) {
        fprintf(stderr, "Could not create socket\n");
        me->theSock = 0;
        return 0;
    }
    while ((rvStat = connect(me->theSock,
        (struct sockaddr *)&addr, sizeof(addr))) ==
```



```
        SOCKET_ERROR && (errno == EINTR));
    if (SOCKET_ERROR == rvStat) {
        fprintf(stderr, "Could not connect to server
        at %s port %d\n Error No. : %d\n", hostName,
        (int)nSocketPort, errno);
        return 0;
    }
    return 1;
}
```

receive

This method waits on the socket to receive the data.

Signature

```
int RiCOSSocket_receive (RiCOSSocket *const me,
    char *buf, int bufLen);
```

Parameters

me

The RiCOSSocket object

buf

The string buffer in which data will be stored

bufLen

The length of the buffer

Returns

The method returns one of the following values:

- ◆ 0—There was an error.
- ◆ n—The number of bytes read.

Example

```
int RiCOSSocket_receive(RiCOSSocket * const me,
    char * buf, int bufLen)
{
    int bytes_read = 0;
    int n;

    if (me==NULL) return -1;

    while (bytes_read < bufLen) {
        n = recv(me->theSock, buf + bytes_read,
            bufLen - bytes_read,0);
        if (SOCKET_ERROR == n) {
            if (errno == EINTR) {
                continue;
            }
        }
    }
}
```

```
        else {
            return -1;
        }
    } else {
        if (0 == n) { /* Connection closed. */
            return -1;
        }
        bytes_read += n;
    }
    return bytes_read;
}
```

See Also

[send](#)

send

This method sends data through the socket.

Signature

```
int RiCOSSocket_send (RiCOSSocket *const me,
    const char *buf, int bufLen);
```

Parameters

me

The RiCOSSocket object

buf

The constant string buffer that contains the data to be sent

bufLen

The length of the buffer

Returns

The method returns one of the following values:

- ◆ 0—There was an error.
- ◆ n—The number of bytes sent.

Example

```
int RiCOSSocket_send(RiCOSSocket * const me,
    const char *buf, int bufLen)
{
    int bytes_writ = 0;
    int n;
```

```
if (me==NULL) return -1;

while (bytes_writ < buflen) {
    n = send(me->theSock, (char *) (buf + bytes_writ),
            buflen - bytes_writ, 0);
    if (SOCKET_ERROR == n) {
        if (errno == EINTR) {
            continue;
        }
        else {
            return -1;
        }
    }
    bytes_writ += n;
}
return bytes_writ;
}
```

See Also

[receive](#)

RiCOSTask Class

The RiCOSTask class provides the basic tasking features.

Creation Summary

<u>create</u>	Creates an RiCOSTask object
<u>destroy</u>	Destroys an RiCOSTask object
<u>cleanup</u>	Cleans up after an RiCOSTask object
<u>init</u>	Initializes an RiCOSTask object

Method Summary

<u>endMyTask</u>	Terminates the current task
<u>endOtherTask</u>	Terminates a task other than the current one
<u>exeOnMyTask</u>	Determines whether the method was invoked from the same operating system task as the one on which the object is running
<u>getCurrentTaskHandle</u>	Gets the handle to the active task
<u>getOSHandle</u>	Returns a handle to the underlying operating system task
<u>getTaskEndClbk</u>	Is a callback function that ends the current operating system thread
<u>resume</u>	Resumes a suspended task
<u>setEndOSTaskInCleanup</u>	Determines whether destruction of the RiCOSTask class should kill the operating system task associated with the class
<u>setPriority</u>	Sets the priority for the task
<u>start</u>	Starts executing the task
<u>suspend</u>	Suspends a task

create

This method creates a new RiCOSTask object.

Signature

```
RiCOSTask *RiCOSTask_create (RiCOSTaskEndCallBack tfunc,  
    void *param, const char *name,  
    const long stackSize);
```

Parameters

tfunc

The callback function that ends the current operating system task

param

The parameters of the callback function

name

The name of the task

stackSize

The size of the stack

Returns

The newly created RiCOSTask

Example

```
RiCOSTask * RiCOSTask_create(RiCOSTaskEndCallBack tfunc,  
    void * param, const char * name, const long stackSize)  
{  
    RiCOSTask * me = malloc(sizeof(RiCOSTask));  
    RiCOSTask_init(me, tfunc, param, name, stackSize);  
    return me;  
}
```

destroy

This method destroys the RiCOSTask object.

Signature

```
void RiCOSTask_destroy (RiCOSTask *const me);
```

Parameters

me

The RiCOSTask object to destroy

Example

```
void RiCOSTask_destroy(RiCOSTask * const me)
{
    if (me == NULL) return;
    RiCOSTask_cleanup(me);
    free(me);
}
```

cleanup

This method cleans up the memory after a RiCOSTask object is deleted.

Signature

```
void RiCOSTask_cleanup (RiCOSTask *const me);
```

Parameters

me

The RiCOSTask object to clean up after

Example

```
void RiCOSTask_cleanup(RiCOSTask * const me)
{
    if (me == NULL) return;

    if (!me->isWrapperThread) {
        RiCOSEventFlag_cleanup(&me->m_SuspEventFlag);
        /* Remove the thread. */
        if (me->endOSTaskInCleanup) {
            RiCBoolean onMyTask = RiCOSTask_exeOnMyTask(me);
            if (!(RiCOSTask_endOfProcess) &&
                RiCOSTask_exeOnMyTask(me)) {
                /* Do not kill the OS thread if this is the
                 end of process and the running thread
                 is 'this' - you need the OS thread to do
                 some cleanup, and then you kill it
                 explicitly. */
                RiCOSTaskEndCallBack theOSTaskEndClb = NULL;
                void * arg1 = NULL;
                /* Get a callback function to end the OS
                 thread. */
                (void)RiCOSTask_getTaskEndClbk(me,
                    &theOSTaskEndClb, &arg1, onMyTask);
                if (theOSTaskEndClb != NULL) {
                    /* End the OS thread */
                    (*theOSTaskEndClb)(arg1);
                }
            }
        }
    }
}
```

init

This method initializes the `RiCOSTask` object.

Signature

```
RiCBoolean RiCOSTask_init (RiCOSTask *const me,  
    RiCOSTaskEndCallBack tfunc, void *param,  
    const char *name, const long stackSize);
```

Parameters

`me`

The `RiCOSTask` object to initialize

`tfunc`

The callback function that ends the current operating system task

`param`

The parameters to the callback function

`name`

The name of the task

`stackSize`

The size of the stack

Returns

The method returns `RiCTRUE` if successful.

Example

```
RiCBoolean RiCOSTask_init(RiCOSTask * const me,  
    RiCOSTaskEndCallBack tfunc, void * param,  
    const char * name, const long stackSize)  
{  
    size_t i, len = 0;  
    char* myName = NULL;  
  
    if (me == NULL) {return 0;}  
  
    me->endOSTaskInCleanup = TRUE;  
    me->isWrapperThread = 0;  
  
    /* Copy the thread name. */  
    if (name != NULL) len = strlen(name);  
    /* check for legal name */  
    for (i = 0; i < len; i++) {  
        if ((isalnum((int)name[i]) == 0) &&  
            (name[i] != '_')) {  
            len = 0;  
            break;  
        }  
    }  
}
```

```
    }
    if (len > 0) {
        myName = malloc(len + 1);
        strcpy(myName, name);
    }
    RiCOSEventFlag_init(&me->m_SuspEventFlag);
    RiCOSEventFlag_reset(&me->m_SuspEventFlag);
    /* Create SUSPENDED thread !!!!!!! */
    me->m_ExecFunc = tfunc;
    me->m_ExecParam = param;
    me->hThread = 0;
    me->hThread = taskSpawn(myName,
        /* name of new task (stored at pStackBase) */
        (int) PRIORITY_NORMAL, /* priority of new task */
        0, /* task option word */
        (int) stackSize, /*size (bytes) of stack needed */
        (int (*)()) preExecFunc, /* thread function */
        (int) (void *) me, /* argument to thread function */
        0,0,0,0,0,0,0,0);
    return 1;
}
```

endMyTask

This method terminates the current task.

Signature

```
void RiCOSTask_endMyTask (void * t);
```

Parameters

t

The current task

Example

```
void RiCOSTask_endMyTask(void *hThread)
{
    taskDeleteForce((int)hThread);
    /* Force because this is probably waiting on
       something */
}
```

See Also

[endOtherTask](#)

[exeOnMyTask](#)

[getCurrentTaskHandle](#)

endOtherTask

This method terminates a task other than the current task.

Signature

```
RicBoolean RicOSTask_endOtherTask (void * t);
```

Parameters

t

The task to end

Returns

The method returns RiCTRUE if it successfully terminated the task.

Example

```
RicBoolean RicOSTask_endOtherTask(void *hThread)
{
    taskDeleteForce((int)hThread);
    /* Force because this is probably waiting on
       something */
    return RiCTRUE;
}
```

See Also

[endMyTask](#)

[exeOnMyTask](#)

[getCurrentTaskHandle](#)

exeOnMyTask

This method determines whether the method was invoked from the same operating system task as the one on which the object is running.

Signature

```
RicBoolean RicOSTask_exeOnMyTask (RicOSTask *const me);
```

Parameters

me

The RicOSTask object to compare

Return

The method returns one of the following values:

- ◆ RiCTRUE—The method was invoked from the same operating system task as the one on which the object is running.
- ◆ RicFALSE—The tasks are not the same.

Example

```
RicBoolean RicOSTask_exeOnMyTask(RicOSTask * const me)
{
    RicOSHandle executedOsHandle;
    RicOSHandle myOsHandle;
    RicBoolean res;

    if (me == NULL) return RicFALSE;

    /* A handle to the thread that executes the delete */
    executedOsHandle = RicOSTask_getCurrentTaskHandle();
    /* A handle to 'this' thread */
    myOsHandle = RicOSTask_getOSHandle(me);
    res = ((executedOsHandle == myOsHandle) ?
          RiCTRUE : RicFALSE);
    return res;
}
```

See Also

[endMyTask](#)

[endOtherTask](#)

[getCurrentTaskHandle](#)

getCurrentTaskHandle

This method gets the handle to the active task.

Signature

```
RiCOSHandle RiCOSTask_getCurrentTaskHandle();
```

Returns

The handle to the active task

Example

```
RiCOSHandle RiCOSTask_getCurrentTaskHandle()  
{  
    return (RiCOSHandle)taskIdSelf();  
}
```

See Also

[getOSHandle](#)

getOSHandle

This method returns a handle to the underlying operating system task.

Signature

```
RiCOSHandle RiCOSTask_getOSHandle (RiCOSTask *const me);
```

Parameters

me

The RiCOSTask object whose handle you want to retrieve

Returns

The operating system handle

Example

```
RiCOSHandle RiCOSTask_getOSHandle(RiCOSTask * const me)  
{  
    if (me == NULL) {return 0;}  
    return (RiCOSHandle)me->hThread;  
}
```

See Also

[getCurrentTaskHandle](#)

getTaskEndClbk

This method is a callback function that ends the current operating system thread.

Signature

```
int RiCOSTask_getTaskEndClbk (RiCOSTask * const me,
    RiCOSTaskEndCallBack * clb_p, void ** arg1_p,
    RiCBoolean onExecuteTask);
```

Parameters

me

The RiCOSTask object.

clb_p

A pointer to the callback function that ends the thread. This can be either endMyTask() or endOtherTask().

arg1_p

The argument for the callback function.

onExecuteTask

Set this to one of the following Boolean values:

RiCTRUE—The object should kill its own task.

RiCFALSE—Another object should kill the task.

Returns

The status. The possible values are as follows:

- ◆ 1—Success
- ◆ 0—Failure

Example

```
int RiCOSTask_getTaskEndClbk(RiCOSTask * const me,
    RiCOSTaskEndCallBack * clb_p,
    void ** arg1_p, RiCBoolean onExecuteTask)
{
    if (me == NULL) return 0;

    if (onExecuteTask) {
        /* Ask for a callback to end my own thread. */
        *clb_p = (RiCOSTaskEndCallBack)&
            RiCOSTask_endMyTask;
        *arg1_p = (void*)me->hThread;
    }
    else {
        /* Ask for a callback to end my thread by
            someone else. */
    }
}
```

```
        *clb_p = (RiCOSTaskEndCallBack)&
                RiCOSTask_endOtherTask;
        /* My thread handle. */
        *arg1_p = (void*)me->hThread;
    }
    return 1;
}
```

resume

This method resumes a suspended task. This method is not used in generated code—it is used only for advanced scheduling.

The suspend and resume methods provide a way of stopping and restarting a task. Tasks usually block when waiting for a resource, such as a mutex or an event flag, so both are rarely used.

Signature

```
RiCOSResult RiCOSTask_resume (RiCOSTask *const me);
```

Parameters

me

The RiCOSTask object to resume

Returns

The RiCOSResult object, as defined in the RiCOS*.h files

Example

```
RiCOSResult RiCOSTask_resume(RiCOSTask * const me)
{
    if (me == NULL) {return 0;}
    (void)taskResume(me->hThread);
    return 1;
}
```

See Also

[start](#)

[suspend](#)

setEndOSTaskInCleanup

This method determines whether destruction of the `RiCOSTask` class should kill the operating system task associated with the class. If the method returns `RiCTRUE`, the task will be ended at the `RiCOSTask` cleanup.

Signature

```
int RiCOSTask_setEndOSTaskInCleanup (
    RiCOSTask *const me, RiCBoolean val);
```

Parameters

`me`

The `RiCOSTask` object.

`val`

The possible values are as follows:

`RiCTRUE`—The task is ended as part of the object's destruction process.

`RiCFALSE`—The task is not ended when the object is destroyed.

Returns

The status. The possible values are as follows:

- ◆ 1—Success
- ◆ 0—Failure

Example

```
int RiCOSTask_setEndOSTaskInCleanup(
    RiCOSTask * const me, RiCBoolean val)
{
    if (me == NULL) {return 0;}

    me->endOSTaskInCleanup = val;
    return 1;
}
```

setPriority

This method sets the priority for the task.

Signature

```
RiCOSResult RiCOSTask_setPriority (RiCOSTask *const me,  
    int pr);
```

Parameters

me

The RiCOSTask object.

pr

The integer value of the priority. This parameter varies by operating system.

Returns

The RiCOSResult object, as defined in the RiCOS*.h files

Example

```
RiCOSResult RiCOSTask_setPriority(  
    RiCOSTask * const me, int pr)  
{  
    if (me == NULL) {return 0;}  
  
    taskPrioritySet(me->hThread, pr);  
    return 1;  
}
```

See Also

[start](#)

start

This method starts executing the task. Initially, tasks are suspended until `start` is called.

Signature

```
RiCOSResult RiCOSTask_start (RiCOSTask *const me);
```

Parameters

me

The RiCOSTask object to start

Returns

The RiCOSResult object, as defined in the RiCOS*.h files

Example

```
RiCOSResult RiCOSTask_start(RiCOSTask * const me)
{
    if (me == NULL) {return 0;}

    if (RiCOSEventFlag_exists(&me->m_SuspEventFlag)) {
        RiCOSEventFlag_signal(&me->m_SuspEventFlag);
        RiCOSEventFlag_cleanup(&me->m_SuspEventFlag);
    }
    else {
        RiCOSTask_resume(me);
    }
    return 1;
}
```

See Also

[resume](#)

[suspend](#)

suspend

This method suspends a task. This method is not used in generated code—it is used only for advanced scheduling.

Signature

```
RiCOSResult RiCOSTask_suspend (RiCOSTask *const me);
```

Parameters

me

The RiCOSTask object to suspend

Returns

The RiCOSResult object, as defined in the RiCOS*.h files

Example

```
RiCOSResult RiCOSTask_suspend(RiCOSTask * const me)
{
    if (me == NULL) {return 0;}

    (void)taskSuspend(me->hThread);
    return 1;
}
```

See Also

[resume](#)

[start](#)

RiCOSTimer

The `RiCOSTimer` class is a building block for `RiCTimerManager`, which provides basic timing services for the execution framework. In the Rational Rhapsody implementation, the timer runs on its own task. Therefore, the target operating system must support multitasking.

Creation Summary

create	Creates an <code>RiCOSTimer</code> object
destroy	Destroys an <code>RiCOSTimer</code> object
cleanup	Cleans up after an <code>RiCOSTimer</code> object
init	Initializes an <code>RiCOSTimer</code> object

create

This method creates an `RiCOSTimer` object.

Signature

```
RiCOSTimer * RiCOSTimer_create (timeUnit ptime,
    void (*cbkfunc)(void *), void * params);
```

Parameters

`ptime`

The time between each tick of the timer. In most adapters, the time unit is milliseconds; however, this depends on the specific adapter implementation.

`cbkfunc`

The tick-timer call-back function used to notify the timer client that a tick occurred.

`params`

The parameters to the callback function.

Returns

The newly created `RiCOSTimer`

Example

```
RiCOSTimer * RiCOSTimer_create(timeUnit ptime,
    void (*cbkfunc)(void *), void * params)
{
    RiCOSTimer * me = malloc(sizeof(RiCOSTimer));
    RiCOSTimer_init(me, ptime, cbkfunc, params);
    return me;
}
```

destroy

This method destroys the `RiCOSTimer` object.

Signature

```
void RiCOSTimer_destroy (RiCOSTimer *const me);
```

Parameters

`me`

The `RiCOSTimer` object to destroy

Example

```
void RiCOSTimer_destroy(RiCOSTimer * const me)
{
    if (me == NULL) return;

    RiCOSTimer_cleanup(me);
    free(me);
}
```

cleanup

This method cleans up the memory after an `RiCOSTimer` object is deleted.

Signature

```
void RiCOSTimer_cleanup (RiCOSTimer * const me);
```

Parameters

`me`

The `RiCOSTimer` object to clean up after

Example

```
void RiCOSTimer_cleanup(RiCOSTimer * const me)
{
    if (me == NULL) return;

    if (me->hThread) {
        RiCOSHandle executedOsHandle =
            RiCOSTask_getCurrentTaskHandle();
        /* A handle to this 'thread' */
        RiCOSHandle myOsHandle = me->hThread;
        RiCBoolean onMyThread = ((executedOsHandle ==
            myOsHandle) ? TRUE : FALSE);
        if (onMyThread) {
            RiCOSTask_endMyTask((void*)myOsHandle);
        }
        else {
```

```

        RiCOSTask_endOtherTask((void*)myOsHandle);
    }
    me->hThread = 0;
}
}

```

init

This method initializes the RiCOSTimer object.

Signature

```

RiCBoolean RiCOSTimer_init (RiCOSTimer *const me,
    timeUnit ptime, void (*cbkfunc)(void *),
    void *params);

```

Parameters

me

The RiCOSTimer object to initialize.

ptime

The time between each tick of the timer. In most adapters, the time unit is milliseconds; however, this depends on the specific adapter implementation.

cbkfunc

The tick-timer call-back function used to notify the timer client that a tick occurred.

params

The parameters to the callback function.

Returns

The method returns RiCTRUE if successful.

Example

```

RiCBoolean RiCOSTimer_init(RiCOSTimer * const me,
    timeUnit ptime, void (*cbkfunc)(void *), void *params)
{
    if (me == NULL) return RiCFALSE;
    me->cbkfunc = cbkfunc;
    me->param = params;

    if (((RiCTimerManager*)params)->realTimeModel) {
        /**** VxWorks TickTimer(Real Time)****/
        me->m_Time = ptime;
        /* Create a thread that runs the bridge, passing
           this as an argument. */
        me->ticks = cvrtTmInMStoTicks(me->m_Time);
        me->hThread = taskSpawn("timer", PRIORITY_HIGH, 0,
            SMALL_STACK, (int (*)())bridge,
            (int)(void *)me /*p1*/, 0,0,0,0,0,0,0,0,0,0);
        return me->hThread != ERROR;
    }
}

```

```
    }
    else {
        /*** IdleTimer (Simulated Time)***/
        me->m_Time = 0; /* Just create context-switch
            until the system enters idle mode. */
        me->hThread = taskSpawn("timer", PRIORITY_LOW, 0,
            SMALL_STACK, (int (*)())bridge, (int)(void*)me,
            0,0,0,0,0,0,0,0,0);
        return RiTRUE;
    }
}
```

RiHandleCloser Class

OSAL interface contains `RiCOSTask_endMyTask` method which should be used if a thread should be deleted by itself (for example, if active reactive class entered into terminate connector).

But in some RTOSes it is forbidden for thread perform such operation directly. The `RiHandleCloser` class solves this problem. It is an active reactive singleton class with a statechart containing one state. This state receives only one event (`CloseEvent`) and performs only one action (`doCloseHandle()` call) when it is received.

`OMHandleClose` thread is initialized in the `OMOS::initEpilog()`:

```
void RiCOSOXFInitEpilog(void)
{

    (void)RiHandleCloser_startBehavior(RiHandleCloser_Instance(RiInt_doCloseHandle));
}

```

If some thread is going to exit it calls (from framework) `endMyTask()` function which sends `CloseEvent` message(`event`) to the `HandleCloser` thread.

```
void RiCOSTask_endMyTask( RiC_CONST_TYPE void *const hThread )
{
    if( hThread != NULL )
    {
        RiHandleCloser_genCloseEvent(hThread);
        Exit( OUL );
    }
}

```

This message contains the handle of the thread, which should be deleted.

The `doCloseHandle` is static function, which is called by `HandleCloser` thread when `CloseEvent` event is processed.

You can see `HandleCloser` usage in Integrity adapter (`Share/LangC/oxf/RiCOSIntegrity.c` file).

Note

A similar mechanism is implemented in C++ framework.

Rational Rhapsody Developer for C++

The C++ classes for the abstract interface are as follows:

- ◆ [OMEventQueue Class](#)
- ◆ [OMMessageQueue Class](#)
- ◆ [OMOS Class](#)
- ◆ [OMOSConnectionPort Class](#)
- ◆ [OMOSEventFlag Class](#)
- ◆ [OMOSFactory Class](#)
- ◆ [OMOSMessageQueue Class](#)
- ◆ [OMOSMutex Class](#)
- ◆ [OMOSSemaphore Class](#)
- ◆ [OMOSSocket Class](#)
- ◆ [OMOSThread Class](#)
- ◆ [OMOSTimer Class](#)
- ◆ [OMTMessageQueue Class](#)

OMEventQueue Class

OMEventQueue inherits from `OMTMessageQueue<>` with `OMEvent` as a parameter. In other words, `OMEventQueue` is a list (vector/queue) of events.

Construction Summary

OMEventQueue	Creates an <code>OMOSEventQueue</code> object
------------------------------	-----------------------------------------------

Method Summary

getOsQueue	Retrieves the event queue
----------------------------	---------------------------

OMEventQueue

This method constructs an `OMEventQueue` object and initializes the `OMTMMMessageQueue<OMEvent>` superclass of the event queue, with the given size and ability to grow dynamically.

Visibility

Public

Signature

```
OMEventQueue(const long messageQueueSize =
    OMOSThread::DefaultMessageQueueSize,
    OMBoolean dynamicMessageQueue = TRUE) :
    OMTMMMessageQueue<OMEvent>(messageQueueSize,
    dynamicMessageQueue)
```

Parameters

`messageQueueSize`

The size of the message queue. If not overridden, the message queue size is initialized to the value of the static constant `DefaultMessageQueueSize` in `OMOSThread`.

`dynamicMessageQueue`

A Boolean value that specifies whether the message queue size is dynamic (`TRUE`) or fixed (`FALSE`). By default, the message queue size is dynamic.

getOsQueue

This method retrieves the event queue.

Visibility

Public

Signature

```
OMOSMessageQueue * getOsQueue()
```

OMMessageQueue Class

OMMessageQueue inherits from OMTMessageQueue<> with OMSDData as a parameter. In other words, OMMessageQueue is a list (vector/queue) of serialized data. The OMMessageQueue<OMSDData> parameterized class is declared only if instrumentation is defined.

OMSDData is the base class for all messages passed between the aom and tom libraries during instrumentation.

OMOS Class

The OMOS class defines the operating system-specific actions to take at the end of OXF::init after the environment is set (such as the main thread, timer, and so on) and before the return from the function.

Method Summary

endApplication	Ends a running application
endProlog	Ends the prolog
initEpilog	Executes operating system-specific actions to be taken at the end of OXF::init after the environment has been set (that is, the main thread and the timer have been started) and before it returns

endApplication

This method ends a running application. This operation should be implemented in the concrete adapter for the target operating system.

Visibility

Public

Signature

```
static void endApplication(int errorCode);
```

Parameters

errorCode

The error code to be passed to the operating system, if required

endProlog

This method ends the prolog.

Visibility

Public

Signature

```
static void endProlog();
```

initEpilog

This method executes operating system-specific actions to be taken at the end of `OXF::init` after the environment has been set (that is, the main thread and the timer have been started) and before it returns. This operation should be implemented in the concrete adapter for the target operating system.

Visibility

Public

Signature

```
static void initEpilog();
```

OMOSConnectionPort Class

The connection port is used for interprocess communication between instrumented applications and Rational Rhapsody. The factory's `createOMOSConnectionPort()` method creates a connection port.

Construction Summary

~OMOSConnectionPort	Destroys the <code>OMOSConnectionPort</code> object.
-------------------------------------	------------------------------------------------------

Method Summary

Connect	Connects to the specified port
Send	Sends data from the connection port
SetDispatcher	Sets the dispatcher function

~OMOSConnectionPort

This method destroys the `OMOSConnectionPort` object. You must declare the destructor explicitly, rather than letting the compiler add it, because it must be made virtual.

Visibility

Public

Signature

```
virtual ~OMOSConnectionPort()
```

Connect

This method connects a process to the instrumentation server at a given socket address and port.

Visibility

Public

Signature

```
virtual int Connect (const char* SocketAddress = NULL,  
                    unsigned int nSocketPort = 0) = 0;
```

Parameters

SocketAddress

The socket address. If you do not specify a socket address, its default value is a NULL string.

nSocketPort

The port number of the socket. If you do not specify a port number, the value 0 is used.

Returns

The connection status. The possible values are as follows:

- ◆ 1—Success
- ◆ 0—Failure

Send

This method sends data out from the connection port. This operation should be thread protected.

Visibility

Public

Signature

```
virtual int Send (OMSDData *m) = 0;
```

Parameters

m

The data to be sent from the port. The data is of type `OMSDData`, which is defined in `omCom\omsdata.h`. It encapsulates the methods by which serialized data is passed between an instrumented application and the animation/tracing server.

Return

An integer that represents the number of bytes that were sent through the socket

SetDispatcher

This method sets the dispatcher function, which is called whenever there is an input on the connection port (input from the socket).

This method was created for two reasons:

- ◆ To provide flexibility by allowing for different dispatch routines. For example, the Rational Rhapsody framework uses `SetDispatcher(portToMessageQueue)` in `aomdisp.cpp`.
- ◆ To allow the dispatch routine to be located in a different place and to be set only after creation of the connection port.

Visibility

Public

Signature

```
virtual void SetDispatcher (void dispfunc(OMSData*)) = 0;
```

Parameters

`dispfunc`

The dispatcher function

OMOSEventFlag Class

An *event flag* is a synchronization object used for signaling between 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. The event flag is initially in the unsignaled (reset) state.

With the Rational Rhapsody implementation of event flags, at least one of the waiting threads is released when an event flag is reset. This is in contrast to the regular semantics in some operating systems, in which all waiting threads are released when an event flag is reset.

The operating system factory's `createOMOSEventFlag` method creates a new event flag.

Construction Summary

~OMOSEventFlag	Destroys the <code>OMOSEventFlag</code> object
--------------------------------	------------------------------------------------

Method Summary

getOsHandle	Retrieves the thread's operating system ID
reset	Forces the event flag into a known state
signal	Releases a blocked thread
wait	Blocks the thread making the call until some other thread releases it by calling <code>signal</code> on the same event flag instance

~OMOSEventFlag

This method destroys the `OMOSEventFlag` object. You must declare the destructor explicitly, rather than letting the compiler add it, because it must be made virtual.

Visibility

Public

Signature

```
virtual ~OMOSEventFlag()
```

getOsHandle

This method retrieves the thread's operating system ID. This value varies by operating system.

Visibility

Public

Signature

```
virtual void* getOsHandle() const = 0;
```

Return

The operating system ID

reset

This method forces the event flag into a known state. This method is often called immediately prior to a wait.

Visibility

Public

Signature

```
virtual void reset() = 0;
```

signal

This method releases a blocked thread. If more than one task is waiting for an event flag, a call to this method releases at least one of them.

Visibility

Public

Signature

```
virtual void signal() = 0;
```

wait

This method blocks the thread making the call until some other thread releases it by calling `signal` on the same event flag instance.

Visibility

Public

Signature

