



# TIBCO Rendezvous®

## C++ Reference

Version 8.7.0 | October 2023

# Contents

<b>Concepts</b>	<b>13</b>
Implementation	14
Wrapper Architecture	14
Uncaught C++ Exceptions	14
Strings and Character Encodings	16
Custom Datatypes	17
<b>Programmer's Checklist</b>	<b>18</b>
Checklist	19
Code	19
Compile	19
Link	19
Run	19
Programming Restrictions	20
Include These Header Files	21
Link These Library Files	22
UNIX	22
Microsoft Windows	23
Source Code Distribution	27
<b>Rendezvous Environment</b>	<b>28</b>
Tibrv	29
Tibrv::close()	30
Tibrv::defaultQueue()	32
Tibrv::open()	33
Tibrv::processTransport()	35
Tibrv::version()	36
TibrvSdContext	37
TibrvSdContext:setDaemonCert()	38
TibrvSdContext:setUserCertWithKey()	40

TibrvSdContext:setUserCertWithKeyBin()	42
TibrvSdContext:setUserNameWithPassword()	44
<b>Data</b>	<b>45</b>
Validity of Data Extracted From Message Fields	46
Scalar Snapshot	46
Pointer Snapshot	47
Deleting Snapshot References	50
Multiple Subscription Snapshots	51
Field Names and Field Identifiers	52
Finding a Field Instance	53
TibrvMsg	55
TibrvMsg()	59
~TibrvMsg()	62
TibrvMsg::addField()	63
Add Scalar	66
Add Array	68
Add Nested Message	70
Add String	71
Add Opaque Byte Sequence	72
Add XML Byte Sequence	73
Add DateTime	74
TibrvMsg::clearReferences()	76
TibrvMsg::convertToString()	77
TibrvMsg::createCopy()	78
TibrvMsg::detach()	79
TibrvMsg::expand()	81
TibrvMsg::getAsBytes()	82
TibrvMsg::getAsBytesCopy()	83
TibrvMsg::getByteSize()	85
TibrvMsg::getCurrentTime()	86

TibrvMsg::getField()	87
Get Scalar	90
Get Array	92
Get Nested Message	94
Get String	96
Get Opaque Byte Sequence	98
Get XML Byte Sequence	100
Get DateTime	102
TibrvMsg::getFieldByIndex()	104
TibrvMsg::getFieldInstance()	105
TibrvMsg::getEvent()	107
TibrvMsg::getHandle()	108
TibrvMsg::getNumFields()	109
TibrvMsg::getReplySubject()	110
TibrvMsg::getSendSubject()	112
TibrvMsg::getStatus()	113
TibrvMsg::isDetached()	114
TibrvMsg::markReferences()	115
TibrvMsg::removeField()	117
TibrvMsg::removeFieldInstance()	119
TibrvMsg::reset()	120
TibrvMsg::setReplySubject()	121
TibrvMsg::setSendSubject()	122
TibrvMsg::updateField()	123
Update Scalar	126
Update Array	128
Update Nested Message	130
Update String	132
Update Opaque Byte Sequence	134
Update XML Byte Sequence	136
Update DateTime	138

TibrvMsgField .....	139
TibrvMsgField() .....	141
TibrvMsgField::getCount() .....	142
TibrvMsgField::getData() .....	143
TibrvMsgField::getId() .....	144
TibrvMsgField::getName() .....	145
TibrvMsgField::getSize() .....	146
TibrvMsgField::getType() .....	147
TibrvMsgDateTime .....	148
TibrvMsgDateTime() .....	152
<b>Events and Queues .....</b>	<b>153</b>
Event Overview .....	154
TibrvEvent .....	155
TibrvEvent::destroy() .....	157
TibrvEvent::getClosure() .....	159
TibrvEvent::getHandle() .....	160
TibrvEvent::getType() .....	161
TibrvEvent::getQueue() .....	162
TibrvEvent::isIOEvent() .....	163
TibrvEvent::isListener() .....	164
TibrvEvent::isTimer() .....	165
TibrvEvent::isValid() .....	166
TibrvEvent::isVectorListener() .....	167
TibrvCallback .....	168
TibrvCallback::onEvent() .....	169
TibrvEventOnComplete .....	170
TibrvEventOnComplete::onComplete() .....	171
TibrvListener .....	175
TibrvListener::create() .....	178
TibrvListener::getSubject() .....	180

TibrvListener::getTransport()	181
TibrvMsgCallback	182
TibrvMsgCallback::onMsg()	183
TibrvVectorListener	185
TibrvVectorListener::create()	187
TibrvVectorListener::getSubject()	192
TibrvVectorListener::getTransport()	193
TibrvVectorCallback	194
TibrvVectorCallback::onMsgs()	195
TibrvTimer	197
TibrvTimer::create()	200
TibrvTimer::getInterval()	202
TibrvTimer::resetInterval()	203
TibrvTimerCallback	204
TibrvTimerCallback::onTimer()	205
TibrvIOEvent	206
TibrvIOEvent::create()	209
TibrvIOEvent::getIOSource()	211
TibrvIOEvent::getIOType()	212
TibrvIOCallback	213
TibrvIOCallback::onIOEvent()	214
TibrvDispatchable	215
TibrvDispatchable::destroy()	217
TibrvDispatchable::dispatch()	218
TibrvDispatchable::getDispatchable()	219
TibrvDispatchable::isValid()	220
TibrvDispatchable::poll()	221
TibrvDispatchable::timedDispatch()	222
TibrvQueue	224
TibrvQueue::create()	227
TibrvQueue::destroy()	228

TibrvQueue::dispatch()	229
TibrvQueue::getCount()	230
TibrvQueue::getHandle()	231
TibrvQueue::getLimitPolicy()	232
TibrvQueue::getName()	234
TibrvQueue::getPriority()	235
TibrvQueue::isValid()	236
TibrvQueue::poll()	237
TibrvQueue::setLimitPolicy()	238
TibrvQueue::setName()	240
TibrvQueue::setPriority()	241
TibrvQueue::timedDispatch()	242
TibrvQueueOnComplete	244
TibrvQueueOnComplete::onComplete()	245
TibrvQueueGroup	247
TibrvQueueGroup::add()	249
TibrvQueueGroup::create()	250
TibrvQueueGroup::destroy()	251
TibrvQueueGroup::dispatch()	252
TibrvQueueGroup::getHandle()	253
TibrvQueueGroup::isValid()	254
TibrvQueueGroup::poll()	255
TibrvQueueGroup::remove()	256
TibrvQueueGroup::timedDispatch()	257
TibrvDispatcher	259
TibrvDispatcher::create()	261
TibrvDispatcher::destroy()	263
TibrvDispatcher::getDispatchable()	264
TibrvDispatcher::getHandle()	265
TibrvDispatcher::getName()	266
TibrvDispatcher::isValid()	267

TibrvDispatcher::setName()	268
<b>Transports</b>	<b>269</b>
TibrvTransport	270
TibrvTransport::createInbox()	273
TibrvTransport::destroy()	275
TibrvTransport::getDescription()	276
TibrvTransport::getHandle()	277
TibrvTransport::isValid()	278
TibrvTransport::requestReliability()	279
TibrvTransport::send()	281
TibrvTransport::sendReply()	282
TibrvTransport::sendRequest()	284
TibrvTransport::setDescription()	286
tibrvTransportBatchMode	287
TibrvProcessTransport	288
TibrvNetTransport	290
TibrvNetTransport::create()	292
TibrvNetTransport::getDaemon()	295
TibrvNetTransport::getNetwork()	296
TibrvNetTransport::getService()	297
TibrvNetTransport::setBatchMode()	298
<b>Virtual Circuits</b>	<b>299</b>
TibrvVcTransport	300
TibrvVcTransport::createAcceptVc()	303
TibrvVcTransport::createConnectVc()	305
TibrvVcTransport::waitForVcConnection()	307
<b>Fault Tolerance</b>	<b>309</b>
Fault Tolerance Road Map	310
tibrvftAction	311



TibrvFtMember .....	313
TibrvFtMember::create() .....	315
TibrvFtMember::destroy() .....	319
TibrvFtMember::getClosure() .....	320
TibrvFtMember::getGroupName() .....	321
TibrvFtMember::getHandle() .....	322
TibrvFtMember::getQueue() .....	323
TibrvFtMember::getTransport() .....	324
TibrvFtMember::getWeight() .....	325
TibrvFtMember::isValid() .....	326
TibrvFtMember::setWeight() .....	327
TibrvFtMemberCallback .....	329
TibrvFtMemberCallback::onFtAction() .....	330
TibrvFtMemberOnComplete .....	332
TibrvFtMemberOnComplete::onComplete .....	333
TibrvFtMonitor .....	335
TibrvFtMonitor::create() .....	337
TibrvFtMonitor::destroy() .....	339
TibrvFtMonitor::getClosure() .....	340
TibrvFtMonitor::getGroupName() .....	341
TibrvFtMonitor::getHandle() .....	342
TibrvFtMonitor::getQueue() .....	343
TibrvFtMonitor::getTransport() .....	344
TibrvFtMonitor::isValid() .....	345
TibrvFtMonitorCallback .....	346
TibrvFtMonitorCallback::onFtMonitor() .....	347
TibrvFtMonitorOnComplete .....	348
TibrvFtMonitorOnComplete::onComplete .....	349
<b>Certified Message Delivery .....</b>	<b>351</b>
TibrvCmListener .....	352

TibrvCmListener::confirmMsg()	355
TibrvCmListener::create()	357
TibrvCmListener::destroy()	359
TibrvCmListener::getSubject()	360
TibrvCmListener::getTransport()	361
TibrvCmListener::isValid()	362
TibrvCmListener::setExplicitConfirm()	363
TibrvCmTransport	364
TibrvCmTransport::addListener()	368
TibrvCmTransport::allowListener()	370
TibrvCmTransport::create()	371
TibrvCmTransport::createInbox()	375
TibrvCmTransport::destroy()	376
TibrvCmTransport::disallowListener()	378
TibrvCmTransport::expireMessages()	380
TibrvCmTransport::getDefaultTimeLimit()	382
TibrvCmTransport::getLedgerName()	383
TibrvCmTransport::getName()	384
TibrvCmTransport::getRequestOld()	385
TibrvCmTransport::getSyncLedger()	386
TibrvCmTransport::getTransport()	387
TibrvCmTransport::removeListener()	388
TibrvCmTransport::removeSendState()	390
TibrvCmTransport::reviewLedger()	392
TibrvCmTransport::send()	394
TibrvCmTransport::sendReply()	396
TibrvCmTransport::sendRequest()	398
TibrvCmTransport::setDefaultTimeLimit()	400
TibrvCmTransport::setPublisherInactivityDiscardInterval()	401
TibrvCmTransport::syncLedger()	403
TibrvCmTransportOnComplete	404

TibrvCmTransportOnComplete::onComplete .....	405
TibrvCmReviewCallback .....	406
TibrvCmReviewCallback::onLedgerMsg() .....	407
TibrvCmMsg .....	410
TibrvCmMsg::getSender() .....	411
TibrvCmMsg::getSequence() .....	412
TibrvCmMsg::getTimeLimit() .....	414
TibrvCmMsg::setTimeLimit() .....	415
TibrvCmMsgCallback .....	417
TibrvCmMsgCallback::onCmMsg() .....	418
<b>Distributed Queue .....</b>	<b>420</b>
TibrvCmQueueTransport .....	421
TibrvCmQueueTransport::create() .....	425
TibrvCmQueueTransport::destroy() .....	429
TibrvCmQueueTransport::getCompleteTime() .....	430
TibrvCmQueueTransport::getUnassignedMessageCount() .....	431
TibrvCmQueueTransport::getWorkerWeight() .....	432
TibrvCmQueueTransport::getWorkerTasks() .....	433
TibrvCmQueueTransport::setCompleteTime() .....	434
TibrvCmQueueTransport::setTaskBacklogLimit...() .....	435
TibrvCmQueueTransport::setWorkerWeight() .....	437
TibrvCmQueueTransport::setWorkerTasks() .....	438
<b>Datatypes .....</b>	<b>440</b>
Wire Format Datatypes .....	441
C Datatypes .....	444
Datatype Conversion .....	446
General Rules .....	446
Converting to Boolean .....	447
<b>Status and Errors .....</b>	<b>449</b>

TibrvStatus .....	450
TibrvStatus::getCode() .....	456
TibrvStatus::getText() .....	457
<b>TIBCO Documentation and Support Services .....</b>	<b>458</b>
<b>Legal and Third-Party Notices .....</b>	<b>461</b>

# Concepts

---

This section presents concepts specific to the TIBCO Rendezvous® C++ language interface. For concepts that pertain to Rendezvous software in general, see the book *TIBCO Rendezvous Concepts*.

# Implementation

The Rendezvous C++ API consists of a thin wrapper around the C API.

## Wrapper Architecture

It features a lightweight object model, in which C++ objects refer to C objects using a pointer or handle.

### General Case

In general, C++ constructors declare typed variables and create hollow objects. In contrast, create methods make hollow objects operational, by creating a corresponding C object, and storing its handle in the C++ object.

Similarly, C++ destroy methods destroy the corresponding C object—however, although they invalidate the C++ object, they do not free its storage. In contrast, C++ destructors first call the corresponding destroy method, and then destroy the C++ object, reclaiming its storage.

### Data Objects

Data objects (messages, message fields, and datetime objects) present exceptions to these rules. C++ data objects have constructors and destructors, but not create and destroy methods.

Message constructors declare typed variables, but do not allocate storage; the first operation on a message variable creates a C++ object.

Datetime and field objects are identical to the corresponding C structs. Their constructors declare typed variables and allocate struct storage.

## Uncaught C++ Exceptions

The Rendezvous C library (within the C++ wrapper) does not support uncaught C++ exceptions. Throwing uncaught exceptions through Rendezvous C library code can leave Rendezvous code in undefined states.

Application callback methods in your C++ programs must ensure that they catch all exceptions thrown within their code (and exceptions thrown within application program code that they call). Such callback methods include [TibrvMsgCallback::onMsg\(\)](#), [TibrvEventOnComplete::onComplete\(\)](#), and others.

# Strings and Character Encodings

Rendezvous software uses strings in several roles:

- String data inside message fields
- Field names
- Subject names (and other *associated* strings that are not strictly *inside* the message)
- Certified delivery correspondent names
- Group names (fault tolerance)

All these strings (both in C++ and in wire format) use the character encoding appropriate to the ISO locale of the *sender*. For example, the United States is locale `en_US`, and uses the Latin-1 character encoding (also called ISO 8859-1); Japan is locale `ja_JP`, and uses the Shift-JIS character encoding.

When two programs exchange messages within the same locale, strings are always correct. However, when a message sender and receiver use different character encodings, the receiving program must convert between encodings as needed. Rendezvous software does not convert automatically.



# Custom Datatypes

The Rendezvous C++ API does not include classes to support custom datatypes. However, the C API machinery is available in C++ programs. The Rendezvous distribution includes a C++ source code example.

## See Also

Custom Datatypes in TIBCO Rendezvous C Reference

# Programmer's Checklist

---

Developers of Rendezvous programs can use this checklist during the four phases of the development cycle: installing Rendezvous software, coding your C++ program, compiling your C++ program, and running your program.

# Checklist

## Code

- Include the appropriate Rendezvous C++ header files; see [Include These Header Files](#).

## Compile

- Compile your programs.

## Link

- Link with the appropriate Rendezvous C++ and C library files; see [Link These Library Files](#).
- If name-mangling incompatibilities prevent your program from linking with the Rendezvous C++ library, you might need to recompile the C++ library. For more information, see [Source Code Distribution](#).
- To create a shared library that uses the Rendezvous C++ API, you must recompile the Rendezvous C++ library. (This step is not necessary on Windows and TruUNIX64 platforms.) For more information, see [Source Code Distribution](#).

## Run

- Be sure that the Rendezvous daemon can run on each application host computer. The user's path must contain a version appropriate for the application host. For more information, see Rendezvous Daemon (rvd) in TIBCO Rendezvous Administration.

# Programming Restrictions



## Warning

In general, it is illegal to call Rendezvous methods from inside signal handlers.

# Include These Header Files

Rendezvous C++ programs must include the appropriate header files from this list.

## Header Files

Header File	Description
<b>Communications, Events and Data All programs must include this API header file.</b>	
tibrv/tibrvcpp.h	Include this header file for the Rendezvous C++ API. This header automatically includes tibrv.h.
<b>Certified Message Delivery and Distributed Queue</b>	
tibrv/cmcpp.h	Include this header file for the Rendezvous certified message delivery and distributed queue C++ API. This header automatically includes cm.h.
<b>Fault Tolerance</b>	
tibrv/ftcpp.h	Include this header file for the Rendezvous fault tolerance C++ API. This header automatically includes ft.h.
<b>Secure Daemons and OpenSSL</b>	
Programs that connect to secure daemons (rvsd, rvsrd) must include this header file.	
tibrv/sdcpp.h	Include this header file for the Rendezvous secure daemon C++ API. This header automatically includes sd.h.

# Link These Library Files

Rendezvous C++ programs must link the appropriate library files. Choose from the appropriate table based on operating system platform.

## UNIX

In UNIX environments, both shared and static libraries are available. We recommend shared libraries to ease forward migration.

### Linker Flags for UNIX

Linker Flag	Description
<b>Rendezvous C++ Library</b>	
All programs must link this library.	
-ltibrvcpp	All programs must link using this library flag.
<b>Communications, Data and Event Manager</b>	
All programs must link <i>only one</i> of these two libraries.	
-ltibrv	All programs must link this library.
<b>Secure Daemon</b>	
-ltibrvsd	Programs that connect to secure daemons (rvsd, rvsrd) must link using this library flag.
<b>Certified Message Delivery, Fault Tolerance, and Distributed Queues</b>	
Programs may also link one or more of these libraries as needed.	
-ltibrvcmm	Programs that use <b>certified message delivery</b> must link using this library flag.
	Programs that use <b>distributed queues</b> must link using this library flag.

Linker Flag	Description
-ltibrvft	<p>Programs that use <b>fault tolerance</b> features must link using this library flag.</p> <p>Programs that use <b>distributed queues</b> must link using this library flag.</p>
-ltibrvcmq	<p>Programs that use <b>distributed queues</b> must link using this library flag.</p> <p>In addition, distributed queue programs also use communications, certified message delivery, and fault tolerance libraries; they must link with appropriate flags from those groups.</p>

## Microsoft Windows

The C++ library is only available as a static library; the C library is available as a DLL and as a static library. To create a C++ DLL, you must recompile the Rendezvous C++ library; see [Source Code Distribution](#).

### Library Files for Microsoft Windows

Library File	Description
<b>Rendezvous C++ Library</b>	
All programs must link <i>only one</i> of these C++ libraries.	
libtibrvcpp.lib	<p>Rendezvous C++ libraries.</p> <p>Static libraries, for use with the /MT compiler option.</p>
libtibrvcppmd.lib	<p>Rendezvous C++ libraries.</p> <p>Static libraries, for use with the /MD compiler option.</p>

### Communications, Data and Event Manager

All programs must link *only one* of these C libraries.

Library File	Description
tibrv.lib	Rendezvous C library. DLL (import library).
libtibrv.lib	Rendezvous C library. Static library, for use with the /MT compiler option.
libtibrvmd.lib	Rendezvous C library. Static library, for use with the /MD compiler option.

## Secure Daemon

Programs that connect to **secure daemons** (rvsd, rvsrd) must link only one of these *sets* of libraries.

tibrvsd.lib libcrypto.lib ssleay32.lib	Secure daemon additions. DLL (import library).
libtibrvsd.lib libcrypto.lib ssleay32.lib	Secure daemon additions. Static library, for use with the /MT compiler option.
libtibrvsdmd.lib libcrypto.lib ssleay32.lib	Secure daemon additions. Static library, for use with the /MD compiler option.

## Certified Message Delivery

Programs that use **certified message delivery** must link *only one* of these C libraries.



Library File	Description
Programs that use <b>distributed queues</b> must link <i>only one</i> of these libraries.	
tibrvcmlib	Rendezvous certified message delivery software.  DLL (import library).
libtibrvcmlib	Rendezvous certified message delivery software.  Static library, for use with the /MT compiler option.
libtibrvcmmmdlib	Rendezvous certified message delivery software.  Static library, for use with the /MD compiler option.

## Fault Tolerance

Programs that use **fault tolerance** must link *only one* of these C libraries.

Programs that use **distributed queues** must link *only one* of these libraries.

tibrvftlib	Rendezvous fault tolerance software.  DLL (import library).
libtibrvftlib	Rendezvous fault tolerance software.  Static library, for use with the /MT compiler option.
libtibrvftmmdlib	Rendezvous fault tolerance software.  Static library, for use with the /MD compiler

Library File	Description
	option.

## Distributed Queue

Programs that use **distributed queues** must link *only one* of these C libraries.

In addition, distributed queue programs also use certified message delivery, and fault tolerance libraries; they must link appropriate libraries from those groups.

tibrvcmq.lib	Rendezvous distributed queue software.  DLL (import library).
libtibrvcmq.lib	Rendezvous distributed queue software.  Static library, for use with the /MT compiler option.
libtibrvcmqmd.lib	Rendezvous distributed queue software.  Static library, for use with the /MD compiler option.

# Source Code Distribution

We compile the C++ API libraries on each platform using the vendor's primary compiler. This default case works well in most environments.

For maximum flexibility we also distribute the C++ API as source code. Some environments are incompatible with the default compilation settings. In these situations, you can recompile the Rendezvous C++ API libraries to match your environment. For example:

- The compiler in your environment differs from our primary compiler in its *name-mangling* technique. By recompiling the Rendezvous C++ libraries, you can ensure compatibility with other software compiled in your environment.
- Our static compilation is not *position independent* (on most platforms). To use Rendezvous software to produce shared libraries or executables, you must recompile with appropriate flags.

# Rendezvous Environment

---

This brief section describes the methods that open and close the internal machinery upon which Rendezvous software depends.

# Tibrv

*Class*

## Declaration

```
class Tibrv
```

## Purpose

The Rendezvous environment.

## Remarks

Programs do not create instances of [Tibrv](#). Instead, programs use its static methods to open and close the Rendezvous environment, and extract utility objects.

Method	Description
<b>Environment Life Cycle and Properties</b>	
<a href="#">Tibrv::open()</a>	Start Rendezvous internal machinery.
<a href="#">Tibrv::close()</a>	Stop and destroy Rendezvous internal machinery.
<a href="#">Tibrv::version()</a>	Identify the Rendezvous API release number.
<b>Utility Objects</b>	
<a href="#">Tibrv::defaultQueue()</a>	Extract the default queue object.
<a href="#">Tibrv::processTransport()</a>	Extract the intra-process transport object.

# Tibrv::close()

*Method*

## Declaration

```
static TibrvStatus close();
```

## Purpose

Stop and destroy Rendezvous internal machinery.

## Remarks

After [Tibrv::close\(\)](#) destroys the internal machinery, Rendezvous software becomes inoperative:

- Events no longer arrive in queues.
- All events, queues and queue groups are unusable, so programs can no longer dispatch events.
- All transports are unusable, so programs can no longer send outbound messages.

After closing a [Tibrv](#) object, all events, transports, queues and queue groups associated with that environment are invalid; it is illegal to call any methods of these objects.

## Reference Count

A reference count protects against interactions between programs and third-party packages that call [Tibrv::open\(\)](#) and [Tibrv::close\(\)](#). Each call to [Tibrv::open\(\)](#) increments an internal counter; each call to [Tibrv::close\(\)](#) decrements that counter. A call to [Tibrv::open\(\)](#) actually creates internal machinery only when the reference counter is zero; subsequent calls merely increment the counter, but do not duplicate the machinery. A call to [Tibrv::close\(\)](#) actually destroys the internal machinery only when the call decrements the counter to zero; other calls merely decrement the counter. In each program, the number of calls to [Tibrv::open\(\)](#) and [Tibrv::close\(\)](#) must match.

## See Also

[Tibrv::open\(\)](#)

# Tibrv::defaultQueue()

*Method*

## Declaration

```
static TibrvQueue* defaultQueue();
```

## Purpose

Extract the default queue object.

## Remarks

If Rendezvous is not open, the default queue is invalid; nevertheless, this method returns a pointer to it.

Each process has exactly one default queue; the call [Tibrv::open\(\)](#) automatically creates it. Programs must not destroy the default queue.

## See Also

[TibrvQueue](#).



# Tibrv::open()

*Method*

## Declaration

```
static TibrvStatus open();
```

## Purpose

Start Rendezvous internal machinery.

## Remarks

This call creates the internal machinery that Rendezvous software requires for its operation:

- Internal data structures.
- Default event queue.
- Intra-process transport.
- Event driver.

Until the first call to [Tibrv::open\(\)](#) creates the internal machinery, all events, transports, queues and queue groups are unusable. Messages and their methods do not depend on the internal machinery.

## Reference Count

A reference count protects against interactions between programs and third-party packages that call [Tibrv::open\(\)](#) and [Tibrv::close\(\)](#). Each call to [Tibrv::open\(\)](#) increments an internal counter; each call to [Tibrv::close\(\)](#) decrements that counter. A call to [Tibrv::open\(\)](#) actually creates internal machinery only when the reference counter is zero; subsequent calls merely increment the counter, but do not duplicate the machinery. A call to [Tibrv::close\(\)](#) actually destroys the internal machinery only when the call decrements the counter to zero; other calls merely decrement the counter. In each program, the number of calls to [Tibrv::open\(\)](#) and [Tibrv::close\(\)](#) must match.

## See Also

[Tibrv::close\(\)](#)

# Tibrv::processTransport()

*Method*

## Declaration

```
static TibrvProcessTransport* processTransport();
```

## Purpose

Extract the intra-process transport object.

## Remarks

If Rendezvous is not open, the intra-process transport is invalid; nevertheless, this method returns a pointer to it.

Each process has exactly one intra-process transport; the call [Tibrv::open\(\)](#) automatically creates it. Programs must not destroy the intra-process transport.

## See Also

[TibrvProcessTransport](#)

# Tibrv::version()

*Method*

## Declaration

```
static const char* version();
```

## Purpose

Identify the Rendezvous API release number.

# TibrvSdContext

*Class*

## Declaration

```
class TibrvSdContext
```

## Purpose

This class defines static methods for interacting with secure Rendezvous daemons.

## Remarks

Programs do not create instances of [TibrvSdContext](#). Instead, programs use its static methods to configure user names, passwords and certificates, and to register trust in daemon certificates.

Method	Description
<a href="#">TibrvSdContext:setDaemonCert()</a>	Register trust in a secure daemon.
<a href="#">TibrvSdContext:setUserCertWithKey()</a>	Register a (PEM) certificate with private key for identification to secure daemons.
<a href="#">TibrvSdContext:setUserCertWithKeyBin()</a>	Register a (PKCS #12) certificate with private key for identification to secure daemons.
<a href="#">TibrvSdContext:setUserNameWithPassword()</a>	Register a user name with password for identification to secure daemons.

# TibrvSdContext:setDaemonCert()

*Method*

## Declaration

```
static TibrvStatus setDaemonCert(  
    const char*  daemonName,  
    const char*  daemonCert);  
#define TIBRV_SECURE_DAEMON_ANY_NAME  (NULL)  
#define TIBRV_SECURE_DAEMON_ANY_CERT  (NULL)
```

## Purpose

Register trust in a secure daemon.

## Remarks

When any program transport connects to a secure daemon, it verifies the daemon's identity using TLS protocols. Certificates registered using this method identify trustworthy daemons. Programs divulge user names and passwords to daemons that present registered certificates.

Parameter	Description
daemonName	Register a certificate for a secure daemon with this name. For the syntax and semantics of this parameter, see <a href="#">Daemon Name</a> , below.
daemonCert	Register this public certificate. The text of this certificate must be in PEM encoding.  See also <a href="#">Certificate</a> .

## Daemon Name

The daemon name is a three-part string of the form:

```
ssl:host:port_number
```

This string must be identical to the string you supply as the daemon argument to the transport creation call; see [TibrvNetTransport::create\(\)](#).

Colon characters (:) separate the three parts.

*ssl* indicates the protocol to use when attempting to connect to the daemon.

*host* indicates the host computer of the secure daemon. You can specify this host either as a network IP address, or a hostname. Omitting this part specifies the local host.

*port\_number* specifies the port number where the secure daemon listens for TLS connections.

(This syntax is similar to the syntax connecting to remote daemons, with the addition of the prefix *ssl*.)

In place of this three-part string, you can also supply the constant `TIBRV_SECURE_DAEMON_ANY_NAME`. This form lets you register a catch-all certificate that applies to any secure daemon for which you have not explicitly registered another certificate. For example, you might use this form when several secure daemons share the same certificate.

## Certificate

For important details, see [CA-Signed Certificates in TIBCO Rendezvous Administration](#).

In place of an actual certificate, you can also supply the constant `TIBRV_SECURE_DAEMON_ANY_CERT`. The program accepts any certificate from the named secure daemon. For example, you might use this form when testing a secure daemon configuration, before generating any actual certificates.

## Any Name and Any Certificate

Notice that the constants `TIBRV_SECURE_DAEMON_ANY_NAME` and `TIBRV_SECURE_DAEMON_ANY_CERT` each eliminate one of the two security checks before transmitting sensitive identification data to a secure daemon. We strongly discourage using both of these constants simultaneously, because that would eliminate all security checks, leaving the program vulnerable to unauthorized daemons.

# TibrvSdContext:setUserCertWithKey()

*Method*

## Declaration

```
static TibrvStatus setUserCertWithKey(  
    const char* userCertWithKey,  
    const char* password);
```

## Purpose

Register a (PEM) certificate with private key for identification to secure daemons.

## Remarks

When any program transport connects to a secure daemon, the daemon verifies the program's identity using TLS protocols.

The Rendezvous API includes two methods that achieve similar effects:

- This call accepts a certificate in PEM text format.
- [TibrvSdContext:setUserCertWithKeyBin\(\)](#) accepts a certificate in PKCS #12 binary format.

Parameter	Description
userCertWithKey	Register this user certificate with private key. The text of this certificate must be in PEM encoding.
password	Use this password to decrypt the private key.



### Important

For important information about password security, see Security Factors in TIBCO Rendezvous Administration.



## CA-Signed Certificate

You can also supply a certificate signed by a certificate authority (CA). To use a CA-signed certificate, you must supply not only the certificate and private key, but also the CA's public certificate (or a chain of such certificates). Concatenate these items in one string. For important details, see [CA-Signed Certificates](#) in TIBCO Rendezvous Administration.

## Errors

Error status code [TIBRV\\_INVALID\\_FILE](#) can indicate either disk I/O failure, or invalid certificate data, or an incorrect password.

## See Also

[TibrvSdContext:setUserCertWithKeyBin\(\)](#)

# TibrvSdContext:setUserCertWithKeyBin()

*Method*

## Declaration

```
static TibrvStatus setUserCertWithKeyBin(  
    const void*   userCertWithKey,  
    tibrv_u32     userCertWithKey_size,  
    const char*   password);
```

## Purpose

Register a (PKCS #12) certificate with private key for identification to secure daemons.

## Remarks

When any program transport connects to a secure daemon, the daemon verifies the program's identity using TLS protocols.

The Rendezvous API includes two methods that achieve similar effects:

- This call accepts a certificate in PKCS #12 binary format.
- [TibrvSdContext:setUserCertWithKey\(\)](#) accepts a certificate in PEM text format.

Parameter	Description
userCertWithKey	Register this user certificate with private key. The binary data of this certificate must be in PKCS #12 format.
userCertWithKey_size	The length (in bytes) of the certificate data.
password	Use this password to decrypt the private key.



### Important

For important information about password security, see Security Factors in TIBCO Rendezvous Administration.

## CA-Signed Certificate

You can also supply a certificate signed by a certificate authority (CA). To use a CA-signed certificate, you must supply not only the certificate and private key, but also the CA's public certificate (or a chain of such certificates). For important details, see CA-Signed Certificates in TIBCO Rendezvous Administration.

## Errors

Error status code [TIBRV\\_INVALID\\_FILE](#) can indicate either disk I/O failure, or invalid certificate data, or an incorrect password.

## See Also

[TibrvSdContext:setUserCertWithKey\(\)](#)

# TibrvSdContext:setUserNameWithPassword()

*Method*

## Declaration

```
static TibrvStatus setUserNameWithPassword(  
    const char*  userName,  
    const char*  password);
```

## Purpose

Register a user name with password for identification to secure daemons.

## Remarks

When any program transport connects to a secure daemon, then daemon verifies the program's identity using TLS protocols.

Parameter	Description
userName	Register this user name for communicating with secure daemons.
password	Register this password for communicating with secure daemons.



### Important

For important information about password security, see Security Factors in TIBCO Rendezvous Administration.

# Data

---

This section describes messages and the data they contain.

## See Also

[Strings and Character Encodings](#)

[Datatypes](#)

[Custom Datatypes](#) in TIBCO Rendezvous C Reference

# Validity of Data Extracted From Message Fields

To extract data values from the fields of a message, programs use a set of *get* convenience methods. All of these methods extract a *snapshot* of the message data—that is, the data value as it exists at a particular time. If the program later modifies the message by removing or updating the field, the snapshot remains unchanged.

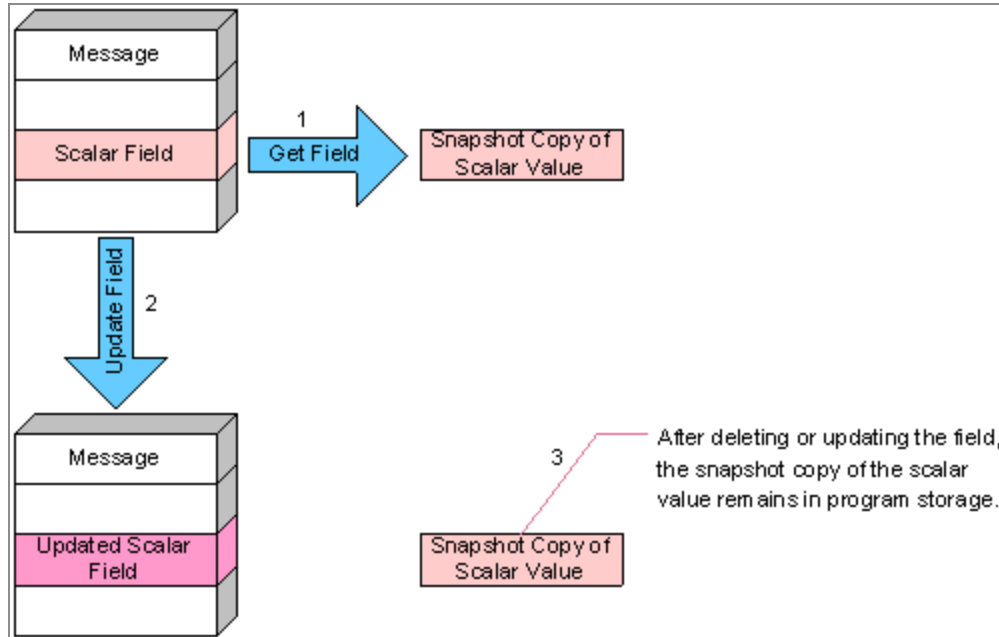
Rendezvous messages implement snapshot semantics using two separate strategies for scalar data and pointer data.

## Scalar Snapshot

To extract the value of a *scalar* field, a program declares a scalar in program-owned storage, and passes its address to the *get* method; the *get* method copies a snapshot of the scalar field value from the message into program storage.

The program can modify its snapshot at any time without affecting the original message. The program can update or delete the message field at any time without affecting the snapshot copy.

Figure 1: Extracting a Scalar Field

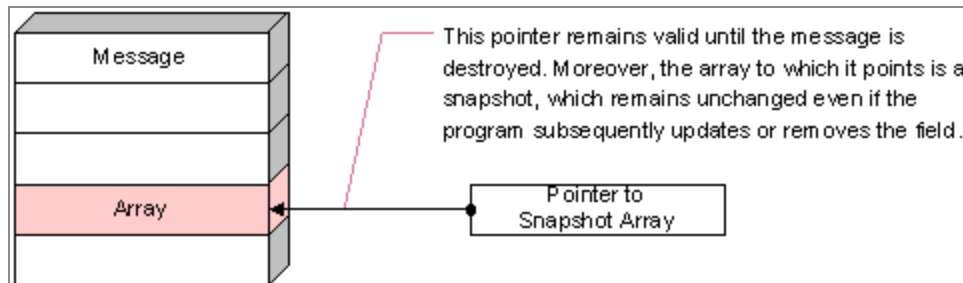


## Pointer Snapshot

Pointer data is a broad category, which includes arrays, strings, opaque byte sequences, XML data, and submessages.

To extract the value of an array, string, XML, or opaque field, a program declares a typed pointer variable in program-owned storage, and passes its reference to the *get* method; the *get* method copies a pointer to the field value into the program's variable. The method does *not* copy data into program-owned storage; the data still resides in storage associated with the message. Nonetheless, Rendezvous software protects the integrity of snapshot pointer data from subsequent changes to the message field.

Figure 2: Extracting a Pointer Field



(Schematic diagrams in this section illustrate the general principles of *snapshot* semantics as they apply to pointer data of message fields. However, these diagrams do *not* accurately reflect the storage allocation and geometry of messages, nor do they reflect the underlying implementation of snapshots.)

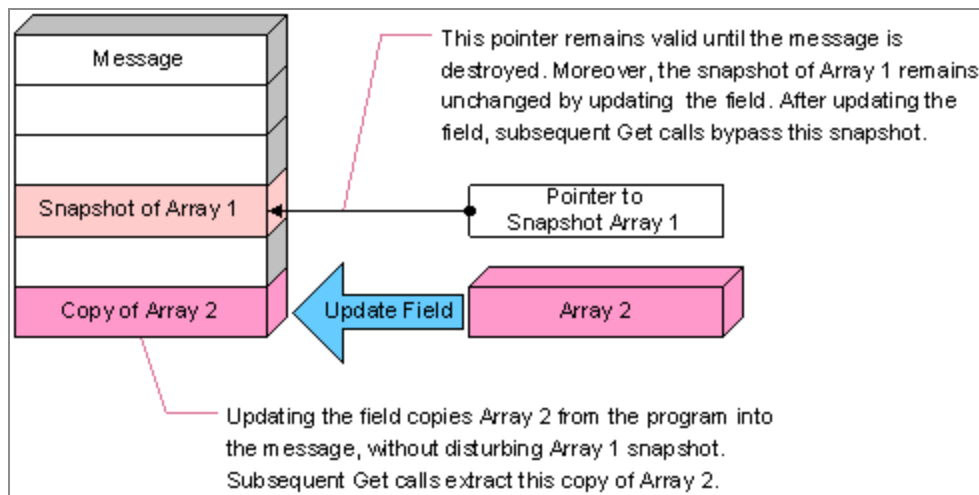
## Rendezvous Protects Pointer Snapshots from Changes to the Message

If the program *removes* the field from the message, then Rendezvous software protects the integrity of the snapshot data by retaining it in storage associated with the message; the program's pointer to the snapshot data remains valid until the message is destroyed, even though the data is no longer accessible through the message.

