Telelogic

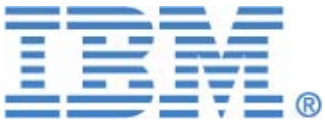
Rhapsody

RTOS Adapter Guide

IBM

Rhapsody®

RTOS Adapter Guide



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

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

Contents

The Deployment Environment

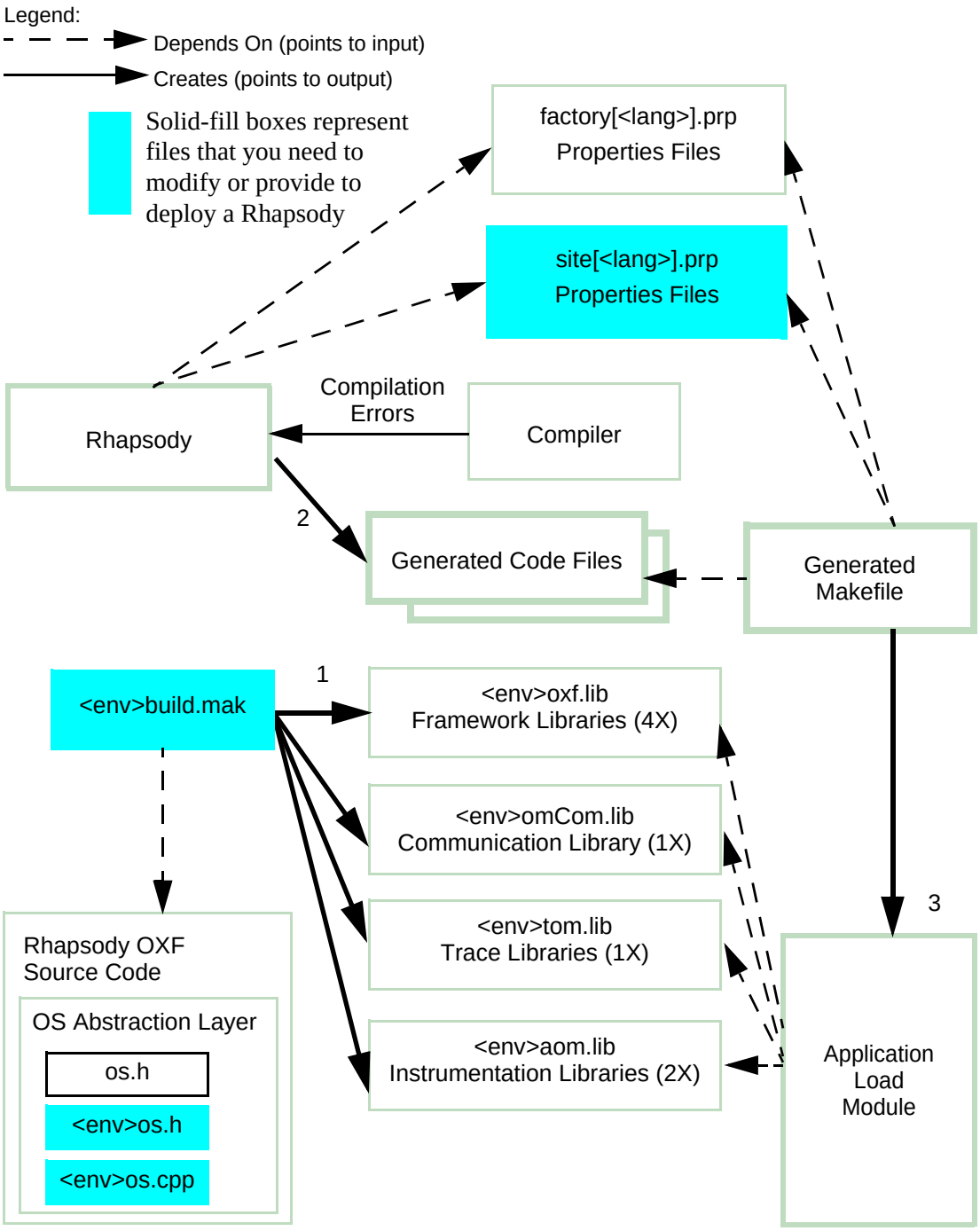
This section provides an overview of the Rhapsody deployment environment. The topics are as follows:

Basic Concepts

The deployment environment is the set of tools and third-party software required to develop and deploy a 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

The following diagram shows the dependencies between the Rhapsody components and the deployment environment.