```
virtual void wait (int tminms = -1) = 0;
```

Parameters

`tminms`

The length of time, in milliseconds, that the thread should remain blocked. The default value is `-1`, which means to wait indefinitely.

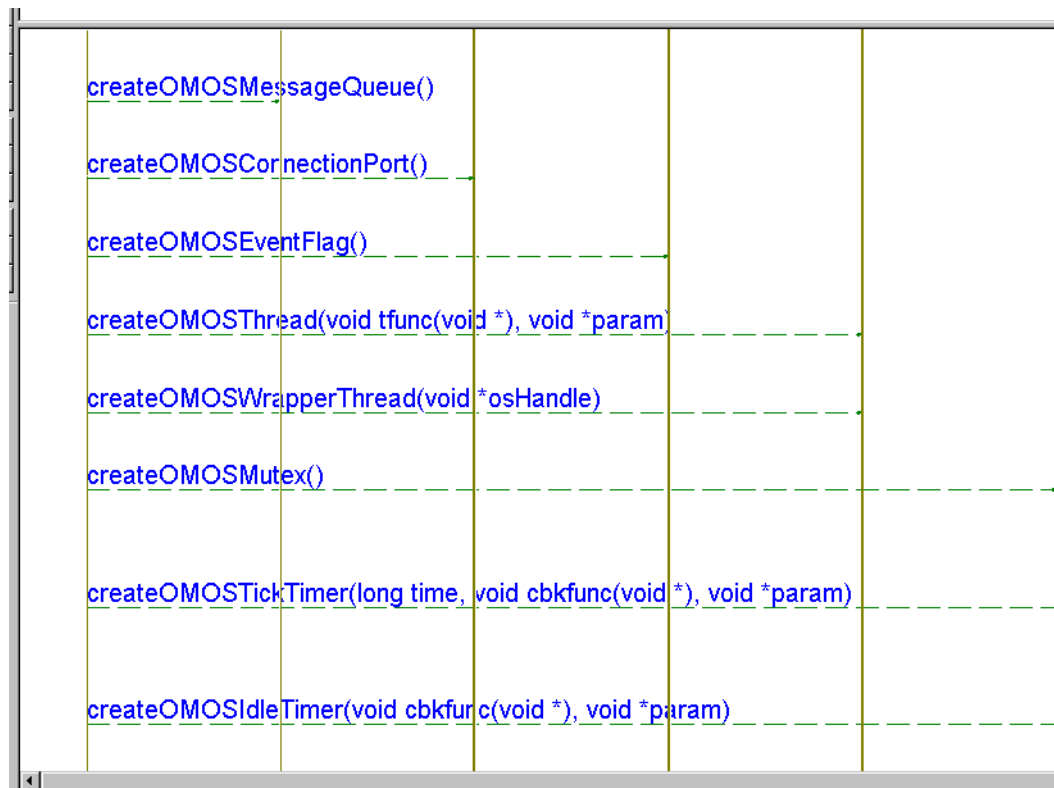
Notes

If an operating system does not support the ability to wait on an event flag with a timeout (for example, Solaris), the Rational Rhapsody framework implements `wait` with timeouts by slicing the time to 50 ms intervals, then checks every 50 ms to see if the event flag was signaled.

OMOSFactory Class

Each concrete `OSFactory` inherits publicly from the abstract class `OMOSFactory`. `OMOSFactory` hides the RTOS mechanisms for tasking and synchronization. In addition, the `OSFactory` provides other operating system-dependent services that the Rational Rhapsody framework requires, such as obtaining a handle to the current thread.

The following sequence diagram shows the `OSFactory` creating various operating system entities, such as `OMOSMessageQueue` and `OMOSConnectionPort`.



Construction Summary

instance	Creates a single instance of the <code>OMOSFactory</code>
--------------------------	-----------------------------------------------------------

Method Summary

createOMOSConnectionPort	Creates a connection port
createOMOSEventFlag	Creates an event flag
createOMOSIdleTimer	Creates an idle timer
createOMOSMessageQueue	Creates a message queue
createOMOSMutex	Creates a mutex
createOMOSSemaphore	Creates a semaphore
createOMOSThread	Creates a thread
createOMOSTickTimer	Creates a tick timer
createOMOSWrapperThread	Creates a wrapper thread
delayCurrentThread	Delays the current thread for the specified length of time
getCurrentThreadHandle	Gets the handle to the current thread
waitOnThread	Waits on the thread for the specified length of time

instance

This method creates a single instance of `OMOSFactory`. This function must be implemented for a given RTOS to return a pointer to the operating system adapter factory designed specifically for that RTOS.

Visibility

Public

Signature

```
static OMOFactory* instance();
```

Notes

To create an operating system entity, you call one of the methods through the pointer returned by `instance`. For example, to create an event flag, use the following call:

```
instance()->createOMOSEventFlag()
```

createOMOSConnectionPort

This method creates a connection port.

Visibility

Public

Signature

```
virtual OMOSConnectionPort* createOMOSConnectionPort()  
    = 0;
```

Return

The new connection port

createOMOSEventFlag

This method creates an event flag.

Visibility

Public

Signature

```
virtual OMOSEventFlag* createOMOSEventFlag() = 0;
```

Return

The new event flag

createOMOSIdleTimer

This method creates an idle timer.

Visibility

Public

Signature

```
virtual OMOSTimer* createOMOSIdleTimer(  
    void cbkfunc (void *), void *param) = 0;
```

Parameters

cbkfunc

The callback function

param

The parameters for the callback function

Return

The new idle timer

createOMOSMessageQueue

This method creates a message queue.

Visibility

Public

Signature

```
virtual OMOSMessageQueue* createOMOSMessageQueue(  
    OMBoolean shouldGrow = TRUE,  
    const long messageQueueSize =  
    OMOSThread::DefaultMessageQueueSize) = 0;
```

Parameters

shouldGrow

A Boolean value that determines whether the size of the message queue can be increased to yield more room

messageQueueSize

The default size of the message queue

Return

The new message queue

createOMOSMutex

This method creates a mutex.

Visibility

Public

Signature

```
virtual OMOSMutex* createOMOSMutex() = 0;
```

Return

The new mutex

createOMOSSemaphore

This method creates a semaphore.

Visibility

Public

Signature

```
virtual OMOSSemaphore* createOMOSSemaphore(  
    unsigned long semFlags = 0, unsigned long  
    initialCount = 1, unsigned long maxCount = 1,  
    const char * const name = NULL) = 0;
```

Parameters

semFlags

The semaphore flags

initialCount

The initial count of tokens available on the semaphore

maxCount

The maximum number of tokens

name

The name of the semaphore

Return

The new semaphore

createOMOSThread

This method creates a thread.

Visibility

Public

Signature

```
virtual OMOSThread* createOMOSThread (void tfunc(void *),  
    void *param, const char* const threadName = NULL,  
    const long stackSize = OMOSThread::DefaultStackSize)  
    = 0;
```

Parameters

tfunc

The thread function

param

The parameters for tfunc

threadName

The name of the thread

stackSize

The stack size

Return

The new thread

createOMOSTickTimer

This method creates a new tick timer.

Visibility

Public

Signature

```
virtual OMOSTimer* createOMOSTickTimer (timeUnit time,  
    void cbkfunc(void *), void *param) = 0;
```

Parameters

time

The time between ticks

cbkfunc

The callback function

param

The parameters for cbkfunc

Return

The new tick timer

createOMOSWrapperThread

This method creates a wrapper thread.

Visibility

Public

Signature

```
virtual OMOSThread* createOMOSWrapperThread(  
    void* osHandle) = 0;
```

Parameters

osHandle

The handle to the operating system

Return

The new wrapper thread

delayCurrentThread

This method delays the current thread for the specified length of time.

The `OXFTDelay(timInMs)` macro provides a convenient shortcut for calling [delayCurrentThread](#).

Visibility

Public

Signature

```
virtual void delayCurrentThread (timeUnit ms) = 0;
```

Parameters

ms

The length of time, in milliseconds, to delay processing on the current thread

getCurrentThreadHandle

This method returns the native operating system handle to the thread. This handle is used to identify a thread or to apply operating system-specific operations to a thread.

Visibility

Public

Signature

```
virtual void* getCurrentThreadHandle() = 0;
```

Return

The `OSThreadHandle`

waitOnThread

This method waits for a thread to terminate.

Visibility

Public

Signature

```
virtual OMBoolean waitOnThread (void* osHandle,  
                                timeUnit ms) = 0;
```

Parameters

osHandle

The operating system handle

ms

The length of time to wait, in milliseconds

Return

The method returns one of the following Boolean values:

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

OMOSMessageQueue Class

An important building block for the execution framework class `OMThread`, the message queue is initially empty. The factory's `createOMOSMessageQueue` method creates an operating system message queue.

The default message queue size is set by the static constant variable `OMOSThread::DefaultMessageQueueSize`. You can override the default value by passing a different value as the second argument to the factory's `createOMOSMessageQueue` method when you create the message queue.

The maximum length of the message queue is operating system- and implementation-dependent. It is usually set in the adapter for a particular operating system.

Construction Summary

<u>~OMOSMessageQueue</u>	Destroys the <code>OMOSMessageQueue</code> object
------------------------------------------	---------------------------------------------------

Method Summary

<u>get</u>	Retrieves the message at the beginning of the queue
<u>getMessageList</u>	Retrieves the list of messages
<u>getOsHandle</u>	Returns the native operating system handle to the thread
<u>isEmpty</u>	Determines whether the queue is empty
<u>isFull</u>	Determines whether the queue is full
<u>pend</u>	Blocks the thread making the call until there is a message in the queue
<u>put</u>	Adds a message to the queue
<u>setOwnerProcess</u>	Sets the thread that owns the message queue

~OMOSMessageQueue

This method destroys the `OMOSMessageQueue` object. You must declare the destructor explicitly, rather than letting the compiler add it, because it must be made virtual.

Visibility

Public

Signature

```
virtual ~OMOSMessageQueue()
```

get

This method retrieves the message at the beginning of the queue.

Visibility

Public

Signature

```
virtual void *get() = 0;
```

Return

The first message in the queue

getMessageList

This method retrieves the list of messages. It is used for two reasons:

- ◆ To cancel events.
When a reactive class is destroyed, it notifies its thread to cancel all events in the queue that are targeted for that reactive class. The thread iterates over the queue, using `getMessageList` to retrieve the data, and marks all events whose target is the reactive class as canceled.
- ◆ To show the data in the event queue during animation.

Visibility

Public

Signature

```
virtual void getMessageList (OMList<void*>& c) = 0
```

Parameters

c

The list of messages in the event (message) queue.

The list is of type `OMList<void*>`, a parameterized type defined in `oxf\omlist.h` that encapsulates all the operations typically performed on lists, such as adding items to the list and removing items from the list.

getOsHandle

This method returns the native operating system handle to the thread. This handle is used to identify a thread or to apply operating system-specific operations to a thread.

Visibility

Public

Signature

```
virtual void* getOsHandle() const = 0;
```

Return

The handle

isEmpty

This method determines whether the message queue is empty.

Visibility

Public

Signature

```
virtual int isEmpty() = 0;
```

Return

The method returns one of the following values:

- ◆ 0—The queue is not empty.
- ◆ 1—The queue is empty.

isFull

This method determines whether the queue is full.

Visibility

Public

Signature

```
virtual OMBBoolean isFull() = 0;
```

Return

The method returns one of the following values:

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

pend

This method blocks the thread making the call until there is a message in the queue. A reader generally waits until the queue contains a message that it can read.

Visibility

Public

Signature

```
virtual void pend() = 0;
```

put

This method adds a message to the end of the message queue.

Visibility

Public

Signature

```
virtual OMBoolean put (void* m, OMBoolean fromISR = FALSE)  
    = 0;
```

Parameters

m

The message to be added to the queue.

fromISR

A Boolean value that specifies whether the message being added was generated from an interrupt service routine (ISR). The default value is `FALSE`.

Return

The method returns one of the following Boolean values:

- ◆ `TRUE`—The method successfully added the message to the queue.
- ◆ `FALSE`—The method was unsuccessful.

setOwnerProcess

This method sets the thread that owns the message queue. This operation was added to support the OSE environment.

Visibility

Public

Signature

```
virtual void setOwnerProcess (void* handle)
```

Parameters

handle

The handle to the owner process

OMOSMutex Class

The factory's `createOMOSMutex` method creates a *mutex*, which stands for *mutual exclusion*. A mutex is the basic synchronization mechanism used to protect critical sections within a thread. Mutexes are used to implement protected objects.

The mutex allows one thread mutually exclusive access to a resource. Mutexes are useful when only one thread at a time can be allowed to modify data or some other controlled resource. For example, adding nodes to a linked list is a process that should only be allowed by one thread at a time. By using a mutex to control the linked list, only one thread at a time can gain access to the list.

The Rational Rhapsody implementation of a mutex is as a recursive lock mutex. This means that the same thread can lock the mutex several times without blocking itself. In other words, the mutex is actually a counted semaphore. When implementing `OMOSMutex` for the target environment, you should implement it as a recursive lock mutex.

Mutexes can be either free or locked (they are initially free). When a task executes a `lock` operation and finds a mutex locked, it must wait. The task is placed on the waiting queue associated with the mutex, along with other blocked tasks, and the CPU scheduler selects another task to execute. If the `lock` operation finds the mutex free, the task places a lock on the mutex and enters its critical section. When any task releases the mutex by calling `free`, the first blocked task in the waiting queue is moved to the ready queue, where it can be selected to run according to the CPU scheduling algorithm.

The same thread can nest `lock` and `free` calls of the same mutex without indefinitely blocking itself. Nested locking by the same thread does not block the locking thread. However, the nested locks are counted so the proper `free` actually releases the mutex.

Construction Summary

~OMOSMutex	Destroys the <code>OMOSMutex</code> object
----------------------------	--------------------------------------------

Method Summary

free	Releases the lock on the mutex
getOsHandle	Returns the native operating system handle to the thread
lock	Locks the mutex
unlock	Releases the lock on the mutex

~OMOSMutex

This method destroys the `OMOSMutex` object. You must declare the destructor explicitly, rather than letting the compiler add it, because it must be made virtual.

Visibility

Public

Signature

```
virtual ~OMOSMutex()
```

free

This method releases the lock, possibly causing the underlying operating system to reschedule threads.

This method provides backward-compatibility support for non-OSE applications.

Visibility

Public

Signature

```
void free() = 0;
```

getOsHandle

This method returns the native operating system handle to the thread. This handle is used to identify a thread or to apply operating system-specific operations to a thread.

Visibility

Public

Signature

```
virtual void* getOsHandle() const = 0;
```

Return

The handle

lock

This method determines whether the mutex is free and reacts accordingly:

- ◆ If the mutex is free, this operation locks it and allows the calling task to enter its critical section.
- ◆ If the mutex is already locked, this operation places the calling task on a waiting queue with other blocked tasks.

Visibility

Public

Signature

```
virtual void lock() = 0;
```

unlock

This method releases the lock, possibly causing the underlying operating system to reschedule threads.

Visibility

Public

Signature

```
virtual void unlock() = 0;
```

OMOSSemaphore Class

A *semaphore* is a synchronization device that 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.

Semaphores are useful in controlling access to a shared resource that can support only a limited number of users. The current count of the semaphore is the number of additional users allowed. When the count reaches zero, all attempts to use the resource controlled by the semaphore are inserted into a system queue and wait until they either time out or the count again rises above zero. The maximum number of users who can access the controlled resource at one time is specified at construction time.

The Rational Rhapsody framework itself does not use semaphores. However, the `OMOSSemaphore` primitive is provided as a service for environments that need it (such as Windows NT and pSOSystem).

Construction Summary

~OMOSSemaphore	Destroys the <code>OMOSSemaphore</code> object
--------------------------------	------------------------------------------------

Method Summary

getOsHandle	Returns the native operating system handle to the thread
signal	Releases a semaphore token
wait	Obtains a semaphore token

~OMOSSemaphore

This method destroys the `OMOSSemaphore` object. You must declare the destructor explicitly, rather than letting the compiler add it, because it must be made virtual.

Visibility

Public

Signature

```
virtual ~OMOSSemaphore()
```

getOsHandle

This method returns the native operating system handle to the thread. This handle is used to identify a thread or to apply operating system-specific operations to a thread.

Visibility

Public

Signature

```
virtual void* getOsHandle() const = 0;
```

Return

The handle

signal

This method releases a semaphore token.

Visibility

Public

Signature

```
virtual void signal() = 0;
```

wait

This method obtains a semaphore token.

Visibility

Public

Signature

```
virtual OMBoolean wait (long timeout = -1) = 0;
```

Parameters

timeout

The number of ticks to lock on a semaphore before timing out. The possible values are < 0 (wait indefinitely); 0 (do not wait), and > 0 (the number of ticks to wait). For Solaris systems, a value of > 0 means to wait indefinitely.

OMOSSocket Class

The `OMOSSocket` class represents the socket through which data is passed between Rational Rhapsody and an instrumented application.

`OMOSSocket` is generally used for animation, but it can also be used for other connections, as long as you provide a host name and port number. `OMOSSocket` represents the client side of the connection, and assumes that somewhere over the network there is a server listening to the connection. You can modify the definition of the `OMOSSocket` class to remove the `_OMINSTRUMENT` macro definition from the relevant places to provide a socket implementation for non-instrumented configurations. In addition, you might need to modify the definition of the `SOCK_LIB` macro inside the `MakeFileContent` property to be similar to that for tracing and animation.

If an animation session appears to hang, it might be because the high volume of messages passed between Rational Rhapsody and the application causes the internal buffer of the socket to fill up, which might cause a major delay in communication between Rational Rhapsody and the application. The solution to this problem is to increase the size of the socket internal buffer, which is 8K by default. For example, in the Windows NT implementation, you can add the following code to the `Create()` function for `NTSocket`:

```
int NTSocket::Create(
    const char* SocketAddress /*= NULL*/,
    unsigned int nSocketPort /*= 0*/)
{
    ...

    if ((theSock = socket(AF_INET, SOCK_STREAM, proto))
        == INVALID_SOCKET)
    {
        NOTIFY_TO_ERROR("Could not create socket\n");
        theSock = 0;
        return 0;
    }
    int internalBufferSizes = 64 * 1024; // 64k
    setsockopt(theSock, SOL_SOCKET, SO_RCVBUF,
               (char*) &internalBufferSizes, sizeof(int));
    setsockopt(theSock, SOL_SOCKET, SO_SNDBUF,
               (char*) &internalBufferSizes, sizeof(int));
    ...
}
```

Note: This solution has been checked for Windows NT systems only.

Construction Summary

~OMOSSocket	Destroys the OMOSSocket object
-----------------------------	--------------------------------

Method Summary

Close	Closes the socket
Create	Creates a new socket
Receive	Receives data through the socket
Send	Sends data through the socket

~OMOSSocket

This method destroys the `OMOSSocket` object. You must declare the destructor explicitly, rather than letting the compiler add it, because it must be made virtual.

Visibility

Public

Signature

```
virtual ~OMOSSocket()
```

Close

This method closes the socket.

Visibility

Public

Signature

```
virtual void Close()
```

Create

This method creates a new socket.

Visibility

Public

Signature

```
virtual int Create (const char* SocketAddress = NULL,  
                  unsigned int nSocketPort = 0) = 0;
```

Parameters

SocketAddress

The socket address. This can be set to a host name that is a character string. The default value is NULL.

nSocketPort

The socket port number. The default value is 0.

Return

The method returns one of the following values:

- ◆ 0—The operation failed.
- ◆ 1—The operation was successful.

Receive

This method receives data through the socket.

Visibility

Public

Signature

```
virtual int Receive (char* lpBuf, int nBufLen) = 0;
```

Parameters

lpBuf

The string buffer in which data will be stored

nBufLen

The length of the buffer

Return

The method returns one of the following values:

- ◆ 0—There was an error.
- ◆ n—The number of bytes read.

Send

This method sends data through the socket.

Visibility

Public

Signature

```
virtual int Send (const char *lpBuf, int nBufLen) = 0;
```

Parameters

lpBuf

A constant string buffer that contains the data to be sent

nBufLen

The length of the buffer

Return

The method returns one of the following values:

- ◆ 0—There was an error.
- ◆ n—The number of bytes written.

OMOSThread Class

The `OMThread` class in the execution framework aggregates `OMOSThread` to provide the basic threading features. The operating system factory's `createOMOSThread` method creates a raw thread. No constructor is declared for `OMOSThread` because any C++ compiler knows how to add a constructor if it not defined explicitly.

`OMOSThread` has the following static constant variables, which provide default values for user-controllable parameters: stack size, message queue size, and thread priority. Each static variable can be initialized with constants whose values can vary depending on the operating system being targeted, as shown in the following table.

Static Constant Variables	Initialization Constants
<code>DefaultStackSize</code>	<code>SMALL_STACK</code> or <code>DEFAULT_STACK</code>
<code>DefaultMessageQueueSize</code>	<code>MQ_DEFAULT_SIZE</code>
<code>DefaultThreadPriority</code>	<code>PRIORITY_HIGH</code> , <code>PRIORITY_NORMAL</code> , or <code>PRIORITY_LOW</code>

Construction Summary

<u><code>~OMOSThread</code></u>	Destroys the <code>OMOSThread</code> object
-------------------------------------------------	---------------------------------------------

Method Summary

<u><code>exeOnMyThread</code></u>	Determines whether the method was invoked from the same operating system thread as the one on which the object is running
<u><code>getOsHandle</code></u>	Retrieves the thread's operating system ID
<u><code>getThreadEndClbk</code></u>	Is a callback function that ends the current operating system thread
<u><code>resume</code></u>	Resumes a suspended thread
<u><code>setEndOSThreadInDtor</code></u>	Determines whether destruction of the <code>OMOSThread</code> class should kill the operating system thread associated with the class
<u><code>setPriority</code></u>	Sets the thread's operating system priority
<u><code>start</code></u>	Starts thread processing
<u><code>suspend</code></u>	Suspends the thread

~OMOSThread

This method destroys the `OMOSThread` object. You must declare the destructor explicitly, rather than letting the compiler add it, because it must be made virtual.

Visibility

Public

Signature

```
virtual ~OMOSThread()
```

exeOnMyThread

This method determines whether the method was invoked from the same operating system thread as the one on which the object is running.

Visibility

Public

Signature

```
virtual OMBoolean exeOnMyThread();
```

Return

The method returns one of the following Boolean values:

- ◆ `TRUE`—The method was invoked from the same operating system thread as the one on which the object is running.
- ◆ `FALSE`—The threads are not the same.

getOsHandle

This method retrieves the thread's operating system ID. This value varies by operating system.

Visibility

Public

Signature

```
virtual void* getOsHandle() const = 0;  
virtual void* getOsHandle (void*& osHandle) const = 0;
```

Parameters

oshandle

The operating system handle

Return

The operating system ID

getThreadEndClbk

This method is a callback function that ends the current operating system thread.

Visibility

Public

Signature

```
virtual void getThreadEndClbk(  
    OMOSThreadEndCallBack * clb_p, void ** arg1_p,  
    OMBoolean onExecuteThread) = 0;
```

Parameters

clb_p

A pointer to the callback function that ends the thread. This can be either `endMyThread()` or `endOtherThread()`. The function pointer is of type `OMOSThreadEndCallBack`, which is defined in `OMOSThread` as follows:

```
typedef void (*OMOSThreadEndCallBack)(void *);
```

arg1_p

The argument for the callback function.

onExecuteThread

Set this to one of the following Boolean values:

TRUE—The object should kill its own thread.

FALSE—Another object should kill the thread.

Notes

On some operating systems, there are different calls to kill the current thread versus killing other threads. For example, on Windows NT, you kill the current thread by generating a new `OMNTCloseHandleEvent`; to kill another thread, you call `TerminateThread`.

The concrete operating system adapter makes sure that other threads are killed first by providing two static thread functions:

- `static void endMyThread(void *)`;
- Implement this method to handle the case in which the object kills its own thread.
- `static void endOtherThread(void *)`;
- Implement this method to handle the case in which another object kills the thread.

The [getThreadEndCbK](#) operation returns the address of either of the static functions `endMyThread` or `endOtherThread`. The implementation of these two functions could be different (as on Windows NT), or the same, as on `pSOSystem`, where both functions call `t_restart`.

resume

This method resumes a suspended thread. This method is not used in generated code—it is used only for advanced scheduling.

The `suspend` and `resume` methods provide a way of stopping and restarting a thread. Threads usually block when waiting for a resource, such as a mutex or an event flag, so both are rarely used.

Visibility

Public

Signature

```
virtual void resume() = 0;
```

setEndOSThreadInDtor

This method determines whether destruction of the `OMOSThread` class should kill the operating system thread associated with the class.

Visibility

Public

Signature

```
virtual void setEndOSThreadInDtor (OMBoolean val) = 0;
```

Parameters

val

This value is determined by the value of the Boolean data member `endOSThreadInDtor`, which must be defined in the `<env>Thread` class that inherits from `OMOSThread`. The possible values are as follows:

TRUE—The thread is ended as part of the object's destruction process.

FALSE—The thread is not ended when the object is destroyed.

setPriority

This method sets the thread's operating system priority.

Visibility

Public

Signature

```
virtual void setPriority (int pr) = 0;
```

Parameters

pr

The integer value of the priority. This parameter varies by operating system.

start

This method starts thread processing. Initially, threads are suspended until `start` is called.

Visibility

Public

Signature

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

suspend

This method suspends the thread. This method is not used in generated code—it is used only for advanced scheduling.

Visibility

Public

Signature

```
virtual void suspend() = 0;
```

OMOSTimer Class

The abstract class `OMOSTimer` is a building block for `OMTimerManager`, which provides basic timing services for the execution framework. In the Rational Rhapsody implementation, the timer runs on its own thread. Therefore, the target operating system must support multi-threading.

Construction Summary

~OMOSTimer	Destroys the <code>OMOSTimer</code> object
----------------------------	--------------------------------------------

Method Summary

getOsHandle	Retrieves the thread's operating system ID
-----------------------------	--------------------------------------------

~OMOSTimer

This method destroys the operating system entity that the instance wraps and stops the timer. You must declare the destructor explicitly, rather than letting the compiler add it, because it must be made virtual.

Visibility

Public

Signature

```
virtual ~OMOSTimer()
```

getOsHandle

This method retrieves the thread's operating system ID. This value varies by operating system.

Visibility

Public

Signature

```
virtual void* getOsHandle() const = 0;
```

Return

The operating system ID

OMTMMessageQueue Class

The `OMTMMessageQueue` class implements a message queue. It is the base class for `OMEventQueue` and `OMMessageQueue`. The base class `OMTMMessageQueue` has an `OMOSMessageQueue`, called `theQueue`, as a protected data member.

Construction Summary

<u>OMTMMessageQueue</u>	Creates an <code>OMTMMessageQueue</code> object.
<u>-OMTMMessageQueue</u>	Destroys the <code>OMTMMessageQueue</code> object

Method Summary

<u>get</u>	Retrieves the message at the beginning of the queue
<u>getMessageList</u>	Retrieves the list of messages
<u>getOsHandle</u>	Returns the native operating system handle to the thread
<u>isEmpty</u>	Determines whether the queue is empty
<u>pend</u>	Blocks the thread making the call until there is a message in the queue
<u>putVisibility</u>	Adds a message to the queue

OMTMMessageQueue

This method is the constructor for the OMTMMessageQueue class. It allocates theQueue, the OMOSMessageQueue member of OMTMMessageQueue, with a given size and the ability to grow dynamically. In addition, it initializes the following:

- ◆ messageQueueSize—If not overridden, the message queue size is initialized to the value of the static constant DefaultMessageQueueSize in OMOSThread.
- ◆ dynamicMessageQueue—If the default value of TRUE is not overridden, the message queue size is dynamic rather than fixed.

Visibility

Public

Signature

```
OMTMMessageQueue (const long messageQueueSize =  
    OMOSThread::DefaultMessageQueueSize,  
    OMBoolean dynamicMessageQueue = TRUE)
```

Parameters

messageQueueSize

The initial size of the queue

dynamicMessageQueue

A Boolean value that specifies whether the queue is dynamic or fixed

~OMTMMessageQueue

This method deletes memory allocated for the message queue. You must declare the destructor explicitly, rather than letting the compiler add it, because it must be made virtual.

Visibility

Public

Signature

```
virtual ~OMTMMessageQueue ()
```

get

This method calls the message queue's `get` operation to retrieve the first message in the queue.

Visibility

Public

Signature

```
virtual Msg *get()
```

Return

The first message in the queue

getMessageList

This method calls the message queue's `getMessageList` operation to retrieve the list of messages.

Visibility

Public

Signature

```
virtual void getMessageList (OMList<Msg*>& l)
```

Parameters

1

The list of messages in the event (message) queue.

The list is of type `OMList<void*>`, a parameterized type defined in `oxf\omlist.h` that encapsulates all the operations typically performed on lists, such as adding items to the list and removing items from the list.

getOsHandle

This method calls the message queue's `getOsHandle` operation to retrieve the native operating system handle to the thread. This handle is used to identify a thread or to apply operating system-specific operations to a thread.

Visibility

Public

Signature