If the program *updates* the message field (see [Updating a Pointer Field](#)), then Rendezvous software protects the integrity of the snapshot data by retaining it in storage associated with the message; the program's pointer to the snapshot data remains valid until the message is destroyed, and the program's view of the snapshot data remains unchanged—even though the *get* method would extract updated data from the message.

These semantics apply to all pointer data—arrays, strings, opaque byte sequences, XML data, and submessages.

Figure 3: Updating a Pointer Field



## Do Not Modify Pointer Snapshots

Programs must treat array, string, XML data and opaque pointer data as *read-only snapshots*, and must not modify the data to which those pointers refer. For example, it is



*illegal* for programs to change any element in a snapshot array; it is *illegal* for programs to change any byte in snapshot strings, XML data or opaque byte sequences.

Although Rendezvous software does not enforce this restriction, violating this rule is dangerous, and can result in erroneous program behavior. Do not attempt to modify the elements of an array snapshot, nor the bytes of a string, XML data or opaque snapshot.

[Updating a Pointer Field](#) illustrates the correct way to modify the value of pointer data within a message field. Instead of directly modifying storage associated with the message, supply the new value through an *update* call, which replaces the whole value of the field. (Even after updating or removing the field, it is still illegal to modify the snapshot.)

Although superseded snapshot data remains in storage associated with the message, it is not included when sending the message, nor when accessing message fields.

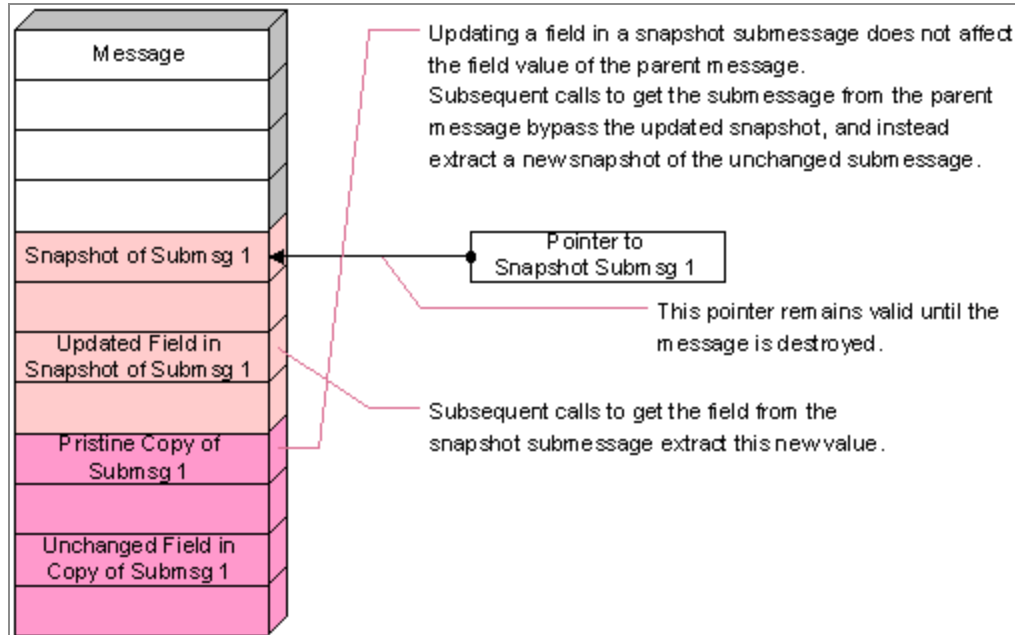
## Rendezvous Protects the Message from Changes to Submessage Snapshots

In contrast to other pointer data, programs may legally modify snapshot submessages (since a submessage is also a message in its own right). Field modification methods apply equally to ordinary messages and to submessage snapshots extracted using *get* calls.

Moreover, modifying a snapshot submessage does not affect the original field in the parent message. Field modification methods protect the integrity of the parent message when updating or removing a field in a submessage snapshot, or when adding a new field to the submessage snapshot.

[Updating a Submessage Field](#) illustrates this protection for parent messages. After updating a field in a snapshot submessage, subsequent *get* calls extract a pristine copy of the submessage from the field of the parent message, creating a second snapshot. Meanwhile, the modified snapshot submessage remains in storage owned by the parent message; it remains valid until the parent message is destroyed. (However, if the program detaches the snapshot submessage, it remains valid until the program explicitly destroys the submessage.)

Figure 4: Updating a Submessage Field



## Deleting Snapshot References

Ordinarily, snapshot references remain part of the message until the program destroys the message. However, in *rare* situations snapshots can accumulate within a program, causing unbounded memory growth.

For example, consider the result of a program that calls a method repeatedly on the same message, where each call creates a new snapshot within the storage associated with that message. Message storage grows, and destroying the message is the only way to free that storage.

A pair of methods give programs explicit control over snapshot references, so you can avoid such situations:

- `TibrvMsg::markReferences()`
- `TibrvMsg::clearReferences()`

When a program repeatedly extracts snapshot references data *and* does not destroy the parent messages, consider using these methods to control the proliferation of references.

## See Also

[TibrvMsg::markReferences\(\)](#)

# Multiple Subscription Snapshots

Rendezvous software also protects the integrity of messages distributed to multiple subscriptions. When a callback method modifies an inbound message (whether detached or not), Rendezvous software still presents the original message content to subsequent callback methods.

# Field Names and Field Identifiers

Rendezvous programs can specify fields in two ways:

- A *field name* is a character string. Each field can have at most one name. Several fields can have the same name.
- A *field identifier* is a 16-bit unsigned integer, which must be unique within the message. That is, two fields in the same message cannot have the same identifier. However, a nested submessage is considered a separate identifier space from its enclosing parent message and any sibling submessages.

Message methods specify fields using a combination of a field name and a unique field identifier. When absent, the default field identifier is zero.

To compare the speed and space characteristics of these two options, see [Search Characteristics](#).

## Rules and Restrictions

NULL is a legal field name *only* when the identifier is zero. It is *illegal* for a field to have *both* a non-zero identifier *and* a NULL field name.

However, NULL is not the same as "" (the empty string). It is *legal* for a field to have a non-zero identifier and the empty string as its field name.

## Field Search Algorithm

The methods that *get* message fields (including methods that *update* or *remove* fields, since these methods call *get* internally) all use this algorithm to find a field within a message, as specified by a field identifier and a field name.

### Procedure

1. If the program supplied a *non-zero* field identifier, then search for the field with that identifier. On failure, continue to step 2. (If the identifier is zero, skip to step 3.)
2. If the identifier search (in step 1) fails, and the program supplied a non-NULL field name, then search for a field with that name. On failure, or if the program supplied NULL as the field name, return the status code `TIBRV_NOT_FOUND`. If the name search succeeds, but the actual identifier in the field is non-NULL (so it does not match the identifier supplied) then return the status code `TIBRV_ID_CONFLICT`.

3. Begin here when the program supplied *zero* as the identifier (or omitted a field identifier).

Search for a field with the specified name—even if that name is NULL. On failure, return the status code [TIBRV\\_NOT\\_FOUND](#).

If a message contains several fields with the same name, searching by name finds the first instance of the field with that name.

## Adding a New Field

When a program adds a new field to a message, it can attach a field name, a field identifier, or both. If the program supplies an identifier, Rendezvous software checks that it is unique within the message; if the identifier is already in use, the operation fails with the status code [TIBRV\\_ID\\_IN\\_USE](#).

## Search Characteristics

In general, an identifier search completes in constant time. In contrast, a name search completes in linear time proportional to the number of fields in the message. Name search is quite fast for messages with 16 fields or fewer; for messages with more than 16 fields, identifier search is faster.

## Space Characteristics

The smallest field name is a one-character string, which occupies three bytes in Rendezvous wire format. That one ASCII character yields a name space of 127 possible field names; a larger range requires additional characters.

Field identifiers are 16 bits, which also occupy three bytes in Rendezvous wire format. However, those 16 bits yield a space of 65535 possible field identifiers; that range is fixed, and cannot be extended.

# Finding a Field Instance

When a message contains several field instances with the same field name, these methods find a specific instance by name and number (they do not use field identifiers):

- [TibrvMsg::getFieldInstance\(\)](#).

- [TibrvMsg::removeFieldInstance\(\)](#).

# TibrvMsg

*Class*

## Declaration

```
class TibrvMsg
```

## Purpose

Represent Rendezvous messages.

## Remarks

This class lacks `create()` and `destroy()` methods; use the constructor and destructor instead.

Method	Description
<b>Message Life Cycle and Properties</b>	
<a href="#">TibrvMsg()</a>	Create a message object.
<a href="#">~TibrvMsg()</a>	Destroy a message object.
<a href="#">TibrvMsg::convertToString()</a>	Format a message as a string.
<a href="#">TibrvMsg::createCopy()</a>	Store a copy of this message in another message object.
<a href="#">TibrvMsg::detach()</a>	Detach an inbound message from Rendezvous storage; the program assumes responsibility for destroying the message.
<a href="#">TibrvMsg::expand()</a>	Enlarge a message by allocating additional storage.

Method	Description
<a href="#">TibrvMsg::getAsBytes()</a>	Extract the data from a message as a byte sequence.
<a href="#">TibrvMsg::getAsBytesCopy()</a>	Extract a copy of the data from a message as a byte sequence.
<a href="#">TibrvMsg::getByteSize()</a>	Return the size of a message (in bytes).
<a href="#">TibrvMsg::getHandle()</a>	Extract the C message handle from a C++ message object.
<a href="#">TibrvMsg::getNumFields()</a>	Extract the number of fields in a message.
<a href="#">TibrvMsg::getStatus()</a>	Extract the status code stored by the message constructor.
<a href="#">TibrvMsg::isDetached()</a>	Determine the ownership of the message.
<a href="#">TibrvMsg::reset()</a>	Clear a message, preparing it for re-use.
<b>Fields</b>	
<a href="#">TibrvMsg::addField()</a>	Add a field object to a message.
<a href="#">TibrvMsg::getField()</a>	Get a specified field from a message.
<a href="#">TibrvMsg::getFieldByIndex()</a>	Get a field from a message by an index.
<a href="#">TibrvMsg::getFieldInstance()</a>	Get a specified instance of a field from a message.



Method	Description
<a href="#">TibrvMsg::removeField()</a>	Remove a field from a message.
<a href="#">TibrvMsg::removeFieldInstance()</a>	Remove a specified instance of a field from a message.
<a href="#">TibrvMsg::updateField()</a>	Update a field within a message.
<b>Address Information</b>	
<a href="#">TibrvMsg::getReplySubject()</a>	Extract the reply subject from a message.
<a href="#">TibrvMsg::getSendSubject()</a>	Extract the subject from a message.
<a href="#">TibrvMsg::setReplySubject()</a>	Set the reply subject for a message.
<a href="#">TibrvMsg::setSendSubject()</a>	Set the subject for a message.
<b>Field References</b>	
<a href="#">TibrvMsg::clearReferences()</a>	Clear references in this message.
<a href="#">TibrvMsg::markReferences()</a>	Mark references in this message.
<b>Event Dispatched</b>	
<a href="#">TibrvMsg::getEvent()</a>	Extract the event associated with a (dispatched) message object.
<b>Time String</b>	
<a href="#">TibrvMsg::getCurrentTime()</a>	Get the current date and time.

## See Also

[Strings and Character Encodings](#)

[TibrvMsgField](#)

## Datatypes

# TibrvMsg()

*Constructor*

## Declaration

```
TibrvMsg(tibrv_u32 initialSize=512); // Declare empty msg obj.  
TibrvMsg(const TibrvMsg& msg);      // Copy constructor.  
TibrvMsg(const void* bytes);        // Create from bytes.  
TibrvMsg(tibrvMsg Cmsg,            // Create from C handle.  
         tibrv_bool detached);
```

## Purpose

Create a message object.

## Remarks

The first form of this constructor merely declares a C++ variable. It does not allocate storage; it does not create a C message object. (Subsequent operations on the variable automatically allocates storage and create the inner structure.) This form works efficiently with methods such as [TibrvTransport::sendRequest\(\)](#), which requires a variable to receive its reply message and overwrites the contents of that variable.

The other forms of this constructor simultaneously declare a variable, allocate storage, create an object with inner structure, and store the status code in the object. The status code indicates any conditions that might make the message object unusable; if the object is unusable, any subsequent operation on the message will return the same status code.

None of the constructors place address information (for example, the subject) on the new message object.

This class has no `destroy()` method; use the destructor instead (see [~TibrvMsg\(\)](#)).

Parameter	Description
initialSize	Optional.

Parameter	Description
	This form of the constructor declares an empty message variable. When a subsequent operation triggers storage allocation, allocate a storage block of this size (in bytes).
<code>msg</code>	Create an independent copy of this message.
<code>bytes</code>	<p>Create a message with fields populated from this byte array. The new message is completely independent of the byte array.</p> <p>For example, programs can create such byte arrays from messages using the method <a href="#">TibrvMsg::getAsBytes()</a>, and store them in files; after reading them from such files, programs can reconstruct a message from its byte array.</p>
<code>Cmsg</code>	Create a C++ message object that embeds this C message handle. This form of the constructor does not copy the C message.
<code>detached</code>	<p>Specify whether the <a href="#">Cmsg</a> is detached—that is, whether the program owns the message.</p> <p><code>TIBRV_TRUE</code> indicates a detached inbound message, or a message created in the program. The destructor also destroys the embedded C message.</p> <p><code>TIBRV_FALSE</code> indicates an attached inbound message. The destructor does not destroy the embedded C message.</p>

## See Also

[TibrvMsg::createCopy\(\)](#)

[TibrvMsg::detach\(\)](#)

[TibrvMsg::getAsBytes\(\)](#)

[TibrvMsg::getStatus\(\)](#)

[TibrvMsg::getHandle\(\)](#)

tibrvMsg in TIBCO Rendezvous C Reference

# ~TibrvMsg()

*Destructor*

## Declaration

```
~TibrvMsg();
```

## Purpose

Destroy a message object.

## See Also

[TibrvMsg::detach\(\)](#)

# TibrvMsg::addField()

*Method*

## Declaration

```
TibrvStatus addField(const TibrvMsgField& field)
```

## Purpose

Add a field object to a message.

## Remarks

This method copies the data into the new message field. All related convenience methods behave similarly.

Parameter	Description
field	Add this field to the message.

## Adding Fields to a Nested Message

Rendezvous programs must use this three-step process:

### Procedure

1. Extract the nested submessage (see [Get Nested Message](#)).
2. Add the new fields to the extracted submessage, using type-specific convenience methods or this method. The field is added to a snapshot copy of the submessage, and modifies the copy rather than the original parent message.
3. Store the modified submessage back into the field of the parent message (see [Update Nested Message](#)).

## Avoiding Common Mistakes

Whenever possible, we recommend storing arrays in message fields using one of the Rendezvous array types. This strategy makes the most efficient use of storage space, processor time, and network bandwidth.

If you must store array elements as individual fields, be careful mapping array indices to field identifiers. Zero-based arrays are common in C++ programs, but zero has a special meaning as a field identifier—it represents the absence of an identifier. Do not map the zero<sup>th</sup> element of an array (`myArray[0]`) to a field with identifier zero; it is impossible to retrieve such a field by its identifier (because it does not have one).

It is illegal to add a field that has both a NULL field name, and a non-zero field identifier.

## Reserved Field Name



### Warning

The field name `_data_` is reserved. Programs may not add fields with this name.

More generally, all fields that begin with the underbar character are reserved.

## Field Name Length

The constant `TIBRVMSG_FIELDNAME_MAX` (127) determines the longest possible field name.

## Convenience Methods

When the datatype of a field is determined during execution, use this general method. When you can determine the datatype of a field before compile-time, we recommend using type-specific convenience methods instead of this general method. Type-specific methods yield these advantages when adding fields:

- Code readability.
- Type checking.
- Accept constants and literals directly.

(Type-specific methods yield even further advantages when extracting or updating fields.)

These categories of type-specific convenience methods add a new field:



- [Add Scalar.](#)
- [Add Array.](#)
- [Add Nested Message.](#)
- [Add String.](#)
- [Add Opaque Byte Sequence.](#)
- [Add XML Byte Sequence.](#)
- [Add DateTime.](#)

## See Also

[Add Scalar](#)

# Add Scalar

## Convenience Methods

### Declaration

```
TibrvStatus addtype(  
    const char*  fieldName,  
    tibrv_type  value,  
    tibrv_u16   fieldId=0);
```

### Purpose

Add a field containing a scalar value.

Method Name	Field Value Type	Type Description
addBool	tibrv_bool	boolean
addF32	tibrv_f32	32-bit floating point
addF64	tibrv_f64	64-bit floating point
addI8	tibrv_i8	8-bit integer
addI16	tibrv_i16	16-bit integer
addI32	tibrv_i32	32-bit integer
addI64	tibrv_i64	64-bit integer
addU8	tibrv_u8	8-bit unsigned integer
addU16	tibrv_u16	16-bit unsigned integer
addU32	tibrv_u32	32-bit unsigned integer
addU64	tibrv_u64	64-bit unsigned integer

Method Name	Field Value Type	Type Description
<code>addIPAddr32</code>	<code>tibrv_ipaddr32</code>	4-byte IP address
<code>addIPPort16</code>	<code>tibrv_ipport16</code>	2-byte IP port

Parameter	Description
<code>fieldName</code>	<p>Create the field with this name.</p> <p>NULL is a legal name. However, if <code>fieldId</code> is non-zero, then <code>fieldName</code> must be non-NULL.</p>
<code>value</code>	<p>Add a new field with this value (which may be a literal or stored in a variable).</p> <p>The convenience method must correspond to the datatype of this value. We recommend casting the data to match the convenience method.</p> <p>The method copies the value into the new message field.</p>
<code>fieldId</code>	<p>Create the new field with this identifier. Zero is a special value that signifies no field identifier. All non-zero field identifiers must be unique within each message.</p> <p>It is illegal to add a field that has both a NULL field name, and a non-zero field identifier.</p>

# Add Array

## Convenience Methods

### Declaration

```
TibrvStatus addelement_typeArray(
    const char*      fieldName,
    const element_type* value,
    tibrv_u32        numElements,
    tibrv_u16        fieldId=0);
```

### Purpose

Add a field containing an array value.

Method Name	Element Type	Type Description
addF32Array	tibrv_f32	32-bit floating point array
addF64Array	tibrv_f64	64-bit floating point array
addI8Array	tibrv_i8	8-bit integer array
addI16Array	tibrv_i16	16-bit integer array
addI32Array	tibrv_i32	32-bit integer array
addI64Array	tibrv_i64	64-bit integer array
addU8Array	tibrv_u8	8-bit unsigned integer array
addU16Array	tibrv_u16	16-bit unsigned integer array
addU32Array	tibrv_u32	32-bit unsigned integer array
addU64Array	tibrv_u64	64-bit unsigned integer array

Parameter	Description
fieldName	<p>Create the new field with this name.</p> <p>NULL is a legal name. However, if fieldId is non-zero, then fieldName must be non-NULL.</p>
value	<p>Add a new field that contains this array.</p> <p>The method signature must correspond to the datatype of this value.</p> <p>The method <i>copies</i> the array into the new message field.</p>
numElements	<p>When adding an array type, the program supplies the count of array elements in this parameter.</p>
fieldId	<p>Create the new field with this identifier. Zero is a special value that signifies no field identifier. All non-zero field identifiers must be unique within each message.</p> <p>It is illegal to add a field that has both a NULL field name, and a non-zero field identifier.</p>

# Add Nested Message

*Convenience Method*

## Declaration

```
TibrvStatus addMsg(  
    const char*      fieldName,  
    const TibrvMsg&  msg,  
    tibrv_u16        fieldId=0);
```

## Purpose

Add a field containing a nested submessage.

## Remarks

This method adds only the data portion of the nested message (value); it does not include any address information or certified delivery information.

Parameter	Description
fieldName	Create the new field with this name.
msg	Add a new field that contains this submessage.  The method <i>copies</i> the data into the new field.
fieldId	Create the new field with this identifier. Zero is a special value that signifies no field identifier. All non-zero field identifiers must be unique within each message.  It is illegal to add a field that has both a NULL field name, and a non-zero field identifier.

# Add String

*Convenience Method*

## Declaration

```
TibrvStatus addString(  
    const char*  fieldName,  
    const char*  value,  
    tibrv_u16    fieldId=0);
```

## Purpose

Add a field containing a string.

## Remarks

The string cannot contain interior NULL bytes, because this method expects a NULL-terminated string. To add a string with interior NULL bytes, use the generic method [TibrvMsg::addField\(\)](#).

Parameter	Description
fieldName	Create the new field with this name.
value	<p>Add a new field that contains this string (which may be a literal or stored in a variable).</p> <p>The method <i>copies</i> the data into the new field.</p>
fieldId	<p>Create the new field with this identifier. Zero is a special value that signifies no field identifier. All non-zero field identifiers must be unique within each message.</p> <p>It is illegal to add a field that has both a NULL field name, and a non-zero field identifier.</p>

# Add Opaque Byte Sequence

*Convenience Method*

## Declaration

```
TibrvStatus addOpaque(  
    const char*   fieldName,  
    const void*   value,  
    tibrv_u32     size  
    tibrv_u16     fieldId=0);
```

## Purpose

Add a field containing an opaque byte sequence.

Parameter	Description
fieldName	Create the new field with this name.
value	Add a new field that contains this opaque buffer.  The method <i>copies</i> the data into the new message field.
size	When adding an opaque buffer, the program supplies the size (in bytes) of the data in this parameter.
fieldId	Create the new field with this identifier. Zero is a special value that signifies no field identifier. All non-zero field identifiers must be unique within each message.  It is illegal to add a field that has both a NULL field name, and a non-zero field identifier.



# Add XML Byte Sequence

*Convenience Method*

## Declaration

```
TibrvStatus addXml(  
    const char*   fieldName,  
    const void*   value,  
    tibrv_u32     size  
    tibrv_u16     fieldId=0);
```

## Purpose

Add a field containing an XML byte sequence.

## Remarks

XML data is a byte sequence. Adding a field of type [TIBRVMSG\\_XML](#) compresses the bytes. Extracting data from the field uncompresses it to obtain the original byte sequence.

Parameter	Description
fieldName	Create the new field with this name.
value	Add a new field that contains the XML data in this buffer.  The method <i>copies</i> the data into the new message field.
size	When adding XML data, the program supplies the size (in bytes) of the data in this parameter.
fieldId	Create the new field with this identifier. Zero is a special value that signifies no field identifier. All non-zero field identifiers must be unique within each message.  It is illegal to add a field that has both a NULL field name, and a non-zero field identifier.

# Add DateTime

*Convenience Method*

## Declaration

```
TibrvStatus addDateTime(
    const char*      fieldName,
    const TibrvMsgDateTime& value,
    tibrv_u16        fieldId=0);
```

## Purpose

Add a field containing a Rendezvous datetime value.

Parameter	Description
fieldName	Create the new field with this name.
value	Add a new field that contains this datetime value. The method <i>copies</i> the data into the new message field.
fieldId	Create the new field with this identifier. Zero is a special value that signifies no field identifier. All non-zero field identifiers must be unique within each message.  It is illegal to add a field that has both a NULL field name, and a non-zero field identifier.

## Example

This example code converts a `time_t` value to a datetime value, and adds the datetime to a message field. Programs can adapt this example as appropriate. (For corresponding code to extract a datetime value from a message field, and convert it to a `time_t` value, see the example at [Get DateTime](#).)

```
#include <limits.h>
TibrvStatus addAsTimeT(
```

```
TibrvMsg& msg,  
const char* field,  
time_t value)  
{  
    examples/c/ d;  
    d.sec = (tibrv_i64)value;  
    return msg.addDateTime(field,d);  
}
```

## See Also

[TibrvMsgDateTime](#)

# TibrvMsg::clearReferences()

*Method*

## Declaration

```
TibrvStatus clearReferences();
```

## Purpose

Clear references in this message.

## Remarks

This method clears references back to the most recent mark.

For a description and example, see [TibrvMsg::markReferences\(\)](#).

## See Also

[Validity of Data Extracted From Message Fields](#)

[Deleting Snapshot References](#)

[TibrvMsg::markReferences\(\)](#)

# TibrvMsg::convertToString()

*Method*

## Declaration

```
TibrvStatus convertToString(const char*& string)
```

## Purpose

Format a message as a string.

## Remarks

Programs can use this method to obtain a string representation of the message for printing.

For most datatypes, this method formats the full value of the field to the string; these types are an exceptions:

[TIBRVMSG\\_OPAQUE](#) This method abbreviates the value of an opaque field; for example, [472 opaque bytes].

[TIBRVMSG\\_XML](#) This method abbreviates the value of an XML field; for example, [XML document: 472 bytes].

The size measures *uncompressed* data.

This method formats [TIBRVMSG\\_IPADDR32](#) fields as four dot-separated decimal integers.

This method formats [TIBRVMSG\\_IPPORT16](#) fields as one decimal integer.

Parameter	Description
string	The program supplies a variable in this parameter; the method stores the string representation in that variable.

## See Also

[Converting DateTime to Strings](#)

# TibrvMsg::createCopy()

*Method*

## Declaration

```
TibrvStatus createCopy(TibrvMsg& copy)
```

## Purpose

Store a copy of this message in another message object.

## Remarks

Despite its name, this method does not create a new object. Instead, it stores an independent copy in the variable it receives as a parameter.

Parameter	Description
copy	The program supplies a variable in this parameter; the method stores an independent copy of the message in that variable.

## See Also

[TibrvMsg\(\)](#)

# TibrvMsg::detach()

*Method*

## Declaration

```
TibrvMsg* detach();
```

## Purpose

Detach an inbound message from Rendezvous storage; the program assumes responsibility for destroying the message.

## Remarks

When Rendezvous software creates a message, it owns that message. This situation occurs only in the case of inbound messages presented to a data callback method; Rendezvous software destroys such messages when the callback method returns, unless the program explicitly detaches the message. After a detach operation, the program owns the message, and must explicitly destroy it to reclaim the storage.

Programs may detach inbound messages. A program cannot detach a message that it already owns; attempts to do so return NULL.

Programs can use this method to detach either an entire message, or a submessage. After detaching a submessage, the program owns that submessage, even after the destruction of the surrounding parent message.

Callback methods receive inbound messages in stack parameters, which evaporate when the callback methods return. Programs detach messages to allow the messages to persist beyond the local context. Consequently, the detach operation returns a new C++ object, which the program can store in a variable with wider scope. The original inbound message becomes invalid, transferring its embedded C message to the new C++ message object.

**Warning**

Programs may modify messages only with methods that add, remove or update fields. It is illegal to directly modify data storage in a message field. Detaching a message does not change this restriction. For further information, see [Do Not Modify Pointer Snapshots](#), or more generally, [Validity of Data Extracted From Message Fields](#).

**See Also**[~TibrvMsg\(\)](#)[TibrvCallback::onEvent\(\)](#)[TibrvMsgCallback::onMsg\(\)](#)[TibrvCmMsgCallback::onCmMsg\(\)](#)



# TibrvMsg::expand()

*Method*

## Declaration

```
TibrvStatus expand(tibrv_u32 additionalStorage);
```

## Purpose

Enlarge a message by allocating additional storage.

## Remarks

When adding data to a message would overflow the allocated space, the message automatically expands by allocating additional storage. However, reallocation (whether explicit or automatic) is a slow operation; to optimize program performance, we recommend allocating sufficient storage initially, so that reallocation is not required.

In most cases, the message expands in place (without copying). In some cases, this method copies the message to a new location. In all cases, existing pointers to the message, its fields, and its field values remain valid (even after copying).

If no space is available, this method returns the error code [TIBRV\\_NO\\_MEMORY](#).

Parameter	Description
additionalStorage	Enlarge the message by this amount (in bytes) to allocate for the message. If the message was <i>oldSize</i> bytes before this call, it is <i>oldSize</i> + <i>additionalStorage</i> when the method returns.

# TibrvMsg::getAsBytes()

*Method*

## Declaration

```
TibrvStatus getAsBytes(const void*& bytePtr);
```

## Purpose

Extract the data from a message as a byte sequence.

## Remarks

Return a copy of the message data as a byte sequence, suitable for archiving in a file. To reconstruct the message from bytes, see [TibrvMsg\(\)](#).

The byte data includes the message header and all message fields in Rendezvous wire format. It does not include address information, such as the subject and reply subject, nor certified delivery information.

The byte sequence can contain interior NULL bytes.

Parameter	Description
bytePtr	The program supplies a variable. The method stores a pointer to the byte sequence in that variable.

## See Also

[TibrvMsg\(\)](#)

[TibrvMsg::getAsBytesCopy\(\)](#)

# TibrvMsg::getAsBytesCopy()

*Method*

## Declaration

```
TibrvStatus getAsBytesCopy(  
    void* bytePtr,  
    tibrv_u32 byteSize);
```

## Purpose

Extract a copy of the data from a message as a byte sequence.

## Remarks

Return the data as a byte sequence, suitable for archiving in a file.

This method copies the message data into a buffer, which the program owns and may modify.

The byte data includes the message header and all message fields in Rendezvous wire format. It does not include address information, such as the subject and reply subject.

To allocate appropriate storage, programs determine the length of the byte sequence using [TibrvMsg::getByteSize\(\)](#).

The byte sequence can contain interior NULL bytes.

Parameter	Description
bytePtr	The program supplies a byte buffer. The method copies the byte sequence into that buffer.
byteSize	The size of the buffer.

## See Also

[TibrvMsg::getAsBytes\(\)](#)

`TibrvMsg::getByteSize()`

`TibrvMsg()`

# TibrvMsg::getByteSize()

*Method*

## Declaration

```
TibrvStatus getByteSize(tibrv_u32& byteSize) const;
```

## Purpose

Return the size of a message (in bytes).

## Remarks

This measurement accounts for the actual space that the message occupies (in wire format), including its header and its fields. It does not include storage that is allocated but unused. It does not include address information, such as the subject or reply subject.

Programs can use this method as part of these tasks:

- Measure the size of message before allocating space to store a copy—as with [TibrvMsg::getAsBytesCopy\(\)](#).
- Assess throughput rates.
- Limit output rates (also called *throttling*).

Parameter	Description
byteSize	The program supplies a variable. The method stores the message size (in bytes) in that variable.

## See Also

[TibrvMsg::getAsBytes\(\)](#)

[TibrvMsg::getAsBytesCopy\(\)](#)

# TibrvMsg::getCurrentTime()

*Method*

## Declaration

```
static TibrvStatus getCurrentTime(  
    TibrvMsgDateTime& dateTime);
```

## Purpose

Get the current date and time.

## Remarks

This static method uses an operating system call to get the current time.

Programs can call this method without reference to any message object.

Parameter	Description
dateTime	The program supplies a variable; the method stores the current time in that variable.

## Example

```
TibrvMsgDateTime myTime;  
...  
if (TibrvMsg::getCurrentTime(myTime) == TIBRV_OK)  
...  
...
```

# TibrvMsg::getField()

*Method*

## Declaration

```
TibrvStatus getField(  
    const char*    fieldName,  
    TibrvMsgField& field,  
    tibrv_u16      fieldId=0);
```

## Purpose

Get a specified field from a message.

## Remarks

Programs specify the field to retrieve using the `fieldName` and `fieldId` parameters. For details, see [Field Names and Field Identifiers](#).

The method takes a snapshot of the field, and stores that information in the `field` argument, overwriting the field object supplied as an argument.

The method copies scalar data into the program's field object. Pointer data (such as strings, arrays, submessages, XML data, or opaque byte sequences) extracted from the field remain valid until the message is destroyed; that is, even removing the field or updating the field's value does *not* invalidate pointer data.

Programs can use a related method to loop through all the fields of a message. To retrieve each field by its integer index number, see [TibrvMsg::getFieldByIndex\(\)](#).

Parameter	Description
<code>fieldName</code>	Get a field with this name.
<code>field</code>	The program supplies a <a href="#">TibrvMsgField</a> object; the method overwrites the contents of the object with a snapshot of the information from the message field.

Parameter	Description
fieldId	Get the field with this identifier. Zero is a special value that signifies no field identifier. All non-zero field identifiers must be unique within each message.

## Field Search Algorithm

This method, and related methods that *get* message fields, all use this algorithm to find a field within a message, as specified by a field identifier and a field name.

### Procedure

1. If the identifier is zero, skip to step 3.  
If the program supplied a *non-zero* field identifier, then search for the field with that identifier.  
If the search succeeds, return the field.  
On failure, continue to step 2.
2. If the identifier search (in step 1) fails, and the program supplied a non-NULL field name, then search for a field with that name.  
If the name search succeeds, and the identifier in the field is NULL, return the field.  
If the name search succeeds, but the actual identifier in the field is non-NULL (so it does not match the identifier supplied) then return the status code [TIBRV\\_ID\\_CONFLICT](#).  
If the name search fails, or if the program supplied NULL as the field name, return the status code [TIBRV\\_NOT\\_FOUND](#).
3. When the program supplied *zero* as the identifier, search for a field with the specified name—even if that name is NULL.  
If the search succeeds, return the field.  
On failure, return the status code [TIBRV\\_NOT\\_FOUND](#).

If a message contains several fields with the same name, searching by name finds the first instance of the field with that name.



## Extracting Fields from a Nested Message

Rendezvous programs must separately extract the nested submessage using `TibrvMsg::getMsg()`, and then get the desired fields from the submessage.

## Convenience Methods

In most situations, we recommend using type-specific convenience methods instead of this general method. Type-specific methods yield these advantages when extracting fields:

- Code readability.
- Type checking.
- Automatic type conversion.

However, we do recommend the general method in two specific situations:

- No convenience method exists.
- The program must extract the data exactly as it appears in the message, without automatic type conversion. (Convenience methods always convert extracted data to a specific type.)

These categories of type-specific convenience methods find a field and get its data:

- [Get Scalar](#).
- [Get Array](#).
- [Get Nested Message](#).
- [Get String](#).
- [Get Opaque Byte Sequence](#).
- [Get XML Byte Sequence](#).
- [Get DateTime](#).

## See Also

[Field Names and Field Identifiers](#)

[Get Scalar](#)

[TibrvMsgField](#)

# Get Scalar

## Convenience Methods

### Declaration

```
TibrvStatus getscalar_type(  
    const char*    fieldName,  
    tibrv_scalar_type& value,  
    tibrv_u16      fieldId=0);
```

### Purpose

Get the value of a field as a scalar value.

### Remarks

Each convenience method in this family retrieves a field and extracts its data. If the field's type (as it exists) does not match the type of the convenience method, then the method attempts to convert the data (see [Datatype Conversion](#)). If conversion is not possible, the method returns `TIBRV_CONVERSION_FAILED`.

Method Name	Type	Type Description
getBool	tibrv_bool	boolean
getF32	tibrv_f32	32-bit floating point
getF64	tibrv_f64	64-bit floating point
getI8	tibrv_i8	8-bit integer
getI16	tibrv_i16	16-bit integer
getI32	tibrv_i32	32-bit integer
getI64	tibrv_i64	64-bit integer

Method Name	Type	Type Description
getU8	tibrv_u8	8-bit unsigned integer
getU16	tibrv_u16	16-bit unsigned integer
getU32	tibrv_u32	32-bit unsigned integer
getU64	tibrv_u64	64-bit unsigned integer
getIPAddr32	tibrv_ipaddr32	4-byte IP address
getIPPort16	tibrv_ipport16	2-byte IP port

Parameter	Description
fieldName	Get a field with this name.
value	When extracting a scalar type, the program supplies a variable in this parameter, and the method <i>copies</i> the scalar value of the field to that variable.
fieldId	Get the field with this identifier. Zero is a special value that signifies no field identifier. All non-zero field identifiers must be unique within each message.

## See Also

[Validity of Data Extracted From Message Fields](#)

[Field Names and Field Identifiers](#)

# Get Array

## Convenience Methods

### Declaration

```
TibrvStatus getelement_typeArray(
    const char*      fieldName,
    tibrv_element_type*& value,
    tibrv_u32&       numElementsAddr,
    tibrv_u16        fieldId=0);
```

### Purpose

Get the value of a field as an array.

### Remarks

Each convenience method in this family retrieves a field and extracts its data. If the field's type (as it exists) is does not match the type of the convenience method, then the method attempts to convert the data (see [Datatype Conversion](#)). If conversion is not possible, the method returns `TIBRV_CONVERSION_FAILED`.

Pointer data extracted from the field remain valid until the message is destroyed; that is, even removing the field or updating the field's value does *not* invalidate pointer data.

These methods produce values that are *read-only snapshots* of the field data (see [Pointer Snapshot](#)). Programs must not modify elements of the value array.

Method Name	Element Type	Type Description
getF32Array	tibrv_f32	32-bit floating point array
getF64Array	tibrv_f64	64-bit floating point array
getI8Array	tibrv_i8	8-bit integer array
getI16Array	tibrv_i16	16-bit integer array

Method Name	Element Type	Type Description
getI32Array	tibrv_i32	32-bit integer array
getI64Array	tibrv_i64	64-bit integer array
getU8Array	tibrv_u8	8-bit unsigned integer array
getU16Array	tibrv_u16	16-bit unsigned integer array
getU32Array	tibrv_u32	32-bit unsigned integer array
getU64Array	tibrv_u64	64-bit unsigned integer array

Parameter	Description
fieldName	Get a field with this name.
value	When extracting an array type, the program supplies a variable in this parameter, and the method stores a <i>pointer</i> to the array in that variable.
numElementsAddr	When extracting an array type, the program supplies a variable in this parameter, and the method stores the number of array elements in that variable.
fieldId	Get the field with this identifier. Zero is a special value that signifies no field identifier. All non-zero field identifiers must be unique within each message.

## See Also

[Validity of Data Extracted From Message Fields](#)

[Field Names and Field Identifiers](#)

# Get Nested Message

*Convenience Method*

## Declaration

```
TibrvStatus getMsg(  
    const char*   fieldName,  
    TibrvMsg&     subMessage,  
    tibrv_u16     fieldId=0);
```

## Purpose

Get the value of a field as a Rendezvous message.

## Remarks

Since it is not possible to convert any other datatype to a message, the field must already contain a message. Otherwise, the method returns [TIBRV\\_CONVERSION\\_FAILED](#).