Rhapsody Applications and the RTOS

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 *OS abstraction layer* (OSAL), which is part of the 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.

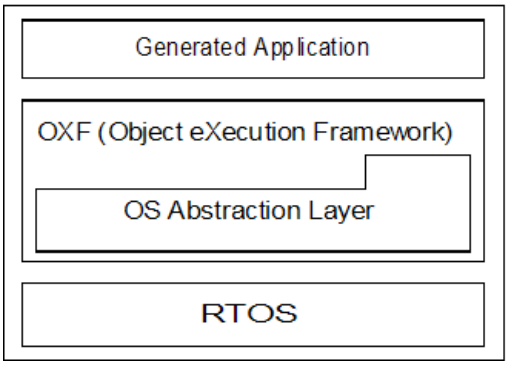
Each target environment requires a special OXF version. Preparing the OXF is primarily the process of providing an implementation for the OSAL. Each implementation of the OSAL for a particular target is known as an *adapter*.

Using the 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 shown in the following figure.



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.

Tasking Services

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 Rhapsody framework).

Setting the 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 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, 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 Rhapsody framework.

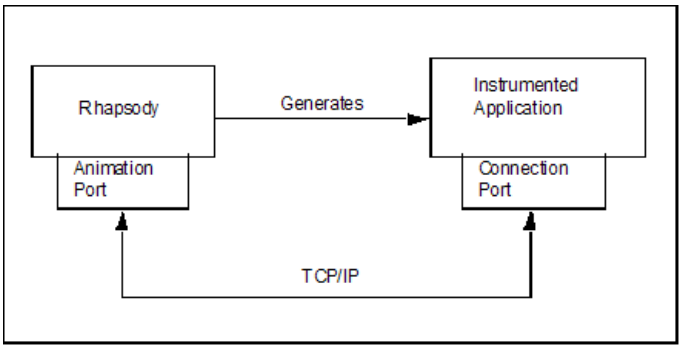
In Rhapsody applications, the `BaseNumberOfInstances` property (under `CG::Event`) specifies the initial size of the memory pool that is allocated for events. This pool is dynamically allocated at program initialization. The `AdditionalNumberOfInstances` property (under `CG::Event`) specifies the size 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 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 Rhapsody to communicate with processes running on either the same machine or different machines. This allows Rhapsody to communicate, for example, with an animated application running on a remote target board.

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:

- 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 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**.

Adapting Rhapsody to a New RTOS

To adapt Rhapsody to a new RTOS, follow these steps:

1. Install Rhapsody with the **Custom** option. In the Select Components screen of the installation program, select the **Runtime Sources** option to obtain the framework source files.
2. Implement the operating system adapter classes for the new environment, using the closest existing environment as a starting point.
3. Create new makefiles for building the framework libraries for the new environment.
4. Build the framework libraries for the new environment.
5. Create a set of code generation properties for the new environment and a batch file that sets its compiler environment. You can use the properties and batch file for the closest existing compiler and linker combination as a starting point.
6. Validate the new adapter.

After performing these steps, you can create a new configuration and select the new RTOS as its target environment, then generate and make code in the new environment.

The following sections describe each of these steps in detail.

Step 1: Installing the Run-Time Sources

When you install the run-time source files for your language (C or C++), Rhapsody copies both the implementation and specification files to the Rhapsody directory `\Share\Lang<Language>\oxf`. For example, if you installed the runtime source files for C++, the directory `\Share\LangCPP\oxf` contains both `.h` and `.cpp` files.

Step 2: 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 `$OMROOT\Lang<lang>\oxf` subdirectory of the Rhapsody installation. In this path, `$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 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 OMOSFactory {