```
virtual void* getOsHandle() const
```

Return

The handle

isEmpty

This method calls the message queue's `isEmpty` operation to determine whether the queue is empty.

Visibility

Public

Signature

```
virtual int isEmpty()
```

Return

The method returns one of the following values:

- ◆ 0—The queue is not empty.
- ◆ 1—The queue is empty.

pend

This method calls the message queue's `pend` operation to block the caller until there is a message in the queue.

Visibility

Public

Signature

```
virtual void pend()
```

putVisibility

Public

Description

This method calls the message queue's `put` operation to add a message to the end of the queue.

Signature

```
virtual OMBBoolean out (Msg *m, OMBBoolean fromISR = FALSE)
```

Parameters

`m`

The message to be added to the queue.

`fromISR`

A Boolean value that specifies whether the message being added was generated from an interrupt service routine (ISR). The default value is `FALSE`.

Return

The method returns one of the following Boolean values:

- ◆ `TRUE`—The method successfully added the message to the queue.
- ◆ `FALSE`—The method was unsuccessful.

Rebuilding the Rational Rhapsody Framework

When you modify your Rational Rhapsody-built application to operate in a different target environment, you must rebuild the Rational Rhapsody framework for that target environment. Because language objects are compiler-specific, you must rebuild these libraries—even if you move from one Windows-based environment to another, such as Borland.

You might need to reinstall Rational Rhapsody before you rebuild the Rational Rhapsody framework. You should reinstall Rational Rhapsody in the following situations:

- ◆ The source files for the framework were not included in your original installation.
- ◆ You installed Rational Rhapsody for a different environment other than the new compiler or environment you now want to target.
- ◆ You installed Rational Rhapsody before installing the new compiler or environment.

During the reinstallation, be sure to select the correct target environment. This enables Rational Rhapsody to prepare the appropriate make (.mak) file for your target environment. Note that reinstalling Rational Rhapsody does not erase your license file or any projects you have under the Rational Rhapsody root directory.

This section describes how to rebuild the Rational Rhapsody framework for the different supported adapters for Windows systems for Rational Rhapsody Developer for C and C++.

Note

Refer to the IBM Rational Rhapsody Release Notes for detailed information about the supported environments.

Borland

To rebuild the Rational Rhapsody framework for the Borland environment, follow these steps:

1. Make sure the file <Borland_dir>\bin\Bcc32.cfg contains the following lines:

```
-I<Borland_Dir>\include  
-L<Borland_Dir>\lib
```

2. Make sure the file `<Borland_dir>\bin\ilink32.cfg` contains the following line:

```
-L"<Borland_Dir>\lib"
```

3. Set following environment variables:

```
set BCROOT=<Borland_install>
set PATH=%BCROOT%\Bin;%PATH%
```

4. Navigate to the `<Rhapsody_install>\Share\Lang<lang>` directory and execute the following command:

```
make -f bc5build.mak
```

5. If you are going to webify your model, add `%BCROOT%\Bin` to your system variables.

INTEGRITY

To rebuild the Rational Rhapsody framework for the INTEGRITY environment (C++ only), follow these steps:

1. Edit the `<Rhapsody_install>\Share\LangCpp\IntegrityBuild.bat` file to set the option `:target` to the target BSP name. For example:

```
:target=mbx800
```

2. Pass the INTEGRITY environment path and target BSP name as command-line parameters to the `IntegrityBuild.bat` file and run this batch file to build all the libraries for the specified target BSP.

For example, to build libraries for `mbx800`, use the following command:

```
<Rhapsody_install>\Share\LangCpp\IntegrityBuild.bat
C:\GHS\int404 mbx800
```

This command builds the following debug libraries for INTEGRITY under the directory `<Rhapsody_install>\Share\LangCpp\lib`:

- a. `IntegrityOxfMbx800.a`
- b. `IntegrityOxfInstMbx800.a`
- c. `IntegrityAomAnimMbx800.a`
- d. `IntegrityOmComApplMbx800.a`
- e. `IntegrityAomTraceMbx800.a`
- f. `IntegrityTomTraceMbx800.a`
- g. `IntegrityOxfInstTraceMbx800.a`

h. IntegrityWebComponentsMbx800.a

In addition, the build generates the following debug information files for each debug library:

i. IntegrityOxfMbx800.dba

j. IntegrityOxfInstMbx800.dba

k. IntegrityAomAnimMbx800.dba

l. IntegrityOmComApplMbx800.dba

m. IntegrityAomTraceMbx800.dba

n. IntegrityTomTraceMbx800.dba

o. IntegrityOxfInstTraceMbx800.dba

p. IntegrityWebComponentsMbx800.dba

Once the libraries are built, you can compile, build, and run the Rational Rhapsody samples.

Compiling and Building a Rational Rhapsody Sample

To compile and build a Rational Rhapsody sample in the INTEGRITY environment, follow these steps:

1. Start Rational Rhapsody and open the project. For example:

```
<Rhapsody_install>\Samples\CplusplusSamples\Dishwasher.rpy
```

2. Select **File > Project Properties**.
3. Set the `CPP_CG::INTEGRITY::RemoteHost` property to the IP address of the machine on which Rational Rhapsody is running. (To get the IP address under the Windows environment, enter the following command at the command prompt:

```
ipconfig
```

4. Set the active configuration for the sample. For example, for the Dishwasher sample, set `EXE::Host` as the active configuration.
5. Open the Features dialog box for the active configuration and set the following values:
 - a. Set the **Instrumentation Mode** field to **Animation**.
 - b. Set the **Environment** field to **INTEGRITY**.
6. Select the Properties tab, then click the All filter.

7. Set the `CPP_CG::INTEGRITY::BLDTarget` property to set the target BSP. By default, this value is set to `mbx800`. If desired, set this to a different value.

You can set additional options and defines by changing the `BLDAdditionalOptions` and `BLDAdditionalDefines` properties.

8. Click **OK** to apply your changes and dismiss the dialog box.
9. Select **Code > Generate <configuration>** to generate the code and the build file for the active configuration.
10. Select **Code > Build <ActiveComponent>.mod** to compile and link the application source code. This will generate the following `INTEGRITY` executable files:
 - a. `<ActiveComponent>.mod`—This is a dynamically download type of image. This image can be downloaded on a running kernel on the target board using the `TFTP` server utility.
 - b. `<ActiveComponent>`—This is an Integrity Application type of image. This image must be integrated with the kernel to form a composite image that can be downloaded on the target using the `ocdserv` utility.

In these names, *ActiveComponent* is the name of the component currently selected as the active component within Rational Rhapsody.

Downloading the Image and Running the Application

To run the sample, perform all the steps described in the following sections.

Modifying the Files

Perform the following steps:

1. Edit the `Default.ld` file in the `<GreenDir>\mbx800 dsp` directory as follows:
 - a. Increase the `.heap` section to 1Mb (`0x100000`).
 - b. Increase the `.download` section to 1.5Mb (`0x180000`).
2. Edit the `Integrity.ld` file in `<GreenDir>` directory to increase the `.heap` section to 256K (`0x40000`). This is used for application build. You can check it in the `.map` file of the application.

Building the Kernel

To build the kernel, follow these steps:

1. From the Windows Start menu, invoke the ADAMULTI IDE.

2. Select **File > Open Project in Builder**, navigate to the `mbx800 BSP` directory under your Green Hills installation (for example, `<GreenDir>\mbx800`), select the project `default.bld`, and open it.
3. Navigate to the project `kernel.bld` and double-click on it. You will see a `global_table.c` file. You must modify this file according to your board specifications. Make the following changes:

- a. Uncomment the following statement:

```
#define HARD_CODE_NETWORK_CONFIGURATION
```

- b. Define the ethernet address for your board. For example:

```
#define ETHERADDR 0x00, 0x01, 0xAF, 0x01, 0x10, 0xCC
```

- c. Define the IP address of the board. For example:

```
#define IP1 194  
#define IP2 90  
#define IP3 28  
#define IP4 151
```

- d. Define the gateway for the board. For example:

```
#define GW1 194  
#define GW2 90  
#define GW3 28  
#define GW4 1
```

- e. Set the netmask. For example:

```
#define NM1 255  
#define NM2 255  
#define NM3 252  
#define NM4 0
```

- f. Make sure the target board using TCP/IP is on the same subnet as any system with which it communicates.

4. Select **Project > FileOptions** for `kernel.bld`. Set the libraries option as follows:

- a. Remove the `log` library.
- b. Add the `tcpip` library.

5. Select **Build > Rebuild all**. This command rebuilds your kernel.

Downloading the Kernel Dynamically

Because two different executable files are created during code generation, there are two ways to download the kernel on the target board. The following sections describe both methods.

To download the kernel on the target board, follow these steps:

1. Make sure the variable `on_board_ram_size` in the file `<GreenDir>\mbx800\mbx800.ocd` is 16 (for the MBX860 board).
2. Select **Target > Connect to Target**. The Connection Chooser command window opens.
3. Enter the following command, then click **OK**:

```
ocdserv lpt1 ppc800 -s <GreenDir>\mbx800\mbx800.ocd
```

4. Select **Debug > Debug kernel** to open the Debug window.
5. Click the **GO** toolbar button to download the kernel on to the board and run it.
6. Invoke another instance of ADAMULTI IDE.
7. Select **Target > Connect to Target** to open the Connection Chooser command window.
8. Enter the following command, then click **OK**:

```
rtserv -port udp@<hostname>
```

In this command, *hostname* is the IP address of the target board. For example:

```
rtserv -port udp@194.90.28.151
```

This command opens the Task window. You can see some kernel tasks running in the kernel space on the Task window. Select **Target > Show Target** windows to see IO and target windows.

9. From the Windows Start menu, invoke the TFTP server.
10. Set the base directory in the TFTP server window to the directory where the images are generated (for example, `<Rhapsody_install>\Samples\CppSamples`).
11. In the `rtserv` Task window, select **Target > Load Module**.
12. Navigate to the path where the dynamically download image (`*.mod`) was generated and select **load**.

Ensure that the TFTP server is running or the download process will be very slow. You can see the download status on the `rtserv` target window. Once the image has been successfully downloaded, the Initial Task will be visible in the `rtserv` Task window in the virtual address space.

Integrating INTEGRITY Application Images with the Kernel

To integrate the INTEGRITY application image with the kernel, follow these steps:

1. Open an application command window and change directory to the directory where the INTEGRITY application image was created.
2. Enter the following command:

```
C:> \..\<path> <GreenDir>\intex -dbo
- lang_7=<executable name>
-kernel=<Target BSP path>\kernel
-target=<Target BSP Path>\default.bsp OutputFileName
```

In this command:

- a. <path> = The path to the application image
- b. <executable name> = Host
- c. <Target BSP Path> = <GreenDir>\mbx800
- d. OutputFileName = Dishwasher

For example:

```
C:\..\Dishwasher\EXE <GreenDir>\intex -dbo
-lang_7=Dishwasher-
kernel=<GreenDir>\mbx800\kernel
-target=<GreenDir>\mbx800\default.bsp Dishwasher
```

3. Invoke the ADAMULTI IDE.
4. Select **Remote > Connect to Target** to open the Remote command window.

Enter the following command:

```
ocdserv lpt1 ppc800 -s <GreenDir>\mbx800\mbx800.ocd
```

The execution of this command opens two windows—the Target window and the IN/OUT window.

5. Select **Debug > Debug Other** and navigate to the path where your Integrity Application image was created, then click **Debug**. This opens the debug window.
6. Click on the toolbar button GO to start downloading your composite image of "Kernel+Application" on the board.
7. Invoke another instance of the ADAMULTI IDE.

8. Select **Remote > Connect to Target** to open the Remote command window.
9. Enter the following command:

```
rtserv -port udp<hostname>
```

In this command, *hostname* is the IP address of the target board. For example:

```
rtserv -port udp@194.90.28.151
```

The execution of this command invokes three windows—the `rtserv` Target window, IN/OUT window, and Task window. In the Task window, you can view the kernel space tasks and the virtual address space task (Initial).

10. Double-click on the Initial Task to bring up its debug window. You can see the debug arrow pointing at the main function for the application. Ensure that the same application is opened in Rational Rhapsody.

Animating the Image

To run the application, follow these steps:

1. Double-click on the Initial Task to bring up its debug window. You can see the debug arrow pointing to the application's main function. Ensure that the same application is open in Rational Rhapsody.
2. To execute this application, click the toolbar button **GO**. You should be able to see the animation toolbar come up in Rational Rhapsody. You can generate events in Rational Rhapsody using the animation toolbar. If there is console output, it is displayed on the rtserv IN/OUT window; animation is displayed in the Rational Rhapsody window.
3. After the execution is complete, quit from animation.
4. The task window shows that the Initial Task and its tcpip client are still alive; these tasks must be killed manually. Close the Initial Task debug window; in the message window, select **QuitandKillProcess** to kill your initial task.
5. In the Task window, in the kernel space, double-click the Client00X (X=1..) task to display the debug window of this task. Close the debug window and select **QuitandKillProcess** to kill the client task.
6. In the rtserv Task window, select **Target > Disconnect from Target** to close your rtserv session.
7. To unload the composite image from the target board, go to the first instance of the ADAMULTI IDE that was opened. Select **Remote > Disconnect from Target** to close your ocdserv session.
8. Close the Debug window of the ocdserv.

Linux

Rational Rhapsody Developer for C++ provides support for the Linux operating system. The following sections describe how to build the Linux libraries, and how to generate Linux code using Rational Rhapsody.

Building the Linux Libraries

You build the Linux libraries on the target machine. Copy the `linuxshare.tar` file installed in the `Share\LangCpp` directory on the host to the Linux machine.

To build the libraries, follow these steps:

1. Change directory to `Share/LangCpp`.
2. Build the libraries using the following command:

```
gmake -f linuxbuild.mak
```

3. Verify that the following library files were created in the directory `Share/LangCpp/lib`:
 - a. `linuxaomanim.a`
 - b. `linuxaomtrace.a`
 - c. `linuxomcoappl.a`
 - d. `linuxoxf.a`
 - e. `linuxoxfinst.a`
 - f. `linuxtomtrace.a`

Creating and Running Linux Applications

You compile, link, and run your Linux application on the Linux machine.

Perform the following steps:

1. Create the Rational Rhapsody project on the host, and select the Linux environment on the configuration's Settings tab.
2. Transfer the generated directory with the sources, headers, and makefiles from the host to the Linux machine (for example, by using `ftp`).
3. On the Linux machine, edit the makefile (`*.mak`) to change the following setting:

```
OMROOT=[LangCPP_Dir]
```

In this syntax, *[LangCPP_Dir]* is the path to the `Share/LangCpp` directory.

4. To compile and link the application, enter the following command:

```
gmake -f xxx.mak
```

In this command, `xxx.mak` is the name of the generated makefile.

5. An executable is created in the current directory. When you run the executable on the target, the Rational Rhapsody animation toolbar opens on the host (for applications using instrumentation).

MultiWin32

To rebuild the Rational Rhapsody OXF for the `MultiWin32` environment (C++ only), follow these steps:

1. Open an application command prompt window.
2. Change directory to `$OMROOT\LangC++`.
3. Assuming that the Green Hills home directory is `C:\GHS`, enter the following commands:

```
> MultiWin32Build.bat C:\GHS\nat35 clean
> MultiWin32Build.bat C:\GHS\nat35
```

You must perform a clean before the build to delete previously generated libraries and debug information. Otherwise, the `MULTI` linker generates errors when you build the Rational Rhapsody generated application.

Stepping Through the Generated Application Using MultiWin32

To step through the generated application, follow these steps:

1. On the Settings tab of the features dialog box for the configuration, set the **Build Set** field to one of the following values:

- a. **Debug**—Turns on the debug information. This option adds the following line to the `_program.bld` file:

```
:defines=_DEBUG line
```

- b. **DebugNoExp**—Turns on the debug and exceptions information. This option adds the following lines to the `_program.bld` file:

```
:defines=_DEBUG
:defines=HAS_NO_EXP
```

This is the default value.

- c. **Release**—No debug information.
- d. **ReleaseNoExp**—No debug or exceptions information. This option adds the following line to the `_program.bld` file:

```
:defines=HAS_NO_EXP
```

2. In Rational Rhapsody, select **Code > Generate/Make/Run**.

3. In MULTI, start debugging by selecting **Debug > Debug Other Executable** and select the Rational Rhapsody generated application's .exe file.

Stepping Through the OXF Using MULTI

The OXF libraries provided with the `MultiWin32` environment do not include debug information. To step through the OXF source code using MULTI, you must rebuild the OXF with debug information enabled.

Perform the following steps:

1. Add the following line to the `$OMROOT\LangCpp\MultiWin32Build.bld` file just below the `:"defines=OM_STL"` line:

```
:"defines=_DEBUG
```

2. Follow the steps described in [Stepping Through the Generated Application Using MultiWin32](#) to rebuild the framework libraries and step through the source code.

OSE

This section describes how to rebuild the Rational Rhapsody OXF for the OSE Soft Kernel (OseSFK) for C++ environments only.

Rebuilding the Framework

To rebuild the OXF framework for the OseSFK environment, follow these steps:

1. Open the command prompt.
2. Navigate to the <Rhapsody install>\Share\LangCpp directory.
3. Call `vcvars32`.
4. Enter the following make command:

```
nmake osesfkbuild.mak
```

To rebuild specific framework libraries only, see [Using Command-Line Attributes and Flags](#).

Using Command-Line Attributes and Flags

For both OSE environments, you can rebuild only part of the framework using the attribute `TARGETS`, where the target is one of the following values:

- ◆ `oxflibs`—Builds the `oxf` and `oxfinst` libraries only
- ◆ `aomlibs`—Builds the `aomtrace` and `aomanim` libraries only
- ◆ `omcomlib`—Builds the `omcom` library only
- ◆ `tomlib`—Builds the `tom` library only

For example:

```
dmake -f oseppcbuild.mak TARGETS=oxflibs
```

To build the framework with debug information, use the flag `USE_PDB=TRUE`. For example:

```
nmake osesfkbuild.mak USE_PDB=TRUE
```

Editing the Batch Files

Before you can execute the model, you must edit the batch files, as shown here.

Add the following line to the file<Rhapsody install>\Share\etc\osesfkRun.bat:

```
set LM_LICENSE_FILE=<OSE license file>;
```

For example:

```
set LM_LICENSE_FILE=744@banana;
```

QNX

To rebuild the Rational Rhapsody framework for the QNX environment, follow these steps:

1. Open an application command prompt window.
2. Set the following environment variables:

```
set QNXROOT=<your_QNX_install_dir>
set QNX_TARGET=%QNXROOT%/target/qnx6
set QNX_HOST=%QNXROOT%/host/win32/x86
set QCC_CONF_PATH=%QNX_HOST%/etc/qcc
set LD_LIBRARY_PATH=%QNXROOT%/target/qnx6/lib
set PATH=%QNXROOT%/host/win32/x86/usr/bin;%PATH%
```

3. (as in `qnxcwmake.bat`) Navigate to the directory `<Rhapsody_install>\Share\LangCpp` and execute one of the following commands:

```
make -f qnxcwbuild.mak CPU=ppc CPU_SUFFIX=be PATH_SEP=\\
or
make -f qnxcwbuild.mak CPU=x86 PATH_SEP=\\
```

In the first command (for `ppc`), `CPU_SUFFIX` can be one of the following values:

- a. `be`—Big-endian
- b. `le`—Little-endian

In the second command (for `x86`), do not include the `CPU_SUFFIX` in the command.

If desired, you can specify the `TARGETS` attribute, which enables you to build only part of the framework. The possible targets are as follows:

- c. `oxflibs`
- d. `aomlibs`
- e. `omcomlib`
- f. `tomlib`
- g. `webcomponentslib`

For example:

```
make -f qnxcwbuild.mak CPU=ppc CPU_SUFFIX=be PATH_SEP=\\
TARGETS=oxflibs
```

4. To execute the model, **Generate** and **Make** the model in Rational Rhapsody, then upload or transfer your executable to the QNX machine using the Momentics Eclipse-based IDE or by using `ftp`, and executing the application on the target machine (suitable for `x86`). The following sections describe this step in detail.

Using Momentics

To upload your executable using Momentics, follow these steps:

1. Open Momentics and select **File > Open**.
2. Choose the new Bourne executable. This will create the Momentics project for your model.
3. For the target settings:
 - a. Set the **QNX Linker panel** to carry the tag `-static`.
 - b. Set **Connection settings > Host Name** to your *target* machine name.
4. On the target machine, run the following process:

```
pdebug 10000
```
5. Execute the model in Momentics.

Using ftp

To transfer your executable using `ftp`, follow these steps:

1. Upload the executable to the target machine using your favorite `ftp` client.
2. Change permissions for the executable using the following command:

```
chmod +x EXE
```

3. Execute your model using the following command:

```
./EXE
```

Message Queue Implementation

The default style is a proprietary-style queue. To use POSIX-style queues, follow these steps:

1. In the makefile, add the flag `OM_POSIX_QUEUES` to `ADDED_CPP_FLAGS`.
2. Rebuild the OXF libraries in the framework, as described in [Building the Framework Libraries](#).

VxWorks

To rebuild the Rational Rhapsody framework for the VxWorks environment, follow these steps:

1. Call the file `<Tornado_dir>\host\x86-win32\bin\torVars.bat`. For example:

```
D:\Tornado\host\x86-win32\bin\torVars.bat
```

2. Navigate to the `<Rhapsody_install>\Share\Lang<lang>` directory and execute the following command:

```
make all -f vxbuild.mak CPU=<cputype>  
PATH_SEP=<path_separator>
```

Note: Set the `PATH_SEP` switch, for example, use `\\` for NT.

3. Change the CPU environment variable to the desired CPU. The CPU switch should not be preceded by `-D`. The CPU can be set, for example, to I80386, I80486 (the default), PPC860, or MC6800.

The makefile `vxbuild.mak` rebuilds all of the framework libraries.

Integrated Development Environment (IDE)

The integrated development environment (IDE) interface is a DLL that exports a set of C definitions, structures, and functions. The DLL header is supplied as part of Rational Rhapsody installation (in <root>/Share/DLLs/ideabs.h). Because this file defines an abstract IDE for Rational Rhapsody, you can use it to create your own DLL to interface to other IDEs.

Defines

Defines represent the IDE interface state. The defines are as follows:

- ◆ `OM_IDE_CONNECTED`—The DLL is connected to the IDE.
- ◆ `OM_IDE_EXEC_DOWNLOADED`—The image was downloaded to the target.
- ◆ `OM_IDE_EXEC_RUNNING`—The image is running on the target.
- ◆ `OM_IDE_EXEC_BREAK`—The image is in a breakpoint.

Note

If the IDE is not connected, the state is 0.

Structures

The `OMIDECallbacks` structure stores a set of callback functions to enable the IDE to call Rational Rhapsody. The following callbacks are called by the IDE interface:

- ◆ `ConnectionClosedNotify` notifies Rational Rhapsody when the connection to the IDE is broken
- ◆ `DoAnimationCommand` makes Rational Rhapsody perform user animation commands (for example, Go Step)
- ◆ `DbgBreakpointNotify` notifies Rational Rhapsody of a breakpoint in either animation or the IDE debugger
- ◆ `DbgContinueNotify` notifies Rational Rhapsody that the user continued execution on the IDE debugger
- ◆ `EnableVCRButtons` forces control to pass to the user (in animation)

Functions

The IDE functions called by Rational Rhapsody are as follows:

```
void OMIDESetCallbacks(/*in*/struct OMIDECallbacks*);
```

Sets the callbacks for the IDE interface.

int OMIDEConnect(*/*inout*/char* InOutConnectParam*);

Connects to the debugger IDE.

The `InOutConnectParam` parameter is a string that contains the information needed to establish the connection.

int OMIDEDisconnect();

Closes the connection with the IDE.

int OMIDEDownload(*/*in*/char* fileName*);

Instructs the IDE to download the specified file to the target.

int OMIDEUnload();

Instructs the IDE to unload the image.

int OMIDERun(*/*in*/char* entryPoint,/*in*/char* language*);

Instructs the IDE to run the image.

The parameters are as follows:

- a. `entryPoint`—The entry point. This parameter is set by Rational Rhapsody based on the value of the `<lang>_CG::<Environment>::EntryPoint` property/
- b. `language`—Specifies the application language, such as C or C++.

int OMIDESTop();

Instructs the IDE to stop execution of the image on the target.

int OMIDEEnd();

Is equivalent to sequence of call of `OMIDESTop()`, `OMIDEUnload()`, and `OMIDEDisconnect()`.

int OMIDEGetStatus();

Returns the IDE interface state. See [Defines](#) for the list of possible states.

int OMIDEContinue();

Instructs the IDE to continue execution, after the image reaches a breakpoint.

Makefiles

The process of building an adapter for a new RTOS is not complete until you define the makefiles that are used to build applications in the new environment. To do this, follow these steps:

1. Define a make batch file.
2. Run the batch file used to build and run applications.
3. Redefine the properties that include such information as compile and link switches needed to interact with the application makefile. These properties provide some of the content for the makefile.
4. Specify a template for the generated makefile by redefining the `MakeFileContent` property (under `<lang>_CG:: <Environment>`) for the new environment.

This section describes these steps in detail.

Creating a Make.bat File

Create a batch file that sets the environment and then calls the generated makefile for the application. Name the batch file `<env>make.bat` and save it to the `$OMROOT\etc` directory. This batch file can be used to build both Rhapsody applications and the framework itself (except for Solaris).

Running the Batch File

In some cases, you will need an `<env>Run.bat` in addition to the make batch file (for example, there is a `jdkrun.bat` for Java). This file is used only to run the application, and is saved to the `$OMROOT\etc` directory. The `InvokeExecutable` property, one of the code generation properties to be redefined for the new environment, might execute the run batch file. For example, the `InvokeExecutable` property for the OSE SFK environment calls the `osesfkRun.bat` file, which sets the `LM_LICENSE_FILE` variable for the OSE environment and then calls an executable file. In this case, you can use the `osesfkRun.bat` file to invoke the `osesfkmake.bat` file to build applications for OSE.

Redefining Makefile-Related Properties

The most crucial code generation properties to modify are the ones that interact with the makefile to build and link the framework libraries for the new environment. These properties are found in the specific environment metaclass under the `<lang>_CG` subject for a given language. For example, the code generation properties for VxWorks in C++ are listed under `CPP_CG: :VxWorks`.

The following table lists the properties help build and link code in the new RTOS.

Property	Description
CompileSwitches	Specifies the compiler the switches to be used for any type of build.
CPPCompileCommand	Specifies the environment-specific compilation command used in the makefile. This command is referenced in the makefile via the <code>OMCPPCompileCommandSet</code> variable. If you modified the generated dependencies section of the <code>MakeFileContent</code> property to generate a new <code>.obj</code> file every time you compile, you need to change the <code>CPPCompileCommand</code> property as follows: <pre>" if exist \$OMFileObjPath del \$OMFileObjPath \$(CPP) \$OMFileCPPCompileSwitches / Fo\"\$OMFileObjPath\" \"\$OMFileImpPath\" "</pre>
CPPCompileDebug	Modifies the makefile compile command with switches for building a Debug version of a component.
CPPCompileRelease	Modifies the makefile compile command with switches for building a Release version of a component.
DependencyRule	Specifies how file dependencies for a configuration are generated in the makefile.
FileDependencies	Specifies which framework source files to include when building model elements. The file inclusions are generated in the makefile.
LinkDebug	Specifies the special link switches used to link in Debug mode.
LinkRelease	Specifies the special link switches used to link in Release mode.
LinkSwitches	Specifies the standard link switches used to link in any mode.
ObjCleanCommand	Specifies the environment-specific command used to clean the object files generated by a previous build.

Redefining the MakeFileContent Property

Finally, you must specify a template for the generated makefile by redefining the `MakeFileContent` property (under `<lang>_CG::<Environment>`) for the new environment. The code generator uses the template defined in this property to generate the makefile used to build a specific model.

A makefile has the following sections:

- ◆ Target type
- ◆ Compilation flags
- ◆ Commands definitions
- ◆ Generated macros
- ◆ Predefined macros
- ◆ Generated dependencies
- ◆ Linking instructions

The following sections describe the contents of the makefile in detail.

Target Type

The target type section of the makefile contains the macros needed to build either a Debug or Release version of a configuration.

For example, the default content of the target type section of a C++ makefile for the Microsoft environment is as follows:

```
##### Target type (Debug/Release) #####
#####
CPPCompileDebug=$OMCPPCompileDebug
CPPCompileRelease=$OMCPPCompileRelease
LinkDebug=$OMLinkDebug
LinkRelease=$OMLinkRelease
BuildSet=$OMBuildSet
SUBSYSTEM=$OMSubSystem
COM=$OMCOM
RPFrameWorkDll=$OMRPFrameWorkDll