Pointer data extracted from the field remain valid until the message is destroyed; that is, even removing the field or updating the field's value does *not* invalidate pointer data.

After extracting a submessage, a program can detach it. A detached submessage remains valid and unchanged even after the parent message is destroyed. The program *must* explicitly destroy the detached submessage.

This method produces values that are *modifiable snapshots* of the field data. Programs may modify the resulting submessage by adding, removing or updating fields. However, these modifications do not change the field in the original parent message; instead, they force Rendezvous software to make a copy of the field (see [Rendezvous Protects the Message from Changes to Submessage Snapshots](#)).

Parameter	Description
fieldName	Get a field with this name.
subMessage	The program supplies a variable in this parameter, and

Parameter	Description
	the method stores the submessage in that variable.
fieldId	Get the field with this identifier. Zero is a special value that signifies no field identifier. All non-zero field identifiers must be unique within each message.

## See Also

[Validity of Data Extracted From Message Fields](#)

[Field Names and Field Identifiers](#)

# Get String

*Convenience Method*

## Declaration

```
TibrvStatus getString(
    const char*  fieldName,
    char*&      value,
    tibrv_u16    fieldId=0);
```

## Purpose

Get the value of a field as a character string.

## Remarks

This convenience method retrieves a field and extracts its data, automatically converting it to a string.

Programs can use this method to obtain a printable representation of field data. For most datatypes, this method formats the full value of the field to the output string; these types are exceptions:

`TIBRVMSG_OPAQUE`      This method abbreviates the value of an opaque field; for example, [472 opaque bytes].

`TIBRVMSG_XML`      This method abbreviates the value of an XML field; for example, [XML document: 472 bytes].

The size measures *uncompressed* data.

Pointer data extracted from the field remain valid until the message is destroyed; that is, even removing the field or updating the field's value does *not* invalidate pointer data.

This method produces values that are *read-only snapshots* of the field data (see [Pointer Snapshot](#)). Programs must not modify the value string.



Parameter	Description
fieldName	Get a field with this name.
value	The program supplies a variable in this parameter, and the method stores a <i>pointer</i> to the field value in that variable.
fieldId	Get the field with this identifier. Zero is a special value that signifies no field identifier. All non-zero field identifiers must be unique within each message.

## See Also

[Validity of Data Extracted From Message Fields](#)

[Field Names and Field Identifiers](#)

[Datatype Conversion](#)

# Get Opaque Byte Sequence

*Convenience Method*

## Declaration

```
TibrvStatus getOpaque(  
    const char*  fieldName,  
    void*&      value,  
    tibrv_u32&   length,  
    tibrv_u16    fieldId=0);
```

## Purpose

Get the value of a field as an opaque byte sequence.

## Remarks

This convenience method retrieves a field and extracts its data.

Since it is not possible to convert any other datatype to an opaque byte sequence, the field must already contain an opaque byte sequence. Otherwise, the method returns [TIBRV\\_CONVERSION\\_FAILED](#).

Pointer data extracted from the field remain valid until the message is destroyed; that is, even removing the field or updating the field's value does *not* invalidate pointer data.

This method produces values that are *read-only snapshots* of the field data (see [Pointer Snapshot](#)). Programs must not modify the value sequence.

Parameter	Description
fieldName	Get a field with this name.
length	The program supplies a variable in this parameter, and the method stores the length of the opaque byte sequence in that variable.

Parameter	Description
value	The program supplies a variable in this parameter, and the method stores a <i>pointer</i> to the field value in that variable.
fieldId	Get the field with this identifier. Zero is a special value that signifies no field identifier. All non-zero field identifiers must be unique within each message.

## See Also

[Validity of Data Extracted From Message Fields](#)

[Field Names and Field Identifiers](#)

# Get XML Byte Sequence

*Convenience Method*

## Declaration

```
TibrvStatus getXml(  
    const char*  fieldName,  
    void*&      value,  
    tibrv_u32&   length,  
    tibrv_u16    fieldId=0);
```

## Purpose

Get the value of a field as an XML byte sequence.

## Remarks

This convenience method retrieves a field and extracts its data.

XML data is a byte sequence. Adding a field of type [TIBRVMSG\\_XML](#) compresses the bytes. Extracting data from the field uncompresses it to obtain the original byte sequence.

Since it is not possible to convert any other datatype to an XML byte sequence, the field must already contain an XML byte sequence. Otherwise, the method returns [TIBRV\\_CONVERSION\\_FAILED](#).

Pointer data extracted from the field remain valid until the message is destroyed; that is, even removing the field or updating the field's value does *not* invalidate pointer data.

This method produces values that are *read-only snapshots* of the field data (see [Pointer Snapshot](#)). Programs must not modify the value sequence.

Parameter	Description
fieldName	Get a field with this name.
length	The program supplies a variable in this parameter, and the method stores the length of the XML byte sequence

Parameter	Description
	in that variable.
value	The program supplies a variable in this parameter, and the method stores a <i>pointer</i> to the field value in that variable.
fieldId	Get the field with this identifier. Zero is a special value that signifies no field identifier. All non-zero field identifiers must be unique within each message.

## See Also

[Validity of Data Extracted From Message Fields](#)

[Field Names and Field Identifiers](#)

# Get DateTime

*Convenience Method*

## Declaration

```
TibrvStatus getDateTIme(  
    const char*    fieldName,  
    TibrvMsgDateTime& value,  
    tibrv_u16      fieldId=0);
```

## Purpose

Get the value of a field as an datetime value.

## Remarks

This convenience method retrieves a field and extracts its data.

Since it is not possible to convert any other datatype to a datetime value, the field must already contain a datetime. Otherwise, the method returns [TIBRV\\_CONVERSION\\_FAILED](#).

Pointer data extracted from the field remain valid until the message is destroyed; that is, even removing the field or updating the field's value does *not* invalidate pointer data.

This method produces values that are *read-only snapshots* of the field data (see [Pointer Snapshot](#)). Programs must not modify the datetime value.

Parameter	Description
fieldName	Get a field with this name.
value	The program supplies a variable in this parameter, and the method stores the field value in that variable.
fieldId	Get the field with this identifier. Zero is a special value that signifies no field identifier. All non-zero field identifiers must be unique within each message.

## Example

This example code extracts a datetime value from a message field, and converts it to a `time_t` value. Programs can adapt this code as appropriate. (For corresponding code to convert a `time_t` value to a datetime value, and add the datetime to a message field, see the example at [Add DateTime](#).)

```
#include <limits.h>
TibrvStatus getAsTimeT(
    TibrvMsg& msg,
    const char* field,
    time_t& value)
{
    TibrvMsgDateTime d;
    TibrvStatus error;
    error = msg.getDateTime(field,d);
    if (error != TIBRV_OK)
        return error;
    if (d.sec > INT_MAX || d.sec < INT_MIN)
        return TIBRV_CONVERSION_FAILED;
    value = (time_t)d.sec;
}
```

## See Also

[Validity of Data Extracted From Message Fields](#)

[Field Names and Field Identifiers](#)

# TibrvMsg::getFieldByIndex()

*Method*

## Declaration

```
TibrvStatus getFieldByIndex(  
    TibrvMsgField& field,  
    tibrv_u32      fieldIndex);
```

## Purpose

Get a field from a message by an index.

## Remarks

Programs can loop through all the fields of a message, to retrieve each field in turn using an integer index.

The method copies scalar data into the program's field object. Pointer data extracted from the field remain valid until the message is destroyed; that is, even removing the field or updating the field's value does *not* invalidate pointer data.

*Add*, *remove* and *update* calls can perturb the order of fields (which, in turn, affects the results when a program gets a field by index).

Parameter	Description
field	The program supplies a <a href="#">TibrvMsgField</a> object; the method overwrites the contents of the object, using information from the message field.
fieldIndex	Get the field with this index. Zero specifies the first field.

## See Also

[TibrvMsg::getField\(\)](#)



# TibrvMsg::getFieldInstance()

*Method*

## Declaration

```
TibrvStatus getFieldInstance(  
    const char*    fieldName,  
    TibrvMsgField& fieldAddr,  
    tibrv_u32      instance);
```

## Purpose

Get a specified instance of a field from a message.

## Remarks

When a message contains several field instances with the same field name, retrieve a specific instance by number (for example, get the *ith* field named foo). Programs can use this method in a loop that examines every field with a specified name.

The argument 1 denotes the first instance of the named field.

The method copies scalar data into the program's field object. Pointer data extracted from the field remain valid until the message is destroyed; that is, even removing the field or updating the field's value does *not* invalidate pointer data.

When the instance argument is greater than the actual number of instances of the field in the message, this method returns the status code `TIBRV_NOT_FOUND`.

## Release 5 Interaction

Rendezvous 5 (and earlier) did not support array datatypes. Some older programs circumvented this limitation by using several fields with the same name to simulate arrays. This work-around is no longer necessary, since release 6 (and later) supports array datatypes within message fields. The method `TibrvMsg::getFieldInstance()` ensures backward compatibility, so new programs can still receive and manipulate messages sent from older programs. Nonetheless, we encourage programmers to use array types as appropriate, and we discourage storing several fields with the same name in a message.

Parameter	Description
fieldName	Get an instance of the field with this name.  NULL specifies the empty string as the field name.
fieldAddr	The program supplies a <a href="#">TibrvMsgField</a> object; the method overwrites the contents of the object, using information from the message field.
instance	Get this instance of the specified field name. The argument 1 denotes the first instance of the named field.

## See Also

[TibrvMsg::getField\(\)](#)

# TibrvMsg::getEvent()

*Method*

## Declaration

```
TibrvEvent* getEvent();
```

## Purpose

Extract the event associated with a (dispatched) message object.

## Remarks

Dispatch associates the message with a listener event.

This call is valid only for an inbound message that has already been dispatched to a listener event. If the message is not associated with a listener event, then this method returns NULL.

## See Also

[TibrvEvent::getClosure\(\)](#)

[TibrvVectorCallback::onMsgs\(\)](#)

# TibrvMsg::getHandle()

*Method*

## Declaration

```
tibrvMsg getHandle() const;
```

## Purpose

Extract the C message handle from a C++ message object.

## See Also

[TibrvMsg\(\)](#)

# TibrvMsg::getNumFields()

*Method*

## Declaration

```
TibrvStatus getNumFields(tibrv_u32& numFields) const;
```

## Purpose

Extract the number of fields in a message.

## Remarks

This method counts the immediate fields of the message; it does not descend into submessages to count their fields recursively.

Parameter	Description
numFields	The program supplies a variable in this parameter; the method stores the number of fields in that variable.

# TibrvMsg::getReplySubject()

*Method*

## Declaration

```
TibrvStatus getReplySubject(const char*& replySubject);
```

## Purpose

Extract the reply subject from a message.

## Remarks

The reply subject string is part of a message's address information—it is *not* part of the message itself.

If the destination subject is not set, this method passes NULL in the return parameter.

Parameter	Description
replySubject	<p>The program supplies a variable in this parameter; the method stores the reply subject in that variable.</p> <p>The method makes a snapshot copy of the reply subject string, and supplies a pointer to that snapshot within message storage. The pointer remains valid as long as the message itself remains valid in the same location. The reply subject pointer becomes <i>invalid</i> if the program destroys the message, or returns from the data callback method that determines the scope of an inbound message. For more information, see <a href="#">Pointer Snapshot</a>, and <a href="#">Deleting Snapshot References</a>.</p>

## See Also

[TibrvMsg::setReplySubject\(\)](#)

## Supplementary Information for Messages in TIBCO Rendezvous Concepts

# TibrvMsg::getSendSubject()

*Method*

## Declaration

```
TibrvStatus getSendSubject(const char*& sendSubject);
```

## Purpose

Extract the subject from a message.

## Remarks

The subject string is part of a message's address information—it is *not* part of the message itself.

If the destination subject is not set, this method passes NULL in the return parameter.

Parameter	Description
sendSubject	<p>The program supplies a variable in this parameter; the method stores the send subject in that variable.</p> <p>The method makes a snapshot copy of the subject string, and supplies a pointer to that snapshot within message storage. The pointer remains valid as long as the message itself remains valid in the same location. The subject pointer becomes <i>invalid</i> if the program destroys the message, or returns from the data callback method that determines the scope of an inbound message. For more information, see <a href="#">Pointer Snapshot</a>, and <a href="#">Deleting Snapshot References</a>.</p>

## See Also

[TibrvMsg::setSendSubject\(\)](#)

Supplementary Information for Messages in TIBCO Rendezvous Concepts



# TibrvMsg::getStatus()

*Method*

## Declaration

```
TibrvStatus getStatus() const;
```

## Purpose

Extract the status code stored by the message constructor.

## Remarks

We recommend that programs test the status after calls to [TibrvMsg\(\)](#) that copy a [TibrvMsg](#), or construct a message from a byte array.

## See Also

[TibrvMsg](#)

[TibrvMsg\(\)](#)

[TibrvStatus](#)

# TibrvMsg::isDetached()

*Method*

## Declaration

```
tibrv_bool isDetached();
```

## Purpose

Determine the ownership of the message.

## Remarks

Ownership of a message belongs either to Rendezvous software, or to your program.

When Rendezvous software owns a message, the message is considered *attached*. Rendezvous software automatically destroys the message and reclaims its storage. For example, Rendezvous software creates an object representing an inbound message, and presents it as a parameter to a listener callback method; when the callback method returns, Rendezvous software destroys the message.

A program can explicitly *detach* an inbound message. Thereafter, the program owns the message, and assumes the responsibility to destroy it and reclaim its storage.

When a program creates a message object, the program owns that message, along with the responsibility to destroy it.

This method determines the ownership of a message. It returns TIBRV\_TRUE when the program owns the message; it returns TIBRV\_FALSE when Rendezvous software owns the message.

## See Also

[TibrvMsg::detach\(\)](#)

# TibrvMsg::markReferences()

*Method*

## Declaration

```
TibrvStatus markReferences();
```

## Purpose

Mark references in this message.

## Remarks

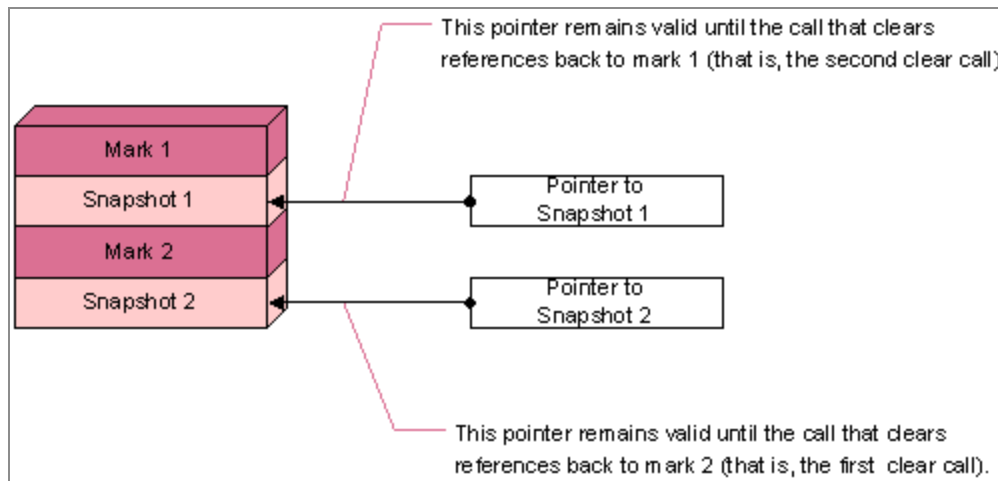
Extracting pointer data from a message field creates a snapshot of that data. The snapshot remains associated with the message until the program destroys the message. However, in *rare* situations snapshots can accumulate within a program, causing unbounded memory growth. This method gives programs explicit control over snapshot references; by clearing references, the program declares that it no longer needs the references that arise as side effects of calls that *get* a message field.

For example, consider a program fragment that repeatedly sends a message, modifying fields within a nested submessage before each send call. Each call to extract the nested message produces a snapshot reference. By surrounding the *get* operation with a *mark* and *clear* pair (with the *clear* call occurring at any time after the *get* call), the program releases the reference, which helps control memory usage.

```
void TimerCallback::onTimer(TibrvTimer* timer)
{
    TibrvMsg* msg = (TibrvMsg*)timer->getClosure();
    msg->markReferences();
    TibrvMsg submsg;
    msg->getMsg("foo",submsg);
    ...
    msg->clearReferences();
    msg->setSendSubject(some_subject);
    /* send a message */
    transport.send(*msg);
}
```

Every call to `TibrvMsg::markReferences()` must be paired with a call to `TibrvMsg::clearReferences()`. It is legal to mark references several times, as long as the program eventually clears all the marks. To understand this idea, it is helpful to think of get and mark as *pushdown* operations, and clear as a *pop* operation. [Mark and Clear References](#) illustrates that each clear call deletes snapshots back to the most recent mark.

Figure 5: Mark and Clear References



Unless a program explicitly marks and clears references, references persist until the message is destroyed or reset.

## See Also

[Validity of Data Extracted From Message Fields](#)

[Deleting Snapshot References](#)

[TibrvMsg::clearReferences\(\)](#)

# TibrvMsg::removeField()

*Method*

## Declaration

```
TibrvStatus removeField(  
    const char*   fieldName,  
    tibrv_u16     fieldId=0);
```

## Purpose

Remove a field from a message.

## Remarks

Pointer data (such as strings, arrays, submessages, XML data, or opaque byte sequences) previously extracted from the field remains valid even after removing the field from the message.

Parameter	Description
fieldName	Remove the field with this name.
fieldId	Remove the field with this identifier. Zero is a special value that signifies no field identifier.

## Field Search Algorithm

This method uses this algorithm to find and remove a field within a message, as specified by a field identifier and a field name.

### Procedure

1. If the identifier is zero, skip to step 3.

If the program supplied a *non-zero* field identifier, then search for the field with that identifier. If the search succeeds, remove the field.

On failure, continue to step 2.

2. If the identifier search (in step 1) fails, and the program supplied a non-NULL field name, then search for a field with that name.

If the program supplied NULL as the field name, return the status code [TIBRV\\_NOT\\_FOUND](#).

If the name search fails, return the status code [TIBRV\\_NOT\\_FOUND](#).

If the name search succeeds, but the actual identifier in the field is non-NULL (so it does not match the identifier supplied) then return the status code [TIBRV\\_ID\\_CONFLICT](#).

If the search succeeds, remove the field.

3. When the program supplied *zero* as the identifier, search for a field with the specified name—even if that name is NULL.

If the search succeeds, remove the field.

If the search fails, return the status code [TIBRV\\_NOT\\_FOUND](#).

If a message contains several fields with the same name, searching by name removes the first instance of the field with that name.

## See Also

[TibrvMsg::removeFieldInstance\(\)](#)

# TibrvMsg::removeFieldInstance()

*Method*

## Declaration

```
TibrvStatus removeFieldInstance(  
    const char*  fieldName,  
    tibrv_u32    instance);
```

## Purpose

Remove a specified instance of a field from a message.

## Remarks

When a message contains several field instances with the same field name, remove a specific instance by number (for example, remove the *ith* field named foo). Programs can use this method in a loop that examines every field with a specified name.

The argument 1 denotes the first instance of the named field.

If the specified instance does not exist, the method returns [TIBRV\\_NOT\\_FOUND](#).

Pointer data (such as strings, arrays, submessages, XML data, or opaque byte sequences) previously extracted from the field remains valid even after removing the field from the message.

Parameter	Description
fieldName	Remove the field with this name.
instance	Remove this instance of the field. The argument 1 specifies the first instance of the named field.

## See Also

[TibrvMsg::removeField\(\)](#)

# TibrvMsg::reset()

*Method*

## Declaration

```
TibrvStatus reset();
```

## Purpose

Clear a message, preparing it for re-use.

## Remarks

When this method returns, the message has no fields; it is like a newly created message. The message's address information is also reset.

Pointer data (such as strings, arrays, submessages, XML data, or opaque byte sequences) previously extracted from fields of the old message are invalid.

## See Also

[TibrvMsg\(\)](#)

[~TibrvMsg\(\)](#)



# TibrvMsg::setReplySubject()

*Method*

## Declaration

```
TibrvStatus setReplySubject(const char* replySubject);
```

## Purpose

Set the reply subject for a message.

## Remarks

A receiver can reply to an inbound message using its reply subject.

Rendezvous routing daemons modify subjects and reply subjects to enable transparent point-to-point communication across network boundaries. This modification does not apply to subject names stored in message data fields; we discourage storing point-to-point subject names in data fields.

Parameter	Description
replySubject	Use this string as the new reply subject, replacing any existing reply subject.  The method copies this string to the message.  The reply subject NULL removes the previous reply subject.  The empty string is <i>not</i> a legal subject name.

## See Also

[TibrvMsg::getReplySubject\(\)](#)

Supplementary Information for Messages in TIBCO Rendezvous Concepts

# TibrvMsg::setSendSubject()

*Method*

## Declaration

```
TibrvStatus setSendSubject(const char* sendSubject);
```

## Purpose

Set the subject for a message.

## Remarks

The subject of a message can describe its content, as well as its destination set.

Rendezvous routing daemons modify subjects and reply subjects to enable transparent point-to-point communication across network boundaries. This modification does not apply to subject names stored in message data fields; we discourage storing point-to-point subject names in data fields.

Parameter	Description
sendSubject	Use this string as the new subject, replacing any existing subject.  The method copies this string to the message.  The subject NULL removes the previous subject, leaving the message unsendable.  The empty string is <i>not</i> a legal subject name.

## See Also

[TibrvMsg::getSendSubject\(\)](#)

Supplementary Information for Messages in TIBCO Rendezvous Concepts

# TibrvMsg::updateField()

*Method*

## Declaration

```
TibrvStatus updateField(  
    const TibrvMsgField& field);
```

## Purpose

Update a field within a message.

For most programs, we recommend type-specific convenience methods instead of this generic method. However, translation engine programs can require generic `TibrvMsg::updateField()`, and would use it in conjunction with generic `TibrvMsg::getField()`. In this paradigm, modify the field returned from `TibrvMsg::getField()` by replacing its `field.value`, and supply it as the `field` argument to `TibrvMsg::updateField()`.

## Remarks

This method locates a field within the message by matching the name and identifier of `field`. Then it updates the message field using the `field` argument. (Notice that the program may not supply a field object with a different field name, field identifier, or datatype.)

If no existing field matches the specifications in the `field` argument, then this method adds the field to the message. Update convenience methods also add the field if it is not present.

The type of the existing field (within the message) and the type of the updating field argument must be identical; otherwise, the method returns the error status code `TIBRV_INVALID_TYPE`. However, when updating array or vector fields, the count (number of elements) can change.

Pointer data (such as strings, arrays, submessages, XML data, or opaque byte sequences) previously extracted from the field remain valid and unchanged until the message is destroyed; that is, even updating the field's value does *not* invalidate pointer data.

Parameter	Description
field	Update the existing message field using this field.

## Field Search Algorithm

This method, and related methods that *update* message fields, all use this algorithm to find and update a field within a message, as specified by a field identifier and a field name.

### Procedure

1. If the identifier is zero, skip to step 3.  
If the program supplied a *non-zero* field identifier, then search for the field with that identifier.  
If the search succeeds, then update that field.  
On failure, continue to step 2.
2. If the identifier search (in step 1) fails, and the program supplied a non-NULL field name, then search for a field with that name.  
If the program supplied NULL as the field name, return the status code [TIBRV\\_NOT\\_FOUND](#).  
If the name search succeeds, but the actual identifier in the field is non-NULL (so it does not match the identifier supplied) then return the status code [TIBRV\\_ID\\_CONFLICT](#).  
If the search fails, *add* the field as specified (with name and identifier).
3. When the program supplied *zero* as the identifier, search for a field with the specified name—even if that name is NULL.  
If the search fails, *add* the field as specified (with name and identifier).

If a message contains several fields with the same name, searching by name finds the first instance of the field with that name.

## Reserved Field Name



### Warning

The field name `_data_` is reserved. Programs may not add fields with this name.

More generally, all fields that begin with the underbar character are reserved.

## Field Name Length

The constant `TIBRVMSG_FIELDNAME_MAX` determines the longest possible field name.

## Convenience Methods

When the datatype of a field is determined during execution, use this general method. When you can determine the datatype of a field before compile-time, we recommend using type-specific convenience methods instead of this general method. Type-specific methods yield these advantages when updating fields:

- Code readability.
- Type checking.
- Automatic type conversion.

These categories of type-specific convenience methods find a field and update its data:

- [Update Scalar](#).
- [Update Array](#).
- [Update Nested Message](#).
- [Update String](#).
- [Update Opaque Byte Sequence](#)
- [Update DateTime](#)

## Nested Message

When the new value is a message object, this method uses only the data portion of the nested message (data); it does not include any address information or certified delivery information.

# Update Scalar

## Convenience Methods

### Declaration

```
TibrvStatus update_scalar_type(  
    const char*    fieldName,  
    tibrv_scalar_type value,  
    tibrv_u16      fieldId=0);
```

### Purpose

Update a field containing a scalar value.

### Remarks

Each convenience method in this family locates a field (by name or identifier) and updates its data.

The type of the existing field (within the message) and the type of the updating value must match.

Method Name	Value Type	Type Description
updateBool	tibrv_bool	boolean scalar
updateI8	tibrv_i8	8-bit integer
updateU8	tibrv_u8	8-bit unsigned integer
updateI16	tibrv_i16	16-bit integer
updateU16	tibrv_u16	16-bit unsigned integer
updateI32	tibrv_i32	32-bit integer

Method Name	Value Type	Type Description
updateU32	tibrv_u32	32-bit unsigned integer
updateI64	tibrv_i64	64-bit integer
updateU64	tibrv_u64	64-bit unsigned integer
updateF32	tibrv_f32	32-bit floating point
updateF64	tibrv_f64	64-bit floating point
updateIPAddr32	tibrv_ipaddr32	4-byte IP address
updateIPPort16	tibrv_ipport16	2-byte IP port

Parameter	Description
fieldName	Update a field with this name.
value	Update the message field to this value (which may be a literal or stored in a variable).  The method copies the value into the new message field.
fieldId	Update the field with this identifier. Zero is a special value that signifies no field identifier. It is illegal to add a field that has both a NULL field name, and a non-zero field identifier.

## See Also

[Field Names and Field Identifiers](#)

# Update Array

## Convenience Methods

### Declaration

```
TibrvStatus updateelement_typeArray(  
    const char*      fieldName,  
    const tibrv_scalar_type value,  
    tibrv_u32        numElements,  
    tibrv_u16        fieldId=0);
```

### Purpose

Update a field containing an array value.

### Remarks

Each convenience method in this family locates a field (by name or identifier) and updates its data.

The type of the existing field (within the message) and the type of the updating value must match. The number of elements can change.

Pointer data previously extracted from the field remain valid and unchanged until the message is destroyed; that is, even updating the field's value does *not* invalidate pointer data. (See [Pointer Snapshot](#).)

Method Name	Element Type	Type Description
updateI8Array	tibrv_i8	8-bit integer array
updateU8Array	tibrv_u8	8-bit unsigned integer array
updateI16Array	tibrv_i16	16-bit integer array
updateU16Array	tibrv_u16	16-bit unsigned integer array



Method Name	Element Type	Type Description
updateI32Array	tibrv_i32	32-bit integer array
updateU32Array	tibrv_u32	32-bit unsigned integer array
updateI64Array	tibrv_i64	64-bit integer array
updateU64Array	tibrv_u64	64-bit unsigned integer array
updateF32Array	tibrv_f32	32-bit floating point array
updateF64Array	tibrv_f64	64-bit floating point array

Parameter	Description
fieldName	Update a field with this name.
value	Update the message field to this array value.  The method <i>copies</i> the new array into the existing field.
numElements	When updating an array type, the program supplies the count of array elements in this parameter.
fieldId	Update the field with this identifier. Zero is a special value that signifies no field identifier. It is illegal to add a field that has both a NULL field name, and a non-zero field identifier.

## See Also

[Field Names and Field Identifiers](#)

# Update Nested Message

*Convenience Method*

## Declaration

```
TibrvStatus updateMsg(  
    const char*    fieldName,  
    const TibrvMsg& value,  
    tibrv_u16      fieldId=0);
```

## Purpose

Update a field containing a nested submessage.

## Remarks

This convenience method locates a field (by name or identifier) and updates its data.

The type of the existing field (within the message) and the type of the updating value must match. The message size (that is, its length in bytes) can change.

Pointer data previously extracted from the field remain valid and unchanged until the message is destroyed; that is, even updating the field's value does *not* invalidate pointer data. (See [Rendezvous Protects the Message from Changes to Submessage Snapshots.](#))

This method uses only the data portion of the nested message (value); it does not include any address information or certified delivery information.

Parameter	Description
fieldName	Update a field with this name.
value	Update the message field to this value. The method <i>copies</i> the new value into the field.
fieldId	Update the field with this identifier. Zero is a special

Parameter	Description
	value that signifies no field identifier. It is illegal to add a field that has both a NULL field name, and a non-zero field identifier.

## See Also

[Field Names and Field Identifiers](#)

# Update String

*Convenience Method*

## Declaration

```
TibrvStatus updateString(  
    const char*   fieldName,  
    const char*   value,  
    tibrv_u16     fieldId=0);
```

## Purpose

Update a field containing a character string.

## Remarks

This convenience method locates a field (by name or identifier) and updates its data.

The type of the existing field (within the message) and the type of the updating value must match. The length of the string can change.

Pointer data previously extracted from the field remain valid and unchanged until the message is destroyed; that is, even updating the field's value does *not* invalidate pointer data. (See [Pointer Snapshot](#).)

Parameter	Description
fieldName	Update a field with this name.
value	Update the message field to this value (which may be a literal or stored in a variable).  The method <i>copies</i> the new value into the existing field.
fieldId	Update the field with this identifier. Zero is a special value that signifies no field identifier. It is illegal to add a field that has both a NULL field name, and a non-zero field identifier.

## See Also

[Field Names and Field Identifiers](#)

# Update Opaque Byte Sequence

*Convenience Method*

## Declaration

```
TibrvStatus updateOpaque(  
    const char*   fieldName,  
    const void*   value,  
    tibrv_u32     size,  
    tibrv_u16     fieldId=0);
```

## Purpose

Update a field containing an opaque byte sequence.

## Remarks

This convenience method locates a field (by name or identifier) and updates its data.

The type of the existing field (within the message) and the type of the updating value must match. The size can change.

Pointer data previously extracted from the field remain valid and unchanged until the message is destroyed; that is, even updating the field's value does *not* invalidate pointer data. (See [Pointer Snapshot](#).)

Parameter	Description
fieldName	Update a field with this name.
value	Update the message field to this value.  The method <i>copies</i> the new value into the existing field.
size	The program supplies the size of the new data in this parameter.

Parameter	Description
fieldId	Update the field with this identifier. Zero is a special value that signifies no field identifier. It is illegal to add a field that has both a NULL field name, and a non-zero field identifier.

---

## See Also

[Field Names and Field Identifiers](#)

# Update XML Byte Sequence

*Convenience Method*

## Declaration

```
TibrvStatus updateXml(  
    const char*   fieldName,  
    const void*   value,  
    tibrv_u32     size,  
    tibrv_u16     fieldId=0);
```

## Purpose

Update a field containing an XML byte sequence.

## Remarks

This convenience method locates a field (by name or identifier) and updates its data.

The type of the existing field (within the message) and the type of the updating value must match. The size can change.

Pointer data previously extracted from the field remain valid and unchanged until the message is destroyed; that is, even updating the field's value does *not* invalidate pointer data. (See [Pointer Snapshot](#).)

XML data is a byte sequence. Adding (or updating) a field of type [TIBRVMSG\\_XML](#) compresses the bytes. Extracting data from the field uncompresses it to obtain the original byte sequence.

Parameter	Description
fieldName	Update a field with this name.
value	Update the message field to this value. The method <i>copies</i> the new value into the existing field.



Parameter	Description
size	The program supplies the size of the new data in this parameter.
fieldId	Update the field with this identifier. Zero is a special value that signifies no field identifier. It is illegal to add a field that has both a NULL field name, and a non-zero field identifier.

## See Also

[Field Names and Field Identifiers](#)

# Update DateTime

*Convenience Method*

## Declaration

```
TibrvStatus updateDateTime(  
    const char*      fieldName,  
    const TibrvMsgDateTime& value,  
    tibrv_u16        fieldId=0);
```

## Purpose

Update a field containing a datetime value.

## Remarks

This convenience method locates a field (by name or identifier) and updates its data.

The type of the existing field (within the message) and the type of the updating value must match.

Parameter	Description
fieldName	Update a field with this name.
value	Update the message field to this value. The method <i>copies</i> the new value into the existing field.
fieldId	Update the field with this identifier. Zero is a special value that signifies no field identifier. It is illegal to add a field that has both a NULL field name, and a non-zero field identifier.

## See Also

[Field Names and Field Identifiers](#)

# TibrvMsgField

*Class*

## Declaration

```
class TibrvMsgField : public tibrvMsgField
```

## Purpose

Represent a message field.

## Remarks

Because the field object is identical to the C field struct, programs can use the struct's accessors to get and set the field's name, data, and other members.

Although a formal destructor is not needed, C++ declares a default destructor, which has no effect.

Method	Description
<a href="#">TibrvMsgField()</a>	Create a message field object.
<a href="#">TibrvMsgField::getCount()</a>	Get the number of elements in a field.
<a href="#">TibrvMsgField::getData()</a>	Get the data value of a field.
<a href="#">TibrvMsgField::getId()</a>	Get the identifier of a field.
<a href="#">TibrvMsgField::getName()</a>	Get the name of a field.
<a href="#">TibrvMsgField::getSize()</a>	Get the size of a field.
<a href="#">TibrvMsgField::getType()</a>	Get the datatype of a field.

Member	Type	Description
name	char*	<p>The field's name. NULL signifies the empty string as the field name.</p> <p>Field name strings use the ISO 8859-1 character encoding.</p>
size	<a href="#">tibrv_u32</a>	<p>The size of the field's data (in bytes).</p> <p>For array types, size reflects the size (in bytes) of one array element. For <b>TIBRVMSG_STRING</b>, size reflects the number of bytes in the string (including the NULL terminator, if present). For <b>TIBRVMSG_OPAQUE</b> and <b>TIBRVMSG_XML</b>, size reflects the number of bytes of opaque data.</p>
count	<a href="#">tibrv_u32</a>	<p>The number of values in an array field. For array types, count is the number of array elements.</p> <p>For example, when a field contains a array of ten 32-bit integers, its size is 4, and its count is 10.</p> <p>(For scalar types, strings, XML data, and opaque byte sequences, count is 1. That is, the field contains one string—not an array of characters; one opaque value—not an array of bytes.)</p>
data	<a href="#">tibrvLocalData</a>	The field's data value.
id	<a href="#">tibrv_u16</a>	The field's identifier. Identifiers are optional, but must be unique within each message.
type	<a href="#">tibrv_u8</a>	A Rendezvous datatype token denoting the type of the field's data.

## See Also

[TibrvMsg::addField\(\)](#)

[TibrvMsg::getField\(\)](#)

[TibrvMsg::updateField\(\)](#)

[tibrvMsgField](#) in TIBCO Rendezvous C Reference

# TibrvMsgField()

*Constructor*

## Declaration

```
TibrvMsgField()  
TibrvMsgField(const TibrvMsgField& field)  
TibrvMsgField(const tibrvMsgField& cField)
```

## Purpose

Create a message field object.

## Remarks

Because the field object is identical to the C field struct, programs can use the struct's accessors to get and set the field's name, data, and other members.

Parameter	Description
field	Create an independent copy of this <a href="#">TibrvMsgField</a> object.
cField	Create an independent copy of this C field struct in a new <a href="#">TibrvMsgField</a> object.

## See Also

[TibrvMsg::addField\(\)](#)

[TibrvMsg::getField\(\)](#)

[TibrvMsg::updateField\(\)](#)

# TibrvMsgField::getCount()

*Method*

## Declaration

```
tibrv_u32 getCount() const;
```

## Purpose

Get the number of elements in a field.

## Remarks

For array types, count is the number of array elements.

For example, when a field contains a array of ten 32-bit integers, its size is 4, and its count is 10.

(For scalar types, strings, opaque byte sequences and XML data, the count is 1. That is, the field contains one string—not an array of characters; one opaque value—not an array of bytes.)

This method returns the value of the member `field.count`.

# TibrvMsgField::getData()

*Method*

## Declaration

```
tibrvLocalData getData() const;
```

## Purpose

Get the data value of a field.

## Remarks

This method returns the value of the member `field.data`.



### Warning

Do not use this method to access opaque data that requires memory alignment; the C struct `tibrvLocalData` does not necessarily preserve alignment. Instead, see [Add Opaque Byte Sequence](#), [Get Opaque Byte Sequence](#), or [Update Opaque Byte Sequence](#).

# TibrvMsgField::getId()

*Method*

## Declaration

```
tibrv_u16 getId() const;
```

## Purpose

Get the identifier of a field.

## Remarks

Identifiers are optional, but must be unique within each message.

This method returns the value of the member `field.id`.



# TibrvMsgField::getName()

*Method*

## Declaration

```
const char* getName() const;
```

## Purpose

Get the name of a field.

## Remarks

NULL signifies the empty string as the field name.

Field name strings use the ISO 8859-1 character encoding.

This method returns the value of the member `field.name`.

# TibrvMsgField::getSize()

*Method*

## Declaration

```
tibrv_u32 getSize() const;
```

## Purpose

Get the size of a field.

## Remarks

Return the size of the field's data (in bytes).

For array types, size reflects the size (in bytes) of one array element. For [TIBRVMSG\\_STRING](#), size reflects the number of bytes in the string (including the NULL terminator, if present). For [TIBRVMSG\\_OPAQUE](#) and [TIBRVMSG\\_XML](#), size reflects the number of data bytes.

This method returns the value of the member `field.size`.

# TibrvMsgField::getType()

*Method*

## Declaration

```
tibrv_u8 getType() const;
```

## Purpose

Get the datatype of a field.

## Remarks

Return the Rendezvous datatype token denoting the type of the field's data.

This method returns the value of the member `field.type`.

# TibrvMsgDateTime

*Class*

## Declaration

```
class TibrvDateTime : public tibrvMsgDateTime
```

## Purpose

Represent date and time.

## Remarks

Because the datetime object is identical to the C field struct, programs can use the struct's accessors to get and set the field's sec and nsec members.

Although a formal destructor is not needed, C++ declares a default destructor, which has no effect.

Method	Description
<a href="#">TibrvMsgDateTime()</a>	Create a Rendezvous datetime object.