    // 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 OMOSMessageQueue *createOMOSMessageQueue(
           OMBoolean /* shouldGrow */ = TRUE,
           const long messageQueueSize =
           OMOSThread::DefaultMessageQueueSize)
    { return (OMOSMessageQueue*)new
                  VxOSMessageQueue(messageQueueSize);}
    virtual OMOSConnectionPort *createOMOSConnectionPort()
    {
    #ifdef _OMINSTRUMENT
           return (OMOSConnectionPort*)new VxConnectionPort();
    #else
           return NULL;
    #endif
    }
    virtual OMOSEventFlag* createOMOSEventFlag() {
           return (OMOSEventFlag *)new VxOSEventFlag(); }
    virtual OMOSThread *createOMOSThread(void tfunc(
           void*), void *param,
           const char* const threadName = NULL,
           const long stackSize=OMOSThread::DefaultStackSize)
           {return (OMOSThread*)new VxThread(tfunc, param,
                  threadName, stackSize);};
    virtual OMOSThread* createOMOSWrapperThread(
           void* osHandle) {
                  if (NULL == osHandle)
                  osHandle = getCurrentThreadHandle();
                  return (OMOSThread*)new VxThread(osHandle);
    }
    virtual OMOSMutex *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(
           unsigned long semFlags = 0,
           unsigned long initialCount = 1,
           unsigned long /* maxCount */ = 1,
           const char * const /* name */ = NULL)
    {
           return (OMOSSemaphore*) 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 OMOSFactory* instance();
```

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

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

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 OMOSMutex {
   OM_DECLARE_FRAMEWORK_MEMORY_ALLOCATION_OPERATORS
   public:
      virtual ~OMOSMutex(){};
      virtual void lock() = 0;
      virtual void unlock() = 0;
      virtual void* getOsHandle() const = 0;
      #ifndef 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 OMOSMutex {
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 `ifstreams` 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
```

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);
```

Step 3: Creating 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 Rhapsody and an instrumented application	`<env>omcom.mak`

The compiled framework libraries are linked to the application generated from the 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 details.

Creating the Batch File and Makefiles

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 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`	`$OMROOT\Lang<lang>`	Calls the other makefiles to build the Rhapsody framework libraries (see **Sample <env>build.mak File**).
`<env>aom.mak`	`$OMROOT\Lang<lang>\aom`	Builds the instrumentation libraries: • `<env>aomtrace` • `<env>aomanim`
`<env>omcom.mak`	`$OMROOT\Lang<lang>\omcom`	Builds the communication library for instrumentation (`<env>omcomappl`)
`<env>oxf.mak`	`$OMROOT\Lang<lang>\oxf`	Builds the OXF libraries: • `<env>oxf` • `<env>oxfinst` See **OXF Versions** for descriptions of the different OXF libraries.
`<env>tom.mak`	`$OMROOT\tom`	Builds the tracing libraries: • `<env>tomtrace` • `<env>tomtraceRiC` (for Rhapsody in 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 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), 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>aomanim.lib` (for example, `vxaomanim.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`.

Step 4: 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. See the *Properties Reference Manual* for the makefile contents for the supported environments, which includes these macros and switches.

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.

Note

See **Adapter-Specific Info** for environment-specific build information for Rhapsody in C and C++.

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 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 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 Rhapsody in 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 Rhapsody in C framework source code. The Rhapsody in C framework is used to enable Rhapsody in 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

Rhapsody in J® provides a real-time framework for Java™ in the form of a 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 Rhapsody (by selecting **Code** > **Generate/Make**).

You can also build the Java framework outside of Rhapsody. However, you must first generate code for the `oxf.rpy` model inside of 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's `tar` file, which is installed when you select the **Solaris 2.x Libraries** option during the Rhapsody installation.

To build the framework, follow these steps:

1. When installing 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
```

Step 5: Creating Properties for a New RTOS

To complete the process of adding a new environment, you must give Rhapsody information about the development tools it uses, such as the compiler, linker, make utility, and libraries. To do this, you must 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.

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

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

Modifying the site<lang>.prp Files

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

1. Open the `factory<lang>.prp` properties file for each language that the new environment supports. For example, if the environment supports C++, open the `factoryC++.prp` file.
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 `factory<lang>.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 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 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`.

Step 6: Validating the New Adapter

To test the new adapter, follow these steps:

1. Try building a simple “Hello World” using Rhapsody and your new adapter. In 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 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 will continue to test the properties you defined in **Step 5: Creating Properties for a New RTOS**.

4. You must test the framework part of the adapter (see **Step 2: 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, 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 have validated a major portion of the adapter. To complete the validation, request a copy of the RTOS Adapter Test Suite from IBM Rhapsody Support. This test suite consists of several models that cover most of the scenarios needed to test an RTOS adapter.

Summary

The OSAL gives you the unique ability to develop and test 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 painlessly 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.

See the *Properties Reference Manual* for the complete list of RTOSes for which the OSAL has been adapted.

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.

Step 1: Creating a Make Batch 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).

Step 2: 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.