ConfigurationCPPCompileSwitches=
    $OMReusableStatechartSwitches
    $OMConfigurationCPPCompile Switches

!IF "$(RPFrameWorkDll)" == "True"
ConfigurationCPPCompileSwitches=
    $(ConfigurationCPPCompileSwitches) /D "FRAMEWORK_DLL"
!ENDIF

!IF "$(COM)" == "True"
SUBSYSTEM=/SUBSYSTEM:windows
!ENDIF
```

Compilation Flags

The compilation flags section of the makefile contains the default compilation flags stored in the `CompileSwitches` property.

For example, the default content of the compilation flags section of a C++ makefile for the Microsoft environment is as follows:

```
##### Compilation flags #####
#####
INCLUDE_QUALIFIER=/I
LIB_PREFIX=MS
```

Commands Definitions

The commands definition section of the makefile specifies programs to execute from the makefile.

For example, the default commands definition section of a C++ makefile for the Microsoft environment is as follows:

```
##### Commands definition #####  
#####  
RMDIR = rmdir  
LINK_CMD=link.exe  
LIB_FLAGS=$OMConfigurationLinkSwitches  
LINK_FLAGS=$OMConfigurationLinkSwitches $(SUBSYSTEM) /  
    MACHINE:I386
```

Generated Macros

The generated macros section of the makefile contains a variable that expands to the Rhapsody-generated macros in the makefile. For example:

```
##### Generated macros #####
#####
$OMContextMacros
OBJ_DIR=$OMObjectsDir

!IF "$(OBJ_DIR)"!="
CREATE_OBJ_DIR=if not exist $(OBJ_DIR) mkdir $(OBJ_DIR)
CLEAN_OBJ_DIR= if exist $(OBJ_DIR) $(RMDIR) $(OBJ_DIR)
!ELSE
CREATE_OBJ_DIR=
CLEAN_OBJ_DIR=
!ENDIF
```

The `$OMContextMacros` keyword expands several macros in the makefile. Each makefile macro has its own keyword. You can use these keywords separately to customize the makefile.

The `$OMContextMacros` variable enables you to modify target-specific variables. Replace the `$OMContextMacros` line in the `MakeFileContent` property with the following:

```
FLAGSFILE=$OMFlagsFile
RULESFILE=$OMRulesFile
OMROOT=$OMROOT
CPP_EXT=$OMImplExt
H_EXT=$OMSpecExt
OBJ_EXT=$OMObjExt
EXE_EXT=$OMExeExt
LIB_EXT=$OMLibExt
INSTRUMENTATION=$OMInstrumentation
TIME_MODEL=$OMTimeModel
TARGET_TYPE=$OMTargetType
TARGET_NAME=$OMTargetName
$OMAllDependencyRule
TARGET_MAIN=$OMTargetMain
LIBS=$OMLibs
INCLUDE_PATH=$OMIncludePath
ADDITIONAL_OBJS=$OMAdditionalObjs
OBJS= $OMObjs
```


Predefined Macros

The predefined macros section of the makefile contains other macros than the Rhapsody-generated macros specified in the generated macros section.

For example, part of the default predefined macros section of a C++ makefile for the Microsoft environment is as follows:

```
##### Predefined macros #####
#####

$(OBS) : $(INST_LIBS) $(OXF_LIBS)
LIB_POSTFIX=
!IF "$(BuildSet)"=="Release"
LIB_POSTFIX=R
!ENDIF

!IF "$(TARGET_TYPE)" == "Executable"
LinkDebug=$(LinkDebug) /DEBUG
LinkRelease=$(LinkRelease) /OPT:NOREF
!ELSEIF "$(TARGET_TYPE)" == "Library"
LinkDebug=$(LinkDebug) /DEBUGTYPE:CV
!ENDIF
.
.
.
```

Generated Dependencies

The generated dependencies section of the makefile contains a variable that expands to Rhapsody-generated dependencies and compilation instructions.

For example, the generated dependencies section of a C++ makefile for the Microsoft environment is as follows:

```
##### Generated dependencies #####
#####
$OMContextDependencies

$OMFileObjPath : $OMMainImplementationFile $(OBS)
$(CPP) $(ConfigurationCPPCompileSwitches) /
Fo"$OMFileObjPath" $OMMainImplementationFile
```

Makefile Linking Instructions

The linking instructions section of the makefile contains the predefined linking instructions.

For example, the default linking instructions section of a C++ makefile for the Microsoft environment is as follows:

```
##### Linking instructions #####
#####
$(TARGET_NAME)$ (EXE_EXT): $(OBJS) $(ADDITIONAL_OBJS) $OMFileObjPath
$OMMakefileName $OMModelLibs

    @echo Linking $(TARGET_NAME)$ (EXE_EXT)
    $(LINK_CMD) $OMFileObjPath $(OBJS) $(ADDITIONAL_OBJS) \
    $(LIBS) \
    $(INST_LIBS) \
    $(OXF_LIBS) \
    $(SOCK_LIB) \
    $(LINK_FLAGS) /out:$(TARGET_NAME)$ (EXE_EXT)

$(TARGET_NAME)$ (LIB_EXT) : $(OBJS) $(ADDITIONAL_OBJS) $OMMakefileName
    @echo Building library $@
    $(LIB_CMD) $(LIB_FLAGS) /out:$(TARGET_NAME)$ (LIB_EXT) $(OBJS)
$(ADDITIONAL_OBJS)

clean:
    @echo Cleanup
    $OMCleanOBJS
    if exist $OMFileObjPath erase $OMFileObjPath
    if exist *$(OBJ_EXT) erase *$(OBJ_EXT)
    if exist $(TARGET_NAME).pdb erase $(TARGET_NAME).pdb
    if exist $(TARGET_NAME)$ (LIB_EXT) erase $(TARGET_NAME)$ (LIB_EXT)
    if exist $(TARGET_NAME).ilk erase $(TARGET_NAME).ilk
    if exist $(TARGET_NAME)$ (EXE_EXT) erase $(TARGET_NAME)$ (EXE_EXT)
    $(CLEAN_OBJ_DIR)
```

Java Users

To generate Java JAR files, invoke the `jar` command from the makefile, using the `MakeFileContent` property. You can specify the manifest file as an external file with a text element in it. You can add additional files to the model for completeness.

There is no specialized support for RMI in Rhapsody. Call the JDK and invoke the relevant tools manually, or via the generated makefile (change the `MakeFileContent` property).

Active Behavior Framework

The *active behavioral framework* or Behavioral package consists of a set of collaborative classes that form the fundamental architecture of an object-oriented, reactive, multi-threaded 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.

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.

Creating a Reactive Class that Consumes Events

To create a reactive class that consumes events without a statechart:

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](#).

Creating a Statechart as Documentation

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

To create a statechart for documentation only:

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

Modifying a Class Event Consumption

To add functionality to a class event consumption:

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

OMReactive Class

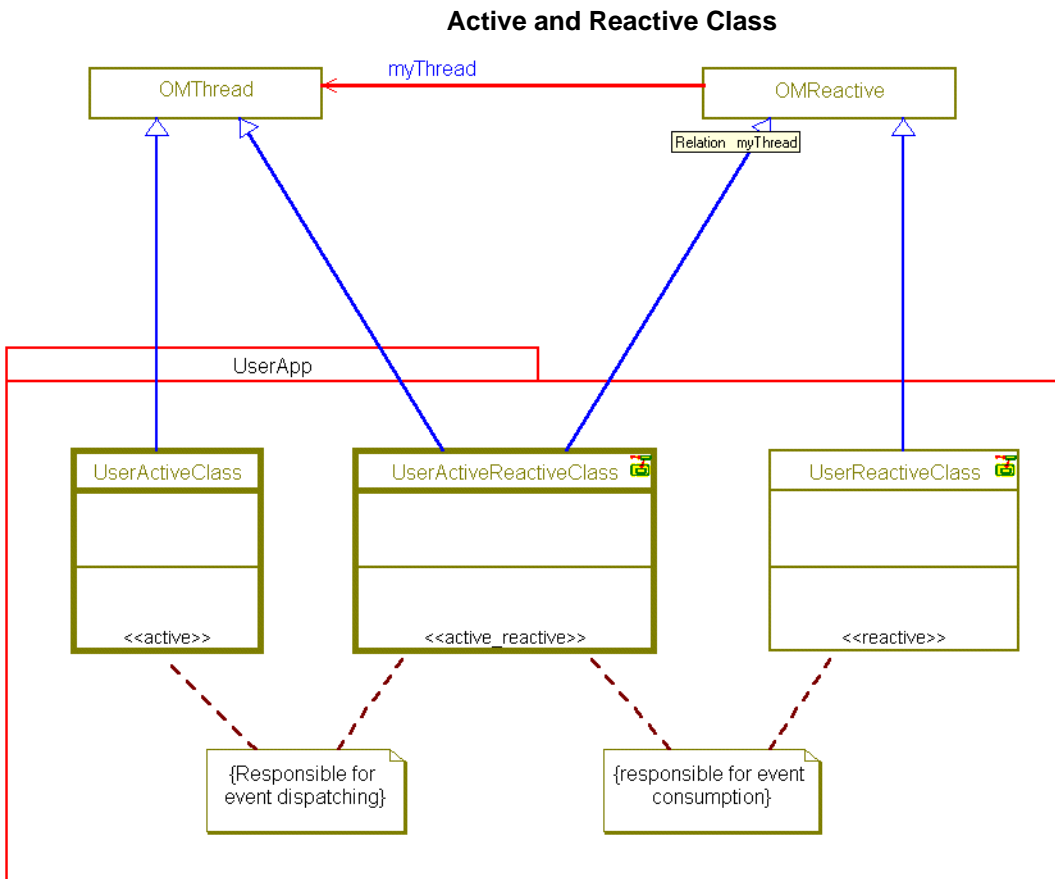
Essentially, a *reactive* class is one that reacts to events; that is, it is an event consumer. A reactive class is represented in the execution framework by the `OMReactive` class (defined in `omreactive.h`), from which every generated reactive class inherits by default. Every reactive class is associated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



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 Classes](#) 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” automatically inherits 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 Classes and Methods](#), 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.

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 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

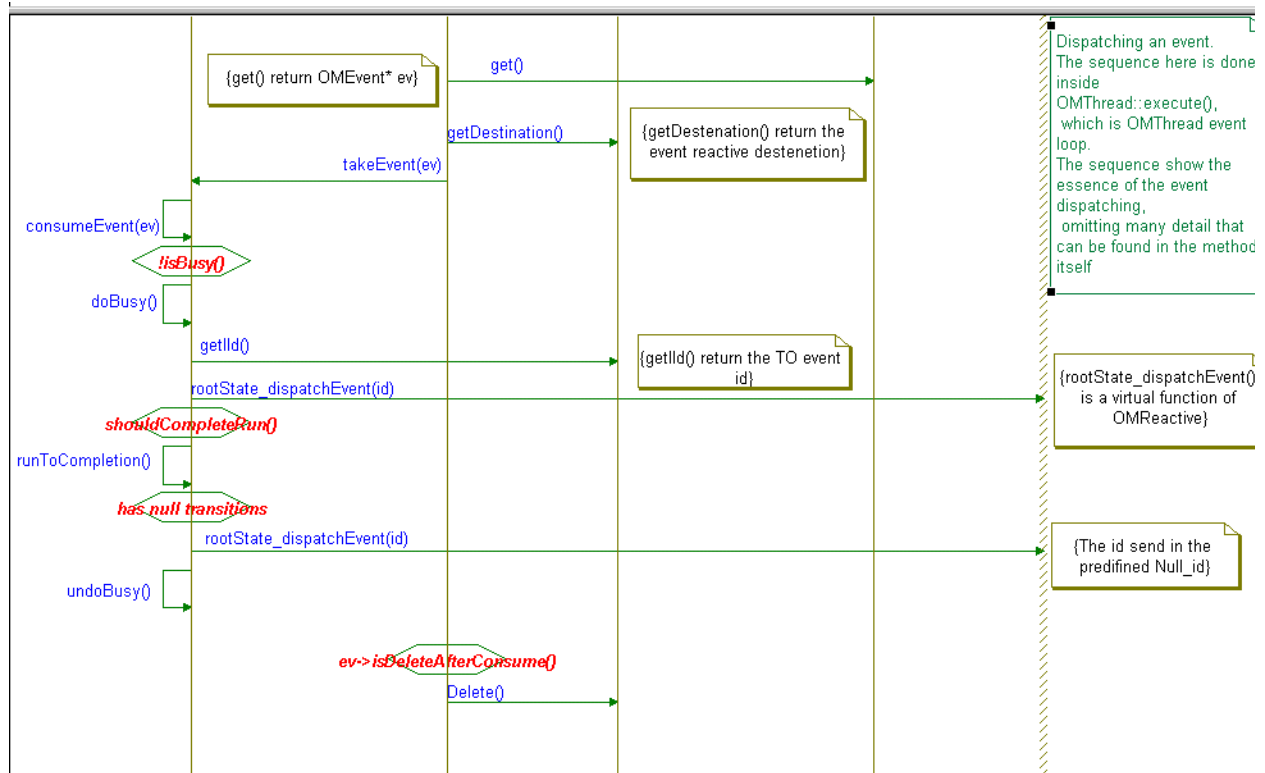
Each class can have *events* and operations defined on it. In the context of statecharts and activity diagrams, events can trigger transitions between states.

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.

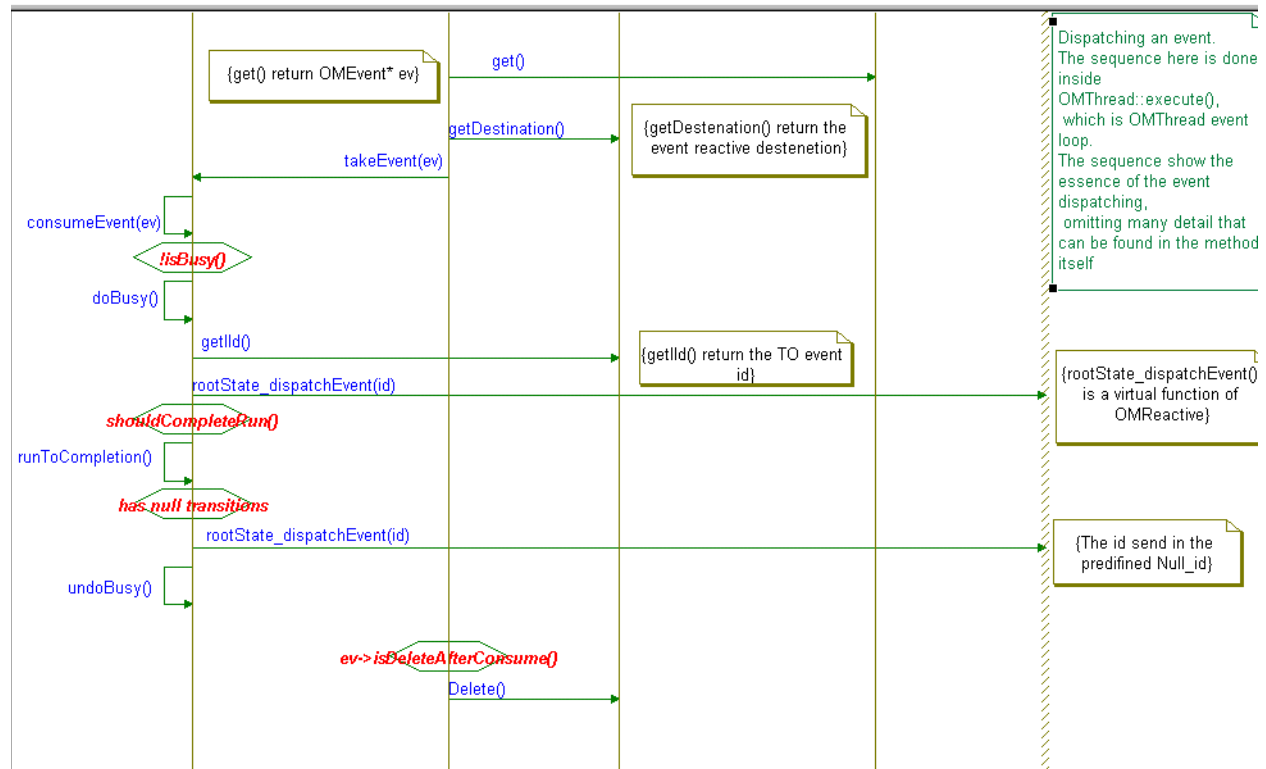


The sequence to generate and queue an event is as follows:

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 [getId](#) 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

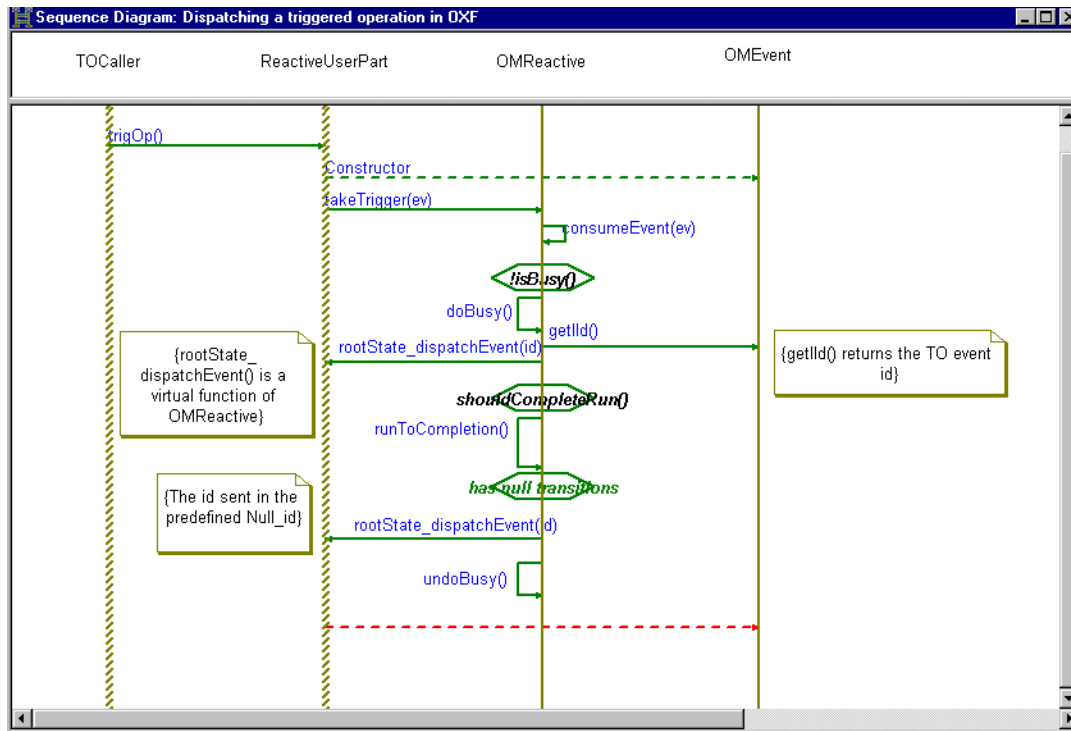
The [cancelEvents](#) method cancels all the events targeted for a specific `OMReactive` instance. It calls `getMessageList` to get a list of all events in the thread's event queue.

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 [getId](#) 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.

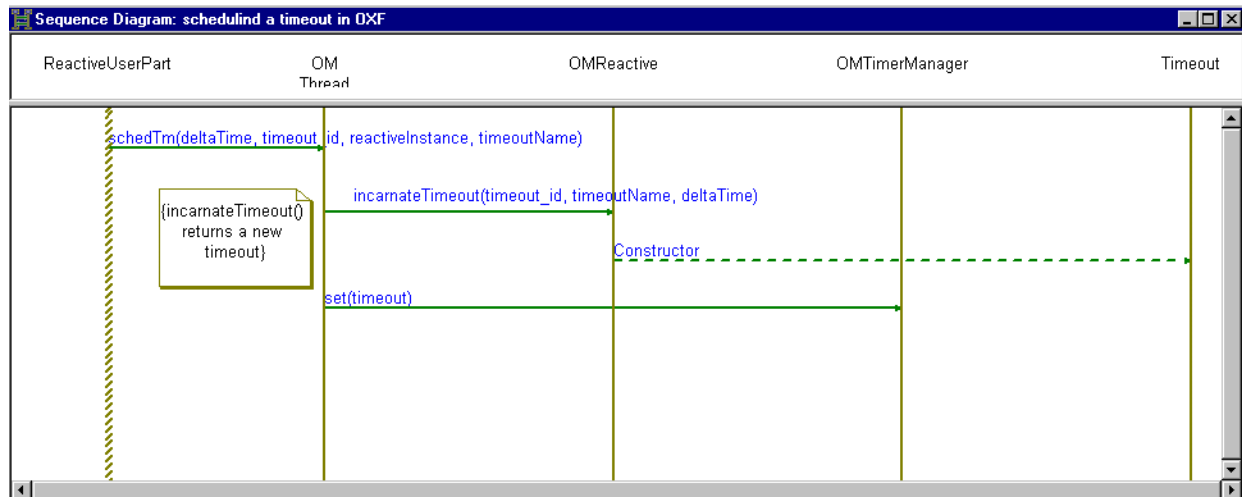
Timeouts

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.

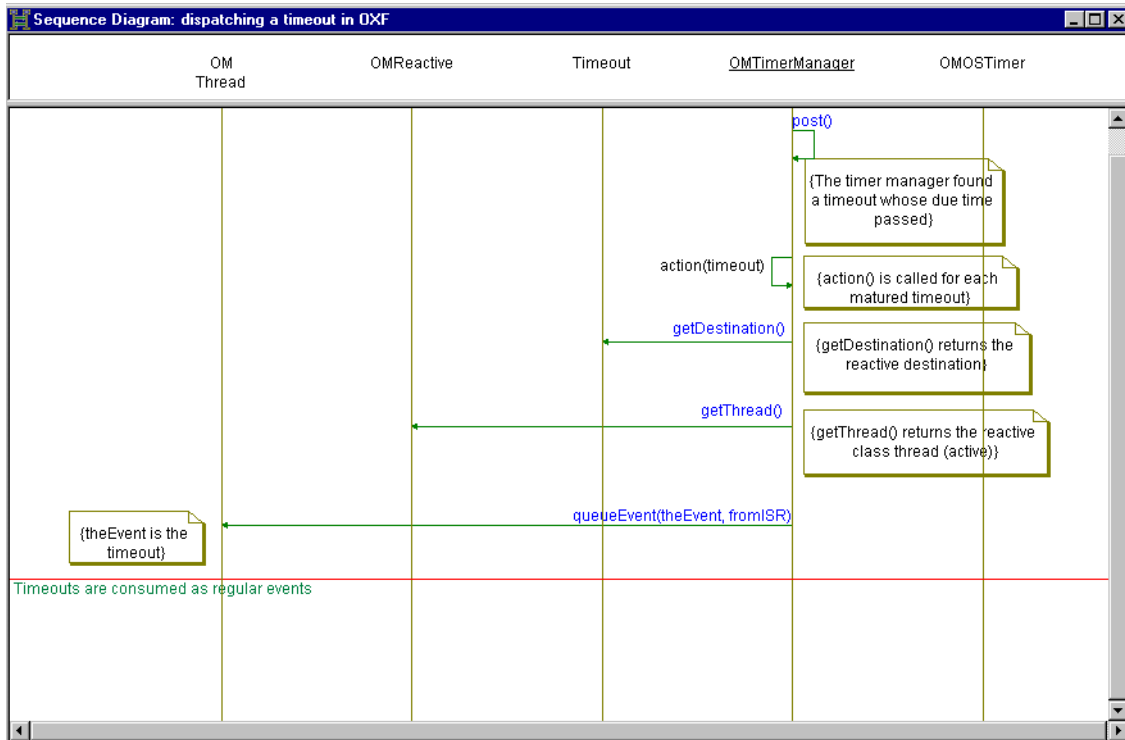


To schedule a timeout, follow these steps:

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

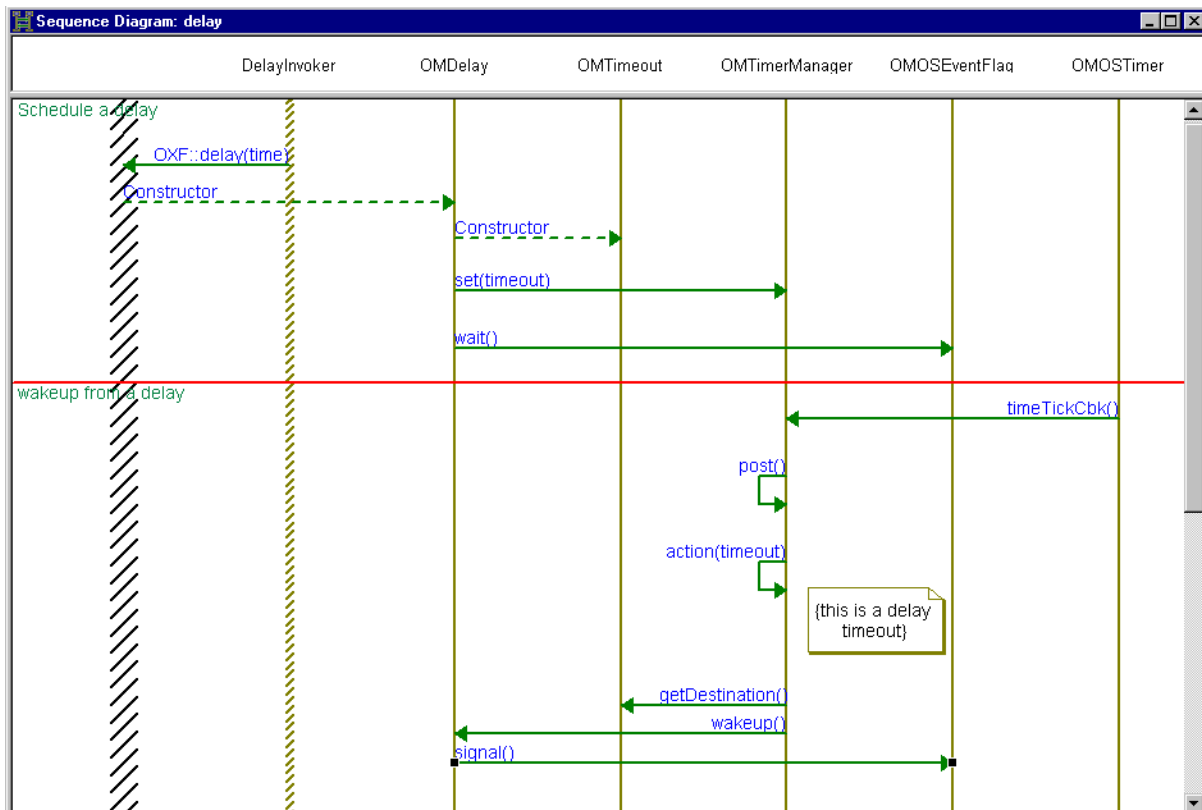
You unschedule a timeout in the following cases:

- ◆ 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.

Analyzing and Customizing

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.

Model-level Debugging and Analysis

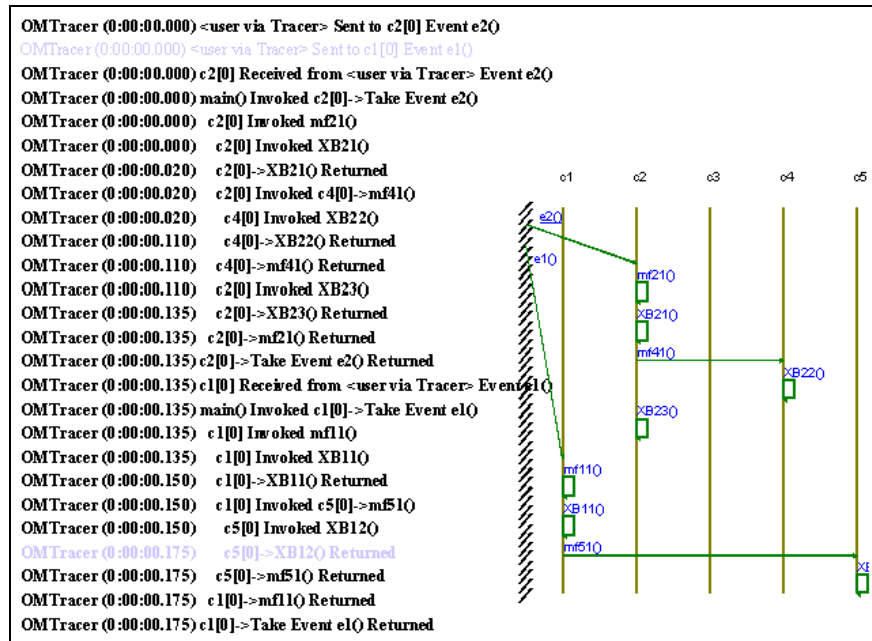
Rhapsody facilitates model-level debugging through animated statecharts and sequence diagrams. You can step through the application at an “object-oriented granularity” (operation call, one event processing, the whole event queue) and visually observe the effect on the statechart (for example, change 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.

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. Assign estimated durations for the execution of operations.
2. 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. 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.

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.

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.

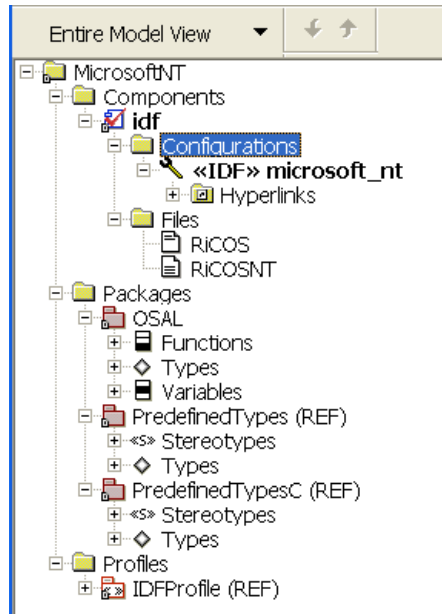
The Rational Rhapsody Interrupt-Driven Framework (IDF)

For systems requiring a solution with a smaller footprint, the OXF provided with Rational Rhapsody Developer for C is not appropriate. To provide a solution for these environments, a limited framework called IDF (Interrupt-Driven Framework) is also provided with Rhapsody. Refer to the [Limitations of the IDF](#) section for the Rational Rhapsody IDF restrictions.

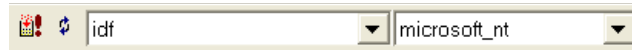
Creating a Sample IDF Project


To learn about the Rhapsody IDF, several IDF components are included as samples that can be adapted for different target systems. To use the IDF features in sample projects, follow these steps:

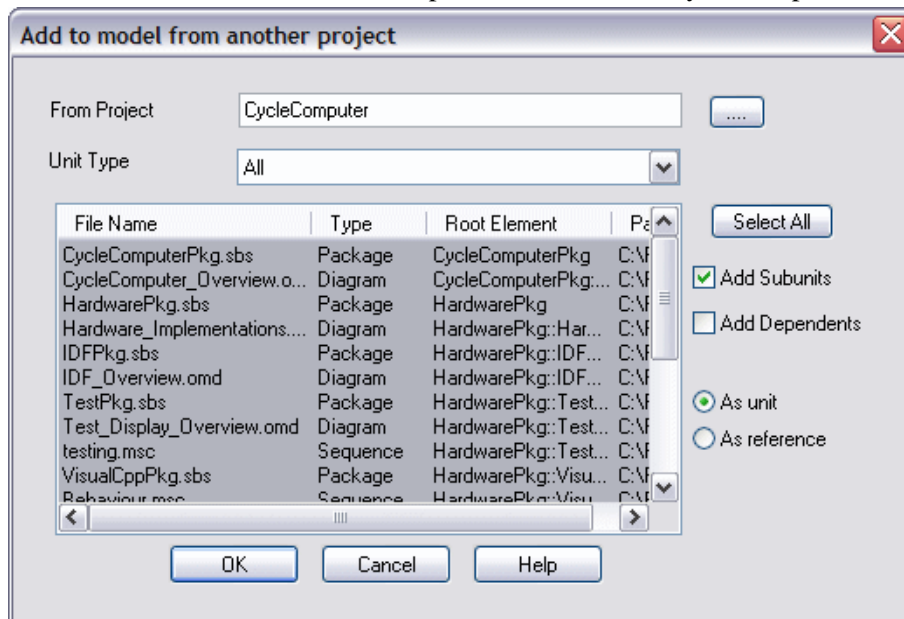
1. Change the **Properties** of the `Share\LangC\idf\Adapters\Microsoft` to remove the “Read-only” restriction and apply this change to the subfolders. Click **OK** to save these changes.
2. Start Rational Rhapsody Developer for C.
3. Open the `Share\LangC\idf\Adapters\Microsoft\MicrosoftNT.rpy` project.



4. Check to be certain the correct “idf” and “microsoft_nt” components are selected (as shown below).



5. Click  to generate to build the `MSidf.lib` library in the `Share\LangC\lib` directory. A message displays questioning if changes have been made manually and asking if you want to continue. Click **Yes**. The build messages display in the output window.
6. Select the **File > Add to Model** option and locate the `CycleComputer`.

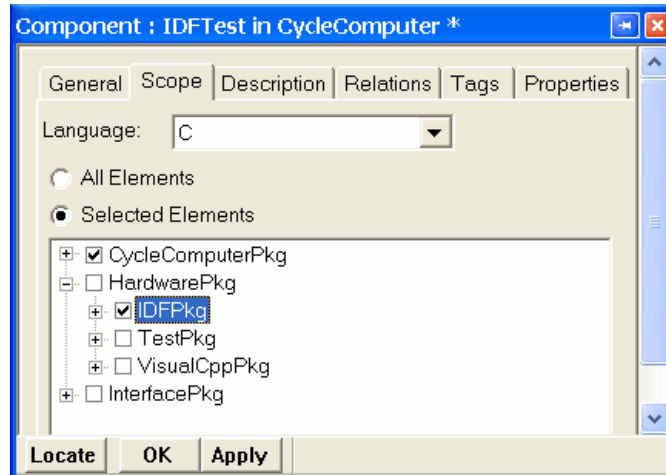



7. Select `Package (*.sbs)` in the **Files of Type** field. In the folder selection area, navigate to the **Rhapsody <version>\Share\Profiles** folder and highlight the `IDFProfile.sbs` file and select **As Reference**.
8. Click **Open** to add this .sbs file to the CycleComputer model.
Note that `IDFProfile (REF)` is now in the Profiles folder in the browser, as shown below.

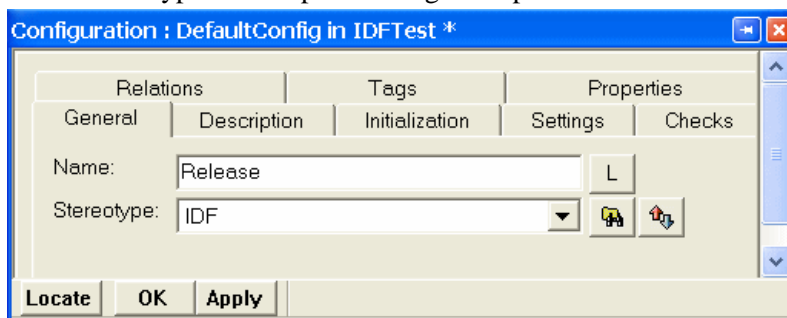


9. In the browser, highlight the **Components** item, right click, and select **Add New Component**. Type `IDFTest` into the area provided for the new component in the browser.

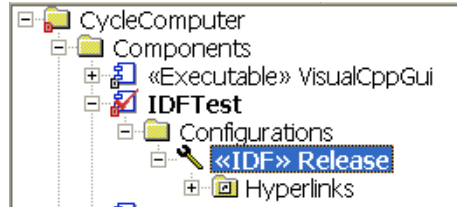
10. Right-click the new `IDFTest` component and select **Features**. In the **Scope** tab of the features dialog box, select the `CycleComputerPkg` and the `IDFPkg` under the `HardwarePkg` (as shown below) and click **OK**.



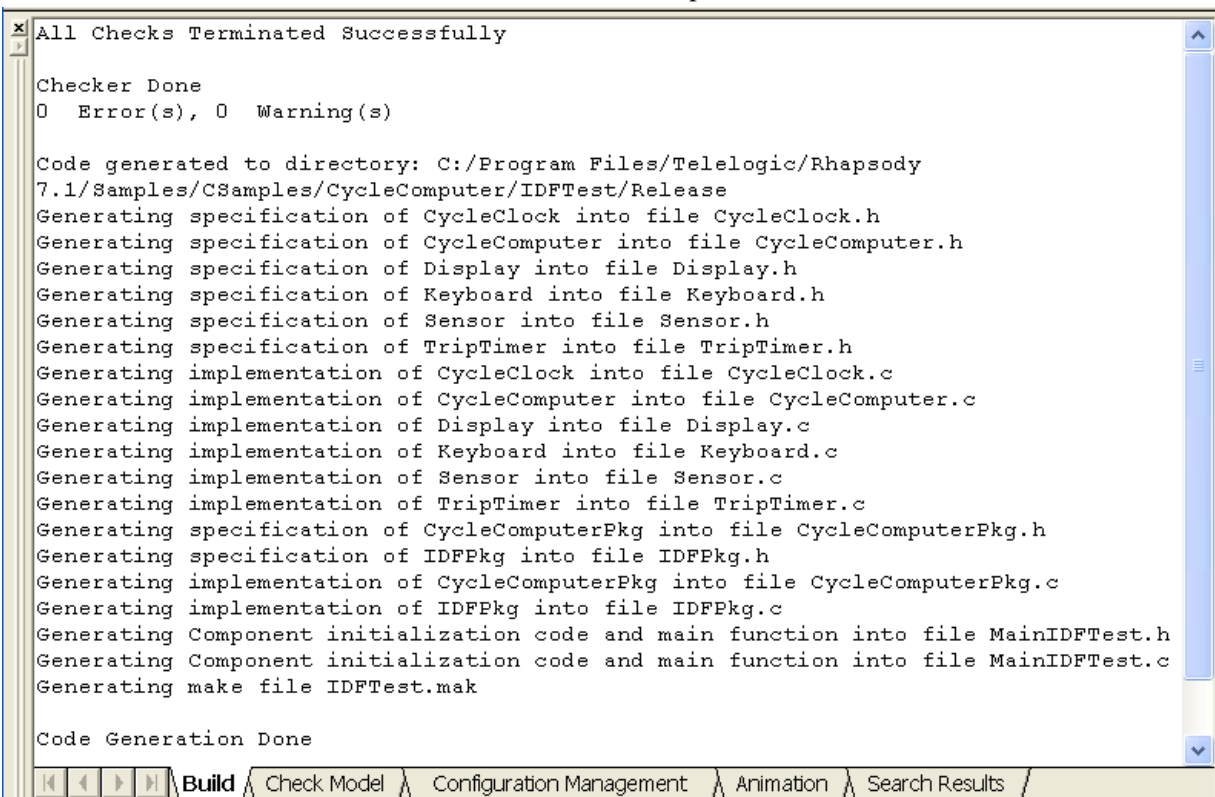
11. In the browser under the new `IDFTest` component, right click the `DefaultConfig` to display the Features dialog box.
12. In the General tab, change the **Name** of the configuration to be `Release`. Beside the **Stereotype** field, click the  and select **IDF** from the Profiles tree. You must apply the IDF stereotype to the configuration to ensure that the properties necessary for using the IDF are set correctly.
13. Click **OK** to return to the General tab. Click **Apply** to save the new configuration name and IDF stereotype and keep the dialog box open.



14. Select the **Settings** tab, from the pull-down menus at the bottom select **MicrosoftIDF** for the Environment and **Release** for the Build Set. Click **OK**. The browser tree for the new configuration should resemble this example.

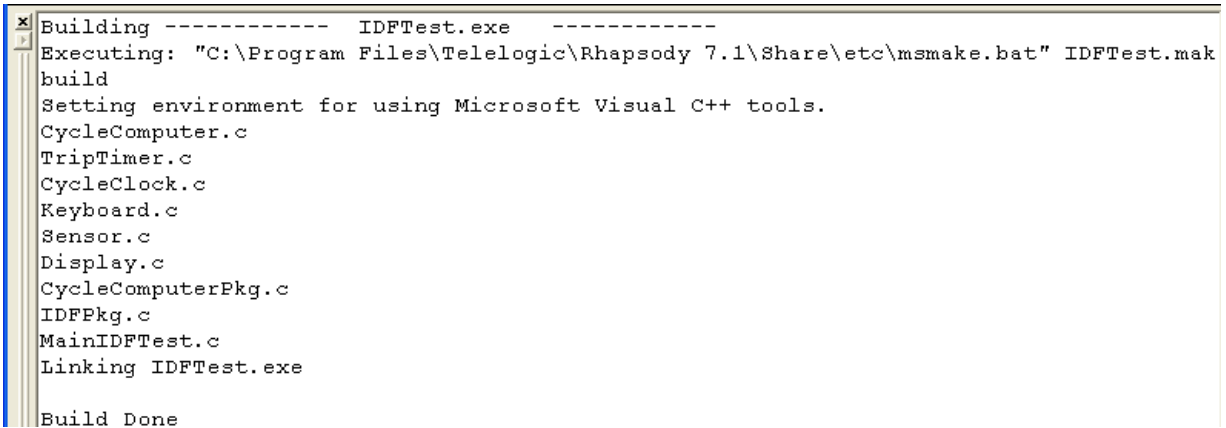


15. Select the **Code > Generate >Release** options. The code generation messages should resemble these listed in the Build tab example.




16. To create the executable, select the **Code > Build IDFTest.exe (F7)** options. The build

messages should resemble those in the following example:



```
Building ----- IDFTest.exe -----
Executing: "C:\Program Files\Telelogic\Rhapsody 7.1\Share\etc\msmake.bat" IDFTest.mak
build
Setting environment for using Microsoft Visual C++ tools.
CycleComputer.c
TripTimer.c
CycleClock.c
Keyboard.c
Sensor.c
Display.c
CycleComputerPkg.c
IDFPkg.c
MainIDFTest.c
Linking IDFTest.exe

Build Done
```

17. To launch the TripTimer display, click the Run  icon. The timer displays on a black background.

You may continue to experiment with this project to learn more about the Rational Rhapsody IDF features. For example, if you do not want to use `printf`, add a define for the `NO_PRINT` macro.

Adapting the Framework for a Specific Target

To adapt the IDF to your target, follow these steps:

1. Create the new component and import the OSAL package from the Microsoft NT model provided (`Share\LangC\idf\Adapters\Microsoft\MicrosoftNT`). The following functions will have to be modified:
 - ◆ `RiCInitTimer`—sets up a periodic interrupt that calls the `RiCTick` operation every `RiC_MS_PER_TICK`.
 - ◆ `RiCExitCriticalRegion`—enables interrupts.
 - ◆ `RiCEnterCriticalRegion`—masks interrupts.
 - ◆ `RiCGetSystemTick`—returns system tick size.
 - ◆ `RiCSleep`—operation is called when there are no events to handle and sleep can be used until the next timeout or when an interrupt occurs.
2. Create `RiCOS<target>.c` and `RiCOS.h` files for the component and add the OSAL package as an element. Some examples of these filename formats are `$OMROOT\LangC\idf\RiCOSNT.c` and for similar target specific files, `$OMROOT\LangC\oxf`.
3. Create a directory called `<target>` under the `idf\Adapters` directory and set it as the output directory for the `RiCOS.h` file.
4. Include a new environment in the `siteC.prp` file for `<Environment>IDF`. Set up the makefile template, compiler flags, etc.
5. The following framework constants and types in the OSAL package must be set:
 - ◆ `RIC_MEMORY_ALLOCATION`—sets up the buffers used for the memory allocation.
 - ◆ `RIC_MAX_EVENTS`—maximum number of simultaneous events.
 - ◆ `RIC_MAX_TIMEOUTS`—maximum number of simultaneous timeouts.
 - ◆ `RIC_MS_PER_TICK`—periodic timeout in milliseconds.
 - ◆ `tRiCCriticalSection`—OS-specific type, which is used during critical section processing.
6. If you need `RiCString`, `RiCMap`, `RiCList`, and `RiCCollection`, then they can be added to the generic configuration scope and the `idf` library should be rebuilt. They are not included by default. To use `RiCString`, the user must `#include RiCString.h`.
7. Change the following properties in `C.CG::<Target>`:
 - ◆ `MakeFileName`—`<target>idf`

- ◆ `MakeFileContent`—change the name of the IDF library to `<target>idf$(LIB_EXT)`
 - ◆ `CppCompileSwitches`- add the `LangC/Adapters/<target>` path.
 - ◆ Add the following property:
`Property ReactiveVtblKind Enum "OXF, IDF" "IDF"`
8. Add the profile `IDFProfile.sbs` to the model, as described in the previous section.

Note

If the message, “*CG MESSAGE: There are no classes in the component scope,*” displays while performing these steps, the message can be ignored.

Limitations of the IDF

While the IDF has a much smaller footprint than the standard framework provided with Rhapsody, the OXF, this size reduction brings with it the following limitations:

- ◆ The IDF is single-threaded and interrupt-driven.
- ◆ It is not possible to use the animation and tracing features with the IDF. All models, however, can be animated using the OXF.
- ◆ The IDF requires the use of the “flat statechart implementation” and “real-time model” options.
- ◆ The event queue and timer heaps are not dynamic. Maximum sizes must be set using the `RIC_MAX_EVENTS` and `RIC_MAX_TIMEOUTS` macro definitions.
- ◆ The property `CG::Events::BaseNumberOfInstances` must be set to a value greater than 0 to allow automatic allocation of memory from the memory pools. The actual number used will be ignored.
- ◆ In order to save RAM, the maximum number of consecutive null transitions has been greatly reduced - from 100 to 7.

OXF Classes and Methods

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.

OMAbstractMemoryAllocator Class

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

Construction Summary

~OMAbstractMemoryAllocator	Destroys the <code>OMAbstractMemoryAllocator</code> object
--------------------------------------------	------------------------------------------------------------

Method Summary

allocPool	Allocates a memory pool big enough to hold the specified number of instances
callMemoryPoolsEmpty	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

This 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

This 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

callMemoryPoolsEmpty

Visibility

Public

Description

This 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

This 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

This 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

This 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

This 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

This 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

This 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

This 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

This 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

This 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 <code>OMAndState</code> object
----------------------------	----------------------------------------------

Method Summary

lock	Locks the mutex of the <code>OMState</code> object
unlock	Unlocks the mutex of the <code>OMState</code> object

OMAndState

Visibility

Public

Description

This 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

This method locks the mutex of the `OMState` object.

Signature

```
void lock();
```

unlock

Visibility

`Public`

Description

This 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

<u><code>OMCollection</code></u>	Constructs an <code>OMCollection</code> object
<u><code>~OMCollection</code></u>	Destroys the <code>OMCollection</code> object

Method Summary

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

OMCollection

Visibility

Public

Description

This 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

This method is the destructor for the `OMCollection` class.

Signature

```
~OMCollection()
```

See Also

[OMCollection](#)

add

Visibility

Public

Description

This 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

This 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

This 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

This method deletes all the elements from the collection.

Signature

```
void removeAll();
```

See Also

[add](#)

[addAt](#)

[remove](#)

[removeByIndex](#)

removeByIndex

Visibility

Public

Description

This 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

This 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 <code>OMComponentState</code> 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

This method is the constructor for the `OMComponentState` class.

Signature

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

Parameters

`par`

The parent

enterState

Visibility

Public

Description

This method specifies the method called on the entry to the state (the entry action).

Signature

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

in

Visibility

Public

Description

This 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

This 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

This 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

This method is the destructor for the OMDelay class.

Signature

```
~OMDelay()
```

See Also

[OMDelay](#)

wakeup

Visibility

Public

Description

This 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 <code>OMReactive</code> instance
frameworkEvent	Specifies whether an event is a framework event
Id	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 <code>OMStartBehavior</code> 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 <code>OMEvent</code> object
~OMEvent	Destroys the <code>OMEvent</code> object

Method Summary

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

This 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

This method is the destructor for the `OMEvent` class.

Signature

```
virtual ~OMEvent()
```

See Also

[OMEvent](#)

Delete

Visibility

Public

Description

This 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

This 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](#)**getId****Visibility**

Public

Description

This method returns the event ID.

Signature

```
short getId() const
```

Return

Id, the value for the event ID

See Also[Id](#)[setId](#)

isCancelledTimeout

Visibility

Public

Description

This 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

This 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

This 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

This 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

This 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

[getId](#)

[lId](#)

[setId](#)

isTypeOf

Visibility

Public

Description

This 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

This 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

This 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

This 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](#)

setId

Visibility

Public

Description

This method sets the event ID.

Signature

```
void setId (short pId)
```

Parameters

pId

Specifies the new event ID

See Also

[getId](#)

[Id](#)

[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 <code>OMFinalState</code> object
------------------------------	------------------------------------------------

Method Summary

getConcept	Returns the current element
----------------------------	-----------------------------

OMFinalState

Visibility

Public

Description

This method is the constructor for the OMFfinalState 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 - statechart owner

par - parent

cmp - component

hdl - handle

getConcept

Visibility

Public

Description

This 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 <code>OMStartBehaviorEvent</code> 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

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

Signature

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

Parameter

`userEventPtr`

A pointer to the event

cserialize

Visibility

Public

Description

This 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

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

Signature

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

serialize

Visibility

Public

Description

This 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 <code>OMFriendTimeout</code> 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

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

Signature

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

Parameters

```
userEventPtr
```

A pointer to the timeout event

cserialize

Visibility

Public

Description

This 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

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

Parameters

```
withParameters
```

A Boolean value that specifies whether to include the parameter values

getEventClass

Visibility

Public

Description

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

Signature

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

serialize

Visibility

Public

Description

This 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

This 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

This 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

This method gets the guard object.

Signature

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

Return

The guard object

lock

Visibility

Public

Description

This method locks the mutex of the OMGuard object.

Signature

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

unlock

Visibility

Public

Description

This 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

<u>OMHeap</u>	Constructs an OMHeap object
<u>~OMHeap</u>	Destroys the OMHeap object

Method Summary

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

OMHeap

Visibility

Public

Description

This 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

This method destroys the OMHeap object.

Signature

```
~OMHeap()
```

See Also

[OMHeap](#)

add

Visibility

Public

Description

This 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

This 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

This 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

This 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

This method moves the iterator to the top of the heap.

Signature

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

Return

The top-most element

trim

Visibility

Public

Description

This 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

This 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

This method increments the iterator by 1.

Signature

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

Return

The new value of the iterator

reset

Visibility

Public

Description

This 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

This 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 <code>OMLeafState</code> 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 <code>TRUE</code> 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

This 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

This method specifies the operation called when the state is entered from a default transition.

Signature

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

enterState

Visibility

Public

Description

This method specifies the state entry action

Signature

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

exitState

Visibility

Public

Description

This method specifies the state exit action.

Signature

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

in

Visibility

Public

Description

This 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

This method is called during animation to send state information.

Signature

```
virtual void serializeStates (AOMState* 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

<u>OMList</u>	Constructs an <code>OMList</code> object
<u>~OMList</u>	Destroys the <code>OMList</code> object

Flag Summary

<u>first</u>	Specifies the first element in the list
<u>last</u>	Specifies the last element in the list

Method Summary

<u>operator []</u>	Returns the element at the specified position
<u>add</u>	Adds the specified element to the end of the list
<u>addAt</u>	Adds the specified element to the list at the given index
<u>addFirst</u>	Adds an element at the beginning of the list
<u>find</u>	Looks for the specified element in the list
<u>getAt</u>	Returns the element found at the specified index
<u>getCount</u>	Returns the number of elements in the list
<u>getCurrent</u>	Is used by the iterator to get the element at the current position in the list
<u>getFirst</u>	Is used by the iterator to get the first position in the list
<u>getFirstConcept</u>	Returns the first <code>Concept</code> element in the list
<u>getLast</u>	Is used by the iterator to get the last position in the list
<u>getLastConcept</u>	Returns the last <code>Concept</code> element in the list
<u>getNext</u>	Is used by the iterator to get the next position in the list
<u>isEmpty</u>	Determines whether the list is empty
<u>_removeFirst</u>	Removes the first item from the list.=
<u>remove</u>	Deletes the first occurrence of the specified element from the list
<u>removeAll</u>	Deletes all the elements from the list
<u>removeFirst</u>	Deletes the first element from the list
<u>removeItem</u>	Deletes the specified element from the list
<u>removeLast</u>	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

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

Signature

```
OMList ()
```

See Also

[~OMList](#)

~OMList

Visibility

Public

Description

This 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

This 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](#)

[removeLast](#)

addAt

Visibility

Public

Description

This 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

This 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

This method looks for the 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

This 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

This 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

This 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

This 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

This 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

This 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

This 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

This 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

This 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

This 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

This 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

This method deletes all the elements from the list.

Signature

```
void removeAll()
```

See Also

[add](#)

[addAt](#)

[remove](#)

[removeFirst](#)

[removeLast](#)

removeFirst

Visibility

Public

Description

This method deletes the first element from the list.

Signature

```
void removeFirst()
```

See Also

[add](#)

[addAt](#)

[remove](#)

[removeAll](#)

[removeLast](#)

removeItem

Visibility

Public

Description

This 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

This 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 <code>OMListItem</code> object
----------------------------	----------------------------------------------

Method Summary

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

OMListItem

Visibility

Public

Description

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

Signature

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

Parameters

`theConcept`

The new list element

connectTo

Visibility

Public

Description

This method connects the specified list item to the list.

Signature

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

Parameters

item

The list item

getNext

Visibility

Public

Description

This 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 (<code>OMThread::execute</code>) of the main thread singleton

~OMMainThread

Visibility

Public

Description

This method is the destructor for the OMMainThread class.

Signature

```
virtual ~OMMainThread()
```

destroyThread

Visibility

Public

Description

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

Signature

```
virtual void destroyThread()
```

instance

Visibility

Public

Description

This 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

This 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

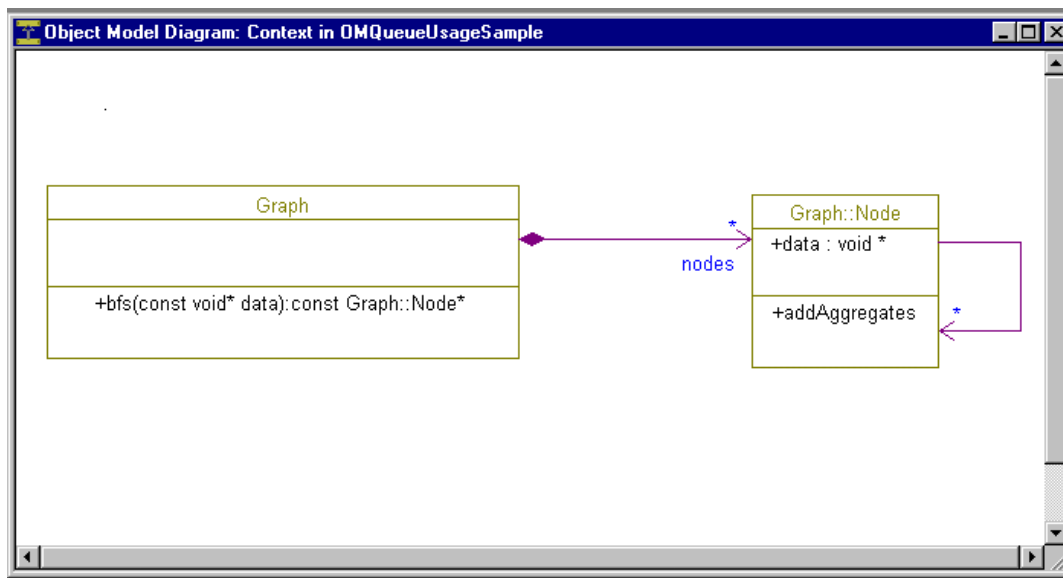
<code>OMMap</code>	Constructs an <code>OMMap</code> object
<code>~OMMap</code>	Destroys the <code>OMMap</code> object

Method Summary

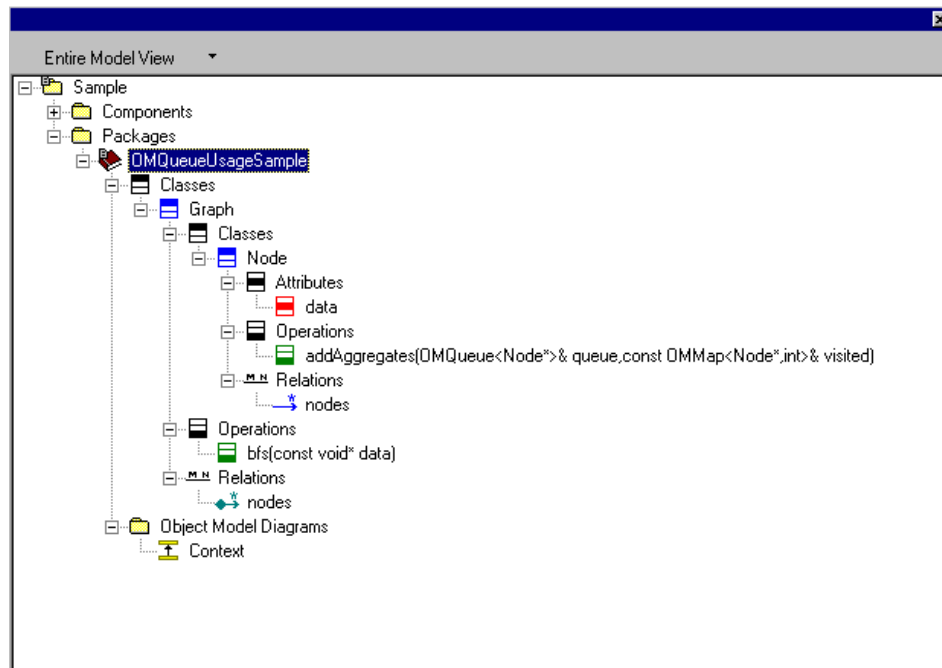
<code>operator []</code>	Returns the element found for the specified key
<code>add</code>	Adds an element to the map
<code>find</code>	Looks for the specified element in the map
<code>getAt</code>	Returns the element for the specified key
<code>getCount</code>	Returns the number of elements in the map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

```

bfs(const void*)
// the queue is used as the search main container
OMQueue<Node*> searchQueue;
// map of the elements we already visited
OMMap<Node*,int> visited;
////////////////////////////////////
// do the BFS
////////////////////////////////////
// set the first node of the search
searchQueue.put(nodes[0]);
// start the search
Node* theNode = NULL;
while ((theNode == NULL) && (!searchQueue.isEmpty())) {
    Node* node = searchQueue.get();
    if (!node) continue;
    // check & add the node to the visited list
    int dummy;
    if (visited.lookup(node, dummy) != 0) continue;
    visited[node] = 1;
    // compare the data
    if (node->getData() == data) {
        // found
        theNode = node;
    }
    else {
        // add the node aggregates to the search queue
        node->addAggregates(searchQueue, visited);
    }
}
return theNode;
    
```

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

```

addAggregates(OMQueue<Node*>&,const OMMMap<Node*,int>&)
int dummy;
OMIterator<Node*> iter(nodes);
for (; *iter; ++iter) {
    Node* node = *iter;
    if (visited.lookup(node, dummy) != 0) continue; // already visited
    queue.put(node);
}
    
```


OMMap

Visibility

Public

Description

This method is the constructor for the OMMap class.