Member	Type	Description
sec	tibrv_i64	Signed 64-bit integer representing seconds.
nsec	<a href="#">tibrv_u32</a>	<p>Unsigned 32-bit integer representing nanoseconds <i>after the seconds value</i>. This value is always non-negative, between zero and 999999999.</p> <p>It modifies the date in whole seconds by specifying the number of nanoseconds <i>after</i> that date. For example, the time 1/2 second</p>

Member	Type	Description
		before midnight of December 31, 1969 is -1 seconds plus 500,000,000 nanoseconds.

Operator	Description
==	<p>Test equality. Operands can be any combination of C++ <a href="#">TibrvMsgDateTime</a> objects and C <a href="#">tibrvMsgDateTime</a> objects.</p> <p>Return TIBRV_TRUE if the two objects represent the same time; TIBRV_FALSE otherwise.</p>
!=	<p>Test inequality. Operands can be any combination of C++ <a href="#">TibrvMsgDateTime</a> objects and C <a href="#">tibrvMsgDateTime</a> objects.</p> <p>Return TIBRV_FALSE if the two objects do <i>not</i> represent the same time; TIBRV_TRUE otherwise.</p>

## Representations

Rendezvous software represents time values in two ways—one within C and C++ programs, and a more compact wire format within messages. [Date and Time Representations](#) compares these two representations. In both representations, zero denotes the epoch, 12:00 midnight, January 1st, 1970. Range limits denote the extreme value on either side of that center. Bold type indicates the primary unit of measurement for each representation.

### Date and Time Representations

Representation	Details
Within C and C++ programs	<p>Seconds as a 64-bit signed integer, plus nanoseconds as a 32-bit unsigned integer.</p> <p>However, values are restricted to the range and granularity supported by Rendezvous wire format. Forcing larger or finer values into this representation produces an error.</p> <p>Two constants bracket the available (restricted) range of values within</p>

Representation	Details
	programs— TIBRVMSG_DATETIME_SEC_MAX and TIBRVMSG_DATETIME_SEC_MIN.
	range in years 292,471,208,677
	<b>range in seconds 9,223,372,036,854,775,807</b>
	restricted range in seconds 549,755,813,887
	restricted range in millisecs 549,755,813,887,000
Rendezvous wire format	Seconds as a 40-bit signed integer, plus microseconds as a 24-bit unsigned integer.
	range in years 17,432
	<b>range in seconds 549,755,813,887</b>
	range in milliseconds 549,755,813,887,000

## Converting DateTime to Strings

[TibrvMsg::convertToString\(\)](#) prints times in UTC format (also known as Zulu time or GMT). The ISO-8601 standard requires appending a Z character in this notation.

[TibrvMsg::convertToString\(\)](#) uses Common Era numbering for years. This system does not include a year zero. So for example, the time 1 second before 0001-01-01 00:00:00Z would print as -0001-12-31 23:59:59Z.

[TibrvMsg::convertToString\(\)](#) uses a proleptic Gregorian calendar. That is, when formatting dates earlier than the adoption of the Gregorian calendar, it projects the Gregorian calendar backward beyond its actual invention and adoption.

## See Also

[Add DateTime](#)

[TibrvMsg::convertToString\(\)](#)

[Get DateTime](#)

[Update DateTime](#)

[tibrvMsgDateTime in TIBCO Rendezvous C Reference](#)

# TibrvMsgDateTime()

*Constructor*

## Declaration

```
TibrvMsgDateTime()  
TibrvMsgDateTime(const TibrvMsgDateTime& dateTime)  
TibrvMsgDateTime(const tibrvMsgDateTime& cDateTime)
```

## Purpose

Create a Rendezvous datetime object.

## Remarks

With no arguments, this method creates an object representing the epoch (0 seconds, 0 nanoseconds).

Because the datetime object is identical to the C field struct, programs can use the struct's accessors to get and set the field's sec and nsec members.

Parameter	Description
dateTime	Create an independent copy of this <a href="#">TibrvMsgDateTime</a> object.
cDateTime	Create an independent copy of this C datetime struct in a new <a href="#">TibrvMsgDateTime</a> object.

## See Also

[TibrvMsgDateTime](#)

[Representations](#)



# Events and Queues

---

This section presents classes and methods associated with event interest and event processing.

# Event Overview

Programs can express interest in events. When an event occurs, it triggers a program callback method to process the event. Events wait in queues until programs dispatch them. Dispatching an event runs its callback method to process the event.

Event queues organize events awaiting dispatch. Programs dispatch events to run callback methods.

Queue groups add flexibility and fine-grained control to the event queue dispatch mechanism. Programs can create groups of queues and dispatch them according to their queue priorities.

# TibrvEvent

*Class*

## Declaration

```
class TibrvEvent
```

## Purpose

Event objects represent program interest in events, and event occurrences.

## Remarks

Programs create instances of event subclasses of [TibrvEvent](#), but not of this superclass.

Each call to a Rendezvous event create method results in a new event object, which represents your program's interest in a set of events. Rendezvous software uses the same event object to signal each occurrence of such an event.

Destroying an event object cancels the program's interest in that event. Destroying the queue or transport of an event automatically destroys the event as well.

Although the fault tolerance classes are technically events, they are sufficiently different from listeners and timers that they require separate description. See [Fault Tolerance](#).

Method	Description
<a href="#">TibrvEvent::destroy()</a>	<a href="#">Destroy an event, canceling interest.</a>
<a href="#">TibrvEvent::getClosure()</a>	<a href="#">Extract the closure data of an event object.</a>
<a href="#">TibrvEvent::getHandle()</a>	<a href="#">Return the C event handle of this C++ event object.</a>
<a href="#">TibrvEvent::getType()</a>	<a href="#">Return the C event type token of an event object.</a>

Method	Description
<a href="#">TibrvEvent::getQueue()</a>	Extract the queue of an event object.
<a href="#">TibrvEvent::isIOEvent()</a>	Test whether this event object is an I/O event.
<a href="#">TibrvEvent::isListener()</a>	Test whether this event object is a listener.
<a href="#">TibrvEvent::isTimer()</a>	Test whether this event object is a timer.
<a href="#">TibrvEvent::isValid()</a>	Test whether an event has been destroyed.
<a href="#">TibrvEvent::isVectorListener()</a>	Test whether this event object is a vector listener.

## Descendants

[TibrvListener](#)

[TibrvVectorListener](#)

[TibrvTimer](#)

[TIBRVFT\\_ACTIVATE](#)

[TibrvCmListener](#)

# TibrvEvent::destroy()

*Method*

## Declaration

```
virtual TibrvStatus destroy(  
    TibrvEventOnComplete* completeCB = NULL);
```

## Purpose

Destroy an event, canceling interest.

## Remarks

Destroying an event object cancels interest in it. Upon return from [TibrvEvent::destroy\(\)](#), the destroyed event is no longer dispatched. However, all active callback methods of this event continue to run and return normally, even though the event is invalid.

It is legal for an event callback method to destroy its own event argument.

Destroying event interest invalidates the event object; subsequent API calls involving the invalid event return error status, unless explicitly documented to the contrary.

This method also destroys the C event handle embedded in the C++ event object.

Although [TibrvEvent::destroy\(\)](#) prevents future dispatch calls from running the destroyed event's callback method, that callback method might be already running in one or more threads that dispatch events from the same queue. In each thread where the callback method is already in progress, that callback method does continue to run until complete. Rendezvous software ensures that the completion method runs when the last callback-in-progress has completed; for important details, see [TibrvEventOnComplete::onComplete\(\)](#).

Parameter	Description
completeCB	Rendezvous software runs the completion callback method immediately after all instances of the event's callback method have completed. If the event's callback

Parameter	Description
	method is not running when the event is destroyed, the destroy call runs it before returning.
	If this parameter is NULL, <a href="#">TibrvEvent::destroy()</a> does not trigger a completion method.

## See Also

[TibrvEvent::isValid\(\)](#)

[TibrvEventOnComplete](#)

[tibrvEvent\\_Destroy\(\)](#) in TIBCO Rendezvous C Reference

# TibrvEvent::getClosure()

*Method*

## Declaration

```
void* getClosure() const;
```

## Purpose

Extract the closure data of an event object.

## Remarks

If no closure data is set for the event object, this method returns NULL.

This method can extract the closure data even from invalid events.

# TibrvEvent::getHandle()

*Method*

## Declaration

```
tibrvEvent getHandle() const;
```

## Purpose

Return the C event handle of this C++ event object.

## Remarks

If the event is invalid, this method returns the constant TIBRV\_INVALID\_ID.

## See Also

tibrvEvent in TIBCO Rendezvous C Reference



# TibrvEvent::getType()

*Method*

## Declaration

```
TibrvStatus getType(  
    tibrvEventType &type) const;
```

## Purpose

Return the C event type token of an event object.

Parameter	Description
type	The program supplies a variable. This method stores the event type in that variable.

## See Also

tibrvEventType in TIBCO Rendezvous C Reference

# TibrvEvent::getQueue()

*Method*

## Declaration

```
TibrvQueue* getQueue() const;
```

## Purpose

Extract the queue of an event object.

## Remarks

If the event is invalid, this method returns NULL.

# TibrvEvent::isIOEvent()

*Method*

## Declaration

```
tibrv_bool isIOEvent() const;
```

## Purpose

Test whether this event object is an I/O event.

## Remarks

This method returns TIBRV\_TRUE when the event is an I/O event. Otherwise, it returns TIBRV\_FALSE.

## See Also

[TibrvListener](#)

[TibrvTimer](#)

[TIBRVFT\\_ACTIVATE](#)

[TibrvCmListener](#)

# TibrvEvent::isListener()

*Method*

## Declaration

```
tibrv_bool isListener() const;
```

## Purpose

Test whether this event object is a listener.

## Remarks

This method returns TIBRV\_TRUE when the event is a listener (including a certified delivery listener). Otherwise, it returns TIBRV\_FALSE.

## See Also

[TibrvListener](#)

[TibrvTimer](#)

[TIBRVFT\\_ACTIVATE](#)

[TibrvCmListener](#)

# TibrvEvent::isTimer()

*Method*

## Declaration

```
tibrv_bool isTimer() const;
```

## Purpose

Test whether this event object is a timer.

## Remarks

This method returns TIBRV\_TRUE when the event is a timer. Otherwise, it returns TIBRV\_FALSE.

## See Also

[TibrvListener](#)

[TibrvTimer](#)

[TIBRVFT\\_ACTIVATE](#)

[TibrvCmListener](#)

# TibrvEvent::isValid()

*Method*

## Declaration

```
tibrv_bool isValid() const;
```

## Purpose

Test whether an event has been destroyed.

## Remarks

This method returns TIBRV\_FALSE if it has been destroyed (using the destroy method); TIBRV\_TRUE otherwise.

Notice that [TibrvEvent::destroy\(\)](#) invalidates the event immediately, even though active callback methods may continue to run.

## See Also

[TibrvEvent::destroy\(\)](#)

# TibrvEvent::isVectorListener()

*Method*

## Declaration

```
tibrv_bool isVectorListener() const;
```

## Purpose

Test whether this event object is a vector listener.

## Remarks

This method returns TIBRV\_TRUE when the event is a vector listener. Otherwise, it returns TIBRV\_FALSE.

## See Also

[TibrvVectorListener](#)

# TibrvCallback

*Class*

## Declaration

```
class TibrvCallback
```

## Purpose

Superclass of event callback interface classes.

## Remarks

Programs can implement this interface to process events. Subclass interfaces process specific types of events.

Method	Description	Page
<a href="#">TibrvCallback::onEvent()</a>	Process events.	<a href="#">TibrvCallback::onEvent()</a>

## Descendants

[TibrvMsgCallback](#)

[TibrvVectorListener](#)

[TibrvCmMsgCallback](#)

[TibrvTimerCallback](#)

[TibrvIOCallback](#)



# TibrvCallback::onEvent()

*Method*

## Declaration

```
virtual void onEvent(  
    TibrvEvent* event,  
    TibrvMsg& msg)  
    = 0;
```

## Purpose

Process events.

## Remarks

Implement this method to process events.

Parameter	Description
event	This parameter receives the event.
msg	<p>When the event is an inbound message, this parameter receives the message object.</p> <p>For all other event types, this parameter receives a message object with no fields. The program must not use this empty message in any way.</p>

# TibrvEventOnComplete

*Class*

## Declaration

```
class TibrvEventOnComplete
```

## Purpose

Run program code after all callback methods of a destroyed event have completed.

## Remarks

Implement this interface to post-process destroyed events.

Method	Description
<a href="#">TibrvEventOnComplete::onComplete()</a>	A program can destroy an event object even when its callback method is running in one or more threads. Multi-threaded programs can define methods of this type to discover when all callback methods in progress have completed.

# TibrvEventOnComplete::onComplete()

Method

## Declaration

```
virtual void onComplete(  
    TibrvEvent* destroyedEvent) = 0;
```

## Purpose

A program can destroy an event object even when its callback method is running in one or more threads. Multi-threaded programs can define methods of this type to discover when all callback methods in progress have completed.

Parameter	Description
destroyedEvent	<p>This parameter receives the event object. This object is identical to the object that the program created to express event interest.</p> <p>However, by the time this method runs, the event is already destroyed; this method cannot use the event object in Rendezvous calls.</p>

## Remarks

This type of method is important in two situations:

- An event callback method calls [TibrvEvent::destroy\(\)](#) to destroy its event, and the program must do additional processing *after* the rest of the callback method has completed.
- Several threads dispatch an event (so the event callback method can be running in several threads) and the program must do additional processing after the callback method has completed *in all threads*.

Upon return from `TibrvEvent::destroy()`, the destroyed event's callback method can no longer begin to run. However, in each thread where the callback method is already in progress, that callback method does continue to run until complete.

The completion callback can still extract closure data from the event, even though the event is already destroyed.

`TibrvEvent::destroy()` accepts an argument of type `TibrvEventOnComplete`. Rendezvous software ensures that the completion method runs when the last callback-in-progress has completed.

## Timing and Context

This completion method can run in two situations:

- **Completion when Callback Methods are in Progress** illustrates a situation in which the program calls `TibrvEvent::destroy()` while callback methods of the destroyed event are in progress. When the last of those callback methods completes, Rendezvous software runs the completion method immediately, in the same thread as the callback method that completes last.
- **Completion when Callback Methods are Not in Progress** illustrates a situation in which the program calls `TibrvEvent::destroy()` when the destroyed event's callback method is not running in any thread. In this case, `TibrvEvent::destroy()` calls the completion method before returning.

Notice that in this situation, the completion method runs in the program context, instead of the usual context of a callback method. In rare instances, deadlock can occur, resulting from unintended interactions between mutex operations in the program context before the destroy call, and mutex operations in the program's completion method code.

To protect against this type of deadlock, programmers can use a straightforward thought-experiment as a preventive test. Expand the completion method code immediately after the call to `TibrvEvent::destroy()`—as it would run when the destroyed event's callback method is not running in any thread. Trace mutex locking activity within this context to determine whether the resulting code could violate established rules for proper use of mutex locks.

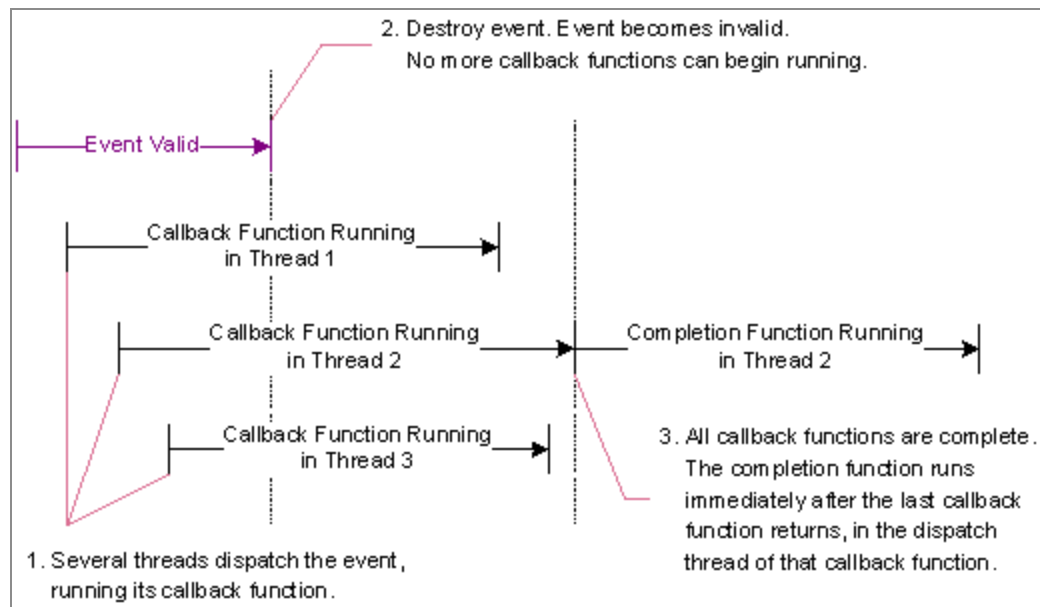
```
...  
mutex lock operations  
...  
myEvent::destroy()
```

*expand completion method code here, and check for violations of mutex rules*

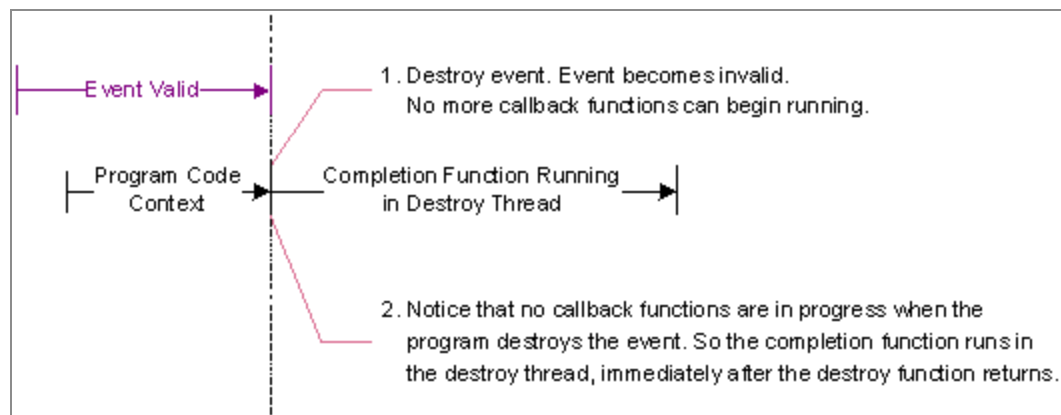
...

Potential violations and conflicts usually become apparent during this exercise. Remember, it is the programmer's responsibility to prevent deadlock.

*Figure 6: Completion when Callback Methods are in Progress*



*Figure 7: Completion when Callback Methods are Not in Progress*



## See Also

[TibrvIOEvent::create\(\)](#)

[TibrvListener::create\(\)](#)

[TibrvTimer::create\(\)](#)

[TibrvEvent::destroy\(\)](#)

# TibrvListener

*Class*

## Declaration

```
class TibrvListener : public TibrvEvent
{
    TibrvListener();           // Construct empty.
    virtual ~TibrvListener();  // Destroy and reclaim storage.
}
```

## Purpose

Listen for inbound messages.

## Remarks

A listener object continues listening for messages until the program destroys it.

The constructor creates a hollow object; the create method makes it operational.

The destructor calls the destroy method, unless the C object is already destroyed.

Destroying the queue or transport of a listener event automatically invalidates the listener as well.

Method	Description
<a href="#">TibrvListener::create()</a>	<a href="#">Listen for inbound messages.</a>
<a href="#">TibrvListener::getSubject()</a>	<a href="#">Extract the subject from a listener event object.</a>
<a href="#">TibrvListener::getTransport()</a>	<a href="#">Extract the transport from a listener event object.</a>

### Inherited Methods

[TibrvEvent::destroy\(\)](#)

[TibrvEvent::getClosure\(\)](#)

[TibrvEvent::getHandle\(\)](#)

[TibrvEvent::getType\(\)](#)

[TibrvEvent::getQueue\(\)](#)

[TibrvEvent::isValid\(\)](#)

[TibrvEvent::isListener\(\)](#)

[TibrvEvent::isTimer\(\)](#)

[TibrvEvent::isIOEvent\(\)](#)

## Activation and Dispatch

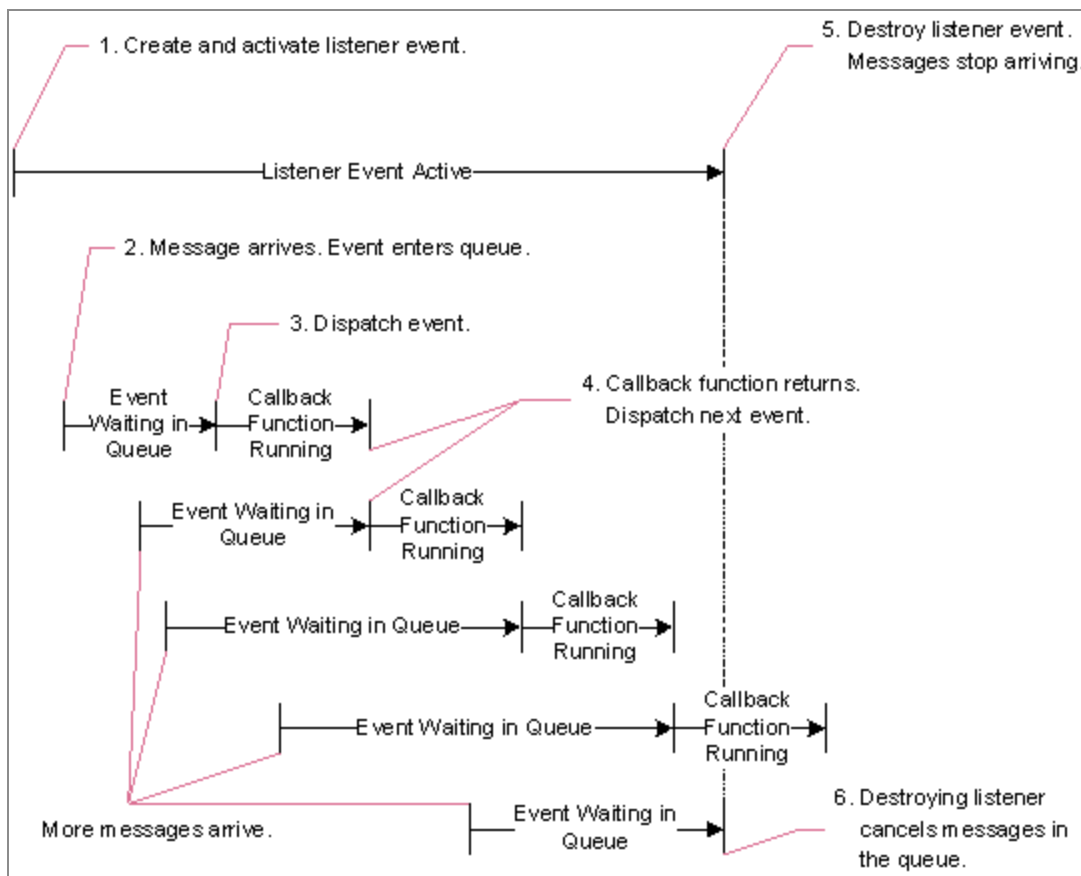
When an inbound message on the transport has a destination subject that matches the listener subject, then the message triggers the event.

The method [TibrvListener::create\(\)](#) creates a C listener event object, and *activates* the event—that is, it begins listening for all inbound messages with matching subjects. When a message arrives, Rendezvous software places the event object and message on its event queue. Dispatch removes the event object from the queue, and runs the callback method to process the message. (To stop receiving inbound messages on the subject, destroy the event object; this action cancels all messages already queued for the listener event; see [TibrvEvent::destroy\(\)](#).)

[Listener Activation and Dispatch](#) illustrates that Rendezvous software does *not* deactivate the listener when it places new message events on the queue (in contrast to I/O events, which are temporarily deactivated). Consequently, several messages can accumulate in the queue while the callback method is processing.



Figure 8: Listener Activation and Dispatch



When the callback method is I/O-bound, messages can arrive faster than the callback method can process them, and the queue can grow unacceptably long. In programs where a delay in processing messages is unacceptable, consider dispatching from several threads to process messages concurrently.

## Related Classes

[TibrvEvent](#)

[TibrvCmListener](#)

# TibrvListener::create()

*Method*

## Declaration

```
TibrvStatus create (  
    TibrvQueue* queue,  
    TibrvCallback* callback,  
    TibrvTransport* transport,  
    const char* subject,  
    const void* closure = NULL);
```

## Purpose

Listen for inbound messages.

## Remarks

This method creates a C listener and stores its handle in the C++ object.

Parameter	Description
queue	For each inbound message, place the event on this event queue.
callback	On dispatch, process the event with this callback object.  This object can be an instance of class <a href="#">TibrvMsgCallback</a> , or its superclass <a href="#">TibrvCallback</a> .
transport	Listen for inbound messages on this transport.
subject	Listen for inbound messages with subjects that match this specification. Wildcard subjects are permitted. The empty string is <i>not</i> a legal subject name.
closure	Store this closure data in the event object.

## Listening for Advisory Messages

Use this method to listen for advisory subjects. We recommend sending advisory message events to the default queue.

### Inbox Listener

To receive unicast (point-to-point) messages, listen to an inbox subject name. First call [TibrvTransport::createInbox\(\)](#) to create the unique inbox name; then call [TibrvListener::create\(\)](#) to begin listening. Remember that other programs have no information about an inbox until the listening program uses it as a reply subject in an outbound message. See also, Inbox Names in TIBCO Rendezvous Concepts.

### See Also

[TibrvListener::getSubject\(\)](#)

[TibrvTransport::createInbox\(\)](#)

# TibrvListener::getSubject()

*Method*

## Declaration

```
TibrvStatus getSubject(const char*& subject) const;
```

## Purpose

Extract the subject from a listener event object.

Parameter	Description
subject	The program supplies a variable. The method stores the subject of the listener event object in that variable.

## Remarks

Programs must not free nor modify the resulting subject string.

# TibrvListener::getTransport()

*Method*

## Declaration

```
TibrvTransport* getTransport() const;
```

## Purpose

Extract the transport from a listener event object.

# TibrvMsgCallback

*Class*

## Declaration

```
class TibrvMsgCallback : public TibrvCallback
```

## Purpose

Process inbound messages (listener events).

## Remarks

Implement this interface to process inbound messages.

Method	Description
<a href="#">TibrvMsgCallback::onMsg()</a>	Process inbound messages (listener events).

## Related Classes

[TibrvCallback](#)

[TibrvCmMsgCallback](#)

## See Also

[TibrvListener::create\(\)](#)

# TibrvMsgCallback::onMsg()

*Method*

## Declaration

```
virtual void onMsg(TibrvListener\* listener, TibrvMsg& msg) = 0;
```

## Purpose

Process inbound messages (listener events).

## Remarks

Implement this method to process inbound messages.

Parameter	Description
listener	This parameter receives the listener event.
msg	This parameter receives the inbound message.

## CM Label Information

The callback method for certified delivery messages can use certified delivery (CM) label information to discriminate among these situations:

- If [TibrvCmMsg::getSender\(\)](#) returns status code [TIBRV\\_NOT\\_FOUND](#), then the message uses the reliable protocol (that is, it was sent from an ordinary transport).
- If [TibrvCmMsg::getSender\(\)](#) returns a valid sender name, then the message uses the certified delivery protocol (that is, it is a labeled message, sent from a CM transport).

## See Also

[TibrvCmListener::create\(\)](#)

[TibrvCmMsg::getSender\(\)](#)

[TibrvCmMsg::getSequence\(\)](#)

[TibrvCmMsg::getTimeLimit\(\)](#)



# TibrvVectorListener

*Class*

## Declaration

```
class TibrvVectorListener : public TibrvEvent
{
    TibrvVectorListener();    // Construct empty.
    virtual ~TibrvVectorListener(); // Destroy and reclaim storage.
}
```

## Purpose

Listen for inbound messages, and receive them in a vector.

## Remarks

A vector listener object continues listening for messages until the program destroys it.

The constructor creates a hollow object; the create method makes it operational.

The destructor calls the destroy method, unless the C object is already destroyed.

Destroying the queue or transport of a vector listener event automatically invalidates the vector listener as well.

Method	Description
<a href="#">TibrvVectorListener::create()</a>	<a href="#">Listen for inbound messages, and receive them in a vector.</a>
<a href="#">TibrvVectorListener::getSubject()</a>	<a href="#">Extract the subject from a vector listener event object.</a>
<a href="#">TibrvVectorListener::getTransport()</a>	<a href="#">Extract the transport from a vector listener event object.</a>

### Inherited Methods

---

[TibrvEvent::destroy\(\)](#)

[TibrvEvent::getClosure\(\)](#)

[TibrvEvent::getHandle\(\)](#)

[TibrvEvent::getType\(\)](#)

[TibrvEvent::getQueue\(\)](#)

[TibrvEvent::isValid\(\)](#)

[TibrvEvent::isListener\(\)](#)

[TibrvEvent::isTimer\(\)](#)

[TibrvEvent::isIOEvent\(\)](#)

---

### Related Classes

[TibrvEvent](#)

[TibrvVectorCallback](#)

# TibrvVectorListener::create()

*Method*

## Declaration

```
TibrvStatus create (  
    TibrvQueue* queue,  
    TibrvVectorCallback* callback,  
    TibrvTransport* transport,  
    const char* subject,  
    const void* closure = NULL);
```

## Purpose

Listen for inbound messages, and receive them in a vector.

## Remarks

This method creates a C vector listener and stores its handle in the C++ object.

Parameter	Description
queue	Place each inbound message on this event queue.
callback	On dispatch, process the message vector with this callback object.  This object must be an instance of class <a href="#">TibrvVectorCallback</a> .
transport	Listen for inbound messages on this transport.
subject	Listen for inbound messages with subjects that match this specification. Wildcard subjects are permitted. The empty string is <i>not</i> a legal subject name.
closure	Store this closure data in the event object.

## Motivation

The standard way of receiving messages—one at a time—has the advantage of simplicity. However, if your application requires high throughput and low latency, consider receiving data messages in a vector instead. Vector listeners can boost performance for programs that receive a large number of messages by reducing the overhead associated with message dispatch. Applications that require high throughput (that is, many messages arriving rapidly) could benefit from vector listeners.



### Warning

We do *not* recommend vector listeners for command messages, administrative messages, advisory messages, nor any other out-of-band purpose.

## Activation and Dispatch

This method creates a vector listener event object, and *activates* the event—that is, it begins listening for all inbound messages with matching subjects. Dispatch removes a group of matching messages from the queue, and runs the callback method to process the message vector.

To stop receiving inbound messages on the subject, destroy the event object; this action cancels all messages already queued for the vector listener event.

## Interoperability

Vector listeners and ordinary listeners can listen on the same queue.

## Grouping Messages into Vectors

When several vector listeners use the same queue, the dispatcher groups messages into vectors with the following properties:

- The sequence of messages in a vector reflect consecutive arrival in the queue.
- All messages in a vector share the same callback object (though they need not match the same listener).

From these properties we can derive further inferences:

- If two vector listeners use the same callback object, then the dispatcher can group messages on their subjects into the same vector.

- If two messages are adjacent in the queue, but require different callback objects, then the dispatcher cannot group them into the same vector.

## Vector Listeners: Same Callback

Two vector listeners, F and P, listen on subjects FOO and PHU, respectively. Both F and P designate the same queue, Q1, and the same callback object, C1, to process their messages. In this situation, the dispatcher for Q1 can group messages on subjects FOO and PHU into the same vector (as long as the messages constitute a contiguous sequence within Q1).

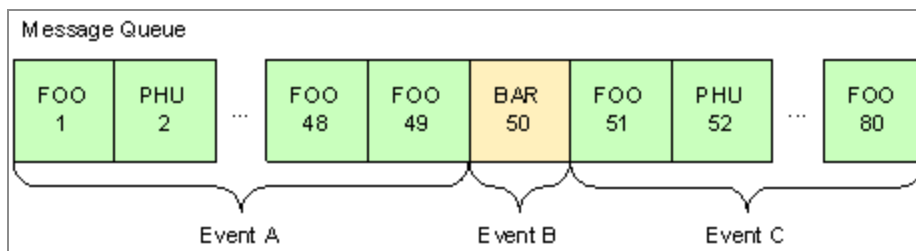
## Vector Listeners: Different Callbacks

Extend the previous example by adding a third vector listener, B, which listens on subject BAR. B designates the same queue, Q1, but uses a new callback object, C2 to process its messages. In this situation, the dispatcher for Q1 must group messages on subject BAR separately from messages on subjects FOO and PHU.

Suppose the Q1 contains 49 messages with subjects FOO or PHU, then 1 message with subject BAR, then 30 more messages with subjects FOO and PHU. [Grouping Messages into Vectors](#) shows this message queue. The dispatcher produces at least three separate events.

Because messages 49 and 50 require different callbacks, the dispatcher must close the vector of FOO and PHU messages at message 49, and start a new vector for message 50 with subject BAR. When the dispatcher encounters message 51 with subject FOO again, it closes the BAR vector after only one message, and starts a third vector for FOO.

*Figure 9: Grouping Messages into Vectors*



## Vector Listeners: Mixing Vector and Ordinary Listeners

Altering the previous example, suppose that B is an ordinary listener, instead of a vector listener. B necessarily specifies a different callback object than F and P (because ordinary listeners and vector listeners require different callback types with different signatures).

The behavior of the dispatcher remains the same as in [Vector Listeners: Different Callbacks](#).

## Dispatch Order vs. Processing Order

Messages dispatch in the order that they arrive in the queue. However, the order in which callbacks process messages can differ from dispatch order. The following examples illustrate this possibility by contrasting three scenarios.

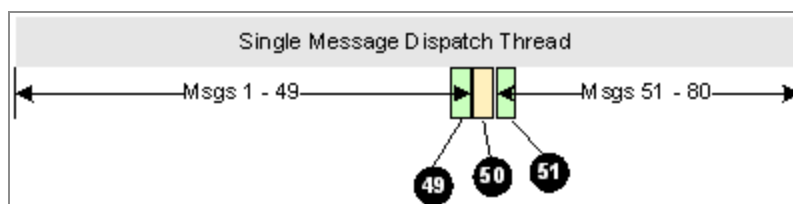
### Vector Listeners: Deliberately Processing Out of Order

The simplest callback (from the programmer's perspective) processes the messages within a vector in order (that is, the order that dispatcher moves them from the queue into the vector, which mirrors the order in which the messages arrive in the queue). Nonetheless you could program a callback that processes messages in reverse order, or any other order (though one would need a convincing reason to do so).

### Vector Listeners: Processing Message Vectors in a Single Dispatcher Thread

[Vector Listener Callbacks in a Single Dispatch Thread](#) shows a closer look at the situation of [Vector Listeners: Different Callbacks](#), in which several vector listeners all designate Q1 for their events. If a single thread dispatches Q1, then the callbacks are guaranteed to run in sequence. If the callbacks process messages in the order that they appear within the vectors, then message processing order is identical to dispatch order, which is also identical to arrival order. [Vector Listener Callbacks in a Single Dispatch Thread](#) shows this effect.

*Figure 10: Vector Listener Callbacks in a Single Dispatch Thread*

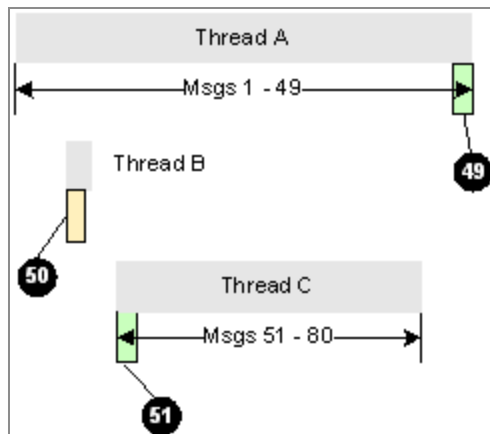


## Vector Listeners: Processing Message Vectors in Separate Threads

However, if several threads dispatch Q1 in parallel, then the callbacks can run concurrently. In this situation, message processing order could differ dramatically from arrival order.

[Vector Listener Callbacks in Multiple Dispatch Threads](#) shows this possibility.

*Figure 11: Vector Listener Callbacks in Multiple Dispatch Threads*



Although message number 49 dispatches (in event A) before message 50 (in event B), it is possible for the BAR callback (in thread B) to process message 50 before the FOO callback (in thread A) processes message 49. Furthermore, it is even possible for the FOO callback (in thread C) to process message 51 before the FOO callback (in thread A) processes message 49.



### Note

Before developing a program that processes inbound message vectors in several threads, consider carefully whether it is important (in the context of your application's semantics) to process messages in order of arrival.

## See Also

[TibrvVectorListener::getSubject\(\)](#)

# TibrvVectorListener::getSubject()

*Method*

## Declaration

```
TibrvStatus getSubject(const char*& subject) const;
```

## Purpose

Extract the subject from a vector listener event object.

Parameter	Description
subject	The program supplies a variable. The method stores the subject of the vector listener event object in that variable.

## Remarks

Programs must not free nor modify the resulting subject string.



# TibrvVectorListener::getTransport()

*Method*

## Declaration

```
TibrvTransport* getTransport() const;
```

## Purpose

Extract the transport from a vector listener event object.

# TibrvVectorCallback

*Class*

## Declaration

```
class TibrvVectorCallback
```

## Purpose

Process inbound message vectors (vector listener events).

## Remarks

Implement this interface to process inbound message vectors.

Method	Description	Page
<a href="#">TibrvVectorCallback::onMsgs()</a>	Process inbound message vectors (vector listener events).	<a href="#">TibrvVectorCallback::onMsgs()</a>

## Related Classes

[TibrvVectorListener](#)

## See Also

[TibrvVectorListener::create\(\)](#)

# TibrvVectorCallback::onMsgs()

*Method*

## Declaration

```
virtual void onMsgs(  
    TibrvMsg* messages[],  
    tibrv_u32 numMessages) = 0;
```

## Purpose

Process inbound message vectors (vector listener events).

Parameter	Description
messages	This parameter receives an array of pointers to inbound messages.
numMessages	This parameter receives the number of messages in the array.

## Remarks

Implement this method to process inbound message vectors.

In the simplest arrangement, your callback method processes the messages in the array. When the callback method returns, the Rendezvous library deallocates the array.

If your application requires a more complex processing arrangement, it can detach individual messages, and pass them to other threads for processing. (If your program detaches a message, then it must also explicitly destroy it.)

It is illegal to pass the message array to a different thread for processing, or to use it as dynamically-allocated storage.

Notice that in contrast to [TibrvMsgCallback::onMsg\(\)](#), this vector callback does not receive the listener event as an argument. You can use [TibrvMsg::getEvent\(\)](#) to get it from the individual message objects.