Step 3: 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 `OMCPPCompileCommandSet` variable.
	If you modified the generated dependencies section of the `MakeFileContent` property to generate a new `.obj` file every time you compile, you need to change the `CPPCompileCommand` property as follows:
	`"  if exist $OMFileObjPath del $OMFileObjPath`
	`$(CPP) $OMFileCPPCompileSwitches / Fo\"$OMFileObjPath\" \"$OMFileImpPath\" "`

`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.

Note

Refer to the *Rhapsody Properties Reference Manual* for the overall structure of properties within Rhapsody and the properties' uses to customize the Rhapsody environment. The individual definitions of properties and their defaults are displayed in the Features dialog box. You may also examine the complete list of Rhapsody property definitions in the *Rhapsody Property Definitions* PDF file available from the **List of Books**. That list can be searched along with the other PDF versions of the Rhapsody documentation to locate specific property definitions and the procedures that use them.

Step 4: 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 ############
#########################################

$(OBJS) : $(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 $(OBJS)
    $(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).

The IDE Interface

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 Rhapsody installation (in `<root>/Share/DLLs/ideabs.h`). Because this file defines an abstract IDE for 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 Rhapsody. The following callbacks are called by the IDE interface:

- `ConnectionClosedNotify`—Notifies Rhapsody when the connection to the IDE is broken
- `DoAnimationCommand`—Makes Rhapsody perform user animation commands (for example, Go Step)
- `DbgBreakpointNotify`—Notifies Rhapsody of a breakpoint in either animation or the IDE debugger
- `DbgContinueNotify`—Notifies 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 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 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.

The OSAL Classes

The operating system adapter is an implementation of the abstract factory pattern[1]. For example, in Rhapsody in 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:

- **Rhapsody in C**
- **Rhapsody in C++**

1.*Design Patterns*, Gamma et al., Addison Wesley 1995

Rhapsody in 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 Rhapsody.

Creation Summary

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

Description

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

Description

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

Description

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

Description

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

Description

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 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 signal on the same event flag instance

create

Description

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

Description

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

Description

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

Description

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

Description

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

Description

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

Description

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

create	Creates an `RiCOSMessageQueue` object
destroy	Destroys `RiCOSMessageQueue` object
cleanup	Cleans up after an `RiCOSMessageQueue` object
init	Initializes an `RiCOSMessageQueue` object

Method Summary

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

create

Description

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

Description

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

Description

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

Description

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

Description

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

Description

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

- To cancel events

  When a reactive class is destroyed, its 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

Description

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

Description

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

Description

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

Description

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 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

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

Method Summary

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

create

Description

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

Description

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

Description

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

Description

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

Description

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

Description

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

- If the mutex if 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

Description

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

Description

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 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 `RiCOSSemaphore` object
destroy	Destroys the `RiCOSSemaphore` object
cleanup	Cleans up after an `RiCOSSemaphore` object
init	Initializes an `RiCOSSemaphore` object

Method Summary

signal	Releases the semaphore token
wait	Waits for a semaphore token

create

Description

The **create** 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

Description

The **destroy** 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

Description

The **cleanup** 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

Description

The **init** 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

Description

The **signal** 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

Description

The **wait** 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 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 `RiCOSSocket` object
destroy	Destroys the `RiCOSSocket` object
cleanup	Cleans up after an `RiCOSSocket` object
init	Initializes an `RiCOSSocket` object

Method Summary

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

create

Description

The **create** 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

Description

The **destroy** 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

Description

The **cleanup** 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

Description

The **init** 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

Description

The **createSocket** 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_inaddr;
    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

Description

The **receive** 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

Description

The **send** 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

create	Creates an `RiCOSTask` object
destroy	Destroys an `RiCOSTask` object
cleanup	Cleans up after an `RiCOSTask` object
init	Initializes an `RiCOSTask` object

Method Summary

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

create

Description

The **create** 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

Description

The **destroy** 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

Description

The **cleanup** 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

Description

The **init** 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 */
           0,0,0,0,0,0,0,0,0);
    return 1;
}
```

endMyTask

Description

The **endMyTask** 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

Description

The endOtherTask 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

Description

The **exeOnMyTask** 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

Description

The **getCurrentTaskHandle** 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

Description

The **getOSHandle** 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

Description

The getTaskEndClbk 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

Description

The **resume** 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

Description

The **setEndOSTaskInCleanup** 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

Description

The **setPriority** 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

Description

The **start** 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

Description

The **suspend** 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 Rhapsody implementation, the timer runs on its own task. Therefore, the target operating system must support multitasking.

Creation Summary

create	Creates an `RiCOSTimer` object
destroy	Destroys an `RiCOSTimer` object
cleanup	Cleans up after an `RiCOSTimer` object
init	Initializes an `RiCOSTimer` object

create

Description

The **create** 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

Description

The **destroy** 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

Description

The **cleanup** 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

Description

The **init** 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 );
           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 RiCTRUE;
    }
}
```