Signature

OMMap ()

See Also

[~OMMap](#)

~OMMap

Visibility

Public

Description

This 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

This 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

This 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

This 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

This 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

This 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

This 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

This 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

This 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

This 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 <code>OMMapItem</code> object
~OMMapItem	Destroys the <code>OMMapItem</code> object

Method Summary

getConcept	Returns the current map item
----------------------------	------------------------------

OMMapItem

Visibility

Public

Description

This 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

This method destroys the `OMMapItem` object.

Signature

```
virtual ~OMMapItem()
```

See Also

[OMMapItem](#)

getConcept

Visibility

Public

Description

This 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 <code>OMMemoryManager</code> object
~OMMemoryManager	Destroys the <code>OMMemoryManager</code> 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.

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

This 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

This method is the destructor for the OMMemoryManager class.

Signature

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

See Also

[OMMemoryManager](#)

getDefaultMemoryManager

Visibility

Public

Description

This 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

This 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

This method returns the current memory manager.

Signature

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

Return

The current memory manager

See Also

[getDefaultMemoryManager](#)

returnMemory

Visibility

Public

Description

This 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

This method is the constructor for the OMMemoryManagerSwitchHelper class.

Signature

```
OMMemoryManagerSwitchHelper()
```

See Also

[~OMMemoryManagerSwitchHelper](#)

~OMMemoryManagerSwitchHelper

Visibility

Public

Description

This method is the destructor for the OMMemoryManagerSwitchHelper class.

Signature

```
~OMMemoryManagerSwitchHelper()
```

See Also

[OMMemoryManagerSwitchHelper](#)

cleanup

Visibility

Public

Description

This method cleans up the allocated memory log.

Signature

```
void cleanup();
```

findMemory

Visibility

Public

Description

This 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

This method returns the singleton instance of the OMMemoryManagerSwitchHelper.

Signature

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

Return

The singleton instance of OMMemoryManagerSwitchHelper

isLogEmpty

Visibility

Public

Description

This 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

This 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

This 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

This 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

This 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

<code>notifyToError</code>	Writes messages to the error log
<code>notifyToOutput</code>	Writes messages to standard output

`notifyToError`

Visibility

Public

Description

This 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

This 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

<u>OMOrState</u>	Constructs an <code>OMOrState</code> object
----------------------------------	---------------------------------------------

Flag Summary

<u>subState</u>	Specifies a substate
---------------------------------	----------------------

Method Summary

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

Flags

subState

Specifies a substate. It is defined as follows:

```
OMState* subState;
```

OMOrState

Visibility

Public

Description

This method is the constructor for the OMOrState class.

Signature

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

Parameters

par

Specifies the parent

entDef

Visibility

Public

Description

This method specifies the operation called when the state is entered from a default transition.

Signature

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

enterState

Visibility

Public

Description

This method specifies the state entry action.

Signature

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

exitState

Visibility

Public

Description

This method specifies the state exit action.

Signature

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

getSubState

Visibility

Public

Description

This method returns the substate.

Signature

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

Return

The substate

in

Visibility

Public

Description

This 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

This 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

This 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

<u>OMProtected</u>	Constructs an OMProtected object
<u>~OMProtected</u>	Destroys the OMProtected object

Macro Summary

<u>OMDECLARE_GUARDED</u>	Aggregates OMProtected objects inside guarded classes instead of inheriting from OMProtected.
------------------------------------------	-----------------------------------------------------------------------------------------------

Method Summary

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

This 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](#)

~OMPProtected

Visibility

Public

Description

This method is the destructor for the `OMPProtected` object. The method deletes (destroys) the operating system entity that the instance wraps.

Signature

```
~OMPProtected()
```

See Also

[OMPProtected](#)

deleteMutex

Visibility

Public

Description

This method deletes the mutex and sets its value to `NULL`.

Signature

```
inline void deleteMutex()
```

free

Visibility

Public

Description

This 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

This 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

This method creates an RTOS mutex, if it has not been created already.

Signature

```
void initializeMutex()
```

lock

Visibility

Public

Description

This method locks the mutex of the `OMPProtected` 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

This method unlocks the mutex of the `OMPProtected` 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

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

Construction Summary

<u>OMQueue</u>	Constructs an <code>OMQueue</code> object
<u>~OMQueue</u>	Destroys the <code>OMQueue</code> object

Method Summary

<u>get</u>	Gets the current element in the queue
<u>getCount</u>	Gets the number of elements in the queue
<u>getInverseQueue</u>	Returns the element that will be returned by <code>get()</code> in the tail of the queue
<u>getQueue</u>	Returns the element that will be returned by <code>get()</code> in the head of the queue
<u>getSize</u>	Returns the size of the memory allocated for the queue
<u>increaseHead</u>	Increases the size of the queue head
<u>increaseTail</u>	Increases the size of the queue tail
<u>isEmpty</u>	Determines whether the queue is empty
<u>isFull</u>	Determines whether the queue is full
<u>put</u>	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;
```

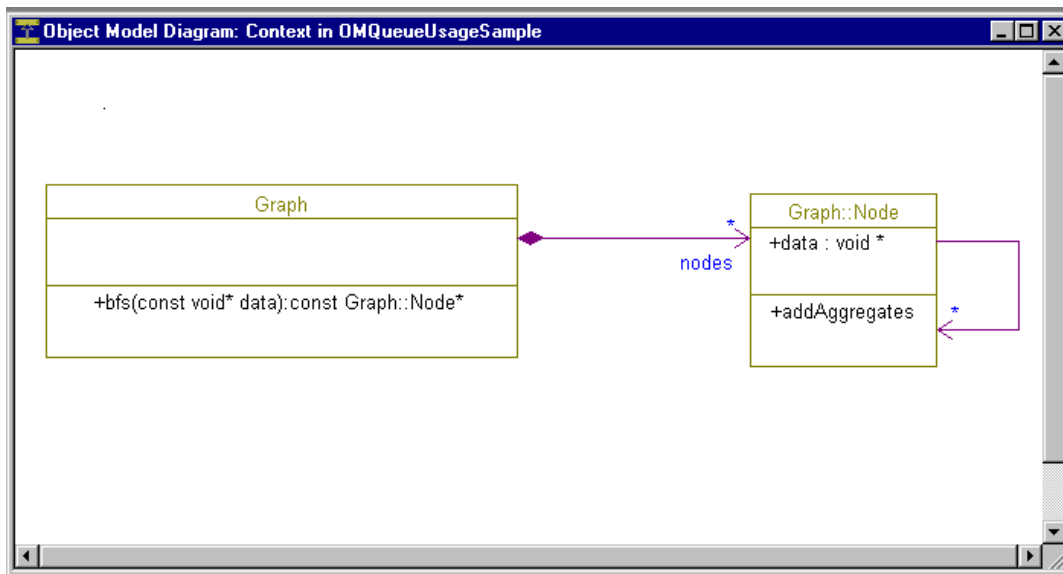
m_tail

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

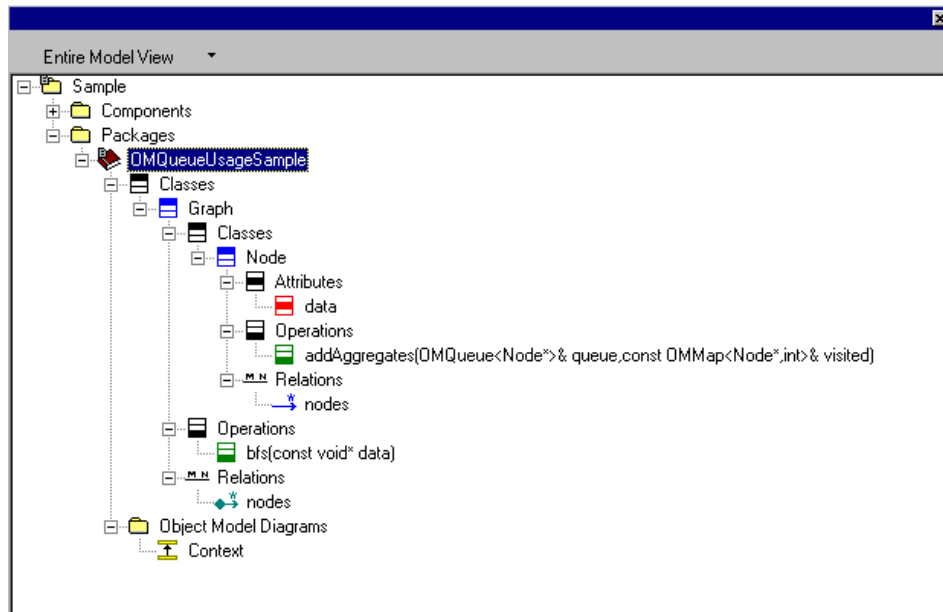
```
int m_tail;
```

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 screenshot shows a code editor window titled "Primitive Operation : bfs in Graph". The editor has tabs for "General", "Implementation", and "Properties". The "Implementation" tab is active, showing the following C++ code:

```

bfs(const void*)

// the queue is used as the search main container
OMQueue<Node*> searchQueue;
// map of the elements we already visited
OMMap<Node*,int> visited;
////////////////////////////////////
// do the BFS
////////////////////////////////////
// set the first node of the search
searchQueue.put(nodes[0]);
// start the search
Node* theNode = NULL;
while ((theNode == NULL) && (!searchQueue.isEmpty())) {
    Node* node = searchQueue.get();
    if (!node) continue;
    // check & add the node to the visited list
    int dummy;
    if (visited.lookup(node, dummy) != 0) continue;
    visited[node] = 1;
    // compare the data
    if (node->getData() == data) {
        // found
        theNode = node;
    }
    else {
        // add the node aggregates to the search queue
        node->addAggregates(searchQueue, visited);
    }
}
return theNode;
    
```

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

The screenshot shows a code editor window titled "Primitive Operation : addAggregates in Node". The editor has tabs for "General", "Implementation", and "Properties". The "Implementation" tab is active, showing the following C++ code:

```

addAggregates(OMQueue<Node*>&,const OMMMap<Node*,int>&)

int dummy;
OMIterator<Node*> iter(nodes);
for (; *iter; ++iter) {
    Node* node = *iter;
    if (visited.lookup(node, dummy) != 0) continue; // already visited
    queue.put(node);
}
    
```


OMQueue

Visibility

Public

Description

This 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

This method destroys the OMQueue object.

Signature

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

See Also

[OMQueue](#)

get

Visibility

Public

Description

This method gets the current element in the queue.

Signature

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

Return

The current element in the queue

getCount

Visibility

Public

Description

This 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

This 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

This 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

This 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

This method increases the size of the queue head.

Signature

```
void increaseHead_();
```

increaseTail_

Visibility

Public

Description

This method increases the size of the queue tail.

Signature

```
void increaseTail_();
```

isEmpty

Visibility

Public

Description

This 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

This 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

This method adds an element to the queue.

Signature

```
virtual OMBBoolean 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 <code>OMReactive</code>) 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

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

Macro Summary

<u>GEN</u>	Generates a new event
<u>GEN BY GUI</u>	Generates an event from a GUI
<u>GEN BY X</u>	Generates a new event from a sender object to a receiver object
<u>GEN ISR</u>	Generates an event from an interrupt service request (ISR)

Relation Summary

<u>event</u>	Specifies the active or current event (the one that is now being processed) for the <code>OMReactive</code> instance
<u>m_eventGuard</u>	Used, in collaboration with the generated code, to protect the event consumption from mutual exclusion between events and triggered operations
<u>myThread</u>	Specifies the active class that queues events and dispatches events (so they are consumed on the active class's thread) for a reactive object
<u>rootState</u>	Defines the root state of the <code>OMReactive</code> 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 <code>omrStatus</code> 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.

<u>rootState_serializeStates</u>	Is a virtual method that performs the actual event consumption.
<u>runToCompletion</u>	Takes all the null transitions (if any) that can be taken after an event has been consumed.
<u>serializeStates</u>	Is called during animation to send state information.
<u>setCompleteStartBehavior</u>	Sets the value of the <u>OMRShouldCompleteStartBehavior</u> attribute.
<u>setEventGuard</u>	Is used to set the event guard flag (<u>m_eventGuard</u>).
<u>setFrameworkInstance</u>	Changes the value of the <u>frameworkInstance</u> attribute.
<u>setInDtor</u>	Specifies that event dispatching should be stopped.
<u>setMaxNullSteps</u>	Sets the maximum number of null transitions (those without a trigger) that can be taken sequentially in the statechart.
<u>setShouldDelete</u>	Specifies whether a reactive object should be deleted by its active object when it reaches a termination connector in its state machine.
<u>setShouldTerminate</u>	Specifies that a reactive instance can be safely destroyed by its active instance.
<u>setThread</u>	Sets the thread of a reactive object.
<u>setToGuardReactive</u>	Specifies the value of the <u>toGuardReactive</u> attribute.
<u>shouldCompleteRun</u>	Checks the value of <u>omrStatus</u> to determine whether there are null transitions to take.
<u>shouldCompleteStartBehavior</u>	Checks the start behavior state.
<u>shouldDelete</u>	Determines whether a reactive object should be deleted by its active object when it reaches a termination connector in its state machine.
<u>shouldTerminate</u>	Determines whether a reactive instance can be safely destroyed by its active instance.
<u>startBehavior</u>	Initializes the behavioral mechanism and takes the initial (default) transitions in the statechart before any events are processed.
<u>takeEvent</u>	Is used by the event loop (within the thread) to make the reactive object process an event.
<u>takeTrigger</u>	Consumes a triggered operation event (synchronous event).
<u>terminate</u>	Sets the <code>OMReactive</code> instance to the terminate state (the statechart is entering a termination connector).
<u>undoBusy</u>	Sets the value of the <code>sm_busy</code> 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.

Multi-thread 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

This 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

This 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

This 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

This 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

This 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

This 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](#) and [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

This 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

This 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

This 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

This 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

This 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

This 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

This 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

This 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

This 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

This 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

This 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 (AOMSSState* aomsState)
    const;
```

Parameters

`aomsState`

Specifies the root state

runToCompletion

Visibility

Public

Description

This 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

This method is called during animation to send state information.

Signature

```
void serializeStates (AOMSSState* s) const;
```

Parameters

`s`

Specifies the state

setCompleteStartBehavior

Visibility

Public

Description

This 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

This 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

This 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

This 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

This 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

This 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

This 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

This 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

This 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

This 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

This 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

This 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

This 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

This 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

This 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 (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 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.
- ◆ The [execute](#) method calls `takeEvent` to process the reactive object an event.

See Also

[consumeEvent](#)

[execute](#)

takeTrigger

Visibility

Public

Description

This 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

This 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

This 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 `OMStack`s. 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 <code>OMStack</code> object
~OMStack	Destroys the <code>OMStack</code> 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

This method is the constructor for the `OMStack` class.

Signature

```
OMStack()
```

See Also

[~OMStack](#)

~OMStack

Visibility

Public

Description

This method destroys the OMStack object.

Signature

```
~OMStack()
```

See Also

[OMStack](#)

getCount

Visibility

Public

Description

This 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

This 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

This method pops the next item off the stack.

Signature

```
Concept pop()
```

Return

The item popped off the stack

push

Visibility

Public

Description

This 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

This 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 <code>OMStartBehaviorEvent</code> class
--------------------------------------	--------------------------------------------------------------------

OMStartBehaviorEvent

Visibility

Public

Description

This 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

<code>parent</code>	Specifies the parent
---------------------	----------------------

Construction Summary

<u><code>OMState</code></u>	Constructs an <code>OMState</code> object
---------------------------------------------	-------------------------------------------

Macro Summary

<u><code>IS_EVENT_TYPE_OF(id)</code></u>	Supports generic derived event handling
<u><code>OM_DECLARE_FRAMEWORK_MEMORY_ALLOCATION_OPERATORS</code></u>	Supports enhanced user control over framework memory allocation

Method Summary

<u><code>entDef</code></u>	Specifies the operation called when the state is entered from a default transition
<u><code>entHist</code></u>	Enters a history connector
<u><code>enterState</code></u>	Specifies the state entry action
<u><code>exitState</code></u>	Specifies the state exit action
<u><code>getConcept</code></u>	Gets the statechart owner
<u><code>getHandle</code></u>	Gets the handle
<u><code>getLastState</code></u>	Gets the last state
<u><code>isCompleted</code></u>	Gets the substate
<u><code>in</code></u>	Returns <code>TRUE</code> when the owner class is in this state
<u><code>isCompleted</code></u>	Determines whether the <code>OR</code> state reached a final state, and therefore can be exited on a null transition
<u><code>serializeStates</code></u>	Is called during animation to send state information
<u><code>setHandle</code></u>	Sets the handle
<u><code>setLastState</code></u>	Sets the last state
<u><code>setSubState</code></u>	Sets the substate
<u><code>takeEvent</code></u>	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

This method is the constructor for the OMState class.

Signature

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

Parameters

par

Specifies the parent

entDef

Visibility

Public

Description

This method specifies the operation called when the state is entered from a default transition.

Signature

```
virtual void entDef()=0;
```

entHist

Visibility

Public

Description

This method enters a history connector.

Signature

```
virtual void entHist();
```

enterState

Visibility

Public

Description

This method specifies the state entry action.

Signature

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

exitState

Visibility

Public

Description

This method specifies the state exit action.

Signature

```
virtual void exitState()=0;
```

getConcept

Visibility

Public

Description

This 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

This method gets the handle. This method is used for animation purposes.

Signature

```
const char * getHandle() const
```

Return

The handle

getLastState

Visibility

Public

Description

This method returns the last state.

Signature

```
virtual OMState* getLastState();
```

Return

The last state

getSubState

Visibility

Public

Description

This method returns the substate.

Signature

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

Return

The substate

in

Visibility

Public

Description

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

Signature

```
virtual int in()=0;
```

isCompleted

Visibility

Public

Description

This 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

This method is called during animation to send state information.

Signature

```
virtual void serializeStates (AOMSSState* s) const = 0;  
  
virtual void serializeStates(void*) //no animation
```

Parameters

s

Specifies the state

setHandle

Visibility

Public

Description

This 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

This method sets the last state.

Signature

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

Parameters

s

Specifies the last state

setSubState

Visibility

Public

Description

This method sets the specified substate.

Signature

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

Parameters

s

Specifies the substate

takeEvent

Visibility

Public

Description

This 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

<code>count</code>	Specifies the number of elements in the static array
<code>theLink</code>	Specifies the link to an element in the static array
<code>size</code>	Specifies the amount of memory allocated for the static array

Construction Summary

<code>OMStaticArray</code>	Constructs an <code>OMStaticArray</code> object
<code>~OMStaticArray</code>	Destroys the <code>OMStaticArray</code> object

Method Summary

<code>operator []</code>	Returns the element at the specified position
<code>add</code>	Adds the specified element to the array
<code>find</code>	Looks for the specified element in the array
<code>getAt</code>	Returns the element found at the specified index
<code>getCount</code>	Determines how many elements are in the array
<code>getSize</code>	Returns the amount of memory allocated for the array
<code>isEmpty</code>	Determines whether the array is empty
<code>removeAll</code>	Deletes all the elements from the array
<code>setAt</code>	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

This 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

This 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

This 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

This 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

This 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

This 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

This method gets the size of the memory allocated for the static array.

Signature

```
int getSize() const
```

Return

The size

isEmpty

Visibility

Public

Description

This 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

This method deletes all the elements from the array.

Signature

```
void removeAll()
```

See Also

[add](#)

setAt

Visibility

Public

Description

This 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

This 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

This 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 ostream and a string.

Signature

```
inline ostream& operator<<(ostream& 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 istream and a string.

Signature

```
istream& operator>>(istream& is, OMString& s)
```

Parameters

is

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

This 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

This method empties the string.

Signature

```
void Empty()
```

GetBuffer

Visibility

Public

Description

This 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

This method returns the length of the string.

Signature

```
int GetLength() const;
```

Returns

The string length

IsEmpty

Visibility

Public

Description

This 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

This 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

This 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

This 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

This 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 <code>OMReactive</code> 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 <code>cleanupAllThreads</code> 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 <code>OMProtected</code> part
getOsHandle	Returns the thread's operating system ID
getOSThreadEndClib	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 <code>OMTimerManager</code>
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 Rational Rhapsody Developer for C++ execution framework (COM)
suspend	Suspends the thread
unlock	Unlocks the thread mutex
unschedTm	Cancel 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

This 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 `OMPProtected` class, a neutral implementation of a mutex. Every `OMThread` instance has a mutex because, in a multi-threaded 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

This 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

This 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 OMBolean allowDeleteInThreadsCleanup()
```

cancelEvent

Visibility

Public

Description

This 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

This 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

This 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

This 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

This 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 adaptor 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

This 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

This 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

This method is used by the framework for animation purposes.

Signature

```
AOMThread* getAOMThread() const
```

getEventQueue

Visibility

Public

Description

This method is used by the framework for animation purposes.

Signature

```
AOMEventQueue* getEventQueue() const;
```

getGuard

Visibility

Public

Description

This 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

This 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

This 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 (  
    OMOThread::OMOThreadEndCallBack *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

This method is used by the framework for animation purposes.

Signature

```
AOMStepper* getStepper() const;
```

lock

Visibility

Public

Description

This method puts a lock on the thread mutex.

Signature

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

omGetEventQueue

Visibility

Public

Description

This 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

This 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

This 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

This 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

This 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

This 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

This 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

This method determines whether the thread should be guarded.

Signature

```
inline OMBBoolean 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

This 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

This method is used to support the DLL version of the Rational Rhapsody Developer for 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

This 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

This method unlocks the thread mutex.

Signature

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

unschedTm

Visibility

Public

Description

This 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

This method is the destructor for the OMThreadTimer class.

Signature

```
RP_FRAMEWORK_DLL virtual ~OMThreadTimer
```

action

Visibility

Public

Description

This 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 OMTimerManager class.

See Also

[getDestination](#)

[getThread](#)

[isNotDelay](#)

[OMDelay](#)

[OMTimerManager](#)

[queueEvent](#)

[wakeup](#)

initInstance

Visibility

Public

Description

This 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 <code>OMTimeout</code> object
~OMTimeout	Destroys the <code>OMTimeout</code> object

Method Summary

operator ==	Determines whether the current values of <code>destination</code> and <code>Timeout</code> are the same as those of the specified timeout
operator >	Determines whether the current value of <code>Timeout</code> is greater than the due time of the specified timeout

<u>operator <</u>	Determines whether the current value of <code>Timeout</code> is less than the due time of the specified timeout
<u>Delete</u>	Deletes a timeout from the heap
<u>getDelay</u>	Returns the current value of <code>delayTime</code>
<u>getDueTime</u>	Returns the due time of a timeout request stored in the heap
<u>getTimeoutId</u>	Returns the current value for <code>timeoutId</code>
<u>isNotDelay</u>	Determines whether a timeout event is a timeout delay
<u>new</u>	Allocates additional memory
<u>setDelay</u>	Sets the value of <code>Timeout</code>
<u>setDueTime</u>	Specifies the value for the <code>Timeout</code> attribute
<u>setRelativeDueTime</u>	Calculates and sets the due time for a timeout based on the current system time and the requested delay time
<u>setState</u>	Used by the framework to set the current state
<u>setTimeoutId</u>	Specifies the value for <code>timeoutId</code>

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

This 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

This 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 ==](#)

[operator >](#)

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
<code>OMEventNullId</code>	-1	The null event ID
<code>OMEventTimeoutId</code>	-2	The timeout event ID
<code>OMEventCancelledEventId</code>	-3	The canceled event ID
<code>OMEventAnyEventId</code>	-4	The ID for all events delegated to a specific <code>OMReactive</code> instance
<code>OMEventStartBehaviorId</code>	-5	The ID reserved for the <code>OMReactive</code> <code>startBehavior</code> event
<code>OMEventOXFEndEventId</code>	-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

This 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

This 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

This 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

This 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

This 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

This 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

This 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
<code>OMEventNullId</code>	-1	The null event ID
<code>OMEventTimeoutId</code>	-2	The timeout event ID
<code>OMEventCancelledEventId</code>	-3	The canceled event ID
<code>OMEventAnyEventId</code>	-4	The ID for all events delegated to a specific <code>OMReactive</code> instance
<code>OMEventStartBehaviorId</code>	-5	The ID reserved for the <code>OMReactive</code> <code>startBehavior</code> event
<code>OMEventOXFEndEventId</code>	-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

<u>overflowMark</u>	Specifies the value used to determine whether the current system time has “overflowed”
-------------------------------------	----------------------------------------------------------------------------------------

Construction Summary

<u>OMTimerManager</u>	Constructs an OMTimerManager object
<u>~OMTimerManager</u>	Destroys the OMTimerManager object

Method Summary

<u>action</u>	Sends a matured timeout request to the relevant thread, where it is then inserted into the thread's event queue
<u>cbkBridge</u>	Is a bridge to get an interrupt from the operating system via the <code>timeTickCbk</code> (private) method
<u>clearInstance</u>	Cleans up the singleton instance of the timer manager
<u>consumeTime</u>	Is used in simulated time mode to simulate time consumption
<u>destroyTimer</u>	Cleans up the timer manager singleton instance
<u>getElapsedTime</u>	Returns the value of <code>m_Time</code> , the current system time.
<u>goNextAndPost</u>	Is used in simulated time mode
<u>init</u>	Starts the timer ticking
<u>initInstance</u>	Initializes the singleton instance
<u>instance</u>	Creates the singleton instance of the timer manager
<u>resume</u>	Is used by the framework to resume the timer during animation
<u>set</u>	Delegates a timeout request to OMTimerManager
<u>setElapsedTime</u>	Sets the value of <code>m_Time</code> , the current system time
<u>softUnschedTm</u>	Removes a specific timeout from the matured list
<u>suspend</u>	Is used by the framework to suspend the timer during animation
<u>unschedTm</u>	Cancel 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

This 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 is 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](#)

[setId](#)

[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 `OMIterator`, 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 <code>OMAbstractContainer</code> 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 `OMUAbstractContainer` 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 <code>OMUCollection</code> object
~OMUCollection	Destroys the <code>OMUCollection</code> 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 `OMUAbstractContainer`.