## See Also

[TibrvMsg::getEvent\(\)](#)

[TibrvVectorListener::create\(\)](#)

# TibrvTimer

*Class*

## Declaration

```
class TibrvTimer : public TibrvEvent
{
    TibrvTimer();           // Construct empty.
    virtual ~TibrvTimer();  // Destroy and reclaim storage
}
```

## Purpose

Timer event.

## Remarks

All timers are repeating timers. To simulate a once-only timer, code the callback method to destroy the timer.

The destructor calls the `destroy` method, unless the C object is already destroyed.

Destroying the queue of a timer automatically destroys the timer as well.

## Activation and Dispatch

The method `TibrvTimer::create()` creates a C timer event object, and *activates* the timer event—that is, it requests notification from the operating system when the timer's interval elapses. When the interval elapses, Rendezvous software places the event object on its event queue. Dispatch removes the event object from the queue, and runs the callback method to process the timer event. When the callback method begins, Rendezvous software automatically reactivates the event, using the same interval. On dispatch Rendezvous software also determines whether the next interval has already elapsed, and requeues the timer event if appropriate. (To stop the cycle, destroy the event object; see `TibrvEvent::destroy()`.)

Notice that time waiting in the event queue until dispatch can increase the effective interval of the timer. It is the programmer's responsibility to ensure timely dispatch of events.

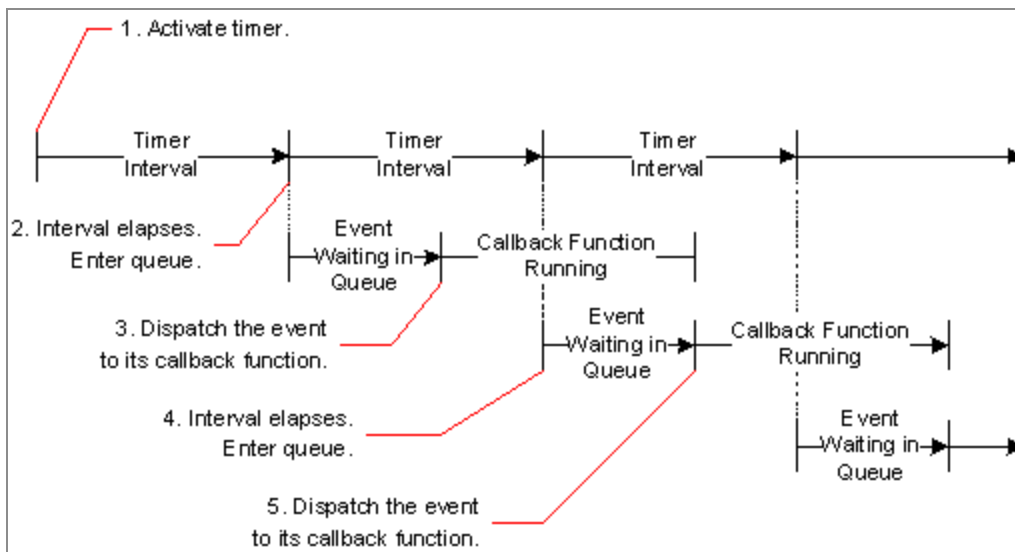
**Timer Activation and Dispatch** illustrates a sequence of timer intervals. The number of elapsed timer intervals directly determines the number of event callbacks.

At any moment the timer object appears on the event queue at most once—not several times as multiple copies. Nonetheless, Rendezvous software arranges for the appropriate number of timer event callbacks based the number of intervals that have elapsed since the timer became active or reset its interval.

Destroying or invalidating the timer object *immediately* halts the sequence of timer events. The timer object ceases to queue new events, and an event already in the queue does not result in a callback. (However, callback methods that are already running in other threads continue to completion.)

Resetting the timer interval *immediately* interrupts the sequence of timer events and begins a new sequence, counting the new interval from that moment. The reset operation is equivalent to destroying the timer and creating a new object in its place.

Figure 12: Timer Activation and Dispatch



## Timer Granularity

Express the timer interval (in seconds) as a 64-bit floating point number. This representation allows microsecond granularity for intervals for over 100 years. The actual granularity of intervals depends on hardware and operating system constraints.

## Zero as Interval

Many programmers traditionally implement user events as timers with interval zero. Instead, we recommend implementing user events as messages on the intra-process transport. For more information, see [Intra-Process Transport and User Events in TIBCO Rendezvous Concepts](#).

Method	Description
<a href="#">TibrvTimer::create()</a>	Start a timer.
<a href="#">TibrvTimer::getInterval()</a>	Extract the interval from a timer event object.
<a href="#">TibrvTimer::resetInterval()</a>	Reset the interval of a timer event object.

### Inherited Methods

[TibrvEvent::destroy\(\)](#)  
[TibrvEvent::getClosure\(\)](#)  
[TibrvEvent::getHandle\(\)](#)  
[TibrvEvent::getType\(\)](#)  
[TibrvEvent::getQueue\(\)](#)  
[TibrvEvent::isValid\(\)](#)  
[TibrvEvent::isListener\(\)](#)  
[TibrvEvent::isTimer\(\)](#)  
[TibrvEvent::isIOEvent\(\)](#)

## Related Classes

[TibrvEvent](#)

# TibrvTimer::create()

*Method*

## Declaration

```
TibrvStatus create (  
    TibrvQueue* queue,  
    TibrvCallback* callback,  
    tibrv_f64 interval,  
    const void* closure = NULL);
```

## Purpose

Start a timer.

Parameter	Description
queue	At each time interval, place the event on this event queue.
callback	On dispatch, process the event with this callback object.  This object can be an instance of class <a href="#">TibrvTimerCallback</a> , or its superclass <a href="#">TibrvCallback</a> .
interval	The timer triggers its callback method at this repeating interval (in seconds).
closure	Store this closure data in the event object.

## Remarks

This method creates a C timer, activates it, and stores its handle in the C++ object.



## Timer Granularity

Express the timer interval (in seconds) as a 64-bit floating point number. This representation allows microsecond granularity for intervals for over 100 years. The actual granularity of intervals depends on hardware and operating system constraints.

## Zero as Interval

Many programmers traditionally implement user events as timers with interval zero. Instead, we recommend implementing user events as messages on the intra-process transport. For more information, see Intra-Process Transport and User Events in TIBCO Rendezvous Concepts.

## See Also

Timer Event Semantics in TIBCO Rendezvous Concepts

[TibrvEvent::destroy\(\)](#)

[TibrvTimerCallback](#)

[TibrvTimerCallback::onTimer\(\)](#)

# TibrvTimer::getInterval()

*Method*

## Declaration

```
TibrvStatus getInterval (tibrv_f64& interval) const;
```

## Purpose

Extract the interval from a timer event object.

Parameter	Description
interval	The program supplies a variable. The method stores the interval of the timer event object in that variable.

## See Also

[TibrvTimer::resetInterval\(\)](#)

# TibrvTimer::resetInterval()

*Method*

## Declaration

```
TibrvStatus resetInterval(tibrv_f64 newInterval);
```

## Purpose

Reset the interval of a timer event object.

## Remarks

The timer begins counting the new interval immediately.

Parameter	Description
newInterval	The timer triggers its callback method at this new repeating interval (in seconds).

## Timer Granularity

Express the timer interval (in seconds) as a 64-bit floating point number. This representation allows microsecond granularity for intervals up to approximately 146 years. The actual granularity of intervals depends on hardware and operating system constraints.

## Limit of Effectiveness

This method can affect a timer only before or during its interval—but not after its interval has elapsed.

This method neither examines, changes nor removes an event that is already waiting in a queue for dispatch. If the next event for the timer object is already in the queue, then that event remains in the queue, representing the old interval. The change takes effect with the subsequent interval. (To circumvent this limitation, a program can destroy the old timer object and replace it with a new one.)

# TibrvTimerCallback

*Class*

## Declaration

```
class TibrvTimerCallback : public TibrvCallback
```

## Purpose

Process timer events.

## Remarks

Implement this interface to process timer events.

Method	Description	Page
<a href="#">TibrvTimerCallback::onTimer()</a>	Process timer events.	<a href="#">TibrvTimerCallback::onTimer()</a>

## Related Classes

[TibrvCallback](#)

## See Also

[TibrvTimer::create\(\)](#)

# TibrvTimerCallback::onTimer()

*Method*

## Declaration

```
virtual void onTimer(TibrvTimer* timer) = 0;
```

## Purpose

Process timer events.

## Remarks

Implement this method to process timer events.

Parameter	Description
timer	This parameter receives the timer event.

# TibrvIOEvent

*Class*

## Declaration

```
class TibrvIOEvent : public TibrvEvent
{
    TibrvIOEvent();           // Create empty.
    virtual ~TibrvIOEvent();  // Destroy and reclaim storage.
};
```

## Purpose

I/O event.

## Remarks

The constructor creates a hollow object; [TibrvIOEvent::create\(\)](#) makes it operational.

The destructor calls the destroy method, unless the C object is already destroyed.

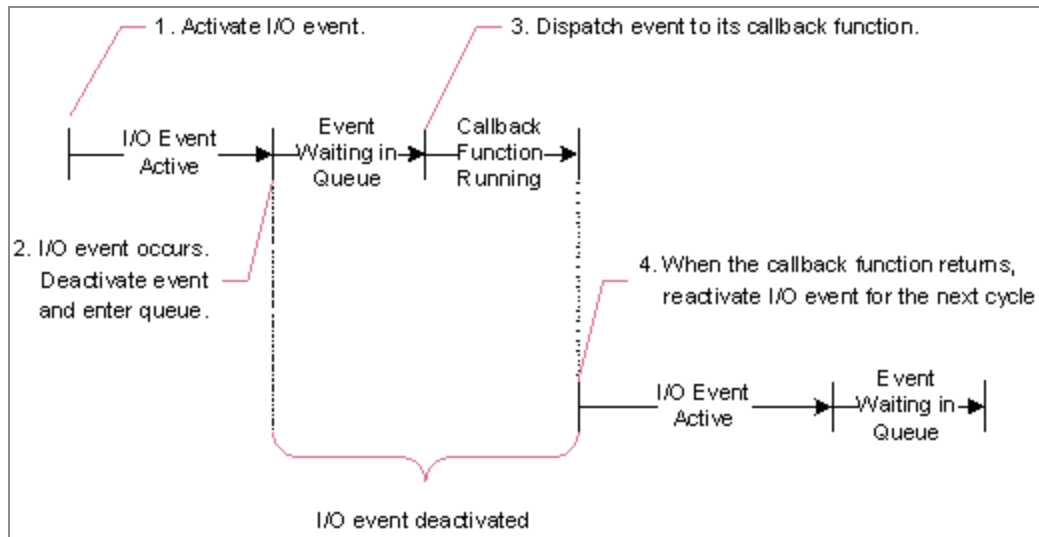
Destroying the queue of an I/O event automatically destroys the I/O event as well.

## Activation and Dispatch

The method [TibrvIOEvent::create\(\)](#) creates an event object that describes an I/O situation, and *activates* the event—that is, it requests notification from the operating system when that I/O situation occurs. When the situation occurs, Rendezvous software deactivates the event, and places the event object on its event queue. Dispatch removes the event object from the queue, and runs the callback method to process the event. When the callback method returns, Rendezvous software automatically reactivates the event. (To stop the cycle, destroy the event object; see [TibrvEvent::destroy\(\)](#).)

[I/O Event Activation and Dispatch](#) illustrates that Rendezvous software temporarily deactivates the I/O event from the time it enters the queue until its callback method returns. Consequently, an I/O object can cause at most one event at a time.

Figure 13: I/O Event Activation and Dispatch



## Semantics of I/O Events

The semantics of all I/O conditions depend on the underlying operating system and event manager. Rendezvous software does not change those semantics.

I/O events trigger when the operating system *reports* that an I/O condition on a monitored socket would succeed (that is, transfer at least one byte without blocking). Nonetheless, Rendezvous software cannot guarantee that a subsequent I/O call will not block.

Method	Description
<a href="#">TibrvIOEvent::create()</a>	Wait for specified I/O situations to occur.
<a href="#">TibrvIOEvent::getIOSource()</a>	Extract the source (socket ID) from an I/O event object.
<a href="#">TibrvIOEvent::getIOType()</a>	Extract the I/O type from an I/O event object.

### Inherited Methods

---

[TibrvEvent::destroy\(\)](#)

[TibrvEvent::getClosure\(\)](#)

[TibrvEvent::getHandle\(\)](#)

[TibrvEvent::getType\(\)](#)

[TibrvEvent::getQueue\(\)](#)

[TibrvEvent::isValid\(\)](#)

[TibrvEvent::isListener\(\)](#)

[TibrvEvent::isTimer\(\)](#)

[TibrvEvent::isIOEvent\(\)](#)

---

### Related Classes

[TibrvEvent](#)

### See Also

[TibrvIOCallback](#)



# TibrvIOEvent::create()

*Method*

## Declaration

```
TibrvStatus create (  
    TibrvQueue* queue,  
    TibrvCallback* callback,  
    tibrv_i32 socketId,  
    tibrvIOType ioType,  
    const void* closure = NULL);
```

## Purpose

Wait for specified I/O situations to occur.

Parameter	Description
queue	At each time interval, place the event on this event queue.
callback	On dispatch, process the event with this callback object.  This object can be an instance of class <a href="#">TibrvIOCallback</a> , or its superclass <a href="#">TibrvCallback</a> .
socketID	Wait for I/O occurrences on this socket.
ioType	Wait for I/O occurrences of this type.  See <a href="#">tibrvIOType</a> in TIBCO Rendezvous C Reference.
closure	Store this closure data in the event object.

## Remarks

This method creates a C I/O event and stores its handle in the C++ object.

All timers are repeating timers. To simulate a once-only timer, code the callback method to destroy the timer.

## See Also

[TibrvEvent::destroy\(\)](#)

[TibrvTimerCallback](#)

[TibrvTimerCallback::onTimer\(\)](#)

I/O Event Semantics in TIBCO Rendezvous Concepts

# TibrvIOEvent::getIOSource()

*Method*

## Declaration

```
TibrvStatus getIOSource (tibrv_i32& ioSource) const;
```

## Purpose

Extract the source (socket ID) from an I/O event object.

Parameter	Description
source	The program supplies a variable. The method stores the source of the I/O event object in that variable.

## See Also

[TibrvIOEvent::create\(\)](#)

# TibrvIOEvent::getIOType()

*Method*

## Declaration

```
TibrvStatus getIOType (tibrvIOType& ioType)) const;
```

## Purpose

Extract the I/O type from an I/O event object.

Parameter	Description
ioType	The program supplies a variable. The method stores the source of the I/O event object in that variable.  See tibrvIOType in TIBCO Rendezvous C Reference.

## See Also

[TibrvIOEvent::create\(\)](#)

# TibrvIOCallback

*Class*

## Declaration

```
class TibrvIOCallback : public TibrvCallback
```

## Purpose

Process I/O events.

## Remarks

Implement this interface to process I/O events.

Method	Description
<a href="#">TibrvIOCallback::onIOEvent()</a>	Process I/O events.

## Related Classes

[TibrvCallback](#)

## See Also

[TIBRVFT\\_ACTIVATE](#)

# TibrvIOCallback::onIOEvent()

*Method*

## Declaration

```
virtual void onIOEvent(TIBRVFT_ACTIVATE* ioEvent) = 0;
```

## Purpose

Process I/O events.

## Remarks

Implement this method to process I/O events.

Parameter	Description
ioEvent	This parameter receives the I/O event.

# TibrvDispatchable

*Class*

## Declaration

```
class TibrvDispatchable
```

## Purpose

Common interface for queues and queue groups.

## Remarks

Programs never instantiate this class.

Both [TibrvQueue](#) and [TibrvQueueGroup](#) implement this interface, so programs can call the common methods on objects of either class. For example, consider a dispatcher routine that receives an object of type [TibrvDispatchable](#); it can call the `dispatch()` method, without needing to determine whether the object is queue or a queue group.

Method	Description
<b>Life Cycle</b>	
<a href="#">TibrvDispatchable::destroy()</a>	Destroy a queue or queue group.
<a href="#">TibrvDispatchable::getDispatchable()</a>	Extract the C handle of a queue or queue group.
<a href="#">TibrvDispatchable::isValid()</a>	Test validity of a queue or queue group.
<b>Dispatch</b>	
<a href="#">TibrvDispatchable::dispatch()</a>	Dispatch an event; if no event is ready, block.

Method	Description
<a href="#">TibrvDispatchable::poll()</a>	Dispatch an event, if possible.
<a href="#">TibrvDispatchable::timedDispatch()</a>	Dispatch an event, but if no event is ready to dispatch, limit the time that this call blocks while waiting for an event.

## Descendants

[TibrvQueue](#)

[TibrvQueueGroup](#)

## See Also

[TibrvQueue](#)

[TibrvQueueGroup](#)



# TibrvDispatchable::destroy()

*Method*

## Declaration

```
virtual TibrvStatus destroy () = 0;
```

## Purpose

Destroy a queue or queue group.

## Remarks

When a queue is destroyed, events that remain in the queue are discarded.

When a queue group is destroyed, the individual queues in the group continue to exist, even though the group has been destroyed.

## See Also

[TibrvQueue::destroy\(\)](#)

[TibrvQueueGroup::destroy\(\)](#)

# TibrvDispatchable::dispatch()

*Method*

## Declaration

```
virtual TibrvStatus dispatch () = 0;
```

## Purpose

Dispatch an event; if no event is ready, block.

## Remarks

If an event is ready to dispatch, then this call dispatches it, and then returns. If no events are waiting, then this call blocks indefinitely while waiting for the object to receive an event.

Both [TibrvQueue](#) and [TibrvQueueGroup](#) implement this method.

## See Also

[TibrvDispatchable](#)

[TibrvQueue::dispatch\(\)](#)

[TibrvQueueGroup::dispatch\(\)](#)

# TibrvDispatchable::getDispatchable()

*Method*

## Declaration

```
virtual tibrvDispatchable getDispatchable() const = 0;
```

## Purpose

Extract the C handle of a queue or queue group.

## Remarks

If the event is invalid, this method returns the constant TIBRV\_INVALID\_ID.

# TibrvDispatchable::isValid()

*Method*

## Declaration

```
virtual tibrv_bool isValid () const = 0;
```

## Purpose

Test validity of a queue or queue group.

## Remarks

Returns TIBRV\_TRUE if the dispatchable object is valid; TIBRV\_FALSE if the dispatchable object has been destroyed.

# TibrvDispatchable::poll()

*Method*

## Declaration

```
virtual TibrvStatus poll();
```

## Purpose

Dispatch an event, if possible.

## Remarks

If an event is ready to dispatch, then this call dispatches it, and then returns. If no events are waiting, then this call returns immediately.

When the call dispatches an event, it returns [TIBRV\\_OK](#). When the call does not dispatch an event, it returns the status code [TIBRV\\_TIMEOUT](#).

This call is equivalent to `timedDispatch(0)`.

Both [TibrvQueue](#) and [TibrvQueueGroup](#) implement this method.

## See Also

[TibrvDispatchable](#)

[TibrvQueue::poll\(\)](#)

[TibrvQueueGroup::poll\(\)](#)

# TibrvDispatchable::timedDispatch()

*Method*

## Declaration

```
virtual TibrvStatus timedDispatch(tibrv_f64 timeout) = 0;
```

## Purpose

Dispatch an event, but if no event is ready to dispatch, limit the time that this call blocks while waiting for an event.

## Remarks

If an event is ready to dispatch, then this call dispatches it, and then returns. If no events are waiting, this call waits for an event to arrive. If an event arrives before the waiting time elapses, then it dispatches the event and returns. If the waiting time elapses first, then the call returns without dispatching an event.

When the call dispatches an event, it returns the status code `.` When the call does not dispatch an event, it returns the status code [TIBRV\\_TIMEOUT](#).

Both [TibrvQueue](#) and [TibrvQueueGroup](#) implement this method.

Parameter	Description
timeout	Maximum time (in seconds) that this call can block while waiting for an event to arrive.  TIBRV_NO_WAIT (zero) indicates no blocking (immediate timeout).  TIBRV_WAIT_FOREVER (-1) indicates no timeout.

## See Also

[TibrvDispatchable](#)

[TibrvQueue::timedDispatch\(\)](#)

[TibrvQueueGroup::timedDispatch\(\)](#)

# TibrvQueue

*Class*

## Declaration

```
class TibrvQueue : TibrvDispatchable
{
    TibrvQueue();           // Construct empty.
    virtual ~TibrvQueue();  // Destroy and reclaim storage.
}
```

## Purpose

Event queue.

## Remarks

Each event is associated with a [TibrvQueue](#) object; when the event occurs, Rendezvous software places the event object in its queue. Programs dispatch queues to process events.

The constructor produces a hollow object; [TibrvQueue::create\(\)](#) makes it operational.

The destructor calls the `destroy` method, unless the C object is already destroyed.

## Default Queue

The method [Tibrv::defaultQueue\(\)](#) returns a pre-defined queue. Programs that need only one event queue can use this default queue (instead of using [TibrvQueue::create\(\)](#) to create one). The default queue has priority 1, can hold an unlimited number of events, and never discards an event (since it never exceeds an event limit).

Rendezvous software places all advisories pertaining to queue overflow on the default queue.

Programs must not destroy the default queue, except as a side effect of [Tibrv::close\(\)](#). Programs cannot change the parameters of the default queue.

## Limit Policy

These constants specify the possible strategies for resolving overflow of queue limit.



Constant	Description
TIBRVQUEUEUE_DISCARD_NONE	Never discard events; use this policy when a queue has no limit on then number of events it can contain.
TIBRVQUEUEUE_DISCARD_FIRST	Discard the first event in the queue (that is, the oldest event in the queue, which would otherwise be the next event to dispatch).
TIBRVQUEUEUE_DISCARD_LAST	Discard the last event in the queue (that is, the youngest event in the queue).
TIBRVQUEUEUE_DISCARD_NEW	Discard the new event (which would otherwise cause the queue to overflow its maximum events limit).

Method	Description
--------	-------------

## Life Cycle

<a href="#">TibrvQueue::create()</a>	Create an event queue.
<a href="#">TibrvQueue::destroy()</a>	Destroy an event queue.
<a href="#">TibrvQueue::getHandle()</a>	Extract the C handle of this queue.
<a href="#">TibrvQueue::isValid()</a>	Test validity of a queue.

## Dispatch

<a href="#">TibrvQueue::dispatch()</a>	Dispatch an event; if no event is ready, block.
<a href="#">TibrvQueue::poll()</a>	Dispatch an event, if possible.
<a href="#">TibrvQueue::timedDispatch()</a>	Dispatch an event, but if no event is ready to dispatch, limit the time that this call blocks

Method	Description
--------	-------------

[while waiting for an event.](#)

## Properties

<a href="#">TibrvQueue::getCount()</a>	Extract the number of events in a queue.
<a href="#">TibrvQueue::getLimitPolicy()</a>	Extract the limit properties of a queue.
<a href="#">TibrvQueue::getName()</a>	Extract the name of a queue.
<a href="#">TibrvQueue::getPriority()</a>	Extract the priority of a queue.
<a href="#">TibrvQueue::setName()</a>	Set the name of a queue.
<a href="#">TibrvQueue::setLimitPolicy()</a>	Set the limit properties of a queue.
<a href="#">TibrvQueue::setPriority()</a>	Set the priority of a queue.

## Inherited Methods

[TibrvDispatchable::getDispatchable\(\)](#)  
[TibrvDispatchable::destroy\(\)](#)  
[TibrvDispatchable::dispatch\(\)](#)  
[TibrvDispatchable::isValid\(\)](#)  
[TibrvDispatchable::poll\(\)](#)  
[TibrvDispatchable::timedDispatch\(\)](#)

## Related Classes

[TibrvDispatchable](#)

[TibrvQueueGroup](#)

# TibrvQueue::create()

*Method*

## Declaration

```
TibrvStatus create ();
```

## Purpose

Create an event queue.

## Remarks

This method creates a C queue and stores its handle in the C++ object.

Upon creation, new queues use these default values for properties.

Property	Default Value	Set Method
limitPolicy	TIBRVQUEUE_DISCARD_NONE	TibrvQueue::setLimitPolicy()
maxEvents	zero (unlimited)	
discardAmount	zero	
name	tibrvQueue	TibrvQueue::setName()
priority	1	TibrvQueue::setPriority()

# TibrvQueue::destroy()

*Method*

## Declaration

```
TibrvStatus destroy ();  
TibrvStatus destroy (  
    TibrvQueueOnComplete* completeCB,  
    const void* closure = NULL);
```

## Purpose

Destroy an event queue.

## Remarks

When a queue is destroyed, events that remain in the queue are discarded.

The destructor calls this method.

When a program destroys a queue, all events associated with the queue become invalid. These invalid events still occupy storage until the program explicitly destroys them, or until the program calls [Tibrv::close\(\)](#).

A program must not call [TibrvQueue::destroy\(\)](#) on the default queue. Closing [Tibrv](#) destroys the default queue; see [Tibrv::close\(\)](#).

Parameter	Description
completeCB	<p>Rendezvous software runs this method immediately after all event callback methods dispatched from the queue have completed. If no event callback methods are running when the queue is destroyed, the destroy call runs the completion method before returning.</p> <p>If this parameter is NULL, this method does not run a completion.</p>
closure	Pass this closure argument to the completion method.

# TibrvQueue::dispatch()

*Method*

## Declaration

```
virtual TibrvStatus dispatch ();
```

## Purpose

Dispatch an event; if no event is ready, block.

## Remarks

If the queue is not empty, then this call dispatches the event at the head of the queue, and then returns. If the queue is empty, then this call blocks indefinitely while waiting for the queue to receive an event.

## See Also

[TibrvDispatchable](#)

[TibrvDispatchable::dispatch\(\)](#)

[TibrvQueue::poll\(\)](#)

[TibrvQueue::timedDispatch\(\)](#)

[TibrvDispatcher](#)

# TibrvQueue::getCount()

*Method*

## Declaration

```
TibrvStatus getCount (tibrv_u32& numEvents) const;
```

## Purpose

Extract the number of events in a queue.

Parameter	Description
numEvents	The program supplies a variable, and the method stores (a snapshot of) the event count of the queue in that variable.

# TibrvQueue::getHandle()

*Method*

## Declaration

```
tibrvQueue getHandle() const;
```

## Purpose

Extract the C handle of this queue.

# TibrvQueue::getLimitPolicy()

*Method*

## Declaration

```
TibrvStatus getLimitPolicy (  
    tibrvQueueLimitPolicy& policy,  
    tibrv_u32& maxEvents,  
    tibrv_u32& discardAmount) const;
```

## Purpose

Extract the limit properties of a queue.

Parameter	Description
policy	<p>Each queue has a policy for discarding events when a new event would cause the queue to exceed its maxEvents limit. For an explanation of the policy values, see <a href="#">Limit Policy</a>.</p> <p>The program supplies a variable, and the method stores the limit policy of the queue in that variable.</p>
maxEvents	<p>Programs can limit the number of events that a queue can hold—either to curb queue growth, or implement a specialized dispatch semantics.</p> <p>Zero specifies an unlimited number of events.</p> <p>The program supplies a variable, and the method stores the maximum event limit of the queue in that variable.</p>
discardAmount	<p>When the queue exceeds its maximum event limit, discard a block of events. This property specifies the number of events to discard.</p>



Parameter	Description
	The program supplies a variable, and the method stores the discard amount of the queue in that variable.

## See Also

tibrvQueueLimitPolicy in TIBCO Rendezvous C Reference

[TibrvQueue::setLimitPolicy\(\)](#)

QUEUE.LIMIT\_EXCEEDED in TIBCO Rendezvous Concepts

# TibrvQueue::getName()

*Method*

## Declaration

```
TibrvStatus getName (const char*& queueName) const;
```

## Purpose

Extract the name of a queue.

## Remarks

Queue names assist programmers and administrators in troubleshooting queues. When Rendezvous software delivers an advisory message pertaining to a queue, it includes the queue's name; administrators can use queue names to identify specific queues within a program.

The default name of every queue is `tibrvQueue`. We strongly recommend that you relabel each queue with a distinct and informative name, for use in debugging.

Parameter	Description
queueName	The program supplies a variable, and the method stores in that variable a string pointer to the queue name.  The program <i>must not</i> modify the string.

## See Also

[TibrvQueue::setName\(\)](#)

# TibrvQueue::getPriority()

*Method*

## Declaration

```
TibrvStatus getPriority (tibrv_u32& priority) const;
```

## Purpose

Extract the priority of a queue.

## Remarks

Each queue has a single priority value, which controls its dispatch precedence within queue groups. Higher values dispatch before lower values; queues with equal priority values dispatch in round-robin fashion.

When the queue is invalid, this method returns [TIBRV\\_INVALID\\_QUEUE](#).

## See Also

[TibrvQueue::setPriority\(\)](#)

# TibrvQueue::isValid()

*Method*

## Declaration

```
tibrv_bool isValid () const;
```

## Purpose

Test validity of a queue.

## Remarks

Returns TIBRV\_TRUE if the queue is valid; TIBRV\_FALSE if the queue has been destroyed.

## See Also

[Tibrv::close\(\)](#)

[TibrvQueue::destroy\(\)](#)

# TibrvQueue::poll()

*Method*

## Declaration

```
virtual TibrvStatus poll ();
```

## Purpose

Dispatch an event, if possible.

## Remarks

If the queue is not empty, then this call dispatches the event at the head of the queue, and then returns. If the queue is empty, then this call returns immediately.

When the call dispatches an event, it returns the status code [TIBRV\\_OK](#). When the call does not dispatch an event, it returns the status code [TIBRV\\_TIMEOUT](#).

This call is equivalent to `timedDispatch(0)`.

## See Also

[TibrvDispatchable](#)

[TibrvDispatchable::poll\(\)](#)

[TibrvQueue::dispatch\(\)](#)

[TibrvQueue::timedDispatch\(\)](#)

# TibrvQueue::setLimitPolicy()

*Method*

## Declaration

```
TibrvStatus setLimitPolicy (  
    tibrvQueueLimitPolicy policy,  
    tibrv_u32 maxEvents,  
    tibrv_u32 discardAmount);
```

## Purpose

Set the limit properties of a queue.

## Remarks

This method simultaneously sets three related properties, which together describe the behavior of a queue in overflow situations. Each call must explicitly specify all three properties.

Parameter	Description
limitPolicy	<p>Each queue has a policy for discarding events when a new event would cause the queue to exceed its maxEvents limit. Choose from the values of <a href="#">tibrvQueueLimitPolicy</a>.</p> <p>When maxEvents is zero (unlimited), the policy must be <a href="#">TIBRVQUEUE_DISCARD_NONE</a>.</p> <p>The program supplies a value, and the method sets the limit policy of the queue to that value.</p>
maxEvents	<p>Programs can limit the number of events that a queue can hold—either to curb queue growth, or implement a specialized dispatch semantics.</p>

Parameter	Description
	<p>Zero specifies an unlimited number of events; in this case, the policy must be <a href="#">TIBRVQUEUE_DISCARD_NONE</a>.</p> <p>The program supplies a value, and the method sets the maximum event limit of the queue to that value.</p>
discardAmount	<p>When the queue exceeds its maximum event limit, discard a block of events. This property specifies the number of events to discard.</p> <p>When discardAmount is zero, the policy must be <a href="#">TIBRVQUEUE_DISCARD_NONE</a>.</p> <p>The program supplies a value, and the method sets the discard amount of the queue to that value.</p>

## See Also

[TibrvQueue::getLimitPolicy\(\)](#)

# TibrvQueue::setName()

*Method*

## Declaration

```
TibrvStatus setName (const char* queueName);
```

## Purpose

Set the name of a queue.

## Remarks

Queue names assist programmers and administrators in troubleshooting queues. When Rendezvous software delivers an advisory message pertaining to a queue, it includes the queue's name; administrators can use queue names to identify specific queues within a program.

The default name of every queue is `tibrvQueue`. We strongly recommend that you relabel each queue with a distinct and informative name, for use in debugging.

Parameter	Description
queueName	Replace the name of the queue with this new name. It is illegal to supply NULL as the new queue name.

## See Also

[TibrvQueue::getName\(\)](#)



# TibrvQueue::setPriority()

*Method*

## Declaration

```
TibrvStatus setPriority(tibrv_u32 priority);
```

## Purpose

Set the priority of a queue.

## Remarks

Each queue has a single priority value, which controls its dispatch precedence within queue groups. Higher values dispatch before lower values; queues with equal priority values dispatch in round-robin fashion.

Changing the priority of a queue affects its position in all the queue groups that contain it.

Parameter	Description
priority	Replace the priority of the queue with this new value.  The priority is a non-negative integer. Priority zero signifies the last queue to dispatch.

## See Also

[TibrvQueue::getPriority\(\)](#)

# TibrvQueue::timedDispatch()

*Method*

## Declaration

```
virtual TibrvStatus timedDispatch(tibrv_f64 timeout);
```

## Purpose

Dispatch an event, but if no event is ready to dispatch, limit the time that this call blocks while waiting for an event.

## Remarks

If an event is already in the queue, this call dispatches it, and returns immediately. If the queue is empty, this call waits for an event to arrive. If an event arrives before the waiting time elapses, then it dispatches the event and returns. If the waiting time elapses first, then the call returns without dispatching an event.

When the call dispatches an event, it returns the status code [TIBRV\\_OK](#). When the call does not dispatch an event, it returns the status code [TIBRV\\_TIMEOUT](#).

Parameter	Description
timeout	Maximum time (in seconds) that this call can block while waiting for an event to arrive in the queue.  TIBRV_NO_WAIT (zero) indicates no blocking (immediate timeout).  TIBRV_WAIT_FOREVER (-1) indicates no timeout.

## See Also

[TibrvDispatchable](#)

[TibrvDispatchable::timedDispatch\(\)](#)

`TibrvQueue::dispatch()`

`TibrvQueue::poll()`

# TibrvQueueOnComplete

*Class*

## Declaration

```
class TibrvQueueOnComplete
```

## Purpose

Run program code after all callback methods of a destroyed queue have completed.

## Remarks

Implement this interface to post-process destroyed queues.

Method	Description
<a href="#">TibrvQueueOnComplete::onComplete()</a>	A program can destroy a queue object even when callback methods from its events are running in one or more threads. Multi-threaded programs can define methods of this type to discover when all event callback methods in progress have completed.

# TibrvQueueOnComplete::onComplete()

*Method*

## Declaration

```
virtual void onComplete(  
    TibrvQueue* queue,  
    void* closure)
```

## Purpose

A program can destroy a queue object even when callback methods from its events are running in one or more threads. Multi-threaded programs can define methods of this type to discover when all event callback methods in progress have completed.

Parameter	Description
queue	<p>This parameter receives the queue object.</p> <p>However, by the time this method runs, the queue is already destroyed; this method cannot use the queue object in Rendezvous calls.</p>
closure	<p>This parameter receives the closure object that the program supplied in the destroy call.</p>

## Remarks

This method is important when several threads dispatch from the same queue, and the program must do additional processing after the callback methods have completed *in all threads*.

Upon return from [TibrvQueue::destroy\(\)](#), the destroyed queue can no longer dispatch events. However, in each thread where an event callback method is already in progress, that callback method does continue to run until complete.

[TibrvQueue::destroy\(\)](#) accepts an argument of type [TibrvQueueOnComplete](#). Rendezvous software ensures that the completion method runs when the last callback-in-progress has completed.

## See Also

[TibrvQueue::destroy\(\)](#)

# TibrvQueueGroup

*Class*

## Declaration

```
class TibrvQueueGroup : public TibrvDispatchable
{
public:
    TibrvQueueGroup();           // Create empty.
    virtual ~TibrvQueueGroup();  // Destroy and reclaim storage.
};
```

## Purpose

Prioritized dispatch of several queues with one call.

## Remarks

Queue groups add flexibility and fine-grained control to the event queue dispatch mechanism. Programs can create groups of queues and dispatch them according to their queue priorities.

The constructor creates a hollow object; [TibrvQueueGroup::create\(\)](#) makes it operational.

The destructor calls the destroy method, unless the C object is already destroyed.

Method	Description
--------	-------------

### Life Cycle

<a href="#">TibrvQueueGroup::create()</a>	Create an event queue group.
<a href="#">TibrvQueueGroup::destroy()</a>	Destroy an event queue group.
<a href="#">TibrvQueueGroup::getHandle()</a>	Extract the C handle of this queue group.
<a href="#">TibrvQueueGroup::isValid()</a>	Test validity of a queue group.

Method	Description
<b>Dispatch</b>	
<a href="#">TibrvQueueGroup::dispatch()</a>	Dispatch an event from a queue group; if no event is ready, block.
<a href="#">TibrvQueueGroup::poll()</a>	Dispatch an event, but if no event is ready to dispatch, return immediately (without blocking).
<a href="#">TibrvQueueGroup::timedDispatch()</a>	Dispatch an event, but if no event is ready to dispatch, limit the time that this call blocks while waiting for an event.
<b>Queues</b>	
<a href="#">TibrvQueueGroup::add()</a>	Add an event queue to a queue group.
<a href="#">TibrvQueueGroup::remove()</a>	Remove an event queue from a queue group.

#### Inherited Methods

[TibrvDispatchable::getDispatchable\(\)](#)  
[TibrvDispatchable::destroy\(\)](#)  
[TibrvDispatchable::dispatch\(\)](#)  
[TibrvDispatchable::isValid\(\)](#)  
[TibrvDispatchable::poll\(\)](#)  
[TibrvDispatchable::timedDispatch\(\)](#)

## Related Classes

[TibrvDispatchable](#)

[TibrvQueue](#)



# TibrvQueueGroup::add()

*Method*

## Declaration

```
virtual TibrvStatus add(  
    TibrvQueue* queue);
```

## Purpose

Add an event queue to a queue group.

## Remarks

If the queue is already in the group, adding it again has no effect.

Parameter	Description
eventQueue	Add this event queue to a queue group.

## See Also

[TibrvQueue](#)

# TibrvQueueGroup::create()

*Method*

## Declaration

```
TibrvStatus create();
```

## Purpose

Create an event queue group.

## Remarks

This method creates a C queue group and stores its handle in the C++ object.

The new queue group is empty.

The queue group remains valid until the program explicitly destroys it.

## See Also

[TibrvQueueGroup::add\(\)](#)

[TibrvQueueGroup::destroy\(\)](#)

# TibrvQueueGroup::destroy()

*Method*

## Declaration

```
TibrvStatus destroy ();
```

## Purpose

Destroy an event queue group.

## Remarks

The individual queues in the group continue to exist, even though the group has been destroyed.

## See Also

[TibrvQueueGroup::create\(\)](#)

# TibrvQueueGroup::dispatch()

*Method*

## Declaration

```
virtual TibrvStatus dispatch();
```

## Purpose

Dispatch an event from a queue group; if no event is ready, block.