RiCHandleCloser 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 RiCHandleCloser 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)RiCHandleCloser_startBehavior(RiCHandleCloser_Instance(RiCInt_doCloseH
andle));
}
```

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 )
    {
                RiCHandleCloser_genCloseEvent(hThread);
        Exit( 0UL );
    }
}
```

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.

Rhapsody in 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
- OMTMMessageQueue Class

OMEventQueue Class

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

Construction Summary

OMEventQueue	Creates an `OMOSEventQueue` object

Method Summary

getOsQueue	Retrieves the event queue

OMEventQueue

Visibility

```
Public
```

Description

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

Signature

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

Visibility

```
Public
```

Description

The **getOsQueue** method retrieves the event queue.

Signature

```
OMOSMessageQueue * getOsQueue()
```

OMMessageQueue Class

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

`OMSData` 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

Visibility

`Public`

Description

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

Signature

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

Parameters

`errorCode`

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

endProlog

Visibility

```
Public
```

Description

The **endProlog** method ends the prolog.

Signature

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

initEpilog

Visibility

```
Public
```

Description

The **initEpilog** 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.

Signature

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

OMOSConnectionPort Class

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

Construction Summary

~OMOSConnectionPort	Destroys the `OMOSConnectionPort` object.

Method Summary

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

~OMOSConnectionPort

Visibility

```
Public
```

Description

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

Signature

```
virtual ~OMOSConnectionPort()
```

Connect

Visibility

```
Public
```

Description

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

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

Visibility

`Public`

Description

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

Signature

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

Parameters

`m`

The data to be sent from the port. The data is of type `OMSData`, 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

Visibility

```
Public
```

Description

The **SetDispatcher** 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 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.

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 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 `OMOSEventFlag` 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 `signal` on the same event flag instance

~OMOSEventFlag

Visibility

```
Public
```

Description

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

Signature

```
virtual ~OMOSEventFlag()
```

getOsHandle

Visibility

Public

Description

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

Signature

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

Return

The operating system ID

reset

Visibility

Public

Description

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

Signature

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

signal

Visibility

Public

Description

The **signal** 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.

Signature

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

wait

Visibility

```
Public
```

Description

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

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.