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

Construction Summary

<code>OMUIterator</code>	Constructs an <code>OMUIterator</code> object
------------------------------------------	-----------------------------------------------

Method Summary

<code>operator *</code>	Returns the current value of the iterator
<code>operator ++</code>	Increments the iterator
<code>reset</code>	Resets the iterator to the first position in the container
<code>value</code>	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 to 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 <code>OMUList</code> object
~OMUList	Destroys the <code>OMUList</code> 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 OMULListItem* 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 <code>OMUListItem</code> 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 <code>OMUMap</code> object
~OMUMap	Destroys the <code>OMUMap</code> 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 <code>OMUMapItem</code> object
~OMUMapItem	Destroys the <code>OMUMapItem</code> 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

<u>animDeregisterForeignThread</u>	Unregisters the external thread
<u>animRegisterForeignThread</u>	Registers an external thread (not an <code>OMThread</code>) in the animation framework
<u>delay</u>	Delays the calling thread for the specified length of time
<u>end</u>	Ends the event processing of the default event dispatching thread
<u>getMemoryManager</u>	Returns the current framework memory manager
<u>getTheDefaultActiveClass</u>	Returns the default active class
<u>getTheTickTimerFactory</u>	Returns the low-level timer factory
<u>init</u>	Initializes the timer, creates the default event dispatching thread, and initializes the framework
<u>setMemoryManager</u>	Specifies the current framework memory manager
<u>setTheDefaultActiveClass</u>	Registers an alternate default active object on the framework
<u>setTheTickTimerFactory</u>	Registers a timer factory on the framework, causing the framework to use the user-defined timers instead of the predefined timers
<u>start</u>	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,
    OMBBoolean 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](#)

setDefaultActiveClass

Visibility

Public

Description

The [setDefaultActiveClass](#) 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 OMBBoolean setDefaultActiveClass (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 OBoolean setTheTickTimerFactory(  
    const OAbstractTickTimerFactory* 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).

Index

Symbols

!= operator 428
* operator
 OMIterator 269
 OMString 432
 OMUIterator 523
+ operator
 OMString 422
++ operator
 OMIterator 270
 OMUIterator 524
+= operator 423
< operator
 OMString 430
<< operator 431
<= operator
 OMString 427
= operator 424
== operator
 OMString 425
 OMTimeout 473
> operator
 OMString 429
>= operator 426
>> operator 431
[] operator
 OMList 280
 OMStaticArray 414
 OMString 421
 OMUMap 547
_gen 364
_removeFirst
 OMList 288
 OMUList 537
~OMAbstractMemoryAllocator destructor 212
~OMCollection destructor 222
~OMDelay destructor 231
~OMEvent destructor 239
~OMGuard destructor 261
~OMHeap destructor 264
~OMList destructor 279
~OMMainThread destructor 295
~OMMap destructor 301
~OMMapItem destructor 310
~OMMemoryManager destructor 314
~OMMemoryManagerSwitchHelper 319

~OMProtected destructor 334
~OMQueue destructor 341
~OMReactive destructor 357
~OMStack destructor 397
~OMStaticArray destructor 413
~OMString destructor 421
~OMThread destructor 444
~OMThreadTimer destructor 466
~OMTimeout destructor 473
~OMTimerManager destructor 489
~OMUAbstractContainer destructor 503
~OMUCollection destructor 508
~OMUList destructor 528
~OMUMap destructor 546
~OMUMapItem destructor 555

A

Abstraction layer 5
action
 OMThreadTimer class 466
 OMTimerManager class 489
Active
 attribute 351
 getTheDefaultActiveClass 560
 isActive 370
 object 177
 setTheDefaultActiveClass 565
Activity diagrams 179, 182, 183, 347
Ada language 21
 framework for animation 21
Adapters 5
 classes 5, 6, 18
 operating systems 7
 validating new 30
add
 OMCollection 223
 OMHeap 265
 OMList 280
 OMMap 303
 OMStaticArray 415
 OMUCollection 510
 OMUList 530
 OMUMap 548
addAt
 OMCollection 223

Index

- OMList 281
- OMUCollection 511
- OMUList 531
- addFirst
 - OMList 282
 - OMUList 532
- AdditionalNumberOfInstances property 9
- Algorithms
 - event consumption 178
 - FIFO 186, 337
 - LIFO 186, 396
 - testing code 11
 - validation 484
- Allocate
 - allocPool 213
 - new 479
 - OMSelfLinkedMemoryAllocator 215
 - setAllocator 216
- allocPool 213
- allowDeleteInThreadsCleanup 445
- AMemAlloc.h file 27
- Analyze 199
- Animated statecharts 199
- Animation
 - Ada framework 21
- animDeregisterForeignThread 557
- animRegisterForeignThread 558
- aomthread attribute 440
- Arrays 186
 - fixed size 186
 - OMStatic 27, 28
 - OMStaticArray 411
- Attributes 351
 - active 351
 - aomthread 440
 - component 272
 - count 412
 - deleteAfterConsume 236
 - destination 236
 - eventConsumed 352
 - eventNotConsumed 352
 - eventQueue 441
 - frameworkEvent 236
 - frameworkInstance 351
 - llId 237
 - m_grow 338
 - m_head 338
 - m_myQueue 338
 - m_tail 338
 - myStartBehaviorEvent 351
 - OMDefaultThread 353
 - OMEvent class 237
 - OMEventAnyEventId 237
 - OMEventCancelledEventId 237
 - OMEventNullId 237
 - OMEventOXFEndEventId 238
 - OMEventTimeoutId 238

- omrStatus 352
- OMStartBehavior_id 238
- OMThread class 440
- OMTimeout class 471
- OMTimerManager 486
- overflowMark 486
- parent 402
- rootState 356
- size 412
- theLink 412
- thread 441
- toGuardReactive 352
- toGuardThread 441

B

- BaseNumberOfInstances property 9
- Batch files 167
 - editing 160
- Behavior 177
 - customizing timeout manager 184
 - implement reactive 3
 - package 4
 - startBehavior 391
- Blocks
 - new 214
 - SetDefaultBlock 436
- Bridge
 - cbkBridge 490
- BSP 450
- Buffer
 - GetBuffer 433

C

- C language 20
 - classes 38
 - IDF 203
 - libraries 17, 21
 - methods 38
 - RiCOSConnectionPort class 39
 - RiCOSemaphore class 67
 - RiCOSEventFlag Interface class 45
 - RiCOSMessageQueue class 50
 - RiCOSMutex class 59
 - RiCOSOXF class 64
 - RiCOSSocket class 73
 - RiCOSTask class 80
 - RiCOSTimer class 93
 - run-time source files 13
 - tracing libraries 15
 - tracing services 17
- C++ language 20
 - classes 98
 - libraries 16, 21
 - OMEventQueue class 98
 - OMMessageQueue class 100

- OMOSClass 100
- OMOSConnectionPort class 102
- OMOSFactory class 108
- OMOSMessageQueue class 117
- OMOSMutex class 123
- OMOSSemaphore class 126
- OMOSSocket class 129
- os.cpp file 29
- rebuild OXF for OSE 159
- run-time source files 13
- Callback functions 165
- Callbacks 217
 - timer manager 217
 - TimerManagerCallBack 219
- callMemoryPoolIsEmpty 213
- Calls 8
 - execute sequence of 191
 - lock and free 59
- Cancel
 - cancelEvents OMReactive class 358
 - cancelEvents OMThread class 446
 - IsCancelledTimeout 242
 - single event 192
- cancelEvent 445
- cancelEvents
 - OMReactive 358
 - OMThread class 446
- cbkBridge 490
- Classes
 - active but not reactive 177
 - adapter 5, 18
 - C++ 98
 - consuming events without SCs 178
 - getEventClass OMFriendStartBehaviorEvent 253
 - getEventClass OMFriendTimeout 256
 - getTheDefaultActiveClass 560
 - OMAbstractMemoryAllocator 212
 - OMAbstractTickTimerFactory 217
 - OMCollection 221
 - OMComponentState 227
 - OMDelay 230
 - OMEventQueue 98
 - OMEvent 233
 - OMFriendTimeout 255
 - OMGuard 258
 - OMHeap 263
 - OMInfiniteLoop 268
 - OMIterator 268
 - OMList 276
 - OMListItem 292
 - OMMainThread 294
 - OMMap 297
 - OMMapItem 309
 - OMMemoryManager 311
 - OMMemoryManagerSwitchHelper 318
 - OMMessageQueue 100
 - OMOrState 327
 - OMOS 100
 - OMOSConnectionPort 102
 - OMOSEventFlag 105
 - OMOSFactory 108
 - OMOSMessageQueue 117
 - OMOSMutex 123
 - OMOSSemaphore 126
 - OMOSSocket 129
 - OMProtected 332
 - OMQueue 337
 - OMReactive 347
 - OMStack 186, 396
 - OMStartBehaviorEvent 400
 - OMState 401
 - OMStaticArray 186, 411
 - OMString 419
 - OMThread 437
 - OMThreadTimer 465
 - OMTimeout 469
 - OMTimerManager 484
 - OMTimerManagerDefaults 502
 - OMUAbstractContainer 503
 - OMUCollection 506
 - OMUIterator 522
 - OMUList 526
 - OMUListItem 542
 - OMUMap 545
 - OMUMapItem 554
 - OSAL 37
 - RiCOSConnectionPort 39
 - RiCOSemaphore 67
 - RiCOSEventFlag Interface 45
 - RiCOSMessageQueue 50
 - RiCOSMutex 59
 - RiCOSOXF 64
 - RiCOSSocket 73
 - RiCOSTask 80
 - RiCOSTimer 93
 - RTOS 12
 - SCs for documentation 178
 - setTheDefaultActiveClass 565
- Cleanup 320
 - AllThreads 447
 - Thread 447
- clearInstance 490
- CM tools
 - Integrity 146
- Code
 - generate implementation 5
 - generation properties 24, 26
- Collections 338, 511
 - adding elements 223
 - adding template-free 510
 - creating 222
 - creating template-free 508
 - destroying 222
 - destroying template-free 508

Index

- finding an element 512
- removing all elements 225
- removing elements 224
- removing elements by index 226
- reorganizing 226
- Command-line
 - attributes 159
 - flags 159
- Commands
 - all 16
 - build framework libraries for C or C++ 21
 - definitions 171
 - make 16
 - RHAP_FLAGS 16
- Communication link 8
- Communication ports 5, 9
- CompareNoCase 432
- Compilation
 - flags 170
- Compilers 5
 - environments 13
 - Forte 23
 - GNAT 21
 - GNU 23
 - Green Hills 19
 - memory allocation control 312
 - Microsoft 20
 - multiple installed 22
 - native 30
 - path to 23
 - provided cross 14
- CompileSwitches property 170
- Complete
 - runToCompletion 379
 - shouldCompleteRun 387
- Component attribute 272
- Concept
 - getFirstConcept 285
 - getLastConcept 286
 - OMFinalState 251
 - OMMapItem 310
 - OMState 404
- Configurations
 - active 147
- connectTo
 - OMListItem 293
 - OMULListItem 543
- Constructors
 - OMAndState 219
 - OMCollection 222
 - OMComponentState 228
 - OMDelay 231
 - OMEvent 238
 - OMFinalState 250
 - OMFriendStartBehaviorEvent 252
 - OMFriendTimeout 255
 - OMGuard 261
 - OMHeap 264
 - OMIterator 269
 - OMLeafState 273
 - OMList 279
 - OMListItem 292
 - OMMap 301
 - OMMapItem 309
 - OMMemoryManager 314
 - OMMemoryManagerSwitchHelper 319
 - OMOrState 328
 - OMProtected 333
 - OMQueue 341
 - OMReactive 357
 - OMStack 396
 - OMStartBehaviorEvent 400
 - OMState 403
 - OMStaticArray 413
 - OMString 420
 - OMThread 442
 - OMTimeout 472
 - OMTimerManager 487
 - OMUCollection 508
 - OMUIterator 523
 - OMUList 528
 - OMUListItem 542
 - OMUMap 546
 - OMUMapItem 554
- consumeEvent 358
- consumeTime 491
- Consumption
 - handleEventNotConsumed 366
 - handleTONotConsumed 367
 - isDeleteAfterConsume 242
 - modifying 178
 - of triggered events 393
 - setDeleteAfterConsume 246
- Container types 186
 - OMAbstractContainer 186
 - OMCollection 186
 - OMHeap 186
 - OMIterator 186
 - OMList 186
 - OMMap 186
 - OMQueue 186
 - OMStack 186
 - OMStaticArray 186
 - OMString 186
- Containers 186
- Containers package 185
- CORBA 234
- Count
 - elements 397
 - getCount 284
 - getCount OMMap 305
 - getCount OMStaticArray 416
 - getCount OMUCollection 513
 - getCount OMUList 535

- getCount OMUMap 550
- count attribute 412
- createRealTimeTimer 217
- createSimulatedTimeTimer 218
- cserialize
 - OMFriendStartBehaviorEvent class 253
 - OMFriendTimeout 256
- Current
 - event 371
 - getCurrentEvent 365
 - OMList 284
- Customizing
 - automated behavior code 3
 - framework 3, 201
 - installation for OS/RTOS 11
 - OMThread 181
 - operator 432
 - timeout framework behavior 184

D

- Debug 199
- Debugging 484
- DECLARE_MEMORY_ALLOCATOR macro 471
- decNonIdleThreadCounter 491
- Decrement
 - null transitions 374
- Default
 - getTheDefaultActiveClass 560
 - setTheDefaultActiveClass 565
 - transition 377
- defaultHost 562
- defaultMaxTM constant 502
- defaultPort 562
- defaultTicktime constant 502
- Defines 165
- Delay 559
 - getDelay 477
 - isNotDelay 479
 - setDelay 480
 - timeout 197
- Delete
 - isDeleteAfterConsume 242
 - OMEvent 240
 - setDeleteAfterConsume 246
 - setShouldDelete 383
 - shouldDelete 389
 - Timeout 476
- deleteAfterConsume attribute 236
- deleteMutex 334
- Dependencies 173
- Deployment environment 5
- Deregister
 - animDeregisterForeignThread 557
- Derived event 245
- Destination
 - attribute 236
- canceling events to 192
- getting 240
- setting 247
- destroyThread
 - OMMainThread class 295
 - OMThread class 448
- destroyTimer 492
- Destructors
 - ~OMCollection 222
 - ~OMDelay 231
 - ~OMEvent 239
 - ~OMGuard 261
 - ~OMHeap 264
 - ~OMList 279
 - ~OMMainThread 295
 - ~OMMap 301
 - ~OMMapItem 310
 - ~OMMemoryManager 314
 - ~OMProtected 334
 - ~OMQueue 341
 - ~OMReactive 357
 - ~OMStack 397
 - ~OMStaticArray 413
 - ~OMString 421
 - ~OMThread 444
 - ~OMThreadTimer 466
 - ~OMTimeout 473
 - ~OMTimerManager 489
 - ~OMUCollection 508
 - ~OMUList 528
 - ~OMUMap 546
 - ~OMUMapItem 555
 - OMAbstractMemoryAllocator 212
 - OMUAbstractContainer 503
- Development environment 24
- Diagrams
 - activity 179, 182, 183, 347
 - object model 8
 - sequence 108, 196, 199
 - statecharts 2, 179, 182, 183, 199, 347
- discarnateTimeout 360
- Dispatch
 - event 191
 - rootState_dispatchEvent 376
 - stopping 373
 - timeout 184
 - timeout SD 196
 - triggered operation 193
- dispatched timeout 196
- doBusy 361
- doExecute 448
- DTOR
 - isInDtor 373
 - setEndOSThreadInDtor 458
 - setInDtor 382
 - START_DTOR_REACTIVE_GUARDED_SECTIO
N 259

Index

- Due time 183
 - getting 477
 - relative 481
 - setting 481
- E**
- Eclipse
 - Momentics IDE 162
- Elements
 - adding to collections 223
 - adding to template-free collections 510
 - adding to template-free collections at a given location 511
 - getElement OMListItem 543
 - getElement OMUMapItem 555
 - remove all from static array 418
 - remove from heap 266
 - remove from list 288
 - remove from template-free list 538, 539
 - remove from template-free map 552, 553
 - removing 224
 - removing all from collections 225
 - removing all from template-free collections 518
 - removing from template-free collections 517
 - setElement 544
- Empty 433
- end 559
- END_REACTIVE_GUARDED_SECTION macro 259
- END_THREAD_GUARDED_SECTION macro 259
- endOfProcess flag 440
- entDef
 - OMLeafState 273
 - OMOrState 328
 - OMState 403
- enterState
 - OMComponentState 228
 - OMLeafState 274
 - OMOrState 328
 - OMState 404
- entHist 403
- EntryPoint property 166
- Environment
 - property 25, 26
 - setting the new 26
- Environment property 26
- Environments
 - changing target 145
 - deployment 5
 - for development 24
- Event
 - setEventGuard 380
- Event flag 45
- Event loop 191
- Event queue
 - omGetEventQueue 454
 - versus communication port 9
- event relation 355
- event.cpp file 27
- event.h file 27
- eventConsumed 352
- eventNotConsumed 352
- eventQueue attribute 441
- Events 182, 189
 - cancelEvent 445
 - cancelEvents 358
 - cancelEvents OMThread class 446
 - canceling a single 192
 - canceling all 192
 - canceling all events 192
 - consumeEvent 358
 - consumption algorithm 178
 - creating 190
 - current 371
 - derived 245
 - dispatching 191
 - flag 105
 - generate & queue 190
 - generating 190
 - generic handling 402
 - getCurrentEvent 365
 - getEventClass OMFriendTimeout 256
 - getEventQueue 451
 - handleEventNotConsumed 366
 - handling 189
 - isFrameworkEvent 243
 - isRealEvent 244
 - modifying consumption 178
 - OMFriendStartBehaviorEvent 252, 253
 - OMReactive 392
 - processing loop 448
 - queueEvent 455
 - queuing 190
 - rootState_dispatchEvent 376
 - setFrameworkEvent 247
 - stopping dispatch 382
 - synchronous 8
 - takeEvent OMComponentState 229
 - takeEvent OMState 410
- Example
 - OMList 278
 - OMMap class 298
 - OMQueue class 339
 - OMStaticArray 412
 - OMUList 527
- Execute 449
 - doExecute 448
- exitState
 - OMLeafState 274
 - OMOrState 329
 - OMState 404
- External thread
 - animRegisterForeignThread 558
 - deregistering 557

F

Factory

getTheTickTimerFactory 561
setTheTickTimerFactory 566

Features dialog box 27

configuration 26
Environment 147
Instrumentation Mode 147

FIFO algorithm 186, 337

Files

.mak 145
batch 14, 160, 167
event.cpp 27
event.h 27
make 5
MemAlloc.h 27
omabscon.h 27
omcollec.h 27
omcon.h 27
omheap.h 27
omlist.h 27
ommap.h 27
omoutput.cpp 28
omoutput.h 28
omprotected.h 28
omqueue.h 28
omreactive.cpp 28
omreactive.h 28
omstack.h 28
omstatic.h 28
omstring.cpp 28
omstring.h 28
omthread.cpp 28
omthread.h 28
omtypes.h 28
os.cpp file 29
os.h 28
os.h 29
oxf.cpp 28
oxf.h 28
rawtypes.h 28
run-time sources 13
sol2shr.tar file for Solaris 22
state.cpp 28
state.h 28
timer.cpp 29
timer.h 28

find

OMHeap 265
OMList 283
OMMap 304
OMStaticArray 415
OMUCollection 512
OMUList 533
OMUMap 549

First

_removeFirst OMList 288
_removeFirst OMUList 537
addFirst OMList 282
addFirst OMUList 532
getFirst OMList 285
getFirst OMUAbstractContainer 504
getFirst OMUCollection 514
getFirst OMUList 536
removeFirst OMList 289
removeFirst OMUList 539

Flags 278, 440

command for Rational Rhapsody 16
command-line 159
compilation 170
event 105
OMComponentState 227
OMList 278
OMUList 527
stopDelay 230
subState 327

frameworkEvent attribute 236

frameworkInstance attribute 351

Frameworks 20, 67

Ada 21
advantages of 1
build libraries 20
C 31
C++ 31
C++ properties 27
customizing 1, 3, 201
initializing 562
isFrameworkEvent 243
isFrameworkInstance 372
Java 22
modifying OXF 31
object execution 5
OXF 3, 562
port number into connection point 9
properties 27
real-time 1
rebuilding 145
setFrameworkEvent 247
setFrameworkInstance 381
Solaris 22
start 567
starting the timer 494
VxWorks 164

free 334

Functions 165
callback 165

G

gen 361
GEN macro 354
GEN_BY_GUI macro 354
GEN_BY_X macro 355

Index

- GEN_ISR macro 355
- Generate
 - event 190
- Generated macros 172
- Generic event handling 402
- Get
 - current system time 492
 - destination 240
 - getId 241
- get 342
- getAOMThread 451
- getAt
 - OMList 283
 - OMMap 304
 - OMStaticArray 416
 - OMUCollection 513
 - OMUList 534
 - OMUMap 549
- GetBuffer 433
- getConcept
 - OMFinalState 251
 - OMMapItem 310
 - OMState 404
- getCount
 - OMList 284
 - OMMap 305
 - OMStack 397
 - OMStaticArray 416
 - OMUCollection 513
 - OMUList 535
 - OMUMap 550
- getCurrent
 - OMUAbstractContainer 504
 - OMUCollection 514
 - OMUList 535
- getCurrentEvent 365
- getDefaultMemoryManager 315
- getDelay 477
- getDestination 240
- getDueTime 477
- getElapsedTime 492
- getElement
 - OMUListItem 543
 - OMUMapItem 555
- getEventClass
 - OMFriendStartBehaviorEvent 253
 - OMFriendTimeout 256
- getEventQueue 451
- getFirst
 - OMList 285
 - OMUAbstractContainer 504
 - OMUCollection 514
 - OMUList 536
- getFirstConcept 285
- getGuard
 - OMGuard 262
 - OMProtected 335
 - OMThread 451
- getHandle 405
- getInverseQueue 343
- getKey
 - OMMap 305
 - OMUMap 550
- getLast 286
- getLastConcept 286
- getLastState 405
- GetLength 434
- getId 241
- getMemory 316
 - OMAbstractMemoryAllocator 214
 - OMMemoryManager 316
- getMemoryManager
 - OMMemoryManager 316
 - OMMemoryManager class 316
 - OXF class 560
- getNext
 - OMList 287
 - OMListItem 293
 - OMUAbstractContainer 505
 - OMUCollection 515
 - OMUList 536
 - OMUListItem 543
- getOsHandle 452
- getOsThreadEndClib 453
- getQueue 343
- getSize 515
 - OMQueue 344
 - OMStaticArray 417
- getStepper 454
- getSubState
 - OMOrState 329
 - OMState 406
- getTheDefaultActiveClass 560
- getTheTickTimerFactory 561
- getThread 365
- getTimeoutId 478
- GNAT compiler 21
- goNextAndPost 493
- Guard
 - END_REACTIVE_GUARDED_SECTION 259
 - END_THREAD_GUARDED_SECTION 259
 - getGuard OMGuard 262
 - getGuard OMProtected 335
 - getGuard OMThread 451
 - GUARD_OPERATION 259
 - OMGuard class 258
 - setEventGuard 380
 - setToGuardReactive 386
 - setToGuardThread 459
 - shouldGuardThread 460
 - START_DTOR_REACTIVE_GUARDED_SECTION 259
 - START_DTOR_THREAD_GUARDED_SECTION

260
 START_REACTIVE_GUARDED_SECTION 260
 START_THREAD_GUARDED_SECTION 260
 GUARD_OPERATION macro 259

H

Handle

derived events 245
 getHandle 405
 getOsHandle 452
 setHandle 408
 handleEventNotConsumed 366
 handleTONotConsumed 367

Head

increaseHead_ 344

Heap

class 263
 empty 266
 finding an element 265
 remove elements from 266
 removing elements 266

Helpers

OMMemoryManagerSwitch 318

Host 562

I

ID

of a timeout event 194
 SetTimeoutId 483

IDE 165

IDE interface 165

IDF 203

Idle timer 10

Implementation 6

in

OMComponentState 228
 OMLeafState 274
 OMOOrState 330
 OMState 406

incarnateTimeout 368

incNonIdleThreadCounter 493

increaseHead_ 344

increaseTail_ 344

Increment

setIncrementNum 216

increment 270

Index

addAt OMCollection 223
 addAt OMUCollection 511
 removing elements collections 226
 removing elements from template-free
 collections 519

init

OXF 562
 init OMTimerManager 494

Initialize

framework 562
 init OMTimerManager 494
 init OXF 562
 initInstance 467
 initInstance OMThreadTimer 467
 instance OMTimerManager 494
 mutex 335
 timer 494

initializeMutex 335

initiatePool 214

initInstance

OMThreadTimer 467
 OMTimerManager 494

inNullConfig 369

Instance

clearInstance 490
 initializing 467
 initInstance OMThreadTimer 467
 initInstance OMTimerManager 494
 isFrameworkInstance 372
 setFrameworkInstance 381
 thread 295

instance

OMMainThread 295
 OMMainThread class 295
 OMMemoryManagerSwitchHelper class 321
 OMTimerManager 495

Integrated Development Environment (IDE) 165

Integrity 146

Interfaces

implementing 6
 RiCOSEventFlag 45

Interprocess

communication 9

Interrupt

cbkBridge 490

InvokeExecutable property 167

InvokeMake property 21

IS_EVENT_TYPE_OF macro 402

isActive 370

isBusy 370

isCancelledTimeout 242

isCompleted 407

isCurrentEvent 371

isDeleteAfterConsume 242

isEmpty 306

OMHeap 266

OMList 287

OMQueue 345

OMStack 398

OMStaticArray 417

OMString 434

OMUCollection 516

OMUList 537

OMUMap 551

isFrameworkEvent 243

Index

isFrameworkInstance 372
isFull 345
isInDtor 373
isLogEmpty 321
isNotDelay 479
isRealEvent 244
isRealTimeModel 563
isTimeout 244
isTypeOf 245
isTypeOf method 245
isValid 373
Item
 removeItem 290
 removeItem OMUList 540

J

Java language 22
 jar command 175
 libraries 17

K

Key
 getKey OMMMap 305
 getKey OMUMap 550
 removing 553

L

Last
 getLast 286
 getLastConcept 286
 getLastState 405
 removeLast OMList 291
 removeLast OMUList 541
 setLastState 409
Length
 GetLength 434
Libraries 159
 build framework 21
 C 17
 C tracing 15
 C++ 16
 framework 20
 Java 17
 OXF 4
 VxWorks 164
lld attribute 237
 setting 248
LIFO algorithm 186, 396
Link 8
List
 empty 287
 finding an element in a list 283
 finding an element in a template-free list 533
 first element 285

 next element 287
 number of elements 284
 removing all elements 289
 removing elements 288
 removing last element 291
lock
 OMAndState 220
 OMGuard 262
 OMProtected 336
 OMThread 454
lookUp
 OMMap 306
 OMUMap 551

M

m_eventGuard constant 356
m_grow attribute 338
m_head attribute 338
m_myQueue attribute 338
m_tail attribute 338
Macro
 DECLARE_MEMORY_ALLOCATOR 471
 defined in omprotected.h 259
 END_REACTIVE_GUARDED_SECTION 259
 END_THREAD_GUARDED_SECTION 259
 GUARD_OPERATION 259
 IS_EVENT_TYPE_OF 402
 OM_DECLARE_FRAMEWORK_MEMORY_ALLOCATION_OPERATORS 402
 OMDECLARE_GUARDED 259
 OMReactive class 354
 START_DTOR_REACTIVE_GUARDED_SECTION 259
 START_DTOR_THREAD_GUARDED_SECTION 260
 START_REACTIVE_GUARDED_SECTION 260
 START_THREAD_GUARDED_SECTION 260
Macros
 generated 172
 OMDECLARE_GUARED 332
 predefined 173
Make files 5
MakeFileContent property 14, 169, 175
Makefiles 14, 145, 167
 creating 14
 creating new 16
 linking 174
 modifying 13
 properties 168
 target type 170
 VxWorks sample file 15
Map
 finding an element in a map 304
 finding an element in a template-free map 549
 getting the key 305
 looking up an element 306