## Remarks

If any queue in the group contains an event, then this call searches the queues in priority order, dispatches an event from the first non-empty queue that it finds, and then returns. If all the queues are empty, then this call blocks indefinitely while waiting for any queue in the group to receive an event.

When searching the group for a non-empty queue, this call searches according to the priority values of the queues. If two or more queues have identical priorities, subsequent dispatch and poll calls rotate through them in round-robin fashion.

## See Also

[TibrvDispatchable](#)

[TibrvDispatchable::dispatch\(\)](#)

[TibrvQueueGroup::timedDispatch\(\)](#)

[TibrvQueueGroup::poll\(\)](#)

# TibrvQueueGroup::getHandle()

*Method*

## Declaration

```
tibrvQueueGroup getHandle() const;
```

## Purpose

Extract the C handle of this queue group.

# TibrvQueueGroup::isValid()

*Method*

## Declaration

```
tibrv_bool isValid() const;
```

## Purpose

Test validity of a queue group.

## Remarks

Returns TIBRV\_TRUE if the queue group is valid; TIBRV\_FALSE if the queue group has been destroyed.

## See Also

[Tibrv::close\(\)](#)

[TibrvQueueGroup::destroy\(\)](#)

# TibrvQueueGroup::poll()

*Method*

## Declaration

```
virtual TibrvStatus poll();
```

## Purpose

Dispatch an event, but if no event is ready to dispatch, return immediately (without blocking).

## Remarks

If any queue in the group contains an event, then this call searches the queues in priority order, dispatches an event from the first non-empty queue that it finds, and then returns. If all the queues are empty, then this call returns immediately.

When searching the group for a non-empty queue, this call searches according to the priority values of the queues. If two or more queues have identical priorities, subsequent dispatch and poll calls rotate through them in round-robin fashion.

When the call dispatches an event, it returns the status code [TIBRV\\_OK](#). When the call does not dispatch an event, it returns the status code [TIBRV\\_TIMEOUT](#).

This call is equivalent to `timedDispatch(0)`.

## See Also

[TibrvDispatchable](#)

[TibrvDispatchable::poll\(\)](#)

[TibrvQueueGroup::dispatch\(\)](#)

[TibrvQueueGroup::timedDispatch\(\)](#)

# TibrvQueueGroup::remove()

*Method*

## Declaration

```
virtual TibrvStatus remove(TibrvQueue* queue);
```

## Purpose

Remove an event queue from a queue group.

## Remarks

If the queue is not in the group, this call returns the status code [TIBRV\\_INVALID\\_QUEUE](#).

Parameter	Description
eventQueue	Remove this event queue from a queue group.

## See Also

[TibrvQueue](#)



# TibrvQueueGroup::timedDispatch()

*Method*

## Declaration

```
virtual TibrvStatus timedDispatch(tibrv_f64 timeout);
```

## Purpose

Dispatch an event, but if no event is ready to dispatch, limit the time that this call blocks while waiting for an event.

## Remarks

If any queue in the group contains an event, then this call searches the queues in priority order, dispatches an event from the first non-empty queue that it finds, and then returns. If the queue is empty, this call waits for an event to arrive in any queue. If an event arrives before the waiting time elapses, then the call searches the queues, dispatches the event, and returns. If the waiting time elapses first, then the call returns without dispatching an event.

When searching the group for a non-empty queue, this call searches according to the priority values of the queues. If two or more queues have identical priorities, subsequent dispatch calls rotate through them in round-robin fashion.

When the call dispatches an event, it returns the status code [TIBRV\\_OK](#). When the call does not dispatch an event, it returns the status code [TIBRV\\_TIMEOUT](#).

Parameter	Description
timeout	Maximum time (in seconds) that this call can block while waiting for an event to arrive in the queue group.  TIBRV_NO_WAIT (zero) indicates no blocking (immediate timeout).  TIBRV_WAIT_FOREVER (-1) indicates no timeout.

## See Also

[TibrvDispatchable](#)

[TibrvDispatchable::timedDispatch\(\)](#)

[TibrvQueueGroup::dispatch\(\)](#)

[TibrvQueueGroup::poll\(\)](#)

# TibrvDispatcher

*Class*

## Declaration

```
class TibrvDispatcher
  TibrvDispatcher();           // Create empty.
  virtual ~TibrvDispatcher();  // Halt, destroy & reclaim storage.
```

## Purpose

Dispatch events from a queue or queue group.

## Remarks

A dispatcher thread repeatedly dispatches a queue or queue group by calling [TibrvDispatchable::timedDispatch\(\)](#) in a loop.

The constructor produces a hollow object; [TibrvDispatcher::create\(\)](#) fills in the C handle, which makes the dispatcher thread operational.

The destructor calls the destroy method, unless the C handle is already destroyed.

This class is a programming convenience. Programs can implement specialized dispatcher threads, and use them instead of this class.

Method	Description
<a href="#">TibrvDispatcher::create()</a>	Start a dispatcher thread.
<a href="#">TibrvDispatcher::destroy()</a>	Destroy a dispatcher thread.
<a href="#">TibrvDispatcher::getDispatchable()</a>	Extract the queue or queue group that this thread dispatches.
<a href="#">TibrvDispatcher::getHandle()</a>	Extract the C handle of this dispatcher thread.

Method	Description
<a href="#">TibrvDispatcher::getName()</a>	Extract the name of a dispatcher thread.
<a href="#">TibrvDispatcher::isValid()</a>	Test whether a dispatcher object has been destroyed.
<a href="#">TibrvDispatcher::setName()</a>	Set the name of a dispatcher thread.

## See Also

[TibrvDispatchable::timedDispatch\(\)](#)

[DISPATCHER.THREAD\\_EXITED](#) on [page 242](#) in TIBCO Rendezvous Concepts

[tibrvDispatcher](#) in TIBCO Rendezvous C Reference

# TibrvDispatcher::create()

*Method*

## Declaration

```
TibrvStatus create(  
    TibrvDispatchable* dispatchable,  
    tibrv_f64 idleTimeout = TIBRV_WAIT_FOREVER);
```

## Purpose

Start a dispatcher thread.

## Remarks

This method creates a C dispatcher thread and stores its handle in the C++ object.

A dispatcher thread repeatedly dispatches a queue or queue group.

Inside the thread, a loop calls [TibrvDispatchable::timedDispatch\(\)](#). If the timed dispatch call returns without dispatching an event (after waiting for `idleTimeout` seconds), then the thread exits by calling [TibrvDispatcher::destroy\(\)](#).

Parameter	Description
<code>dispatchable</code>	The new thread dispatches this object, which can be either a queue or a queue group.
<code>idleTimeout</code>	<p>When this time period (in seconds) elapses without dispatching an event, the thread exits.</p> <p>The special value <code>TIBRV_WAIT_FOREVER</code> instructs the dispatcher to loop indefinitely; the thread does not exit until the program explicitly destroys it.</p> <p>The special value <code>TIBRV_NO_WAIT</code> instructs the dispatcher to poll until no events are ready to dispatch, then exit.</p>

## Stack Size

On UNIX platforms that use the POSIX thread package (pthread), this call sets the `stacksize` attribute of the dispatcher thread to the larger of two candidate values—either the default stack size of the operating system, or 64 kilobytes.

For programs that require a larger stack size, we recommend that you code a custom dispatcher thread.

## See Also

[TibrvDispatchable::timedDispatch\(\)](#)

[TibrvDispatchable](#)

[TibrvDispatcher::destroy\(\)](#)

[TibrvDispatcher::setName\(\)](#)

DISPATCHER.THREAD\_EXITED in TIBCO Rendezvous Concepts

tibrvDispatcher in TIBCO Rendezvous C Reference

# TibrvDispatcher::destroy()

*Method*

## Declaration

```
TibrvStatus destroy();
```

## Purpose

Destroy a dispatcher thread.

## Remarks

We do not recommend destroying a dispatcher thread within the same thread (for example, from within a listener callback function running within that thread). Although it is legal to do so, we discourage this practice, because some operating systems do not properly free internal resources associated with the thread (which can result in memory growth).

## See Also

DISPATCHER.THREAD\_EXITED in TIBCO Rendezvous Concepts

# TibrvDispatcher::getDispatchable()

*Method*

## Declaration

```
TibrvDispatchable* getDispatchable() const;
```

## Purpose

Extract the queue or queue group that this thread dispatches.



# TibrvDispatcher::getHandle()

*Method*

## Declaration

```
tibrvDispatcher getHandle() const;
```

## Purpose

Extract the C handle of this dispatcher thread.

# TibrvDispatcher::getName()

*Method*

## Declaration

```
TibrvStatus getName(  
    const char*& name) const;
```

## Purpose

Extract the name of a dispatcher thread.

Parameter	Description
dispatchName	The program supplies a variable, and the method stores the name of the dispatcher thread in that variable.

# TibrvDispatcher::isValid()

*Method*

## Declaration

```
tibrv_bool isValid() const;
```

## Purpose

Test whether a dispatcher object has been destroyed.

## Remarks

This method returns TIBRV\_TRUE if the dispatcher object is valid, and TIBRV\_FALSE if it has been destroyed.

# TibrvDispatcher::setName()

*Method*

## Declaration

```
TibrvStatus setName (const char* dispatchName);
```

## Purpose

Set the name of a dispatcher thread.

Parameter	Description
dispatchName	Use this name as the name of the dispatcher thread.

# Transports

---

Transports manage network connections and send outbound messages.

This section presents the various transport classes and their methods.

## See Also

[TibrvCmTransport](#)

[TibrvCmQueueTransport](#)

# TibrvTransport

*Class*

## Declaration

```
class TibrvTransport  
    virtual ~TibrvTransport(); // Destroy and reclaim storage.
```

## Purpose

A transport object represents a delivery mechanism for messages.

## Remarks

A transport describes a carrier for messages—whether across a network, among processes on a single computer, or within a process. Transports manage network connections, and send outbound messages.

A transport also defines the delivery scope of a message—that is, the set of *possible* destinations for the messages it sends.

Destroying a transport object invalidates subsequent send calls on that transport, and invalidates any listeners using that transport. The destructor calls the `destroy` method, unless the C object is already destroyed.

This class is the superclass of all other transport classes. Methods defined by this class are implemented by all transport subclasses (except [TibrvCmQueueTransport](#), for which some methods do not apply).

## Intra-Process Transport

Each process has exactly one intra-process transport; the call [Tibrv::open\(\)](#) automatically creates it, and the call [Tibrv::processTransport\(\)](#) extracts it. Programs must not destroy the intra-process transport.

Method	Description
<a href="#">TibrvTransport::createInbox()</a>	Create a unique inbox subject name.
<a href="#">TibrvTransport::destroy()</a>	Destroy a transport.
<a href="#">TibrvTransport::isValid()</a>	Test validity of a transport.
<a href="#">TibrvTransport::getDescription()</a>	Extract the program description parameter from a transport.
<a href="#">TibrvTransport::getHandle()</a>	Extract the C handle of this transport object.
<a href="#">TibrvTransport::requestReliability()</a>	Request reliability interval (message retention time) for a service.
<a href="#">TibrvTransport::send()</a>	Send a message.
<a href="#">TibrvTransport::sendReply()</a>	Send a reply message.
<a href="#">TibrvTransport::sendRequest()</a>	Send a request message and wait for a reply.
<a href="#">TibrvTransport::setDescription()</a>	Set the program description parameter of a transport.

## Descendants

[TibrvProcessTransport](#)

[TibrvNetTransport](#)

[TibrvVcTransport](#)

[TibrvCmTransport](#)

[TibrvCmQueueTransport](#)

## See Also

Transport in TIBCO Rendezvous Concepts



# TibrvTransport::createInbox()

*Method*

## Declaration

```
TibrvStatus createInbox(  
    char* subjectString,  
    tibrv_u32 subjectLimit) const;
```

## Purpose

Create a unique inbox subject name.

## Remarks

This method creates inbox names that are unique throughout the transport scope.

- For network transports, inbox subject names are unique across all processes within the local router domain—that is, anywhere that direct multicast contact is possible. The inbox name is not necessarily unique outside of the local router domain.
- For the intra-process transport, inbox names are unique across all threads of the process.

This method creates only the unique name for an inbox; it does not begin listening for messages on that subject name. To begin listening, pass the inbox name as the subject argument to [TibrvListener::create\(\)](#). The inbox name is only valid for use with the same transport that created it. When calling [TibrvListener::create\(\)](#), you *must* pass the same transport object that created the inbox subject name.

Remember that other programs have no information about an inbox subject name until the listening program uses it as a reply subject in an outbound message.

Use inbox subject names for delivery to a specific destination. In the context of a network transport, an inbox destination specifies unicast (point-to-point) delivery.

Rendezvous routing daemons (rvrd) translate inbox subject names that appear as the send subject or reply subject of a message. They do not translate inbox subject names within the data fields of a message.

This inherited method is disabled for [TibrvCmQueueTransport](#) objects.

**Warning**

This method is the *only* legal way for programs to create inbox subject names.

Parameter	Description
subjectString	The program supplies a string buffer, and the method stores the new inbox subject string in that buffer.
subjectLimit	The number of bytes that the program has allocated to receive the new inbox subject string.

**See Also**

[TibrvMsg::setReplySubject\(\)](#)

# TibrvTransport::destroy()

*Method*

## Declaration

```
TibrvStatus destroy()
```

## Purpose

Destroy a transport.

## Remarks

Programs must explicitly destroy each transport object, either using this method, or by calling [Tibrv::close\(\)](#) (which destroys all transports).

Destroying a transport achieves these effects:

- The transport flushes all outbound data to the Rendezvous daemon.  
This effect is especially important, and neither exiting the program nor calling [Tibrv::close\(\)](#) is sufficient to flush outbound data.
- The transport invalidates (but does not destroy) all associated listeners.
- Subsequent calls that use the destroyed transport return an error status.
- Storage for the transport object is freed.

## See Also

[TibrvTransport::isValid\(\)](#)

# TibrvTransport::getDescription()

*Method*

## Declaration

```
TibrvStatus getDescription(  
    const char** description) const;
```

## Purpose

Extract the program description parameter from a transport.

## Remarks

The description identifies your program to Rendezvous components. Browser administration interfaces display the description string.

## See Also

[TibrvTransport](#)

[TibrvTransport::setDescription\(\)](#)

# TibrvTransport::getHandle()

*Method*

## Declaration

```
tibrvTransport getHandle() const;
```

## Purpose

Extract the C handle of this transport object.

# TibrvTransport::isValid()

*Method*

## Declaration

```
virtual tibrv_bool isValid() const;
```

## Purpose

Test validity of a transport.

## Remarks

Returns TIBRV\_TRUE if the transport is valid; TIBRV\_FALSE if the transport has been destroyed.

## See Also

[Tibrv::close\(\)](#)

[TibrvTransport::destroy\(\)](#)

[TibrvCmTransport::destroy\(\)](#)

[TibrvCmQueueTransport::destroy\(\)](#)

# TibrvTransport::requestReliability()

*Method*

## Declaration

```
TibrvStatus requestReliability(  
    tibrv_f64    reliability);
```

## Purpose

Request reliability interval (message retention time) for a service.

Parameter	Description
reliability	Request this reliability interval (in seconds).  This value must be greater than zero.

## Remarks

This call lets application programs shorten the reliability interval of the specific service associated with a transport object. Successful calls change the daemon's reliability interval for all transports within the application process that use the same service.

An application can request a shorter retention time than the value that governs the daemon as a whole (either the factory default or the daemons `-reliability` parameter). The daemon's governing value silently overrides calls that request a longer retention time.

## Maximum Value Rule

Client transport objects that connect to the same daemon could specify different reliability intervals on the same service—whether by requesting a reliability value, or by using the daemon's effective value. In this situation, the daemon selects the *largest* potential value from among all the transports on that service, and uses that maximum value as the effective reliability interval for the service (that is, for all the transports on the service). This method of resolution favors the more stringent reliability requirements. (Contrast this rule with the Lower Value Rule that applies between two daemons.)

## Recomputing the Reliability

Whenever a transport connects, requests reliability, or disconnects from the daemon, the daemon recalculates the reliability interval for the corresponding service, by selecting the largest value of all transports communicating on that service.

When recomputing the reliability interval would result in a shorter retention time, the daemon delays using the new value until after an interval equivalent to the older (longer) retention time. This delay ensures that the daemon retains message data at least as long as the effective reliability interval at the time the message is sent.

## See Also

[TibrvTransport](#)

Reliability and Message Retention Time in TIBCO Rendezvous Administration

Lower Value Rule in TIBCO Rendezvous Administration

Changing the Reliability Interval within an Application Program in TIBCO Rendezvous Administration

Reliable Message Delivery in TIBCO Rendezvous Concepts



# TibrvTransport::send()

*Method*

## Declaration

```
virtual TibrvStatus send(  
    const TibrvMsg& message);
```

## Purpose

Send a message.

## Remarks

The message must have a valid destination subject; see [TibrvMsg::setSendSubject\(\)](#).

Parameter	Description
message	Send this message.

## See Also

[TibrvMsg::setSendSubject\(\)](#)

# TibrvTransport::sendReply()

Method

## Declaration

```
virtual TibrvStatus sendReply(  
    const TibrvMsg& replyMessage,  
    const TibrvMsg& requestMessage);
```

## Purpose

Send a reply message.

## Remarks

This convenience call extracts the reply subject of an inbound request message, and sends an outbound reply message to that subject. In addition to the convenience, this call is marginally faster than using separate calls to extract the subject and send the reply.

This method overwrites any existing send subject of the reply message with the reply subject of the request message.

Parameter	Description
replyMessage	Send this <i>outbound</i> reply message.
requestMessage	Send a reply to this <i>inbound</i> request message; extract its reply subject to use as the subject of the outbound reply message.



### Warning

Give special attention to the *order* of the arguments to this method. Reversing the inbound and outbound messages can cause an infinite loop, in which the program repeatedly resends the inbound message to itself (and all other recipients).

## See Also

[TibrvMsg::getReplySubject\(\)](#)

# TibrvTransport::sendRequest()

*Method*

## Declaration

```
virtual TibrvStatus sendRequest (
    const TibrvMsg& message,
    TibrvMsg& reply,
    tibrv_f64 timeout);
```

## Purpose

Send a request message and wait for a reply.

## Blocking can Stall Event Dispatch



### Warning

This call blocks all other activity on its program thread. If appropriate, programmers must ensure that other threads continue dispatching events on its queues.

Parameter	Description
message	Send this message.
reply	<p>The program supplies a variable, and the method stores the inbound reply in that variable.</p> <p>The program owns the reply message, and <i>must call its destructor</i> to reclaim storage.</p>
timeout	<p>Maximum time (in seconds) that this call can block while waiting for a reply.</p> <p>TIBRV_WAIT_FOREVER (-1) indicates no timeout (wait without limit for a reply).</p>

## Remarks

The status code [TIBRV\\_TIMEOUT](#) indicates that the specified time expired before receiving a reply.

Programs that receive and process the request message cannot determine that the sender has blocked until a reply arrives.

The request message must have a valid destination subject; see [TibrvMsg::setSendSubject\(\)](#).

## Operation

This method operates in several synchronous steps:

### Procedure

1. Create an inbox name, and an event that listens to it. Overwrite any existing reply subject of message with the inbox name.
2. Send the outbound message.
3. Block until the listener receives a reply; if the time limit expires before a reply arrives, then return [TIBRV\\_TIMEOUT](#). (The reply circumvents the event queue mechanism, so it is not necessary to explicitly call dispatch methods in the program.)
4. Store the reply in the variable specified as the reply parameter.
5. Return.

# TibrvTransport::setDescription()

*Method*

## Declaration

```
TibrvStatus setDescription(  
    const char* description);
```

## Purpose

Set the program description parameter of a transport.

## Remarks

The description identifies your program to Rendezvous components. Browser administration interfaces display the description string of ordinary transport objects (however they do not display the description string of [TibrvCmTransport](#) or [TibrvCmQueueTransport](#) objects).

As a debugging aid, we recommend setting a unique description string for each transport. Use a string that distinguishes both the application and the role of the transport within it.

Parameter	Description
description	Use this string as the new program description.

## See Also

[TibrvTransport](#)

[TibrvTransport::getDescription\(\)](#)

# tibrvTransportBatchMode

*Type*

## Declaration

```
typedef enum {  
    TIBRV_TRANSPORT_DEFAULT_BATCH,  
    TIBRV_TRANSPORT_TIMER_BATCH  
} tibrvTransportBatchMode;
```

## Purpose

Specify the batch mode of a transport.

Value	Description
TIBRV_TRANSPORT_DEFAULT_BATCH	Default batch behavior. The transport transmits outbound messages to rvd as soon as possible.  This value is the initial default for all transports.
TIBRV_TRANSPORT_TIMER_BATCH	Timer batch behavior. The transport accumulates outbound messages, and transmits them to rvd in batches—either when its buffer is full, or when a timer interval expires. (Programs cannot adjust the timer interval.)

## See Also

[TibrvNetTransport::setBatchMode\(\)](#)

Batch Modes for Transports in TIBCO Rendezvous Concepts

# TibrvProcessTransport

*Class*

## Declaration

```
class TibrvProcessTransport : public TibrvTransport
```

## Purpose

The intra-process transport delivers messages among the threads of a program.

## Remarks

The intra-process transport does not access the network.

This class does not introduce any new methods.

The call [Tibrv::open\(\)](#) automatically creates the intra-process transport; [Tibrv::close\(\)](#) automatically destroys it; [Tibrv::processTransport\(\)](#) extracts it from the Rendezvous environment. Programs cannot create additional instances of this class, and cannot destroy the intra-process transport.

### Inherited Methods

[TibrvTransport::createInbox\(\)](#)

[TibrvTransport::destroy\(\)](#)

[TibrvTransport::isValid\(\)](#)

[TibrvTransport::getHandle\(\)](#)

[TibrvTransport::send\(\)](#)

[TibrvTransport::sendReply\(\)](#)

[TibrvTransport::sendRequest\(\)](#)

[TibrvTransport::setDescription\(\)](#)



## Related Classes

[TibrvTransport](#)

[TibrvNetTransport](#)

[TibrvCmTransport](#)

## See Also

[Tibrv::open\(\)](#)

[Tibrv::processTransport\(\)](#)

# TibrvNetTransport

*Class*

## Declaration

```
class TibrvNetTransport : public TibrvTransport
{
    TibrvNetTransport();           // Construct empty.
    virtual ~TibrvNetTransport(); // Destroy and reclaim storage.
};
```

## Purpose

Deliver messages across a network.

## Remarks

This class connects to rvd for network communications.

The constructor creates a hollow object; [TibrvNetTransport::create\(\)](#) makes it operational.

The destructor calls the destroy method, unless the C object is already destroyed.

Method	Description
<a href="#">TibrvNetTransport::create()</a>	Create a transport that connects to a Rendezvous daemon.
<a href="#">TibrvNetTransport::getDaemon()</a>	Extract the daemon parameter from this transport.
<a href="#">TibrvNetTransport::getNetwork()</a>	Return the network interface that this transport uses for communication.
<a href="#">TibrvNetTransport::getService()</a>	Return the effective

Method	Description
	<a href="#">service that this transport uses for communication.</a>
<a href="#">TibrvNetTransport::setBatchMode()</a>	Set the batch mode parameter of a transport.

### Inherited Methods

[TibrvTransport::createInbox\(\)](#)  
[TibrvTransport::destroy\(\)](#)  
[TibrvTransport::isValid\(\)](#)  
[TibrvTransport::getHandle\(\)](#)  
[TibrvTransport::send\(\)](#)  
[TibrvTransport::sendReply\(\)](#)  
[TibrvTransport::sendRequest\(\)](#)  
[TibrvTransport::setDescription\(\)](#)

## Related Classes

[TibrvTransport](#)

[TibrvProcessTransport](#)

[TibrvCmTransport](#)

[TibrvCmQueueTransport](#)

# TibrvNetTransport::create()

*Method*

## Declaration

```
TibrvStatus create(  
    const char* service = NULL,  
    const char* network = NULL,  
    const char* daemon = NULL );  
TibrvStatus createLicensed(  
    const char* service,  
    const char* network,  
    const char* daemon,  
    const char* license_ticket);
```

## Purpose

Create a transport that connects to a Rendezvous daemon.

## Remarks

This method creates a C network transport and stores its handle in the C++ object.

Parameter	Description
service	<p>The Rendezvous daemon divides the network into logical partitions. Each <a href="#">TibrvNetTransport</a> communicates on a single service; a transport can communicate only with other transports on the same service.</p> <p>To communicate on more than one service, a program must create more than one transport—one transport for each service.</p> <p>You can specify the service in several ways. For details, see Service Parameter in TIBCO Rendezvous Concepts.</p> <p>NULL specifies the default rendezvous service.</p>
network	Every network transport communicates with other transports over a

Parameter	Description
	<p>single network interface. On computers with more than one network interface, the <code>network</code> parameter instructs the Rendezvous daemon to use a particular network for all outbound messages from this transport.</p> <p>To communicate over more than one network, programs must create more than one transport.</p> <p>You can specify the network in several ways. For details, see <a href="#">Network Parameter in TIBCO Rendezvous Concepts</a>.</p> <p>NULL specifies the primary network interface for the host computer.</p>
<code>daemon</code>	<p>The <code>daemon</code> parameter instructs the transport object about how and where to find the Rendezvous daemon and establish communication.</p> <p>For details, see <a href="#">Daemon Parameter in TIBCO Rendezvous Concepts</a>.</p> <p>You can specify a daemon on a remote computer. For details, see <a href="#">Remote Daemon in TIBCO Rendezvous Concepts</a>.</p> <p>If you specify a secure daemon, this string must be identical to as the <code>daemonName</code> argument of <a href="#">TibrvSdContext:setDaemonCert()</a>. See also, <a href="#">Secure Daemon in TIBCO Rendezvous Concepts</a>.</p> <p>null specifies the default—find the local daemon on TCP socket 7500. (This default is not valid when the local daemon is a secure daemon.)</p>
<code>licenseTicket</code>	<p>License tickets are no longer required. Values for this parameter are ignored.</p>

## Connecting to the Rendezvous Daemon

Rendezvous daemon processes do the work of moving messages across a network. Every [TibrvNetTransport](#) must connect to a Rendezvous daemon.

If a Rendezvous daemon process with a corresponding daemon parameter is already running, the transport connects to it.

If an appropriate Rendezvous local daemon is *not* running, the transport tries to start it. However, the transport does not attempt to start a *remote* daemon when none is running.

If the transport cannot connect to the Rendezvous daemon, it returns the status code [TIBRV\\_DAEMON\\_NOT\\_CONNECTED](#), and does not create a C transport object.

## Description String

As a debugging aid, we recommend setting a unique description string for each transport. Use a string that distinguishes both the application and the role of the transport within it. See [TibrvTransport::setDescription\(\)](#).

## See Also

[TibrvNetTransport::getDaemon\(\)](#)

[TibrvNetTransport::getNetwork\(\)](#)

[TibrvNetTransport::getService\(\)](#)

# TibrvNetTransport::getDaemon()

*Method*

## Declaration

```
TibrvStatus getDaemon(  
    const char*& daemonString ) const;
```

## Purpose

Extract the daemon parameter from this transport.

Parameter	Description
daemonString	<p>The program supplies a variable, and the method places in that variable a string pointer to the daemon information.</p> <p>The program <i>must not</i> modify nor free the string.</p>

# TibrvNetTransport::getNetwork()

*Method*

## Declaration

```
TibrvStatus getNetwork(  
    const char*& networkString) const;
```

## Purpose

Return the network interface that this transport uses for communication.

Parameter	Description
networkString	<p>The program supplies a variable, and the method places in that variable a string pointer to the network information.</p> <p>The program <i>must not</i> modify nor free the string.</p>



# TibrvNetTransport::getService()

*Method*

## Declaration

```
TibrvStatus getService(  
    const char*& serviceString) const;
```

## Purpose

Return the effective service that this transport uses for communication.

Parameter	Description
serviceString	<p>The program supplies a variable, and the method places in that variable a string pointer to the service information.</p> <p>The program <i>must not</i> modify nor free the string.</p>

# TibrvNetTransport::setBatchMode()

*Method*

## Declaration

```
TibrvStatus setBatchMode(  
    tibrvTransportBatchMode mode);
```

## Purpose

Set the batch mode parameter of a transport.

## Remarks

The batch mode determines when the transport transmits outbound message data to rvd:

- As soon as possible (the initial default for all transports)
- Either when its buffer is full, or when a timer interval expires—either event triggers transmission to the daemon

Parameter	Description
mode	Use this value as the new batch mode.

## See Also

[tibrvTransportBatchMode](#)

Batch Modes for Transports in TIBCO Rendezvous Concepts

# Virtual Circuits

---

Virtual circuits feature Rendezvous communication between two terminals over an exclusive, continuous, monitored connection.

## See Also

Virtual Circuits in TIBCO Rendezvous Concepts

# TibrVcTransport

*Class*

## Declaration

```
class TibrVcTransport : public TibrTransport
    TibrVcTransport();      // Construct empty.
    virtual ~TibrVcTransport(); // Destroy and reclaim storage.
```

## Purpose

A virtual circuit transport object represents a terminal in a potential circuit.

## Remarks

A virtual circuit transport can fill the same roles as an ordinary transport. Programs can use them to create inbox names, send messages, create listeners and other events.

The constructor creates a hollow object; programs call one of the two create methods to makes operational.

The destructor calls the destroy method, unless the C object is already destroyed.

Two methods determine the protocol role of the transport object—one method creates a terminal that *accepts* connections, and another method creates a terminal that attempts to *connect*.

The two types of terminal play complementary roles as they attempt to establish a connection. However, this difference soon evaporates. After the connection is complete, the two terminals behave identically.

Method	Description
<a href="#">TibrVcTransport::createAcceptVc()</a>	Create a virtual circuit accept object.
<a href="#">TibrVcTransport::createConnectVc()</a>	Create a virtual circuit connect

Method	Description
	object.
<code>TibrvVcTransport::waitForVcConnection()</code>	Test the connection status of a virtual circuit.

## Broken Connection

The following conditions can close a virtual circuit connection:

- Contact is broken between the object and its terminal.
- The virtual circuit loses data in either direction (see `DATALOSS` in TIBCO Rendezvous Concepts).
- The partner program destroys its terminal object (or that terminal becomes invalid).
- The program destroys the object.
- The program destroys the object's ordinary transport.

## Destroying VC Transports

Destroying a transport object precludes subsequent communications on that transport. Attempting to use a destroyed transport in any way is an error.

Destroying a virtual circuit transport does *not* affect the ordinary transport that the terminal employs.

To free storage, call the object's destructor. (The `destroy` method does not automatically free storage.)

## Direct Communication

Because virtual circuits rely on point-to-point messages between the two terminals, they can use direct communication to good advantage. To do so, both terminals must use network transports that enable direct communication.

For an overview, see [Direct Communication in TIBCO Rendezvous Concepts](#).

For programming details, see [Specifying Direct Communication in TIBCO Rendezvous Concepts](#).

## Inherited Methods

Legal Methods	<a href="#">TibrvTransport::createInbox()</a>
	<a href="#">TibrvTransport::destroy()</a>
	<a href="#">TibrvTransport::getHandle()</a>
	<a href="#">TibrvTransport::isValid()</a>
	<a href="#">TibrvTransport::send()</a>
	<a href="#">TibrvTransport::sendReply()</a>
	<a href="#">TibrvTransport::sendRequest()</a>

Disabled Methods	<a href="#">TibrvTransport::getDescription()</a>
	<a href="#">TibrvTransport::setDescription()</a>

## See Also

Virtual Circuits in TIBCO Rendezvous Concepts

# TibrvVcTransport::createAcceptVc()

*Method*

## Declaration

```
TibrvStatus createAcceptVc(  
    const char**    connectSubject,  
    TibrvTransport* transport);
```

## Purpose

Create a virtual circuit accept object.

## Remarks

This method creates a C accept transport and stores its handle in the C++ object.

Parameter	Description
connectSubject	<p>The program supplies a location, and the method stores the connect subject of the new virtual circuit accept transport in that location.</p> <p>After this call returns, the program must send a message to another program, inviting it to establish a virtual circuit. Furthermore, the <i>reply subject</i> of that invitation message must be this connect subject. To complete the virtual circuit, the second program must extract this subject from the invitation, and supply it to <a href="#">TibrvVcTransport::createConnectVc()</a>.</p>
transport	<p>The virtual circuit uses this ordinary transport for communications.</p> <p>Programs may use this transport for other purposes.</p> <p>It is illegal to supply a virtual circuit transport object for this parameter (that is, you cannot nest a virtual circuit within another virtual circuit).</p>

## Test Before Using

Either of two conditions indicate that the connection is ready to use:

- The transport presents the [VC.CONNECTED](#) advisory.
- [TibrvVcTransport::waitForVcConnection\(\)](#) returns without error.

### Procedure

Immediately after this call, test *both* conditions with these two steps (in this order):

1. Listen on the virtual circuit transport object for the [VC.CONNECTED](#) advisory.
2. Call [TibrvVcTransport::waitForVcConnection\(\)](#) with zero as the timeout parameter.

For an explanation, see Testing the New Connection in TIBCO Rendezvous Concepts.

## See Also

[TibrvVcTransport::createConnectVc\(\)](#)

[TibrvVcTransport::waitForVcConnection\(\)](#)

[VC.CONNECTED](#) in TIBCO Rendezvous Concepts

[VC.DISCONNECTED](#) in TIBCO Rendezvous Concepts



# TibrvVcTransport::createConnectVc()

*Method*

## Declaration

```
TibrvStatus createConnectVc(  
    const char*    connectSubject,  
    TibrvTransport* transport);
```

## Purpose

Create a virtual circuit connect object.

## Remarks

This method creates a C connect transport and stores its handle in the C++ object.

Parameter	Description
connectSubject	<p>The connect transport uses this connect subject to establish a virtual circuit with an <i>accept</i> transport in another program.</p> <p>The program must receive this connect subject from the accepting program. The call to <a href="#">TibrvVcTransport::createAcceptVc()</a> creates this subject.</p>
transport	<p>The virtual circuit uses this ordinary transport for communications.</p> <p>Programs may use this transport for other purposes.</p> <p>It is illegal to supply a virtual circuit transport object for this parameter (that is, you cannot nest a virtual circuit within another virtual circuit).</p>

## Test Before Using

Either of two conditions indicate that the connection is ready to use:

- The transport presents the VC.CONNECTED advisory.

- [TibrvVcTransport::waitForVcConnection\(\)](#) returns without error.

## Procedure

Immediately after this call, test *both* conditions with these two steps (in this order):

1. Listen on the virtual circuit transport object for the VC.CONNECTED advisory.
2. Call [TibrvVcTransport::waitForVcConnection\(\)](#) with zero as the timeout parameter.

For an explanation, see Testing the New Connection in TIBCO Rendezvous Concepts.

## See Also

[TibrvVcTransport::createAcceptVc\(\)](#)

[TibrvVcTransport::waitForVcConnection\(\)](#)

VC.CONNECTED in TIBCO Rendezvous Concepts

VC.DISCONNECTED in TIBCO Rendezvous Concepts

# TibrvVcTransport::waitForVcConnection()

*Method*

## Declaration

```
TibrvStatus waitForVcConnection(  
    tibrv_f64    timeout);
```

## Purpose

Test the connection status of a virtual circuit.

## Remarks

This method tests (and can block) until this virtual circuit transport object has established a connection with its opposite terminal. You may call this method for either an accept terminal or a connect terminal.

This method produces the same information as the virtual circuit advisory messages—but it produces it synchronously (while advisories are asynchronous). Programs can use this method not only to test the connection, but also to block until the connection is ready to use.

For example, a program can create a terminal object, then call this method to wait until the connection completes.

Parameter	Description
timeout	<p>This parameter determines the behavior of the call:</p> <ul style="list-style-type: none"><li>• For a quick test of current connection status, supply zero. The call returns immediately, without blocking.</li><li>• To wait for a new terminal to establish a connection, supply a reasonable positive value. The call returns either when the connection is complete, or when this time limit elapses.</li></ul>

Parameter	Description
	<ul style="list-style-type: none"> <li>To wait indefinitely for a usable connection, supply -1. The call returns when the connection is complete. If the connection was already complete and is now broken, the call returns immediately.</li> </ul>

---

Status	Description
<a href="#">TIBRV_OK</a>	The connection is complete (ready to use).
<a href="#">TIBRV_TIMEOUT</a>	The connection is not yet complete, but the non-negative time limit for waiting has expired.
<a href="#">TIBRV_VC_NOT_CONNECTED</a>	The connection was formerly complete, but is now irreparably broken.

## See Also

[TibrVcTransport::createAcceptVc\(\)](#)

[TibrVcTransport::createConnectVc\(\)](#)

Testing the New Connection in TIBCO Rendezvous Concepts

VC.CONNECTED in TIBCO Rendezvous Concepts

VC.DISCONNECTED in TIBCO Rendezvous Concepts

# Fault Tolerance

---

Rendezvous fault tolerance software coordinates a group of redundant processes into a fault-tolerant distributed program. Some processes actively fulfill the tasks of the program, while other processes wait in readiness. When one of the active processes fails, another process rapidly assumes active duty.

# Fault Tolerance Road Map

For a complete discussion of concepts and operating principles, see [Fault Tolerance Concepts](#) in TIBCO Rendezvous Concepts.

For suggestions to help you design programs using fault tolerance features, see [Fault Tolerance Programming](#) in TIBCO Rendezvous Concepts.

For step-by-step hints for implementing fault-tolerant systems, see [Developing Fault-Tolerant Programs](#) in TIBCO Rendezvous Concepts.

Fault tolerance software uses advisory messages to inform programs of status changes. For details, see [Fault Tolerance \(RVFT\) Advisory Messages](#) in TIBCO Rendezvous Concepts.

If your application distributes fault-tolerant processes across network boundaries, you must configure the Rendezvous routing daemons to exchange \_RVFT administrative messages. For details, see [Fault Tolerance](#) in TIBCO Rendezvous Administration, and discuss with your network administrator.

# tibrvftAction

*Type*

## Declaration

```
typedef enum
{
    TIBRVFT_PREPARE_TO_ACTIVATE = 1,
    TIBRVFT_ACTIVATE             = 2,
    TIBRVFT_DEACTIVATE           = 3
} tibrvftAction;
```

## Purpose

Instruct fault tolerance callback methods to react to changing circumstances.

## Remarks

Each token of this enumerated type designates a command to a fault tolerance callback method. The program's callback method receives one of these tokens in a parameter, and interprets it as an instruction from the Rendezvous fault tolerance software as described in this table (see also, Fault Tolerance Callback Actions in TIBCO Rendezvous Concepts).