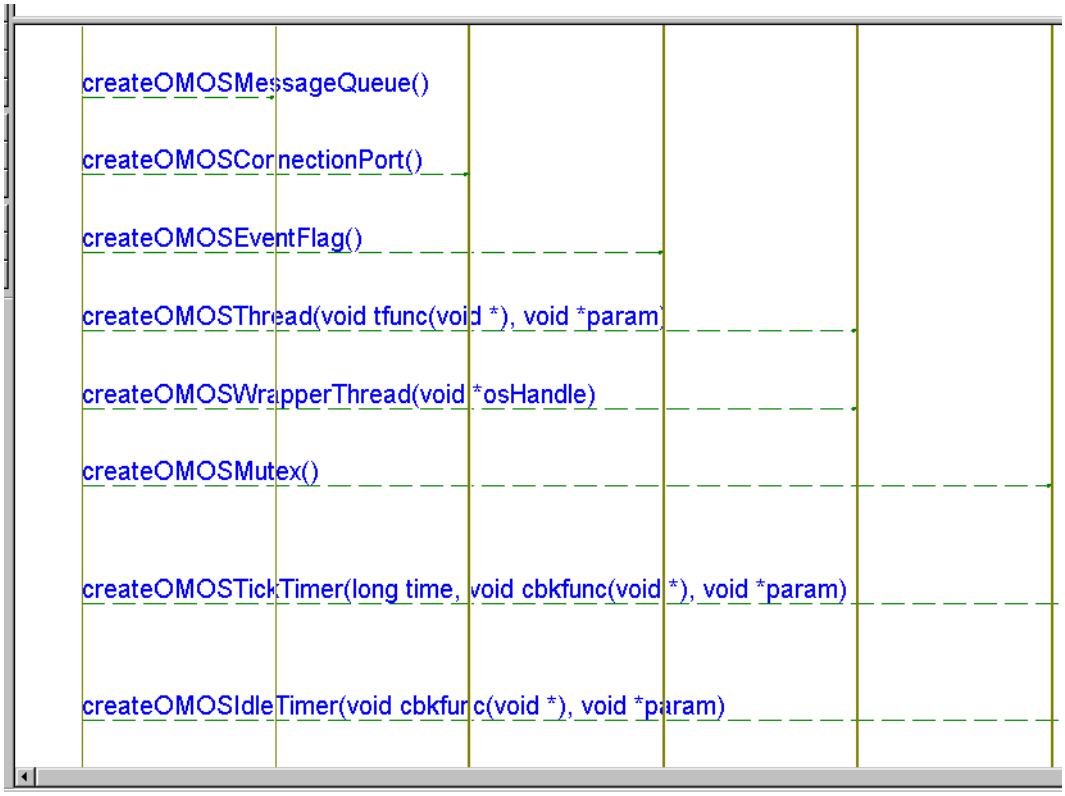
Notes

If an operating system does not support the ability to wait on an event flag with a timeout (for example, Solaris), the 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 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 `OMOSFactory`

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

Visibility

`Public`

Description

The instance 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.

Signature

```
static OMOSFactory* 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

Visibility

```
Public
```

Description

The **createOMOSConnectionPort** method creates a connection port.

Signature

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

Return

The new connection port

createOMOSEventFlag

Visibility

```
Public
```

Description

The **createOMOSEventFlag** method creates an event flag.

Signature

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

Return

The new event flag

createOMOSIdleTimer

Visibility

```
Public
```

Description

The **createOMOSIdleTimer** method creates an idle timer.

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

Visibility

`Public`

Description

The **createOMOSMessageQueue** method creates a message queue.

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

Visibility

`Public`

Description

The **createOMOSIdleTimer** method creates a mutex.

Signature

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

Return

The new mutex

createOMOSSemaphore

Visibility

```
Public
```

Description

This method creates a semaphore.

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

Visibility

Public

Description

This method creates a thread.

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

Visibility

Public

Description

This method creates a new tick timer.

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

Visibility

`Public`

Description

The **createOMOSWrapperThread** method creates a wrapper thread.

Signature

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

Parameters

`osHandle`

The handle to the operating system

Return

The new wrapper thread

delayCurrentThread

Visibility

`Public`

Description

The **delayCurrentThread** method delays the current thread for the specified length of time.

The `OXFTDelay(timInMs)` macro provides a convenient shortcut for calling **delayCurrentThread**.

Signature

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

Parameters

`ms`

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

getCurrentThreadHandle

Visibility

`Public`

Description

The **getCurrentThreadHandle** 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.

Signature

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

Return

The `OSThreadHandle`

waitOnThread

Visibility

```
Public
```

Description

The **waitOnThread** method waits for a thread to terminate.

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

~OMOSMessageQueue	Destroys the `OMOSMessageQueue` object

Method Summary

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

~OMOSMessageQueue

Visibility

```
Public
```

Description

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

Signature

```
virtual ~OMOSMessageQueue()
```

get

Visibility

```
Public
```

Description

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

Signature

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

Return

The first message in the queue

getMessageList

Visibility

```
Public
```

Description

The **getMessageList** 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.

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

Visibility

```
Public
```

Description

The **getOsHandle** 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.

Signature

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

Return

The handle

isEmpty

Visibility

```
Public
```

Description

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

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

Visibility

```
Public
```

Description

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

Signature

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

Return

The method returns one of the following values:

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

pend

Visibility

```
Public
```

Description

The **pend** 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.

Signature

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

put

Visibility

```
Public
```

Description

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

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

Visibility

```
Public
```

Description

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

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 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 `OMOSMutex` 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

Visibility

```
Public
```

Description

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

Signature

```
virtual ~OMOSMutex()
```

free

Visibility

```
Public
```

Description

The **free** method releases the lock, possibly causing the underlying operating system to reschedule threads.

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

Signature

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

getOsHandle

Visibility

```
Public
```

Description

The **getOsHandle** 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.

Signature

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

Return

The handle

lock

Visibility

```
Public
```

Description

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

- If the mutex if 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.

Signature

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

unlock

Visibility