- number of elements 305
- removing an element 307
- Maps
 - remove elements 552
 - remove elements from 307
 - removing all elements 308
 - template-free 553
- Matured list
 - removing a timeout 499
- maxTM 563
- MemAlloc.h file 27
- Memory
 - ~OMAbstractMemoryAllocator 212
 - allocating 479
 - callMemoryPoolIsEmpty 213
 - getDefaultMemoryManager 315
 - getMemory 316
 - getMemory OMAbstractMemoryAllocator 214
 - getMemoryManager 560
 - OMAbstractMemoryAllocator class 212
 - OMSelfLinkedMemoryAllocator 215
 - pool 9
 - returnMemory OMAbstractMemoryAllocator 215
 - setMemoryManager 564
- Memory management
 - getMemory 316
 - getMemoryManager
 - OMMemoryManager class 316
 - OXF class 560
 - package 185
 - returnMemory 317
- Memory pool 9
- Message queue 8
- Messages
 - queues 5, 8
- Methods
 - _gen 364
 - _removeFirst OMList 288
 - _removeFirst OMUList 537
 - ~OMAbstractMemoryAllocator 212
 - ~OMCollection 222
 - ~OMDelay 231
 - ~OMEvent 239
 - ~OMGuard 261
 - ~OMHeap 264
 - ~OMList 279
 - ~OMMainThread 295
 - ~OMMap 301
 - ~OMMapItem 310
 - ~OMMemoryManager 314
 - ~OMProtected 334
 - ~OMQueue 341
 - ~OMReactive 357
 - ~OMStack 397
 - ~OMString 421
 - ~OMThread 444
 - ~OMThreadTimer 466
 - ~OMTimeout 473
 - ~OMTimerManager 489
 - ~OMUMapItem 555
 - action OMThreadTimer 466
 - action OMTimerManager 489
 - add OMCollection 223
 - add OMHeap 265
 - add OMList 280
 - add OMMMap 303
 - add OMStaticArray 415
 - add OMUCollection 510
 - add OMUList 530
 - add OMUMap 548
 - addAt OMCollection 223
 - addAt OMList 281
 - addAt OMUCollection 511
 - addAt OMUList 531
 - addFirst OMList 282
 - addFirst OMUList 532
 - allocPool 213
 - animDeregisterForeignThread 557
 - animRegisterForeignThread 558
 - callMemoryPoolIsEmpty 213
 - cancelEvent 445
 - cancelEvents 358
 - cancelEvents OMThread class 446
 - cbkBridge 490
 - cleanup 320
 - cleanupAllThreads 447
 - cleanupThread 447
 - clearInstance 490
 - CompareNoCase 432
 - connectTo OMListItem 293
 - connectTo OMUListItem 543
 - consumeEvent 358
 - consumeTime 491
 - createRealTimeTimer 217
 - createSimulatedTimeTimer 218
 - cserialize OMFriendStartBehaviorEvent 253
 - cserialize OMFriendTimeout 256
 - decNonIdleThreadCounter 491
 - delay 559
 - Delete 240
 - Delete OMTimeout class 476
 - deleteMutex 334
 - destroyThread OMMainThread class 295
 - destroyThread OMThread class 448
 - destroyTimer 492
 - discarnateTimeout 360
 - doBusy 361
 - doExecute 448
 - Empty 433
 - end 559
 - entDef OMLeafState 273
 - entDef OMOrState 328
 - entDef OMState 403
 - enterState OMComponentState 228

enterState OMLeafState 274
 enterState OMOrState 328
 enterState OMState 404
 entHist 403
 execute 449
 exitState OMLeafState 274
 exitState OMOrState 329
 exitState OMState 404
 find
 OMStaticArray 415
 OMUCollection 512
 find OMHeap 265
 find OMList 283
 find OMMMap 304
 find OMUList 533
 find OMUMap 549
 findMemory 320
 free 334
 gen 361
 get 342
 getAOMThread 451
 getAt
 OMList 283
 OMStaticArray 416
 OMUCollection 513
 OMUMap 549
 getAt OMMMap 304
 getAt OMUList 534
 GetBuffer 433
 getConcept OMFfinalState 251
 getConcept OMMMapItem 310
 getConcept OMState 404
 getCount 284
 getCount OMMMap 305
 getCount OMStack 397
 getCount OMStaticArray 416
 getCount OMUCollection 513
 getCount OMUList 535
 getCount OMUMap 550
 getCurrent OMList 284
 getCurrent OMUAbstractContainer 504
 getCurrent OMUCollection 514
 getCurrent OMUList 535
 getCurrentEvent 365
 getDefaultMemoryManager 315
 getDelay 477
 getDestination 240
 getDueTime 477
 getElapsedTime 492
 getElement OMUListItem 543
 getElement OMUMapItem 555
 getEmpty 306
 getEventClass OMFfriendStartBehaviorEvent 253
 getEventClass OMFfriendTimeout 256
 getEventQueue 451
 getFirst
 OMUAbstractContainer 504
 getFirst OMList 285
 getFirst OMUCollection 514
 getFirst OMUList 536
 getFirstConcept 285
 getGuard OMGuard 262
 getGuard OMProtected 335
 getGuard OMThread 451
 getHandle 405
 getInverseQueue 343
 getKey OMMMap 305
 getKey OMUMap 550
 getLast 286
 getLastConcept 286
 getLastState 405
 GetLength 434
 getId 241
 getMemory
 OMMemoryManager 316
 getMemory OMAbstractMemoryAllocator 214
 getMemoryManager
 OMMemoryManager 316
 OXF class 560
 getNext
 OMList 287
 OMListItem 293
 OMUAbstractContainer 505
 OMUCollection 515
 getNext OMUList 536
 getNext OMUListItem 543
 getOsHandle 452
 getOSThreadEndClib 453
 getQueue 343
 getSize 515
 getSize OMQueue 344
 getSize OMStaticArray 417
 getStepper 454
 getSubState OMOrState 329
 getSubState OMState 406
 getTheDefaultActiveClass 560
 getTheTickTimerFactory 561
 getThread 365
 getTimeoutId 478
 goNextAndPost 493
 handleEventNotConsumed 366
 handleTONotConsumed 367
 in OMComponentState 228
 in OMLeafState 274
 in OMOrState 330
 in OMState 406
 incarnateTimeout 368
 incNonIdleThreadCounter 493
 increaseHead_ 344
 increaseTail_ 344
 increment 270
 init OMTimerManager 494
 init OXF 562
 initializeMutex 335

- initiatePool 214
- initInstance OMThreadTimer 467
- initInstance OMTimerManager 494
- inNullConfig 369
- instance
 - OMMemoryManagerSwitchHelper 321
 - OMTimerManager 495
- instance OMMainThread 295
- isActive 370
- isBusy 370
- isCancelledTimeout 242
- isCompleted 407
- isCurrentEvent 371
- isDeleteAfterConsume 242
- IsEmpty
 - OMString 434
- isEmpty
 - OMList 287
 - OMQueue 345
 - OMStack 398
 - OMStaticArray 417
 - OMUCollection 516
 - OMUMap 551
- isEmpty OMHeap 266
- isEmpty OMMMap 306
- isEmpty OMUList 537
- isFrameworkEvent 243
- isFrameworkInstance 372
- isFull 345
- isInDtor 373
- isLogEmpty 321
- isNotDelay 479
- isRealEvent 244
- isTimeout 244
- isTypeOf 245
- isTypeof 245
- isValid 373
- lock
 - OMProtected 336
 - OMThread 454
- lock OMAAndState 220
- lock OMGuard 262
- lookUp
 - OMUMap 551
- lookUp OMMMap 306
- new 479
- OMAndState 219
- OMCollection 222
- OMComponentState 228
- OMDelay 231
- OMDestructiveString2X 435
- OMEvent 238
- OMFinalState 250
- OMFriendStartBehaviorEvent 252
- omGetEventQueue 454
- OMGuard 261
- OMHeap 264
- OMLeafState 273
- OMList 279
- OMListItem 292
- OMMap 301
- OMMapItem 309
- OMMemoryManager 314
- OMMemoryManagerSwitchHelper 319
- OMProtected 333
- OMQueue 341
- OMReactive 357, 392
- OMSelfLinkedMemoryAllocator 215
- OMStack 396, 399
- OMStartBehaviorEvent 400
- OMState 403, 410
- OMString 420
- OMThread 442
- OMTimeout 472
- OMTimerManager 487
- OMUMapItem 554
- OSAL 34
- pop 398
- popNullConfig 374
- push 399
- pushNullConfig 375
- put 346
- queueEvent 455
- recordMemoryAllocation 322
- recordMemoryDeallocation 323
- registerWithOMReactive 375
- remove
 - OMUCollection 517
- remove OMCollection 224
- remove OMHeap 266
- remove OMList 288
- remove OMMMap 307
- remove OMUList 538
- remove OMUMap 552
- removeAll OMCollection 225
- removeAll OMList 289
- removeAll OMMMap 308
- removeAll OMStaticArray 418
- removeAll OMUCollection 518
- removeAll OMUList 539
- removeAll OMUMap 553
- removeByIndex
 - OMUCollection 519
- removeByIndex OMCollection 226
- removeFirst OMList 289
- removeFirst OMUList 539
- removeItem OMList 290
- removeItem OMUList 540
- removeKey 553
- removeLast OMList 291
- removeLast OMUList 541
- reorganize OMCollection 226
- reorganize OMUCollection 520
- reset

- OMIterator 271
 - OMUIterator 524
 - resetSize 435
 - resume 456
 - returnMemory 215
 - OMMemoryManager 317
 - returnMemory OMAbstractMemoryAllocator 215
 - rootState_dispatchEvent 376
 - rootState_entdef 377
 - rootState_serializeStates 378
 - runToCompletion 379
 - schedTm 456
 - serialize OMFriendStartBehaviorEvent 254
 - serialize OMFriendTimeout 257
 - serializeStates
 - OMReactive 379
 - OMState 408
 - serializeStates OMLeafState 275
 - serializeStates OMorState 330
 - set 497
 - setAllocator 216
 - SetAt 436
 - setAt
 - OMStaticArray 418
 - OMUCollection 521
 - setCompleteStartBehavior 380
 - setDefaultBlock 436
 - setDelay 480
 - setDeleteAfterConsume 246
 - setDestination 247
 - setDueTime 481
 - setElapsedTime 498
 - setElement 544
 - setEndOSThreadInDtor 458
 - setEventGuard 380
 - setFrameworkEvent 247
 - setFrameworkInstance 381
 - setHandle 408
 - setIncrementNum 216
 - setInDtor 382
 - setLastState 409
 - setId 248
 - setMaxNullSteps 382
 - setMemoryManager 564
 - setPriority 459
 - setRelativeDueTime 481
 - setShouldDelete 383
 - setShouldTerminate 384
 - setState 482
 - setSubState OMorState 331
 - setSubState OMState 409
 - setDefaultActiveClass 565
 - setTheTickTimerFactory 566
 - setThread 385
 - setTimeoutId 483
 - setToGuardReactive 386
 - setToGuardThread 459
 - setUpdateState 324
 - shouldCompleteRun 387
 - shouldCompleteStartBehavior 388
 - shouldDelete 389
 - shouldGuardThread 460
 - shouldTerminate 390
 - shouldUpdate 324
 - softUnschedTm 499
 - start
 - OMMainThread class 296
 - OMThread class 460
 - OXF 567
 - startBehavior 391
 - stopAllThreads 461
 - suspend 462
 - takeEvent OMComponentState 229
 - takeTrigger 393
 - terminate 394
 - TimerManagerCallback 219
 - top
 - OMStack 399
 - top OMHeap 267
 - trim 267
 - undoBusy 395
 - unlock
 - OMProtected 336
 - OMThread 462
 - unlock OMAndState 220
 - unlock OMGuard 262
 - unschedTm
 - OMThread class 463
 - OMTimerManager class 500
 - update 267
 - value
 - OMIterator 271
 - OMUIterator 525
 - wakeup 232
 - Model
 - testing 199
 - Modify
 - event consumption 178
 - Momentics IDE 162
 - Mutex 8, 59
 - deleting 334
 - free in guarded operation 258
 - goNextAndPost 493
 - initializeMutex 335
 - lock 220
 - unlock 220
 - Mutual exclusion (mutex) 123
 - myStartBehaviorEvent attribute 351
 - myThread relation 356
- ## N
- Next (getNext)
 - OMList 287

- OMListItem 293
 - OMUAbstractContainer 505
 - OMUCollection 515
 - OMUList 536
 - OMUListItem 543
 - Null transition
 - decrementing 374
 - incrementing 375
 - maximum number 382
 - taking 379
 - Null transitions 369
 - numProgArgs 562
- O**
- Object execution framework (OXF) 5
 - Objects
 - active 181
 - OM_DECLARE_FRAMEWORK_MEMORY_ALLOC
ATION_OPERATORS macro 402
 - OM_ENABLE_MEMORY_MANAGER_SWITCH
switch 322, 323
 - omabscon.h file 27
 - OMAbstractMemoryAllocator class 212
 - OMAbstractTickTimerFactory class 217
 - OMAndState class 219
 - constructor 219
 - omcollec.h file 27
 - OMCollection 186
 - OMCollection class 221
 - constructor 222
 - OMComponentState class 227
 - constructor 228
 - omcon.h file 27
 - OMDECLARE_GUARDED macro 259
 - OMDefaultThread attribute 353
 - OMDelay class 181, 230
 - constructor 231
 - stopDelay flag 230
 - OMDestructiveString2X 435
 - OMEvent class 182, 233
 - attributes 237
 - Behavior package 177
 - constructor 238
 - OMEventAnyEventId attribute 237
 - OMEventCancelledEventId attribute 237
 - OMEventNullId attribute 237
 - OMEventOXFEndEventId attribute 238
 - OMEventTimeoutId attribute 238
 - OMFinalState class 249
 - constructor 250
 - OMFriendStartBehaviorEvent class 252
 - constructor 252
 - OMFriendTimeout class 255
 - constructor 255
 - omGetEventQueue method 454
 - OMGuard class 182, 258
 - 4.0 changes 258
 - constructor 261
 - OMHeap 186
 - OMHeap class 263
 - constructor 264
 - omheap.h file 27
 - OMInfiniteLoop class 268
 - OMIterator 186
 - OMIterator class 268
 - constructor 269
 - OMLeafState class 272
 - constructor 273
 - OMList 186
 - OMList class 276
 - constructor 279
 - omlist.h file 27
 - OMListItem class 292
 - constructor 292
 - OMMainThread class 181, 294
 - OMMap 186, 302
 - OMMap class 297
 - constructor 301
 - example 298
 - ommap.h file 27
 - OMMapItem class 309
 - OMMapItem constructor 309
 - OMMemoryManager class 311
 - constructor 314
 - OMMemoryManagerSwitchHelper constructor 319
 - OMMemoryManagerSwitchHelper switch 318
 - OMOrState class 327
 - constructor 328
 - OMOSConnectionPort class 12
 - OMOSEventFlag class 12
 - OMOSFactory class 12
 - OMOSMessageQueue class 12
 - OMOSMutex class 12
 - OMOSSemaphore class 12
 - OMOSSocket class 12
 - OMOSThread class 12
 - OMOSTimer class 12
 - timer service 10
 - omoutput.cpp file 28
 - omoutput.h file 28
 - OMProtected class 177, 182
 - constructor 333
 - declaration 332
 - omprotected.h file 28
 - OMQueue 186
 - OMQueue class 337
 - constructor 341
 - example 339
 - omqueue.h file 28
 - OMRDefaultStatus constant 352
 - OMReactive class 177, 179, 358
 - attributes 351
 - constants 352

- constructor 357
- declaration 347
- defines and macros 354
- relations 355
- omreactive.cpp file 28
- omreactive.h file 28
- OMRInDtor constant 353
- OMRNullConfig constant 353
- OMRNullConfigMask constant 353
- OMROOT 31
- OMRShouldCompleteStartBehavior constant 353
- OMRShouldDelete constant 354
- OMRShouldTerminate constant 354
- omrStatus
 - doBusy 361
 - isBusy 370
 - undoBusy 395
- omrStatus attribute 352
- OMSelfLinkedMemoryAllocator 215
- OMStack 186
- OMStack class 396
 - constructor 396
- omstack.h file 28
- OMStartBehavior_id attribute 238
- OMStartBehaviorEvent class 400
 - constructor 400
- OMState class 401
 - constructor 403
- omstatic.h file 28
- OMStaticArray 27, 28, 186
- OMStaticArray class 411
 - constructor 413
- OMString 186
- OMString class 419
 - constructor 420
- omstring.cpp file 28
- omstring.h file 28
- OMThread class 177, 181
 - attributes 440
 - constructor 442
 - declaration 437
- omthread.cpp file 28
- omthread.h file 28
- OMThreadTimer class 184
 - declaration 465
- OMTimeout 473
- OMTimeout class 183, 469
 - attribute 471
 - constructor 472
- OMTimerManager attribute 486
- OMTimerManager class 177, 183
 - constructor 487
 - declaration 484
- OMTimerManagerDefaults class 184
 - declaration 502
- omtypes.h file 28
- OMUAbstractContainer class 503
- OMUCollection class 506
 - constructor 508
- OMUIterator class 522
 - constructor 523
- OMUList class 526
 - constructor 528
 - example 527
- OMUListItem class 542
 - constructor 542
- OMUMap class 545
 - constructor 546
- OMUMapItem class 554
 - constructor 554
- Operating system
 - services 5
- Operating systems
 - adapters 7
 - new as default 25
 - real-time 5
 - RTOS 11
 - services 5, 32
 - Solaris 19
- Operation
 - unconsumed 367
- Operations
 - dispatch triggered 193
 - triggered 182, 347
 - virtual 32
- operator 280
- Operators
 - != 428
 - *
 - OMIterator 269
 - OMString 432
 - OMUIterator 523
 - +
 - OMString 422
 - ++
 - OMIterator 270
 - OMUIterator 524
 - += 423
 - <
 - OMString 430
 - << 431
 - <= 427
 - = 424
 - == 473
 - OMString 425
 - >
 - OMString 429
 - >= 426
 - >> 431
 - []
 - OMStaticArray 414
 - OMString 421
 - OMUMap 547
 - new 479

- OMMap 302
- OMTimeout 474, 475
- OMUCollection 509
- OMUList 529
- OS abstraction layer (OSAL) 5
- os.h file 28
- OSAL 5
 - AbstractLayer package 11
 - classes 37
 - methods 34
 - services 5
- OSLayer package 4
- OSWrappers package 12
- overflowMark attribute 486
- OXF 3, 5
 - end 559
 - general class 556
 - init 562
 - library 4
 - start 567
 - working with 3
- oxf.cpp file 28
- oxf.h file 28

P

- Packages
 - AbstractLayer for OSAL 11
 - behavioral 4, 177
 - containers 185
 - operating system 4
 - services 4
- parent attribute 402
- Pool
 - allocPool 213
 - callMemoryPoolIsEmpty 213
 - initiatePool 214
- pop 398
- popNullConfig 374
- Port 562
- Ports
 - animation 9
 - communication 9
 - number 9
- Predefined macros 173
- Priority
 - setPriority 459
- Processes
 - communication 8
 - lightweight 7
- progArgs 562
- Properties 24
 - AdditionalNumberOfInstances 9
 - BaseNumberOfInstances 9
 - C++ framework 27
 - CompileSwitches 170
 - customizing for a new RTOS 24

- Environment 26
- framework 27
- InvokeExecutable 167
- makefile 168
- MakeFileContent 169, 175
- modifying 13
- RTOS 24
- push 399
- pushNullConfig 375
- put 346

Q

- Quality of service 200
- Queue
 - event 190
 - full 345
 - get 343
 - getEventQueue 451
 - getInverseQueue 343
 - getQueue 343
 - increasing the head 344
 - increasing the tail 344
 - size 344
- queueEvent 455
- Queues
 - dynamically sized 186
 - message 8

R

- Rate monotonic analysis 199
- Rational Rhapsody
 - adapt to a new RTOS 13
 - deployment environment 5
 - framework 67
 - host machine 9
 - OXF 3
 - statecharts 2
 - supported container types 186
- rawtypes.h file 28
- Reactive object 177
- Real time 563
- Real-time 563
 - createRealTimeTimer 217
 - frameworks 1
- Real-time operating system (RTOS) 5
- Real-time Operating System (RTOS) 11
- recordMemoryAllocation 322
- recordMemoryDeallocation 323
- Register
 - animRegisterForeignThread 558
- register
 - registerWithOMReactive 375
- registerWithOMReactive 375
- Relations
 - OMReactive class 355

Index

- Remove
 - _removeFirst 288
 - all elements from list 289
 - element from list 290
 - element from map 307
 - first element from list 289
 - first element from template-free list 539
 - item from template-free list 540
 - key 553
 - last element 291
 - last element from template-free list 541
 - remove
 - OMCollection 224
 - OMHeap 266
 - OMList 288
 - OMMap 307
 - OMUCollection 517
 - OMUList 538
 - OMUMap 552
 - removeAll
 - OMCollection 225
 - OMList 289
 - OMMap 308
 - OMStaticArray 418
 - OMUCollection 518
 - OMUList 539
 - OMUMap 553
 - removeByIndex
 - OMCollection 226
 - OMUCollection 519
 - removeFirst
 - OMList 289
 - OMUList 539
 - removeItem
 - OMList 290
 - OMUList 540
 - removeKey 553
 - removeLast
 - OMList 291
 - OMUList 541
 - reorganize
 - OMCollection 226
 - OMUCollection 520
 - reset
 - OMIterator 271
 - OMUIterator 524
 - resetSize 435
 - resume
 - OMThread 456
 - OMTimerManager 496
 - returnMemory 317
 - OMAbstractMemoryAllocator 215
 - OMMemoryManager 317
 - RiCOSTimerManager 93
 - rootState attribute 356
 - rootState_dispatchEvent 376
 - rootState_entDef 377
 - rootState_serializeStates 378
 - RTOS 5, 11
 - adapting Rational Rhapsody to 13
 - classes 12
 - creating properties for 24
 - layered approach 6
 - makefile creating new 16
 - relation to Rational Rhapsody applications 5
 - with Rational Rhapsody applications 5
 - Run-time sources 13
 - runToCompletion 379
- ## S
- Samples
 - IDF 203
 - schedTm 456
 - Schedule
 - delay 198
 - timeout 195
 - Semaphores 8, 67, 126
 - Sequence diagrams 108, 196, 199
 - dispatch triggered operations 193
 - dispatched event 191
 - for documentation 178
 - generation & queue events 190
 - serialize 254
 - cserialize OMFriendStartBehaviorEvent 253
 - OMFriendStartBehaviorEvent 254
 - OMFriendTimeout 257
 - serializeStates
 - OMLeafState 275
 - OMOrState 330
 - OMReactive 379
 - OMState 408
 - Services
 - communication 9
 - operating system 5
 - operating systems 32
 - package 4, 185
 - synchronization 5, 8
 - tasking 7
 - timer 10
 - timing 93
 - tracing in C 17
 - Set
 - destination 247
 - setDeleteAfterConsume 246
 - set 497
 - setAllocator 216
 - SetAt 436
 - setAt
 - OMStaticArray 418
 - OMUCollection 521
 - setCompleteStartBehavior 380
 - SetDefaultBlock 436
 - setDelay 480

-
- setDeleteAfterConsume 246
 - setDestination 247
 - setDueTime 481
 - setElapsedTime 498
 - setElement 544
 - setEndOSThreadInDtor 458
 - setEventGuard 380
 - setFrameworkEvent 247
 - setFrameworkInstance 381
 - setHandle 408
 - setIncrementNum 216
 - setInDtor 382
 - setLastState 409
 - setId 248
 - setMaxNullSteps 382
 - setMemoryManager 564
 - setPriority 459
 - setRelativeDueTime 481
 - setShouldDelete 383
 - setShouldTerminate 384
 - setState 482
 - setSubState
 - OMOrState 331
 - OMState 409
 - setDefaultActiveClass 565
 - setTheTickTimerFactory 566
 - setThread 385
 - setTimeoutId 483
 - setToGuardReactive 386
 - setToGuardThread 459
 - setUpdateState 324
 - shouldCompleteRun 387
 - shouldCompleteStartBehavior 388
 - shouldDelete 389
 - shouldGuardThread 460
 - shouldTerminate 390
 - shouldUpdate 324
 - Simulated time 563
 - consumeTime 491
 - createSimulatedTimeTimer 218
 - goNextAndPost 493
 - Size
 - getSize 515
 - getSize OMQueue 344
 - getSize OMStaticArray 417
 - resetSize 435
 - stack 7
 - size attribute 412
 - Sockets 129
 - softUnschedTm 499
 - Solaris 19
 - makefile 16
 - Solaris systems 22
 - Stack 396
 - Stack size 7
 - Stacks 186
 - Start 567
 - start
 - OMMainThread class 296
 - OMThread class 460
 - START_DTOR_REACTIVE_GUARDED_SECTION
 - macro 259
 - START_DTOR_THREAD_GUARDED_SECTION
 - macro 260
 - START_REACTIVE_GUARDED_SECTION
 - macro 260
 - START_THREAD_GUARDED_SECTION macro 260
 - startBehavior 391
 - State
 - enterState OMComponentState 228
 - enterState OMState 404
 - exitState 404
 - getLastState 405
 - getSubState OMOrState 329
 - OMState 401
 - serializeState OMLeafState 275
 - serializeState OMOrState 330
 - setLastState 409
 - setState 482
 - setSubState OMOrState 331
 - setSubState OMState 409
 - State machine 2
 - null transitions 369
 - shouldDelete 389
 - termination connector 383
 - state.cpp file 28
 - state.h file 28
 - Statecharts 1, 2, 179, 182, 183, 199, 347
 - animated 199
 - Static array
 - empty 417
 - example 412
 - finding an element 415
 - number of elements 416
 - removing all elements 418
 - Stepper
 - getStepper 454
 - stopAllThreads 461
 - String
 - default block 436
 - length 434
 - Substate
 - getSubState OMState 406
 - getting 329
 - setting 331
 - suspend 462
 - Switches
 - OMMemoryManagerSwitchHelper 318
 - Synchronization 59
 - services 5, 8
 - Synchronous event
 - consuming 393
 - System time
 - getting 492
-

setting 498

T

Tail

increasing 344

takeEvent

OMComponentState 229

OMReactive 392

OMState 410

processing events 179

takeTrigger 393

Target

type 170

Target environment

customizing the OXF 11

Targets 159

Tasking services 5, 7

TCP/IP protocol 9

Template-free

adding elements to a location 511

current element 514

number of elements 513

Template-free collection

empty 516

first element 514

removing all elements 518

removing elements 517

removing elements by index 519

reorganizing 520

Template-free list

current element 535

empty 537

finding an element 533

first element 536

number of elements 535

removing all elements 539

removing item 540

Template-free map

empty 551

getting the key 550

looking up an element 551

number of elements 550

Terminate 394

setShouldTerminate 384

shouldTerminate 390

Termination connector

setShouldDelete 383

shouldDelete 389

Test 199

theLink attribute 412

Thread 7

thread attribute 441

Threads 7

action OMThreadTimer class 466

action OMTimerManager class 489

allowDeleteInThreadsCleanup 445

animDeregisterForeignThread 557

animRegisterForeignThread 558

cleanupAllThreads 447

ending 453

getAOMThread 451

getThread 365

instance 295

operating system ID 452

resuming a suspended 456

setEndOSThreadInDtor 458

setPriority 459

setThread 385

shouldGuardThread 460

stack size 7

start 460

stopAllThreads 461

suspending 462

user-defined 7

wrapper 7

Tick timer

getTheTickTimerFactory 561

setTheTickTimerFactory 566

tickTime 563

Time

consumeTime 491

getElapsedTime 492

goNextAndPost 493

real 563

setElapsedTime 498

simulated 563

Timeout class

Behavior package 177

timeoutDelayId attribute 471

Timeouts 2, 194, 469

canceling a request 500

customizing behavior 184

delaying 197

delegating a request 497

deleting from heap 476

discarnateTimeout 360

dispatching 184

dispatching SD 196

getDelay 477

getDueTime 477

getTimeoutId 478

ID 194

incarnateTimeout 368

isCancelledTimeout 242

isNotDelay 479

isTimeout 244

maxTM 563

posting 183

removing from the matured list 499

schedTm 456

scheduling 195

setDelay 480

setDueTime 481

- setRelativeDueTime 481
- setTimeoutId 483
- unschedTm OMThread class 463
- unscheduling 197
- timeouts 488
- Timer service 5
- timer.cpp file 29
- timer.h file 28
- TimerManagerCallBack 219
- TimerMaxTimeouts property 488
- TimerResolution property
 - system timer 488
- Timers 10, 183
 - createRealTimeTimer 217
 - createSimulatedTimeTimer 218
 - getTheTickTimerFactory 561
 - idle 10
 - starting 494
 - tick 10
 - TimerManagerCallBack 219
- Timing services 93
- toGuardReactive attribute 352
 - setting 386
- ToGuardThread attribute
 - setting 459
- toGuardThread attribute 441
- top
 - OMHeap 267
 - OMStack 399
- Transition
 - decrementing null 374
 - default 377
 - incrementing null 375
 - null 369, 382
 - taking null 379
- Trigger 2
 - take Trigger methods 347
 - takeTrigger 393
 - transitions 189

- Triggered operations 182, 347
 - dispatching 193
 - unconsumed 367
- trim 267

U

- UML 1, 2, 3, 200
 - active object 181
 - time event 469
- undoBusy 395
- unlock
 - OMAndState 220
 - OMGuard 262
 - OMProtected 336
 - OMThread 462
- unschedTm
 - OMThread class 463
 - OMTimerManager 500
- Unschedule
 - timeout 197
- update 267

V

- Validate
 - new adapter 30
- value
 - OMIterator 271
 - OMUIterator 525
- Virtual operations 32
- VxWorks 164

W

- wakeup 232
- Windows systems 20
- Wrapper threads 7