Token	Description
TIBRVFT_PREPARE_TO_ACTIVATE	<p>Prepare to activate (hint).</p> <p>Rendezvous fault tolerance software passes this token to the callback method to instruct the program to make itself ready to activate on short notice—so that if the callback method subsequently receives the instruction to activate, it can do so without delay.</p> <p>This token is a hint, indicating that the program might soon receive an instruction to activate. It does not guarantee that an activate instruction will follow, nor that any minimum time will elapse before an activate</p>

Token	Description
	instruction follows.
TIBRVFT_ACTIVATE	Activate immediately.  Rendezvous fault tolerance software passes this token to the callback method to instruct the program to activate.
TIBRVFT_DEACTIVATE	Deactivate immediately.  Rendezvous fault tolerance software passes this token to the callback method to instruct the program to deactivate.

## See Also

[TibrvFtMemberCallback::onFtAction\(\)](#)

[TibrvFtMember::create\(\)](#)



# TibrvFtMember

*Class*

## Declaration

```
class TibrvFtMember
    TibrvFtMember();           // Create empty.
    virtual ~TibrvFtMember(); // Destroy and reclaim storage.
```

## Purpose

Represent membership in a fault tolerance group.

## Remarks

The constructor creates a hollow object; [TibrvFtMember::create\(\)](#) makes it operational and joins a fault tolerance group.

By destroying a member object, the program withdraws its membership in the fault tolerance group. The destructor calls the destroy method, unless the C object is already destroyed.

Destroying the queue or transport of a member object invalidates the member object, but does not destroy it. Programs must explicitly call the member's destructor to reclaim storage.

Method	Description
--------	-------------

## Life Cycle

<a href="#">TibrvFtMember::create()</a>	Create a member of a fault tolerance group.
<a href="#">TibrvFtMember::destroy()</a>	Destroy a member of a fault tolerance group.

Method	Description
<a href="#">TibrvFtMember::getHandle()</a>	Extract the C handle of a fault tolerance member.
<a href="#">TibrvFtMember::isValid()</a>	Test validity of a fault tolerance member object.

## Properties

<a href="#">TibrvFtMember::getClosure()</a>	Extract the closure data of a fault tolerance member.
<a href="#">TibrvFtMember::getGroupName()</a>	Extract the group name of a fault tolerance member.
<a href="#">TibrvFtMember::getQueue()</a>	Extract the event queue of a fault tolerance member.
<a href="#">TibrvFtMember::getTransport()</a>	Extract the transport of a fault tolerance member.
<a href="#">TibrvFtMember::getWeight()</a>	Extract the weight of a fault tolerance member.
<a href="#">TibrvFtMember::setWeight()</a>	Change the weight of a fault tolerance member within its group.

## Related Classes

[TibrvFtMonitor](#)

## See Also

[TibrvFtMemberCallback](#)

# TibrvFtMember::create()

*Method*

## Declaration

```
TibrvStatus create(  
    TibrvQueue* queue,  
    TibrvFtMemberCallback* callback,  
    TibrvTransport* transport,  
    const char* groupName,  
    tibrv_u16 weight,  
    tibrv_u16 activeGoal,  
    tibrv_f64 heartbeatInterval,  
    tibrv_f64 preparationInterval,  
    tibrv_f64 activationInterval,  
    const void* closure = NULL);
```

## Purpose

Create a member of a fault tolerance group.

## Remarks

This method creates a C fault tolerance member object, and stores it in the C++ object. The program becomes a member of the fault tolerance group.

A program may hold simultaneous memberships in several distinct fault tolerance groups. For examples, see Multiple Groups in TIBCO Rendezvous Concepts.

Avoid joining the same group twice. It is illegal for a program to maintain more than one membership in any one fault tolerance group. This method does not guard against this illegal situation, and results are unpredictable.

All arguments are required except for preparationInterval (which may be zero) and closure (which may be NULL).

## Intervals

The heartbeat interval must be less than the activation interval. If the preparation interval is non-zero, it must be greater than the heartbeat interval and less than the activation

interval. It is an error to violate these rules.

In addition, intervals must be reasonable for the hardware and network conditions. For information and examples, see Step 4: Choose the Intervals in TIBCO Rendezvous Concepts.

## Group Name

The group name must be a legal Rendezvous subject name (see Subject Names in TIBCO Rendezvous Concepts). You may use names with several elements; for examples, see Multiple Groups in TIBCO Rendezvous Concepts.

Parameter	Description
queue	Place fault tolerance events for this member on this event queue.
callback	On dispatch, process the event with this callback object.
transport	Use this transport for fault tolerance internal protocol messages (such as heartbeat messages).
groupName	Join the fault tolerant group with this name.  The group name must conform to the syntax required for Rendezvous subject names. For details, see Subject Names in TIBCO Rendezvous Concepts.
weight	Weight represents the ability of this member to fulfill its purpose, relative to other members of the same fault tolerance group. Rendezvous fault tolerance software uses relative weight values to select which members to activate; members with higher weight take precedence over members with lower weight.  Acceptable values range from 1 to 65535. Zero is a special, reserved value; Rendezvous fault tolerance software assigns zero weight to processes with resource errors, so they only activate when no other members are available.  For more information, see Rank and Weight in TIBCO Rendezvous Concepts.
activeGoal	Rendezvous fault tolerance software sends callback instructions to

Parameter	Description
	<p>maintain this number of active members.</p> <p>Acceptable values range from 1 to 65535.</p>
heartbeatInterval	<p>When this member is active, it sends heartbeat messages at this interval (in seconds).</p> <p>The interval must be positive. To determine the correct value, see Step 4: Choose the Intervals in TIBCO Rendezvous Concepts.</p>
preparationInterval	<p>When the heartbeat signal from one or more active members has been silent for this interval (in seconds), Rendezvous fault tolerance software issues an early warning hint (<a href="#">TIBRVFT_PREPARE_TO_ACTIVATE</a>) to the ranking inactive member. This warning lets the inactive member prepare to activate, for example, by connecting to a database server, or allocating memory.</p> <p>The interval must be non-negative. Zero is a special value, indicating that the member does not need advance warning to activate; Rendezvous fault tolerance software never issues a <a href="#">TIBRVFT_PREPARE_TO_ACTIVATE</a> hint when this value is zero. To determine the correct value, see Step 4: Choose the Intervals in TIBCO Rendezvous Concepts.</p>
activationInterval	<p>When the heartbeat signal from one or more active members has been silent for this interval (in seconds), Rendezvous fault tolerance software considers the silent member to be lost, and issues the instruction to activate (<a href="#">TIBRVFT_ACTIVATE</a>) to the ranking inactive member.</p> <p>When a new member joins a group, Rendezvous fault tolerance software identifies the new member to existing members (if any), and then waits for this interval to receive identification from them in return. If, at the end of this interval, it determines that too few members are active, it issues the activate instruction (<a href="#">TIBRVFT_ACTIVATE</a>) to the new member.</p> <p>Then interval must be positive. To determine the correct value,</p>

Parameter	Description
	see Step 4: Choose the Intervals in TIBCO Rendezvous Concepts.
closure	Store this closure data in the member object.

## See Also

[TibrvFtMember](#).

[TibrvFtMemberCallback](#).

[TibrvFtMember::destroy\(\)](#).

Step 1: Choose a Group Name in TIBCO Rendezvous Concepts

Step 2: Choose the Active Goal in TIBCO Rendezvous Concepts

Step 4: Choose the Intervals in TIBCO Rendezvous Concepts

Step 5: Program Start Sequence in TIBCO Rendezvous Concepts

# TibrvFtMember::destroy()

*Method*

## Declaration

```
TibrvStatus destroy(  
    TibrvFtMemberOnComplete* completeCB = NULL);
```

## Purpose

Destroy a member of a fault tolerance group.

## Remarks

By destroying a member object, the program cancels or withdraws its membership in the group. If the member is active, it stops sending the heartbeat signal.

Once a program withdraws from a group, it no longer receives fault tolerance events. One direct consequence is that an active program that withdraws can never receive an instruction to deactivate.

To free storage, call the object's destructor. (The destroy method does not automatically free storage.)

## See Also

[TibrvFtMember::create\(\)](#)

[TibrvFtMember::isValid\(\)](#)

[TibrvFtMemberOnComplete](#)

[TibrvFtMemberOnComplete::onComplete](#)

# TibrvFtMember::getClosure()

*Method*

## Declaration

```
void* getClosure() const;
```

## Purpose

Extract the closure data of a fault tolerance member.

## See Also

[TibrvFtMember::create\(\)](#)



# TibrvFtMember::getGroupName()

*Method*

## Declaration

```
TibrvStatus getGroupName(  
    const char *& groupName) const;
```

## Purpose

Extract the group name of a fault tolerance member.

Parameter	Description
groupName	<p>The program supplies a variable, and the method stores the group name in that variable.</p> <p>The program must not modify this string nor free its storage.</p>

## See Also

[TibrvFtMember::create\(\)](#)

# TibrvFtMember::getHandle()

*Method*

## Declaration

```
tibrvftMember getHandle() const;
```

## Purpose

Extract the C handle of a fault tolerance member.

## See Also

[TibrvFtMember::create\(\)](#)

tibrvftMember in TIBCO Rendezvous C Reference

# TibrvFtMember::getQueue()

*Method*

## Declaration

```
TibrvQueue* getQueue () const;
```

## Purpose

Extract the event queue of a fault tolerance member.

## Remarks

If the member is invalid, this method returns NULL.

## See Also

[TibrvQueue](#)

[TibrvFtMember::create\(\)](#)

# TibrvFtMember::getTransport()

*Method*

## Declaration

```
TibrvTransport* getTransport() const;
```

## Purpose

Extract the transport of a fault tolerance member.

## See Also

[TibrvTransport](#)

[TibrvFtMember::create\(\)](#)

# TibrvFtMember::getWeight()

*Method*

## Declaration

```
TibrvStatus getWeight(  
    tibrv_u16& weight) const;
```

## Purpose

Extract the weight of a fault tolerance member.

Parameter	Description
weight	The program supplies a variable, and the method stores the weight in that variable.

## See Also

[TibrvFtMember::create\(\)](#)

# TibrvFtMember::isValid()

*Method*

## Declaration

```
tibrv_bool isValid() const;
```

## Purpose

Test validity of a fault tolerance member object.

## Remarks

Returns TIBRV\_TRUE if the C member is valid; TIBRV\_FALSE if the member has been destroyed or is otherwise invalid.

## See Also

[TibrvFtMember::destroy\(\)](#)

# TibrvFtMember::setWeight()

*Method*

## Declaration

```
TibrvStatus setWeight(  
    tibrv_u16 weight);
```

## Purpose

Change the weight of a fault tolerance member within its group.

## Remarks

Weight summarizes the relative suitability of a member for its task, relative to other members of the same fault tolerance group. That suitability is a combination of computer speed and load factors, network bandwidth, computer and network reliability, and other factors. Programs may reset their weight when any of these factors change, overriding the previous assigned weight.

You can use relative weights to indicate priority among group members.

Zero is a special value; Rendezvous fault tolerance software assigns zero weight to processes with resource errors, so they only activate when no other members are available. Programs must always assign weights greater than zero.

When Rendezvous fault tolerance software requests a resource but receives an error (for example, the member process cannot allocate memory, or start a timer), it attempts to send the member process a [DISABLING\\_MEMBER](#) advisory message, and sets the member's weight to zero, effectively disabling the member. Weight zero implies that this member is active only as a last resort—when no other members outrank it. (However, if the disabled member process does become active, it might not operate correctly.)

Parameter	Description
weight	The new weight value. See <a href="#">weight</a> .

## See Also

Adjusting Member Weights in TIBCO Rendezvous Concepts.



# TibrvFtMemberCallback

*Class*

## Declaration

```
class TibrvFtMemberCallback
```

## Purpose

Process fault tolerance events for a group member.

## Remarks

Implement this interface to process fault tolerance events.

Method	Description
<a href="#">TibrvFtMemberCallback::onFtAction()</a>	Process fault tolerance events for a group member.

## See Also

[TibrvFtMember::create\(\)](#)

# TibrvFtMemberCallback::onFtAction()

*Method*

## Declaration

```
virtual void onFtAction(  
    TibrvFtMember* ftMember,  
    const char* groupName,  
    tibrvftAction action ) = 0;
```

## Purpose

Process fault tolerance events for a group member.

## Remarks

Each member program of a fault tolerance group must implement this method. Programs register a member callback object (and this method) with each call to [TibrvFtMember::create\(\)](#).

Rendezvous fault tolerance software queues a member action event in three situations. In each case, it passes a different action argument, instructing the callback method to activate, deactivate, or prepare to activate the program.

- When the number of active members drops below the active goal, the fault tolerance callback method (in the ranking inactive member process) receives the token [TIBRVFT\\_ACTIVATE](#); the callback method must respond by assuming the duties of an active member.
- When the number of active members exceeds the active goal, the fault tolerance callback method (in any active member that is outranked by another active member) receives the action token [TIBRVFT\\_DEACTIVATE](#); the callback method must respond by switching the program to its inactive state.
- When the number of active members equals the active goal, and Rendezvous fault tolerance software detects that it might soon decrease below the active goal, the fault tolerance callback method (in the ranking inactive member) receives the action token [TIBRVFT\\_PREPARE\\_TO\\_ACTIVATE](#); the callback method must respond by making the program ready to activate immediately. For example, preparatory steps

might include time-consuming tasks such as connecting to a database. If the callback method subsequently receives the [TIBRVFT\\_ACTIVATE](#) token, it will be ready to activate without delay.

This token is a hint, indicating that the program might soon receive an instruction to activate. It does not guarantee that an activate instruction will follow, nor that any minimum time will elapse before an activate instruction follows.

For additional information see Fault Tolerance Callback Actions in TIBCO Rendezvous Concepts.

Parameter	Description
ftMember	This parameter receives the member object.
groupName	This parameter receives a string denoting the name of the fault tolerance group.
action	This parameter receives a token that instructs the callback method to activate, deactivate or prepare to activate. See <a href="#">tibrvftAction</a> .

## See Also

[TibrvFtMember::create\(\)](#)

Fault Tolerance Callback Actions in TIBCO Rendezvous Concepts

# TibrvFtMemberOnComplete

*Class*

## Declaration

```
class TibrvFtMemberOnComplete
```

## Purpose

A program can destroy a member object even when its callback method is running in one or more threads. Multi-threaded programs can define a subclass of this interface class to discover when all callback methods in progress have completed.

Method	Description
<a href="#">TibrvFtMemberOnComplete::onComplete</a>	A program can destroy a member object even when its callback method is running in one or more threads. Multi-threaded programs can define methods of this type to discover when all callback methods in progress have completed.

## See Also

[TibrvFtMember::create\(\)](#)

# TibrvFtMemberOnComplete::onComplete

Method

## Declaration

```
virtual void onComplete(  
    TibrvFtMember* ftMember) = 0;
```

## Purpose

A program can destroy a member object even when its callback method is running in one or more threads. Multi-threaded programs can define methods of this type to discover when all callback methods in progress have completed.

Parameter	Description
ftMember	<p>This parameter receives the member event object. This object is identical to the object that the program created to join the fault tolerance group.</p> <p>However, by the time this method runs, the member is already destroyed; this method cannot use the member object in Rendezvous calls.</p>

## Remarks

This type of method is important in two situations:

- Internal fault tolerance callback methods run in several threads (because several threads dispatch the member's event queue), and the program must do additional processing after these callback methods have completed *in all threads*.
- A member callback method calls [TibrvFtMember::destroy\(\)](#) to withdraw from a fault tolerance group, and the program must do additional processing *after* the rest of the callback method has completed.

Upon return from [TibrvFtMember::destroy\(\)](#), the destroyed member's callback method can no longer begin to run (this is also true of internal callback methods). However, in each thread where a callback method is already in progress, that callback method does continue to run until complete.

[TibrvFtMember::destroy\(\)](#) accepts a *completion argument* of type [TibrvFtMemberOnComplete](#). Rendezvous software ensures that the completion method runs when the last callback-in-progress has completed.

## Timing and Context

This information is completely analogous to [TibrvEventOnComplete::onComplete\(\)](#). See that section for important details.

## See Also

[TibrvFtMember::create\(\)](#)

# TibrvFtMonitor

*Class*

## Declaration

```
class TibrvFtMonitor
    TibrvFtmonitor();      // Create empty.
    virtual ~TibrvFtmonitor(); // Destroy and reclaim storage.
```

## Purpose

Monitor a fault tolerance group.

## Remarks

The constructor creates a hollow object; [TibrvFtMonitor::create\(\)](#) makes it operational, and monitors a fault tolerance group.

Monitors are passive—they do not affect the group members in any way.

Rendezvous fault tolerance software queues a monitor event whenever the number of active members in the group changes—either it detects a new heartbeat, or it detects that the heartbeat from a previously active member is now silent, or it receives a message from the fault tolerance component of an active member indicating deactivation or termination.

The monitor callback method receives the number of active members as an argument.

By destroying a monitor object, the program stops monitoring the fault tolerance group. The destructor calls the `destroy` method, unless the C object is already destroyed.

Destroying the queue or transport of a monitor automatically destroys the monitor as well.

Method	Description
--------	-------------

## Life Cycle

<a href="#">TibrvFtMonitor::create()</a>	<a href="#">Monitor a fault tolerance group.</a>
--	--

Method	Description
<a href="#">TibrvFtMonitor::destroy()</a>	Stop monitoring a fault tolerance group, and free associated resources.
<a href="#">TibrvFtMonitor::getHandle()</a>	Extract the C handle of a fault tolerance monitor.
<a href="#">TibrvFtMonitor::isValid()</a>	Test validity of a fault tolerance monitor object.
<b>Properties</b>	
<a href="#">TibrvFtMonitor::getClosure()</a>	Extract the closure data of a fault tolerance monitor.
<a href="#">TibrvFtMonitor::getGroupName()</a>	Extract the group name of a fault tolerance monitor.
<a href="#">TibrvFtMonitor::getQueue()</a>	Extract the event queue of a fault tolerance monitor.
<a href="#">TibrvFtMonitor::getTransport()</a>	Extract the transport of a fault tolerance monitor.

## Related Classes

[TibrvFtMember](#)

## See Also

[TibrvFtMonitorCallback](#)



# TibrvFtMonitor::create()

*Method*

## Declaration

```
TibrvStatus create(  
    TibrvQueue* queue,  
    TibrvFtMonitorCallback* callback,  
    TibrvTransport* transport,  
    const char* groupName,  
    tibrv_f64 lostInterval,  
    const void* closure = NULL);
```

## Purpose

Monitor a fault tolerance group.

## Remarks

The monitor callback method receives the number of active members as an argument.

The group need not have any members at the time of this create call.

Parameter	Description
queue	Place events for this monitor on this event queue.
callback	On dispatch, process the event with this callback method.
transport	Listen on this transport for fault tolerance internal protocol messages (such as heartbeat messages).
groupName	Monitor the fault tolerant group with this name.  The group name must conform to the syntax required for Rendezvous subject names. For details, see Subject

Parameter	Description
	Names in TIBCO Rendezvous Concepts.  See also, <a href="#">Group Name</a> .
lostInterval	When the heartbeat signal from an active member has been silent for this interval (in seconds), Rendezvous fault tolerance software considers that member lost, and queues a monitor event.  The interval must be positive. To determine the correct value, see Step 4: Choose the Intervals in TIBCO Rendezvous Concepts.  See also, <a href="#">Lost Interval</a> .
closure	Store this closure data in the monitor object.

## Lost Interval

The monitor uses the lostInterval to determine whether a member is still active. When the heartbeat signal from an active member has been silent for this interval (in seconds), the monitor considers that member lost, and queues a monitor event.

We recommend setting the lostInterval identical to the group's activationInterval, so the monitor accurately reflects the behavior of the group members.

## Group Name

The group name must be a legal Rendezvous subject name (see Subject Names in TIBCO Rendezvous Concepts). You may use names with several elements; for examples, see Multiple Groups in TIBCO Rendezvous Concepts.

## See Also

[TibrvFtMonitorCallback](#).

[TibrvFtMonitor::destroy\(\)](#).

# TibrvFtMonitor::destroy()

*Method*

## Declaration

```
TibrvStatus destroy(  
    TibrvFtMonitorOnComplete* completeCB=NULL);
```

## Purpose

Stop monitoring a fault tolerance group, and free associated resources.

## Remarks

This method returns an error when the monitor object is already invalid, or when its queue or transport are invalid.

## See Also

[TibrvFtMonitor::create\(\)](#)

# TibrvFtMonitor::getClosure()

*Method*

## Declaration

```
void* getClosure() const;
```

## Purpose

Extract the closure data of a fault tolerance monitor.

## See Also

[TibrvFtMonitor::create\(\)](#)

# TibrvFtMonitor::getGroupName()

*Method*

## Declaration

```
TibrvStatus getGroupName(  
    const char*& groupName) const;
```

## Purpose

Extract the group name of a fault tolerance monitor.

Parameter	Description
groupName	The program supplies a variable, and the method stores the group name in that variable.

## See Also

[TibrvFtMonitor::create\(\)](#)

# TibrvFtMonitor::getHandle()

*Method*

## Declaration

```
tibrvftMonitor getHandle() const;
```

## Purpose

Extract the C handle of a fault tolerance monitor.

## See Also

[TibrvFtMonitor::create\(\)](#)

tibrvftMonitor in TIBCO Rendezvous C Reference

# TibrvFtMonitor::getQueue()

*Method*

## Declaration

```
TibrvQueue* getQueue() const;
```

## Purpose

Extract the event queue of a fault tolerance monitor.

## Remarks

If the monitor is invalid, this method returns NULL.

## See Also

[TibrvQueue](#)

[TibrvFtMonitor::create\(\)](#)

# TibrvFtMonitor::getTransport()

*Method*

## Declaration

```
TibrvTransport* getTransport() const;
```

## Purpose

Extract the transport of a fault tolerance monitor.

## See Also

[TibrvTransport](#)

[TibrvFtMonitor::create\(\)](#)



# TibrvFtMonitor::isValid()

*Method*

## Declaration

```
tibrv_bool isValid() const;
```

## Purpose

Test validity of a fault tolerance monitor object.

## Remarks

Returns TIBRV\_TRUE if the C monitor is valid; TIBRV\_FALSE if the monitor has been destroyed or is otherwise invalid.

## See Also

[TibrvFtMonitor::destroy\(\)](#)

# TibrvFtMonitorCallback

*Class*

## Declaration

```
class TibrvFtMonitorCallback
```

## Purpose

Process fault tolerance events for a monitor.

## Remarks

Implement this interface to process fault tolerance monitor events.

Method	Description
<a href="#">TibrvFtMonitorCallback::onFtMonitor()</a>	Process fault tolerance events for a monitor.

## See Also

[TibrvFtMonitor::create\(\)](#)

# TibrvFtMonitorCallback::onFtMonitor()

## Declaration

```
virtual void onFtMonitor(  
    TibrvFtMonitor* ftMonitor,  
    const char* groupName,  
    tibrv_u32 numActiveMembers ) = 0;
```

## Purpose

Process fault tolerance events for a monitor.

## Remarks

A program must define a method of this type as a prerequisite to monitor a fault tolerance group. Programs register a monitor callback method with each call to [TibrvFtMonitor::create\(\)](#).

Rendezvous fault tolerance software queues a monitor event whenever the number of active members in the group changes.

A program need not be a member of a group in order to monitor that group.

Parameter	Description
ftMonitor	This parameter receives the monitor object.
groupName	This parameter receives a string denoting the name of the fault tolerance group.
numActiveMembers	This parameter receives the number of group members now active.

## See Also

[TibrvFtMonitor::create\(\)](#).

# TibrvFtMonitorOnComplete

*Class*

## Declaration

```
class TibrvFtMonitorOnComplete
```

## Purpose

A program can destroy a monitor object even when its callback method is running in one or more threads. Multi-threaded programs can define a subclass of this interface class to discover when all callback methods in progress have completed.

Method	Description
<a href="#">TibrvFtMonitorOnComplete::onComplete</a>	A program can destroy a monitor object even when its callback method is running in one or more threads. Multi-threaded programs can define methods of this type to discover when all callback methods in progress have completed.

## See Also

[TibrvFtMonitor::create\(\)](#)

# TibrvFtMonitorOnComplete::onComplete

Method

## Declaration

```
virtual void onComplete(  
    TibrvFtMonitor* ftMonitor) = 0;
```

## Purpose

A program can destroy a monitor object even when its callback method is running in one or more threads. Multi-threaded programs can define methods of this type to discover when all callback methods in progress have completed.

Parameter	Description
ftMonitor	<p>This parameter receives the monitor event object. This object is identical to the object that the program created to begin monitoring the fault tolerance group.</p> <p>However, by the time this method runs, the monitor is already destroyed; this method cannot use the monitor object in Rendezvous calls.</p>

## Remarks

This type of method is important in two situations:

- Internal fault tolerance callback methods run in several threads (because several threads dispatch the monitor's event queue), and the program must do additional processing after these callback methods have completed *in all threads*.
- A member callback method calls [TibrvFtMonitor::destroy\(\)](#) to stop monitoring a fault tolerance group, and the program must do additional processing *after* the rest of the callback method has completed.

Upon return from [TibrvFtMonitor::destroy\(\)](#), the destroyed monitor's callback method can no longer begin to run (this is also true of internal callback methods). However, in each thread where a callback method is already in progress, that callback method does continue to run until complete.

[TibrvFtMonitor::destroy\(\)](#) accepts a *completion argument* of type [TibrvFtMonitorOnComplete](#). Rendezvous software ensures that the completion method runs when the last callback-in-progress has completed.

## Timing and Context

This information is completely analogous to [TibrvEventOnComplete::onComplete\(\)](#). See that section for important details.

## See Also

[TibrvFtMonitor::create\(\)](#)

# Certified Message Delivery

---

Although Rendezvous communications are highly reliable, some applications require even stronger assurances of delivery. Certified delivery features offers greater certainty of delivery—even in situations where processes and their network connections are unstable.

## See Also

This API implements Rendezvous certified delivery features. For a complete discussion, see [Certified Message Delivery](#) in TIBCO Rendezvous Concepts.

Certified delivery software uses advisory messages extensively. For example, advisories inform sending and receiving programs of the delivery status of each message. For complete details, see [Certified Message Delivery \(RVCM\) Advisory Messages](#) in TIBCO Rendezvous Concepts.

If your application sends or receives certified messages across network boundaries, you must configure the Rendezvous routing daemons to exchange `_RVCM` administrative messages. For details, see [Certified Message Delivery](#) in TIBCO Rendezvous Administration.

Some programs require certified delivery to *one of  $n$*  worker processes. See [Distributed Queue](#) in TIBCO Rendezvous Concepts.

# TibrvCmListener

*Class*

## Declaration

```
class TibrvCmListener : TibrvEvent
    TibrvCmListener();           // Create empty.
    virtual ~TibrvCmListener(); // Destroy and reclaim storage.
```

## Purpose

A certified delivery listener object listens for labeled messages and certified messages.

## Remarks

The constructor creates a hollow object; [TibrvCmListener::create\(\)](#) makes it operational—starting a C certified delivery listener, which represents your program’s listening interest in a stream of labeled messages and certified messages. Rendezvous software uses the same listener object to signal each occurrence of such an event.

We recommend that programs explicitly destroy each certified delivery listener object using [TibrvCmListener::destroy\(\)](#). Destroying a certified listener object cancels the program’s immediate interest in that event; nonetheless, a parameter to the destroy call determines whether certified delivery agreements continue to persist beyond the destroy call.

The destructor calls the `destroy()` method, unless the C object is already destroyed.

Destroying the queue, the certified delivery transport, or the transport of a listener object automatically invalidates the listener as well (but certified delivery agreements continue to persist).

Method	Description
--------	-------------

## Life Cycle

<a href="#">TibrvCmListener::create()</a>	<a href="#">Listen for messages that match the</a>
---	--



Method	Description
	subject, and request certified delivery when available.
<a href="#">TibrvCmListener::destroy()</a>	Destroy a certified delivery listener.
<a href="#">TibrvCmListener::isValid()</a>	Test whether a certified delivery listener has been destroyed.

## Confirmation

<a href="#">TibrvCmListener::confirmMsg()</a>	Explicitly confirm delivery of a certified message.
<a href="#">TibrvCmListener::setExplicitConfirm()</a>	Override automatic confirmation of delivery for this listener.

## Properties

<a href="#">TibrvCmListener::getSubject()</a>	Extract the subject from a certified delivery listener.
<a href="#">TibrvCmListener::getTransport()</a>	Extract the transport from a certified delivery listener.

## Inherited Methods

[TibrvEvent::destroy\(\)](#)  
[TibrvEvent::getClosure\(\)](#)  
[TibrvEvent::getHandle\(\)](#)  
[TibrvEvent::getType\(\)](#)  
[TibrvEvent::getQueue\(\)](#)  
[TibrvEvent::isValid\(\)](#)  
[TibrvEvent::isListener\(\)](#)

### Inherited Methods

---

[TibrvEvent::isTimer\(\)](#)

[TibrvEvent::isIOEvent\(\)](#)

---

## Related Classes

[TibrvEvent](#)

[TibrvListener](#)

# TibrvCmListener::confirmMsg()

*Method*

## Declaration

```
TibrvStatus confirmMsg(  
    const TibrvMsg& msg);
```

## Purpose

Explicitly confirm delivery of a certified message.

## Remarks

Use this method only in programs that override automatic confirmation (see [TibrvCmListener::setExplicitConfirm\(\)](#)). The default behavior of certified listeners is to automatically confirm delivery when the callback method returns.

Parameter	Description
message	Confirm receipt of this message.

## Unregistered Message

When a CM listener receives a labeled message, its behavior depends on context:

- If a CM listener is registered for certified delivery, it presents the supplementary information to the callback method. If the sequence number is present, then the receiving program can confirm delivery.
- If a CM listener is *not* registered for certified delivery with the sender, it presents the sender's name to the callback method, but omits the sequence number. In this case, the receiving program cannot confirm delivery; [TibrvCmListener::confirmMsg\(\)](#) returns the status code [TIBRV\\_NOT\\_PERMITTED](#).

Notice that the first labeled message that a program receives on a subject might not be certified; that is, the sender has not registered a certified delivery agreement with the listener. If appropriate, the certified delivery library automatically requests that

the sender register the listener for certified delivery. (See Discovery and Registration for Certified Delivery in TIBCO Rendezvous Concepts.)

A labeled but uncertified message can also result when the sender explicitly disallows or removes the listener.

## See Also

[TibrvCmListener](#)

[TibrvCmListener::setExplicitConfirm\(\)](#)

# TibrvCmListener::create()

*Method*

## Declaration

```
TibrvStatus create(  
    TibrvQueue* queue,  
    TibrvCmMsgCallback* cmMsgCallback,  
    TibrvCmTransport* cmTransport,  
    const char* subject,  
    const void* closure = NULL);
```

## Purpose

Listen for messages that match the subject, and request certified delivery when available.

Parameter	Description
queue	For each inbound message, place the listener event on this event queue.
cmMsgCallback	On dispatch, process the event with this callback object.
cmTransport	Listen for inbound messages on this certified delivery transport.
subject	Listen for inbound messages with subjects that match this specification. Wildcard subjects are permitted. The empty string is <i>not</i> a legal subject name.
closure	Store this closure data in the event object.

## Activation and Dispatch

Details of listener event semantics are identical to those for ordinary listeners; see [Activation and Dispatch](#).

## Inbox Listener

To receive unicast (point-to-point) messages, listen to a unique inbox subject name. First call [TibrvTransport::createInbox\(\)](#) to create the unique inbox name; then call [TibrvListener::create\(\)](#) to begin listening. Remember that other programs have no information about an inbox until the listening program uses it as a reply subject in an outbound message.

## See Also

[TibrvCmListener](#)

[TibrvCmTransport::destroy\(\)](#)

[TibrvListener::getSubject\(\)](#)

[TibrvTransport::createInbox\(\)](#)

# TibrvCmListener::destroy()

*Method*

## Declaration

```
TibrvStatus destroy(  
    tibrv_bool cancelAgreements,  
    TibrvEventOnComplete* completeCB = NULL);
```

## Purpose

Destroy a certified delivery listener.

Parameter	Description
cancelAgreements	<p>TIBRV_TRUE cancels all certified delivery agreements of this listener; certified senders delete from their ledgers all messages sent to this listener.</p> <p>TIBRV_FALSE leaves all certified delivery agreements in effect, so certified senders continue to store messages.</p>

## Canceling Agreements

When destroying a certified delivery listener, a program can either cancel its certified delivery agreements with senders, or let those agreements persist (so a successor listener can receive the messages covered by those agreements).

When canceling agreements, each (previously) certified sender transport receives a [REGISTRATION.CLOSED](#) advisory. Successor listeners cannot receive old messages.

## See Also

[TibrvEventOnComplete](#)

[TibrvCmListener](#)

# TibrvCmListener::getSubject()

*Method*

## Declaration

```
TibrvStatus getSubject(  
    const char*& subject) const;
```

## Purpose

Extract the subject from a certified delivery listener.

Parameter	Description
subject	<p>The program supplies a variable. The method stores the subject of the listener event object in that variable.</p> <p>The program must not modify this string nor free its storage.</p>

## See Also

[TibrvCmListener](#)



# TibrvCmListener::getTransport()

*Method*

## Declaration

```
TibrvCmTransport* getTransport() const;
```

## Purpose

Extract the transport from a certified delivery listener.

## See Also

[TibrvCmListener](#)

# TibrvCmListener::isValid()

*Method*

## Declaration

```
tibrv_bool isValid() const;
```

## Purpose

Test whether a certified delivery listener has been destroyed.

## Remarks

This method returns TIBRV\_TRUE if the listener is valid, and TIBRV\_FALSE if it has been destroyed.

Notice that [TibrvCmListener::destroy\(\)](#) invalidates the event immediately, even though active callback methods may continue to run.

## See Also

[TibrvCmListener::destroy\(\)](#)

# TibrvCmListener::setExplicitConfirm()

*Method*

## Declaration

```
TibrvStatus setExplicitConfirm();
```

## Purpose

Override automatic confirmation of delivery for this listener.

## Remarks

The default behavior of certified listeners is to automatically confirm delivery when the callback method returns (see [TibrvMsgCallback::onMsg\(\)](#)). This call selectively overrides this behavior for this specific listener (without affecting other listeners).

By overriding automatic confirmation, the listener assumes responsibility to explicitly confirm each inbound certified message by calling [TibrvCmListener::confirmMsg\(\)](#).

Consider overriding automatic confirmation when the processing of inbound messages involves activity that is asynchronous with respect to the message callback method; for example, computations in other threads or additional network communications.

No method exists to restore the default behavior—that is, to reverse the effect of this method.

## See Also

[TibrvCmListener](#)

[TibrvMsgCallback::onMsg\(\)](#)

[TibrvCmListener::confirmMsg\(\)](#)

# TibrvCmTransport

*Class*

## Declaration

```
class TibrvCmTransport : public TibrvTransport
```

## Purpose

A certified delivery transport object implements the CM delivery protocol for messages.

## Remarks

Each certified delivery transport employs a [TibrvTransport](#) for network communications. The [TibrvCmTransport](#) adds the accounting mechanisms needed for delivery tracking and certified delivery.

Several [TibrvCmTransport](#) objects can employ a [TibrvTransport](#), which also remains available for its own ordinary listeners and for sending ordinary messages.

The constructor creates a hollow object; [TibrvCmTransport::create\(\)](#) makes it operational.

The destructor calls the destroy method, unless the C object is already destroyed. Programs must explicitly destroy each certified delivery transport object. Destroying a certified delivery transport object invalidates subsequent certified send calls on that object, invalidates any certified listeners using that transport (while preserving the certified delivery agreements of those listeners).

Method	Description
<a href="#">TibrvCmTransport::create()</a>	Create a transport for certified delivery.
<a href="#">TibrvCmTransport::addListener()</a>	Pre-register an anticipated listener.

Method	Description
<a href="#">TibrvCmTransport::allowListener()</a>	Invite the named receiver to reinstate certified delivery for its listeners, superseding the effect of any previous disallow calls.
<a href="#">TibrvCmTransport::destroy()</a>	Destroy a certified delivery transport.
<a href="#">TibrvCmTransport::disallowListener()</a>	Cancel certified delivery to all listeners at a specific correspondent. Deny subsequent certified delivery registration requests from those listeners.
<a href="#">TibrvCmTransport::getDefaultTimeLimit()</a>	Get the default message time limit for all outbound certified messages from a transport.
<a href="#">TibrvCmTransport::getLedgerName()</a>	Extract the ledger name of a certified delivery transport.
<a href="#">TibrvCmTransport::getName()</a>	Extract the correspondent name of a certified delivery transport or distributed queue member.
<a href="#">TibrvCmTransport::getRequestOld()</a>	Extract the request old messages flag of a certified delivery transport.
<a href="#">TibrvCmTransport::getSyncLedger()</a>	Extract the sync ledger flag of a certified delivery transport.
<a href="#">TibrvCmTransport::getTransport()</a>	Extract the transport employed by a certified

Method	Description
	delivery transport or a distributed queue member.
<a href="#">TibrvCmTransport::removeListener()</a>	Unregister a specific listener at a specific correspondent, and free associated storage in the sender's ledger.
<a href="#">TibrvCmTransport::removeSendState()</a>	Reclaim ledger space from obsolete subjects.
<a href="#">TibrvCmTransport::reviewLedger()</a>	Query the ledger for stored items related to a subject name.
<a href="#">TibrvCmTransport::send()</a>	Send a labeled message.
<a href="#">TibrvCmTransport::sendReply()</a>	Send a labeled reply message.
<a href="#">TibrvCmTransport::sendRequest()</a>	Send a labeled request message and wait for a reply.
<a href="#">TibrvCmTransport::setDefaultTimeLimit()</a>	Set the default message time limit for all outbound certified messages from a transport.
<a href="#">TibrvCmTransport::setPublisherInactivityDiscardInterval()</a>	Set a time limit after which a listening CM transport can discard state for inactive CM senders.
<a href="#">TibrvCmTransport::syncLedger()</a>	Synchronize the ledger to its storage medium.

#### Inherited Methods

[TibrvTransport::createInbox\(\)](#)

## Inherited Methods

[TibrvTransport::destroy\(\)](#)

[TibrvTransport::isValid\(\)](#)

[TibrvTransport::getHandle\(\)](#)

[TibrvTransport::send\(\)](#)

[TibrvTransport::sendReply\(\)](#)

[TibrvTransport::sendRequest\(\)](#)

[TibrvTransport::setDescription\(\)](#)

## Related Classes

[TibrvTransport](#)

[TibrvNetTransport](#)

[TibrvCmQueueTransport](#)

# TibrvCmTransport::addListener()

*Method*

## Declaration

```
TibrvStatus addListener(  
    const char* cmName,  
    const char* subject);
```

## Purpose

Pre-register an anticipated listener.