```
Public
```

Description

The **unlock** method releases the lock, possibly causing the underlying operating system to reschedule threads.

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 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 `OMOSSemaphore` object

Method Summary

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

~OMOSSemaphore

Visibility

```
Public
```

Description

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

Signature

```
virtual ~OMOSSemaphore()
```

getOsHandle

Visibility

```
Public
```

Description

The **getOsHandle** 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.

Signature

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

Return

The handle

signal

Visibility

```
Public
```

Description

The **signal** method releases a semaphore token.

Signature

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

wait

Visibility

```
Public
```

Description

The **wait** method obtains a semaphore token.

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 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 Rhapsody and the application causes the socket's internal buffer to fill up, which might cause a major delay in communication between 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

Visibility

```
Public
```

Description

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

Signature

```
virtual ~OMOSSocket()
```

Close

Visibility

```
Public
```

Description

The Close method closes the socket.

Signature

```
virtual void Close()
```

Create

Visibility

```
Public
```

Description

The **Create** method creates a new socket.

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

Visibility

```
Public
```

Description

The **Receive** method receives data through the socket.

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

Visibility

`Public`

Description

The **Send** method sends data through the socket.

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
`DefaultStackSize`	`SMALL_STACK` or `DEFAULT_STACK`
`DefaultMessageQueueSize`	`MQ_DEFAULT_SIZE`
`DefaultThreadPriority`	`PRIORITY_HIGH`, `PRIORITY_NORMAL`, or `PRIORITY_LOW`

Construction Summary

~OMOSThread	Destroys the `OMOSThread` object

Method Summary

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

~OMOSThread

Visibility

```
Public
```

Description

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

Signature

```
virtual ~OMOSThread()
```

exeOnMyThread

Visibility

```
Public
```

Description

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

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

Visibility

```
Public
```

Description

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

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

Visibility

`Public`

Description

The **getThreadEndClbk** method is a callback function that ends the current operating system thread.

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 **getThreadEndClbk** 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

Visibility

`Public`

Description

The **resume** 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.

Signature

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

setEndOSThreadInDtor

Visibility

Public

Description

The **setEndOSThreadInDtor** method determines whether destruction of the OMOSThread class should kill the operating system thread associated with the class.

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

Visibility

Public

Description

The **setPriority** method sets the thread's operating system priority.

Signature

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

Parameters

pr

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

start

Visibility

Public

Description

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

Signature

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

suspend

Visibility

Public

Description

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

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 Rhapsody implementation, the timer runs on its own thread. Therefore, the target operating system must support multithreading.

Construction Summary

~OMOSTimer	Destroys the `OMOSTimer` object

Method Summary

getOsHandle	Retrieves the thread's operating system ID

~OMOSTimer

Visibility

```
Public
```

Description

The **~OMOSTimer** 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.

Signature

```
virtual ~OMOSTimer()
```

getOsHandle

Visibility

```
Public
```

Description

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

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

OMTMMessageQueue	Creates an `OMTMMessageQueue` object.
~OMTMMessageQueue	Destroys the `OMTMMessageQueue` object

Method Summary

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

OMTMMessageQueue

Visibility

```
Public
```

Description

The **OMTMMessageQueue** 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.

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

Visibility

```
Public
```

Description

The **~OMTMMessageQueue** 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.

Signature

```
virtual ~OMTMMessageQueue()
```

get

Visibility

```
Public
```

Description

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

Signature

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

Return

The first message in the queue

getMessageList

Visibility

```
Public
```

Description

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

Signature

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

Parameters

`l`

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

Visibility

```
Public
```

Description

The **getOsHandle** 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.

Signature

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

Return

The handle

isEmpty

Visibility

```
Public
```

Description

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

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

Visibility

Public

Description

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

Signature

```
virtual void pend()
```

put

Visibility

Public

Description

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

Signature

```
virtual OMBoolean out (Msg *m, OMBoolean 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.

Adapter-Specific Info

When you want to modify your Rhapsody-built application to operate in a different target environment, you must rebuild the 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 Rhapsody before you rebuild the Rhapsody framework. You should reinstall Rhapsody in the following situations:

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

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

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

Note

Refer to the *Release Notes* (`readme.htm`) for detailed information about the supported environments.

The topics are as follows:

For information on rebuilding Rhapsody for other environments, see these sections:

- **Building the Framework for Solaris Systems**
- **Building the Ada Framework**
- **Building the Java Framework**

Borland

To rebuild the 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. Refer to the *Rhapsody User Guide* for more information on the Webify Toolkit.

INTEGRITY

To rebuild the 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 Rhapsody samples.

Compiling and Building a Rhapsody Sample

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

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

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

2. Select **File** > **Project Properties**.

3. Set the `CPP_CG::INTEGRITY::RemoteHost` property to the IP address of the machine on which 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 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 Images

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.

Dynamically Load Files

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.

INTEGRITY Application Images

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 application's main function. Ensure that the same application is opened in 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 Rhapsody.

2. To execute this application, click the toolbar button **GO**. You should be able to see the animation toolbar come up in Rhapsody. You can generate events in Rhapsody using the animation toolbar. If there is console output, it is displayed on the rtserv IN/OUT window; animation is displayed in the 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

Rhapsody in 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 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 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 will be created in the current directory. When you run the executable on the target, the Rhapsody animation toolbar will open on the host (for applications using instrumentation).

MultiWin32

To rebuild the Rhapsody OXF for the `MultiWin32` environment (C++ only), follow these steps:

1. Open an application command prompt window.
2. Change directory to `$OMROOT\LangCpp`.
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 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 Rhapsody, select **Code > Generate/Make/Run**.

3. In MULTI, start debugging by selecting **Debug > Debug Other Executable** and select the 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 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.

Perform the following steps:

1. Add the following lines to the file<Rhapsody install>\Share\etc\oseppcdiabmake.bat:

   ```
   set DIAB_ROOT=<Your Diab Root>
   set LM_LICENSE_FILE=<Your Diab license file>
   ```

2. 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 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 QCC_CONF_PATH=%QNXROOT%/host/win32/etc/qcc
set QNX_TARGET=%QNXROOT%/target/qnx6
set QNX_HOST=%QNXROOT%/host/win32
set LD_LIBRARY_PATH=%QNXROOT%/target/qnx6/lib
set PATH=%QNX_HOST%/binwin;%PATH%
```

3. 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 Rhapsody, then upload or transfer your executable to the QNX machine using the Code Warrior 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 Code Warrior

To upload your executable using Code Warrior, follow these steps:

1. Open Code Warrior and select **File > Open**.
2. Choose the new Bourne executable. This will create the Code Warrior 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 Code Warrior.

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.

VxWorks

To rebuild the 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 -f vxbuild.mak CPU=PPC860 PATH_SEP=\\ all
   ```

3. Change the `CPU` environment variable to the desired CPU.

Quick Reference

This section provides a quick reference to the OSAL methods. The following table briefly describes each 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.

OSAL Method	Description
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.
exeOnMyTask	Determines whether the method was invoked from the same operating system task as the one on which the object is running.
exeOnMyThread	Determines whether the method was invoked from the same operating system thread as the one on which the object is running.
free	Releases the lock, possibly causing the underlying operating system to reschedule threads.
get	Retrieves the message at the beginning of the queue.
getCurrentTaskHandle	Returns the native operating system handle to the task.
getCurrentThreadHandle	Returns the native operating system handle to the thread.
getMessageList	Retrieves the list of messages.
getOSHandle	Retrieves the task's operating system ID.
getOsHandle	Retrieves the thread's operating system ID.
getOsQueue	Retrieves the event queue.
getTaskEndClbk	Is a callback function that ends the current operating system task.
getThreadEndClbk	Is a callback function that ends the current operating system thread.
init	Initializes the new object.
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.
instance	Creates a single instance of `OMOSFactory`.
isEmpty	Determines whether the message queue is empty.
isFull	Determines whether the queue is full.
lock	Determines whether the mutex is free and reacts accordingly.
OMEventQueue	Constructs an `OMEventQueue` object.
OMTMMessageQueue	Constructs an `OMTMMessageQueue` object.
pend	Blocks the thread making the call until there is a message in the queue.

OSAL Method	Description
put	Adds a message to the end of the message queue.
receive	Waits on the socket to receive the data.
Receive	Receives data through the socket.
reset	Forces the event flag into a known state.
resume	Resumes a suspended thread.
RiCOSEndApplication	Ends a running application.
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 `RiCOSTask` class should kill the operating system task associated with the class.
setEndOSThreadInDtor	Determines whether destruction of the `OMOSThread` 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 signal on the same event flag instance.
waitOnThread	Waits for a thread to terminate.

Index

E

F

G

I

J

L

M

O

P

R

S

T

V

W