## Remarks

Some sending programs can anticipate requests for certified delivery—even before the listening programs actually register. In such situations, the sending transport can pre-register listeners, so Rendezvous software begins storing outbound messages in the sender's ledger; when the listener requests certified delivery, it receives the backlogged messages.

If the correspondent with this cmName already receives certified delivery of this subject from this sender transport, then [TibrvCmTransport::addListener\(\)](#) has no effect.

If the correspondent with this cmName is disallowed, [TibrvCmTransport::addListener\(\)](#) returns the status [TIBRV\\_NOT\\_PERMITTED](#). You can call [TibrvCmTransport::allowListener\(\)](#) to supersede the effect of a prior call to [TibrvCmTransport::disallowListener\(\)](#); then call [TibrvCmTransport::addListener\(\)](#) again.

It is not sufficient for a sender to use this method to anticipate listeners; the anticipated listening programs must also require old messages when creating certified delivery transports.

Parameter	Description
cmName	Anticipate a listener from a correspondent with this reusable name.



Parameter	Description
subject	Anticipate a listener for this subject. Wildcard subjects are illegal.

## See Also

[Name](#)

[TibrvCmTransport::allowListener\(\)](#)

[TibrvCmTransport::disallowListener\(\)](#)

[TibrvCmTransport::removeListener\(\)](#)

[Anticipating a Listener in TIBCO Rendezvous Concepts](#)

# TibrvCmTransport::allowListener()

*Method*

## Declaration

```
TibrvStatus allowListener(  
    const char* cmName);
```

## Purpose

Invite the named receiver to reinstate certified delivery for its listeners, superseding the effect of any previous *disallow* calls.

## Remarks

Upon receiving the invitation to reinstate certified delivery, Rendezvous software at the listening program automatically sends new registration requests. The sending program accepts these requests, restoring certified delivery.

Parameter	Description
cmName	Accept requests for certified delivery to listeners at the transport with this correspondent name.

## See Also

[Name](#)

[TibrvCmTransport::disallowListener\(\)](#)

Disallowing Certified Delivery in TIBCO Rendezvous Concepts

# TibrvCmTransport::create()

*Method*

## Declaration

```
TibrvStatus create(  
    TibrvTransport* transport);  
TibrvStatus create(  
    TibrvTransport* transport,  
    const char* cmName,  
    tibrv_bool requestOld,  
    const char* ledgerName = NULL,  
    tibrv_bool syncLedger = TIBRV_FALSE);
```

## Purpose

Create a transport for certified delivery.

## Remarks

This method creates a C certified delivery transport and stores its handle in the C++ object.

The new certified delivery transport must employ a valid transport for network communications.

The certified delivery transport remains valid until the program explicitly destroys it.

Parameter	Description
transport	<p>The new <a href="#">TibrvCmTransport</a> employs this transport object for network communications.</p> <p>Despite the declaration as a <a href="#">TibrvTransport</a>, this object must be an instance of <a href="#">TibrvNetTransport</a>. In particular, it cannot be the <a href="#">TibrvProcessTransport</a> nor <a href="#">TibrvVcTransport</a>.</p> <p>Destroying the <a href="#">TibrvCmTransport</a> does not affect this <a href="#">TibrvTransport</a> object.</p>

Parameter	Description
cmName	<p>Bind this reusable name to the new <a href="#">TibrvCmTransport</a>, so the <a href="#">TibrvCmTransport</a> represents a persistent correspondent with this name.</p> <p>If non-NULL, the name must conform to the syntax rules for Rendezvous subject names. It cannot begin with reserved tokens. It cannot be a non-reusable name generated by another call to <a href="#">TibrvCmTransport::create()</a>. It cannot be the empty string.</p> <p>If omitted or NULL, then <a href="#">TibrvCmTransport::create()</a> generates a unique, non-reusable name for the duration of the transport.</p> <p>For more information, see Name.</p>
requestOld	<p>This parameter indicates whether a persistent correspondent requires delivery of messages sent to a previous certified delivery transport with the same name, for which delivery was not confirmed. Its value affects the behavior of other CM sending transports.</p> <p>If this parameter is <code>TIBRV_TRUE</code> and cmName is non-NULL, then the new <a href="#">TibrvCmTransport</a> requires certified senders to retain unacknowledged messages sent to this persistent correspondent. When the new <a href="#">TibrvCmTransport</a> begins listening to the appropriate subjects, the senders can complete delivery. (It is an error to supply true when cmName is NULL.)</p> <p>If this parameter is <code>TIBRV_FALSE</code> (or omitted), then the new <a href="#">TibrvCmTransport</a> does not require certified senders to retain unacknowledged messages. Certified senders may delete those messages from their ledgers.</p>
ledgerName	<p>If this argument is non-NULL, then the new <a href="#">TibrvCmTransport</a> uses a file-based ledger. The argument must represent a valid file name. Actual locations corresponding to relative file names conform to operating system conventions. We strongly discourage using the empty string as a ledger file name.</p> <p>If omitted or NULL, then the new <a href="#">TibrvCmTransport</a> uses a process-based ledger.</p> <p>For more information, see Ledger File.</p>

Parameter	Description
syncLedger	<p>If this argument is <code>TIBRV_TRUE</code>, then operations that update the ledger file do not return until the changes are written to the storage medium.</p> <p>If this argument is <code>TIBRV_FALSE</code> (or omitted), the operating system writes changes to the storage medium asynchronously.</p>

## Method Forms

With only a transport, create a transient correspondent, with a unique, non-reusable name. (Supplying `NULL` as the `cmName` has the same effect.)

All other parameters are optional, with default values when omitted.

## Name

If `cmName` is `NULL`, then [TibrvCmTransport::create\(\)](#) generates a unique, non-reusable name for the new certified delivery transport.

If `cmName` is non-`NULL`, then the new transport binds that name. A correspondent can persist beyond transport destruction only when it has *both* a reusable name *and* a file-based ledger.

For more information about the use of reusable names, see [CM Correspondent Name in TIBCO Rendezvous Concepts](#), and [Persistent Correspondents in TIBCO Rendezvous Concepts](#). For details of reusable name syntax, see [Reusable Names in TIBCO Rendezvous Concepts](#).

## Ledger File

Every certified delivery transport stores the state of its certified communications in a ledger.

If `ledgerFile` is `NULL`, then the new transport stores its ledger exclusively in process-based storage. When you destroy the transport or the process terminates, all information in the ledger is lost.

If `ledgerFile` specifies a valid file name, then the new transport uses that file for ledger storage. If the transport is destroyed or the process terminates with incomplete certified communications, the ledger file records that state. When a new transport binds the same

reusable name, it reads the ledger file and continues certified communications from the state stored in the file.

Even though a transport uses a ledger file, it may sometimes replicate parts of the ledger in process-based storage for efficiency; however, programmers cannot rely on this replication.

The `syncLedger` parameter determines whether writing to the ledger file is a synchronous operation:

- To specify synchronous writing, supply `TIBRV_TRUE`. Each time Rendezvous software writes a ledger item, the call does not return until the data is safely stored in the storage medium.
- To specify asynchronous writing (the default), supply `TIBRV_FALSE`. Certified delivery calls may return before the data is safely stored in the storage medium, which results in greater speed at the cost of certainty. The ledger file might not accurately reflect program state in cases of hardware or operating system kernel failure (but it is accurate in cases of sudden program failure). Despite this small risk, we strongly recommend this option for maximum performance.

A program that uses an asynchronous ledger file can explicitly synchronize it by calling [TibrvCmTransport::syncLedger\(\)](#).

Destroying a transport with a file-based ledger always leaves the ledger file intact; it neither erases nor removes a ledger file.

The ledger file must reside on the same host computer as the program that uses it.

## See Also

[TibrvCmTransport::destroy\(\)](#)

# TibrvCmTransport::createInbox()

*Method*

## Declaration

```
TibrvStatus createInbox(  
    char* subjectString,  
    tibrv_u32 subjectLimit) const;
```

## Purpose

Create a unique inbox subject name.

## Remarks

This convenience method extracts the network transport, and then calls [TibrvTransport::createInbox\(\)](#). For details, see the documentation for that method.

Parameter	Description
subjectString	The program supplies a string buffer, and the method stores the new inbox subject string in that buffer.
subjectLimit	The number of bytes that the program has allocated to receive the new inbox subject string.

## See Also

[TibrvMsg::setReplySubject\(\)](#)

[TibrvTransport::createInbox\(\)](#)

# TibrvCmTransport::destroy()

*Method*

## Declaration

```
TibrvStatus destroy();  
TibrvStatus destroyEx(  
    TibrvCmTransportOnComplete* completeCB = NULL);
```

## Purpose

Destroy a certified delivery transport.

## Remarks

Destroying a certified delivery transport with a file-based ledger always leaves the ledger file intact; it neither erases nor removes a ledger file.

Parameter	Description
completeCB	<p>Rendezvous software runs this completion callback immediately after all queued tasks are complete.</p> <p>Do not destroy the transport's listeners until after this callback signals that cleanup has completed.</p> <p>See <a href="#">TibrvCmTransportOnComplete::onComplete</a>.</p>

## Distributed Queue

To destroy a distributed queue transport, call `destroyEx()`. With the ordinary `destroy` call, the distributed queue can lose reliable (non-certified) task messages before they are processed. The distributed queue needs the listeners, queues and dispatchers (associated with the transport) to remain operational—programs must wait until after the transport has been completely destroyed before destroying these associated objects.



## See Also

[TibrvCmTransport::create\(\)](#)

[TibrvCmTransportOnComplete](#)

# TibrvCmTransport::disallowListener()

*Method*

## Declaration

```
TibrvStatus disallowListener(  
    const char* cmName);
```

## Purpose

Cancel certified delivery to all listeners at a specific correspondent. Deny subsequent certified delivery registration requests from those listeners.

## Remarks

Disallowed listeners still receive subsequent messages from this sender, but delivery is not certified. In other words:

- The first labeled message causes the listener to initiate registration. Registration fails, and the listener discards that labeled message.
- The listener receives a [REGISTRATION.NOT\\_CERTIFIED](#) advisory, informing it that the sender has canceled certified delivery of all subjects.
- If the sender's ledger contains messages sent to the disallowed listener (for which this listener has not confirmed delivery), then Rendezvous software removes those ledger items, and does not attempt to redeliver those messages.
- Rendezvous software presents subsequent messages (from the canceling sender) to the listener without a sequence number, to indicate that delivery is not certified.

Senders can promptly revoke the acceptance of certified delivery by calling [TibrvCmTransport::disallowListener\(\)](#) within the callback method that processes the [REGISTRATION.REQUEST](#) advisory.

This method disallows a correspondent by name. If the correspondent terminates, and another process instance (with the same reusable name) takes its place, the new process is still disallowed by this sender.

To supersede the effect of [TibrvCmTransport::disallowListener\(\)](#), call [TibrvCmTransport::allowListener\(\)](#).

Parameter	Description
cmName	Cancel certified delivery to listeners of the transport with this name.

## See Also

[Name](#)

[TibrvCmTransport::allowListener\(\)](#)

Disallowing Certified Delivery in TIBCO Rendezvous Concepts

# TibrvCmTransport::expireMessages()

*Method*

## Declaration

```
TibrvStatus expireMessages(  
    const char*  subject,  
    tibrv_u64    sequenceNumber);
```

## Purpose

Mark specified outbound CM messages as expired.

## Remarks

This call checks the ledger for messages that match *both* the subject and sequence number criteria, and *immediately* marks them as expired.

Once a message has expired, the CM transport no longer attempts to redeliver it to registered listeners.

Rendezvous software presents each expired message to the sender in a [DELIVERY.FAILED](#) advisory. Each advisory includes all the fields of an expired message. (This call can cause many messages to expire simultaneously.)



### Warning

Use with extreme caution. This call exempts the expired messages from certified delivery semantics. It is appropriate only in very few situations.

For example, consider an application program in which an improperly formed CM message causes registered listeners to exit unexpectedly. When the listeners restart, the sender attempts to redeliver the offending message, which again causes the listeners to exit. To break this cycle, the sender can expire the offending message (along with all prior messages bearing the same subject).

Parameter	Description
subject	Mark messages with this subject.  Wildcards subjects are permitted, but must exactly reflect the send subject of the message. For example, if the program sends to A.* then you may expire messages with subject A.* (however, A.> does not resolve to match A.*).
sequenceNumber	Mark messages with sequence numbers <i>less than or equal</i> to this value.

## See Also

[DELIVERY.FAILED on page 268](#) in TIBCO Rendezvous Concepts

# TibrvCmTransport::getDefaultTimeLimit()

*Method*

## Declaration

```
TibrvStatus getDefaultTimeLimit(  
    tibrv_f64& timeLimit) const;
```

## Purpose

Get the default message time limit for all outbound certified messages from a transport.

## Remarks

Every labeled message has a time limit, after which the sender no longer certifies delivery.

Sending programs can explicitly set the time limit on a message (see [TibrvCmMsg::setTimeLimit\(\)](#)). If a time limit is not already set for the outbound message, the transport sets it to the transport's default time limit (extractable with this method); if this default is not set for the transport (nor for the message), the default time limit is zero (no time limit).

Time limits represent the minimum time that certified delivery is in effect.

Parameter	Description
timeLimit	The program supplies a variable, and the method stores the time limit in that variable.

## See Also

[TibrvCmTransport::setDefaultTimeLimit\(\)](#)

[TibrvCmMsg::setTimeLimit\(\)](#)

# TibrvCmTransport::getLedgerName()

*Method*

## Declaration

```
TibrvStatus getLedgerName(  
    const char*& ledgerName) const;
```

## Purpose

Extract the ledger name of a certified delivery transport.

Parameter	Description
ledgerName	The program supplies a variable, and the method stores the ledger name in that variable.

## Errors

The status code [TIBRV\\_ARG\\_CONFLICT](#) indicates that the transport does not have a ledger file.

## See Also

[Ledger File](#)

[TibrvCmTransport::create\(\)](#)

# TibrvCmTransport::getName()

*Method*

## Declaration

```
TibrvStatus getName(  
    const char*& cmName) const;
```

## Purpose

Extract the correspondent name of a certified delivery transport or distributed queue member.

Parameter	Description
cmName	The program supplies a variable, and the method stores the correspondent name in that variable.

## See Also

[Name](#)

[TibrvCmTransport::create\(\)](#)

[TibrvCmQueueTransport::create\(\)](#)



# TibrvCmTransport::getRequestOld()

*Method*

## Declaration

```
TibrvStatus getRequestOld(  
    tibrv_bool& requestOld) const;
```

## Purpose

Extract the request old messages flag of a certified delivery transport.

Parameter	Description
requestOld	The program supplies a variable, and the method stores the request old messages flag name in that variable.

## See Also

[requestOld](#)

[TibrvCmTransport::create\(\)](#)

# TibrvCmTransport::getSyncLedger()

*Method*

## Declaration

```
TibrvStatus getSyncLedger(  
    tibrv_bool& syncLedger) const;
```

## Purpose

Extract the sync ledger flag of a certified delivery transport.

Parameter	Description
syncLedger	The program supplies a variable, and the method stores the sync ledger flag name in that variable.

## Errors

The status code [TIBRV\\_ARG\\_CONFLICT](#) indicates that the transport does not have a ledger file.

## See Also

[Ledger File](#)

[TibrvCmTransport::create\(\)](#)

# TibrvCmTransport::getTransport()

*Method*

## Declaration

```
TibrvTransport* getTransport() const;
```

## Purpose

Extract the transport employed by a certified delivery transport or a distributed queue member.

## See Also

[TibrvTransport](#)

[TibrvNetTransport](#)

[TibrvCmTransport::create\(\)](#)

[TibrvCmQueueTransport::create\(\)](#)

# TibrvCmTransport::removeListener()

*Method*

## Declaration

```
TibrvStatus removeListener(  
    const char* cmName,  
    const char* subject);
```

## Purpose

Unregister a specific listener at a specific correspondent, and free associated storage in the sender's ledger.

## Remarks

This method cancels certified delivery of the specific subject to the correspondent with this name. The listening correspondent may subsequently re-register for certified delivery of the subject. (In contrast, [TibrvCmTransport::disallowListener\(\)](#) cancels certified delivery of *all* subjects to the correspondent, *and* prohibits re-registration.)

Senders can call this method when the ledger item for a listening correspondent has grown very large. Such growth indicates that the listener is not confirming delivery, and may have terminated. Removing the listener reduces the ledger size by deleting messages stored for the listener.

When a sending program calls this method, certified delivery software in the sender behaves as if the listener had closed the endpoint for the subject. The sending program deletes from its ledger all information about delivery of the subject to the correspondent with this cmName. The sending program receives a [REGISTRATION.CLOSED](#) advisory, to trigger any operations in the callback method for the advisory.

If the listening correspondent is available (running and reachable), it receives a [REGISTRATION.NOT\\_CERTIFIED](#) advisory, informing it that the sender no longer certifies delivery of the subject.

If the correspondent with this name does not receive certified delivery of the subject from this sender [TibrvCmTransport](#), then [TibrvCmTransport::removeListener\(\)](#) the status code [TIBRV\\_INVALID\\_SUBJECT](#).

Parameter	Description
cmName	Cancel certified delivery of the subject to listeners of this correspondent.
subject	Cancel certified delivery of this subject to the named listener. Wildcard subjects are illegal.

## See Also

[Name](#)

[TibrvCmTransport::addListener\(\)](#)

[TibrvCmTransport::disallowListener\(\)](#)

Canceling Certified Delivery in TIBCO Rendezvous Concepts

# TibrvCmTransport::removeSendState()

*Method*

## Declaration

```
TibrvStatus removeSendState(  
    const char* subject);
```

## Purpose

Reclaim ledger space from obsolete subjects.

## Background

In some programs subject names are useful only for a limited time; after that time, they are never used again. For example, consider a server program that sends certified reply messages to client inbox names; it only sends one reply message to each inbox, and after delivery is confirmed and complete, that inbox name is obsolete. Nonetheless, a record for that inbox name remains in the server's ledger.

As such obsolete records accumulate, the ledger size grows. To counteract this growth, programs can use this method to discard obsolete subject records from the ledger.

The DELIVERY.COMPLETE advisory is a good opportunity to clear the send state of an obsolete subject. Another strategy is to review the ledger periodically, sweeping to detect and remove all obsolete subjects.



### Warning

Do not use this method to clear subjects that are still in use.

Parameter	Description
subject	Remove send state for this obsolete subject.

## Remarks

As a side-effect, this method resets the sequence numbering for the subject, so the next message sent on the subject would be number 1. In proper usage, this side-effect is never detected, since obsolete subjects are truly obsolete.

## See Also

[TibrvCmTransport::reviewLedger\(\)](#)

[TibrvCmTransport::send\(\)](#)

DELIVERY.COMPLETE in TIBCO Rendezvous Concepts

# TibrvCmTransport::reviewLedger()

*Method*

## Declaration

```
TibrvStatus reviewLedger(  
    TibrvCmReviewCallback* reviewCallback,  
    const char* subject,  
    const void* closure=NULL);
```

## Purpose

Query the ledger for stored items related to a subject name.

## Remarks

The callback method receives one message for each matching subject of outbound messages stored in the ledger. For example, when FOO.\* is the subject, [TibrvCmTransport::reviewLedger\(\)](#) calls its callback method separately for each matching subject—once for FOO.BAR, once for FOO.BAZ, and once for FOO.BOX.

However, if the callback method returns non-NULL, then [TibrvCmTransport::reviewLedger\(\)](#) returns immediately.

If the ledger does not contain any matching items, [TibrvCmTransport::reviewLedger\(\)](#) returns normally without calling the callback method.

For information about the content and format of the callback messages, see [TibrvCmReviewCallback::onLedgerMsg\(\)](#).

Parameter	Description
reviewCallback	This object receives the review messages.
subject	Query for items related to this subject name. If this subject contains wildcard characters (* or >),



Parameter	Description
	then review all items with matching subject names. The callback method receives a separate message for each matching subject in the ledger.
closure	Pass this closure data to the review callback method.

## See Also

[TibrvCmReviewCallback::onLedgerMsg\(\)](#)

# TibrvCmTransport::send()

*Method*

## Declaration

```
TibrvStatus send(  
    const TibrvMsg& msg);
```

## Purpose

Send a labeled message.

## Remarks

This method sends the message, along with its certified delivery protocol information: the correspondent name of the [TibrvCmTransport](#), a sequence number, and a time limit. The protocol information remains on the message within the sending program, and also travels with the message to all receiving programs.

Programs can explicitly set the message time limit; see [TibrvCmMsg::setTimeLimit\(\)](#). If a time limit is not already set for the outbound message, this method sets it to the transport's default time limit (see [TibrvCmTransport::setDefaultTimeLimit\(\)](#)); if that default is not set for the transport, the default time limit is zero (no time limit).

Parameter	Description
msg	Send this message.  Wildcard subjects are <i>illegal</i> .

## See Also

[TibrvCmTransport::sendReply\(\)](#)

[TibrvCmTransport::sendRequest\(\)](#)

[TibrvCmTransport::setDefaultTimeLimit\(\)](#)

[TibrvCmMsg::setTimeLimit\(\)](#)

# TibrvCmTransport::sendReply()

*Method*

## Declaration

```
TibrvStatus sendReply(  
    const TibrvMsg& replyMsg,  
    const TibrvMsg& requestMsg);
```

## Purpose

Send a labeled reply message.

## Remarks

This convenience call extracts the reply subject of an inbound request message, and sends a labeled outbound reply message to that subject. In addition to the convenience, this call is marginally faster than using separate calls to extract the subject and send the reply.

This method can send a labeled reply to an ordinary message.

This method automatically registers the requesting CM transport, so the reply message is certified.

Parameter	Description
replyMsg	Send this <i>outbound</i> reply message.
requestMsg	<p>Send a reply to this <i>inbound</i> request message; extract its reply subject to use as the subject of the outbound reply message.</p> <p>If this message has a wildcard reply subject, the method produces an error.</p>

**Warning**

Give special attention to the order of the arguments to this method. Reversing the inbound and outbound messages can cause an infinite loop, in which the program repeatedly resends the inbound message to itself (and all other recipients).

**See Also**

[TibrvCmTransport::send\(\)](#)

[TibrvCmTransport::sendRequest\(\)](#)

# TibrvCmTransport::sendRequest()

*Method*

## Declaration

```
TibrvStatus sendRequest(  
    const TibrvMsg& requestMsg,  
    TibrvMsg& replyMsg,  
    tibrv_f64 timeout);
```

## Purpose

Send a labeled request message and wait for a reply.

## Blocking can Stall Event Dispatch



### Warning

This call blocks all other activity on its program thread. If appropriate, programmers must ensure that other threads continue dispatching events on its queues.

Parameter	Description
requestMsg	Send this request message.  Wildcard subjects are illegal.
replyMsg	The program supplies a variable, and the method stores the inbound reply in that variable.  The program need not create the reply message, nor allocate space for it. However, the program <i>must destroy</i> the reply message, even though it did not create it.
timeout	Maximum time (in seconds) that this call can block while waiting for a reply.

Parameter	Description
	TIBRV_WAIT_FOREVER (-1) indicates no timeout (wait without limit for a reply).

## Remarks

Programs that receive and process the request message cannot determine that the sender has blocked until a reply arrives.

The sender and receiver must already have a certified delivery agreement, otherwise the request is not certified.

The request message must have a valid destination subject; see [TibrvMsg::setSendSubject\(\)](#).

A certified request does not necessarily imply a certified reply; the replying program determines the type of reply message that it sends.

## Operation

This method operates in several synchronous steps:

### Procedure

1. Create a [TibrvCmListener](#) that listens for messages on the reply subject of requestMsg.
2. Label and send the outbound requestMsg.
3. Block until the listener receives a reply; if the time limit expires before a reply arrives, then return the status code [TIBRV\\_TIMEOUT](#). (The reply event uses a private queue that is not accessible to the program.)
4. Store the reply in the variable specified by the replyMsg parameter.
5. Return.

## See Also

[TibrvCmTransport::send\(\)](#)

[TibrvCmTransport::sendReply\(\)](#)

# TibrvCmTransport::setDefaultTimeLimit()

*Method*

## Declaration

```
TibrvStatus setDefaultTimeLimit(  
    tibrv_f64 timeLimit);
```

## Purpose

Set the default message time limit for all outbound certified messages from a transport.

## Remarks

Every labeled message has a time limit, after which the sender no longer certifies delivery.

Sending programs can explicitly set the time limit on a message (see [TibrvCmMsg::setTimeLimit\(\)](#)). If a time limit is not already set for the outbound message, the transport sets it to the transport's default time limit (set with this method); if this default is not set for the transport, the default time limit is zero (no time limit).

Time limits represent the minimum time that certified delivery is in effect.

Parameter	Description
timeLimit	Use this time limit (in whole seconds). The time limit must be non-negative.

## See Also

[TibrvCmTransport::getDefaultTimeLimit\(\)](#)

[TibrvCmMsg::setTimeLimit\(\)](#)



# TibrvCmTransport::setPublisherInactivityDiscardInterval()

*Method*

## Declaration

```
TibrvStatus setPublisherInactivityDiscardInterval(  
    tibrv_i32 timeout);
```

## Purpose

Set a time limit after which a listening CM transport can discard state for inactive CM senders.

## Remarks

The timeout value limits the time that can elapse during which such a sender does not send a message. When the elapsed time exceeds this limit, the listening transport declares the sender inactive, and discards internal state corresponding to the sender.



### Warning

We discourage programmers from using this call except to solve a very specific problem, in which a long-running CM listener program accumulates state for a large number of obsolete CM senders with non-reusable names.

Before using this call, review every subject for which the CM transport has a listener; ensure that only CM senders with non-reusable names send to those subjects. (If senders with reusable names send messages to such subjects, the listening transport can discard their state, and incorrect behavior can result.)

Parameter	Description
timeout	Use this time limit (in whole seconds). The time limit must be non-negative.

# TibrvCmTransport::syncLedger()

*Method*

## Declaration

```
TibrvStatus syncLedger();
```

## Purpose

Synchronize the ledger to its storage medium.

## Remarks

When this method returns, the transport's current state is safely stored in the ledger file.

Transports that use synchronous ledger files need not call this method, since the current state is automatically written to the storage medium before returning. Transports that use process-based ledger storage need not call this method, since they have no ledger file.

## Errors

The status code [TIBRV\\_INVALID\\_ARG](#) indicates that the transport does not have a ledger file.

## See Also

[Ledger File](#)

[TibrvCmTransport::create\(\)](#)

[TibrvCmTransport::getSyncLedger\(\)](#)

# TibrvCmTransportOnComplete

*Class*

## Declaration

```
class TibrvCmTransportOnComplete
```

## Purpose

Callback class to signal completion of clean-up tasks after destroying a CM or CMQ transport.

Method	Description
<a href="#">TibrvCmTransportOnComplete::onComplete</a>	Destroying a CM or CMQ transport involves cleanup operations that can sometimes be lengthy. Programs can define this method to discover when the cleanup has completed.

## See Also

[TibrvCmTransport::destroy\(\)](#)

# TibrvCmTransportOnComplete::onComplete

*Method*

## Declaration

```
virtual void onComplete(  
    TibrvCmTransport* destroyedTransport) = 0;
```

## Purpose

Destroying a CM or CMQ transport involves cleanup operations that can sometimes be lengthy. Programs can define this method to discover when the cleanup has completed.

## Remarks

Programs must not destroy the transport's listeners, nor any queue nor dispatcher that pertains to the transport, until after this callback signals that cleanup has completed.

Parameter	Description
destroyedTransport	<p>This parameter receives the transport object.</p> <p>However, by the time this method runs, the transport is already destroyed; this method cannot use the transport object in Rendezvous calls.</p>

## See Also

[TibrvCmTransport::destroy\(\)](#)

# TibrvCmReviewCallback

*Class*

## Declaration

```
class TibrvCmReviewCallback
```

## Purpose

Process ledger review messages.

## Remarks

Implement this interface to process ledger review messages.

Method	Description
<a href="#">TibrvCmReviewCallback::onLedgerMsg()</a>	Programs define this method to process ledger review messages.

## See Also

[TibrvCmTransport::reviewLedger\(\)](#)

# TibrvCmReviewCallback::onLedgerMsg()

*Method*

## Declaration

```
virtual void* onLedgerMsg(  
    TibrvCmTransport* cmTransport,  
    const char* subject,  
    TibrvMsg& msg,  
    void* closure) = 0;
```

## Purpose

Programs define this method to process ledger review messages.

## Remarks

[TibrvCmTransport::reviewLedger\(\)](#) calls this callback method once for each matching subject stored in the ledger.

To continue reviewing the ledger, return NULL from this callback method. To stop reviewing the ledger, return non-NULL from this callback method;

[TibrvCmTransport::reviewLedger\(\)](#) cancels the review and returns immediately.

Parameter	Description
cmTransport	This parameter receives the transport.
subject	This parameter receives the subject for this ledger item.
msg	This parameter receives a summary message describing the delivery status of messages in the ledger. The table <a href="#">Message Content</a> describes the fields of the summary message.
closure	This parameter receives closure data that the program supplied to <a href="#">TibrvCmTransport::reviewLedger()</a> .

## Message Content

This callback method receives ledger summary messages with these fields.

Field Name	Description
subject	<p>The subject that this message summarizes.</p> <p>This field has datatype <a href="#">TIBRVMSG_STRING</a>.</p>
seqno_last_sent	<p>The sequence number of the most recent message sent with this subject name.</p> <p>This field has datatype <a href="#">TIBRVMSG_U64</a>.</p>
total_msgs	<p>The total number of messages stored at this subject name.</p> <p>This field has datatype <a href="#">TIBRVMSG_U32</a>.</p>
total_size	<p>The total storage (in bytes) occupied by all messages with this subject name.</p> <p>If the ledger contains several messages with this subject name, then this field sums the storage space over all of them.</p> <p>This field has datatype <a href="#">TIBRVMSG_U64</a>.</p>
listener	<p>Each summary message can contain one or more fields named listener. Each listener field contains a nested submessage with details about a single registered listener.</p> <p>This field has datatype <a href="#">TIBRVMSG_MSG</a>.</p>
listener.name	<p>Within each listener submessage, the name field contains the name of the listener transport.</p> <p>This field has datatype <a href="#">TIBRVMSG_STRING</a>.</p>
listener.last_confirmed	<p>Within each listener submessage, the last_confirmed field contains the sequence number of the last message for which the listener confirmed delivery.</p> <p>This field has datatype <a href="#">TIBRVMSG_U64</a>.</p>



## See Also

[TibrvCmTransport::reviewLedger\(\)](#)

# TibrvCmMsg

*Class*

## Declaration

```
class TibrvCmMsg
```

## Purpose

Define methods to manipulate labeled messages.

## Remarks

This class is a collection of accessor methods. Programs do not create instances of [TibrvCmMsg](#). Instead, programs use its static methods to get and set certified delivery information of [TibrvMsg](#) objects.

Method	Description
<a href="#">TibrvCmMsg::getSender()</a>	Extract the correspondent name of the sender from a certified message.
<a href="#">TibrvCmMsg::getSequence()</a>	Extract the sequence number from a certified message.
<a href="#">TibrvCmMsg::getTimeLimit()</a>	Extract the message time limit from a certified message.
<a href="#">TibrvCmMsg::setTimeLimit()</a>	Set the message time limit of a certified message.

## See Also

[TibrvMsg](#)

# TibrvCmMsg::getSender()

*Method*

## Declaration

```
static TibrvStatus getSender(  
    TibrvMsg& msg,  
    const char*& name);
```

## Purpose

Extract the correspondent name of the sender from a certified message.

Parameter	Description
msg	Extract the sender name from this message.
name	The program supplies a variable. The method stores the name in that variable.

## Status

This method returns a status code that discriminates between labeled messages and other messages.

- If the message is from a CM sender, then [TibrvCmMsg::getSender\(\)](#) returns the status code [TIBRV\\_OK](#) and yields a valid CM correspondent name.
- If the message is *not* from a CM sender, then [TibrvCmMsg::getSender\(\)](#) returns the status code [TIBRV\\_NOT\\_FOUND](#).

## See Also

[TibrvCmTransport::create\(\)](#)

[TibrvCmTransport::getName\(\)](#)

# TibrvCmMsg::getSequence()

*Method*

## Declaration

```
static TibrvStatus getSequence(  
    TibrvMsg& msg,  
    tibrv_u64& sequenceNumber);
```

## Purpose

Extract the sequence number from a certified message.

## Remarks

Rendezvous certified delivery sending methods automatically generate positive sequence numbers for outbound labeled messages.

In receiving programs, zero is a special value, indicating that an inbound message is not certified.

Parameter	Description
msg	Extract the sequence number from this message.
sequenceNumber	The program supplies a variable. The method stores the sequence number in that variable.

## Status

This method returns a status code that discriminates between certified messages (with a certified delivery agreement) and other messages.

- If the message is from a CM sender, *and* the CM listener is registered for certified delivery with that sender, then `TibrvCmMsg::getSequence()` returns the status code `TIBRV_OK` and yields a valid sequence number.

- If the message is from a CM sender, but the listener is *not* registered for certified delivery, then `TibrvCmMsg::getSequence()` *in the context of a* `TibrvCmMsgCallback::onCmMsg()` *method* returns the status code `TIBRV_NOT_FOUND`. (In any other context, it returns the actual sequence number stored on the message.)

Notice that the first labeled message that a program receives on a subject might not be certified; that is, the sender has not registered a certified delivery agreement with the listener. If appropriate, the certified delivery library automatically requests that the sender register the listener for certified delivery. (See Discovery and Registration for Certified Delivery in TIBCO Rendezvous Concepts.)

A labeled but uncertified message can also result when the sender explicitly disallows or removes the listener.

- If the message is *not* from a CM sender, then `TibrvCmMsg::getSequence()` (in any context) returns the status code `TIBRV_NOT_FOUND`.

## Release 5 Interaction

In release 6 (and later) the sequence number is a 64-bit unsigned integer, while in older releases (5 and earlier) it is a 32-bit unsigned integer.

When 32-bit senders overflow the sequence number, behavior is undefined.

When 64-bit senders send sequence numbers greater than 32 bits, 32-bit receivers detect malformed label information, and process the message as an ordinary reliable message (uncertified and unlabeled).

## See Also

[TibrvCmTransport::send\(\)](#)

# TibrvCmMsg::getTimeLimit()

*Method*

## Declaration

```
static TibrvStatus getTimeLimit(  
    TibrvMsg& msg,  
    tibrv_f64& timeLimit);
```

## Purpose

Extract the message time limit from a certified message.

## Remarks

Programs can explicitly set the message time limit (see [TibrvCmMsg::setTimeLimit\(\)](#)).

Zero is a special value, indicating no time limit.

If a time limit is not set for a message, this method returns the status code [TIBRV\\_NOT\\_FOUND](#). This situation can occur only for unsent outbound messages, and for inbound unlabeled messages.

Time limits represent the minimum time that certified delivery is in effect.

This value represents the total time limit of the message, *not* the time remaining.

Parameter	Description
msg	Extract the time limit from this message.
timeLimit	The program supplies a variable. The method stores the time limit in that variable.

## See Also

[TibrvCmTransport::send\(\)](#)

[TibrvCmMsg::setTimeLimit\(\)](#)

# TibrvCmMsg::setTimeLimit()

*Method*

## Declaration

```
static TibrvStatus setTimeLimit (  
    TibrvMsg& msg,  
    tibrv_f64 timeLimit);
```

## Purpose

Set the message time limit of a certified message.

## Remarks

Every labeled message has a time limit, after which the sender no longer certifies delivery.

Sending programs can explicitly set the message time limit using this method. If a time limit is not already set for the outbound message, [TibrvCmTransport::send\(\)](#) sets it to the transport's default time limit (see [TibrvCmTransport::setDefaultTimeLimit\(\)](#)); if that default is not set for the transport, the default time limit is zero (no time limit).

Time limits represent the minimum time that certified delivery is in effect.

It is meaningless for receiving programs to call this method.

Parameter	Description
msg	Set the time limit of this message.
timeLimit	Use this time limit (in whole seconds) for the message. The time limit must be non-negative.

## See Also

[TibrvCmTransport::getDefaultTimeLimit\(\)](#)

[TibrvCmTransport::setDefaultTimeLimit\(\)](#)

[TibrvCmMsg::getTimeLimit\(\)](#)



# TibrvCmMsgCallback

*Class*

## Declaration

```
class TibrvCmMsgCallback : public TibrvCallback
```

## Purpose

Process inbound messages (CM listener events).

## Remarks

Implement this interface to process inbound certified delivery messages.

Method	Description
<a href="#">TibrvCmMsgCallback::onCmMsg()</a>	Process inbound certified delivery messages (CM listener events).

## Related Classes

[TibrvCallback](#)

[TibrvMsgCallback](#)

## See Also

[TibrvCmListener::create\(\)](#)

# TibrvCmMsgCallback::onCmMsg()

*Method*

## Declaration

```
virtual void onCmMsg(  
    TibrvCmListener* cmListener,  
    TibrvMsg& msg) = 0;
```

## Purpose

Process inbound certified delivery messages (CM listener events).

## Remarks

Implement this method to process inbound certified delivery messages.

Parameter	Description
cmListener	This parameter receives the listener event.
msg	This parameter receives the inbound message.

## CM Label Information

The callback method for certified delivery messages can use CM label information to discriminate these situations:

- If `TibrvCmMsg::getSender()` returns status code `TIBRV_NOT_FOUND`, then the message uses the reliable protocol (that is, it was sent from an ordinary transport).
- If `TibrvCmMsg::getSender()` returns a valid sender name, then the message uses the certified delivery protocol (that is, it is a labeled message, sent from a CM transport).

Once the callback method determines that the message uses the certified delivery protocol, it can discriminate further:

- If `TibrvCmMsg::getSequence()` returns status code `TIBRV_NOT_FOUND`, then the listener is *not registered* for certified delivery from the sender.
- If `TibrvCmMsg::getSequence()` returns `TIBRV_OK`, then a certified delivery agreement is in effect for this subject with the sender.

## See Also

`TibrvCmListener::create()`

`TibrvCmMsg::getSender()`

`TibrvCmMsg::getSequence()`

`TibrvCmMsg::getTimeLimit()`

# Distributed Queue

---

Programs can use distributed queues for *one of  $n$*  certified delivery to a group of worker processes.

A distributed queue is a group of [TibrvCmQueueTransport](#) objects, each in a separate process. From the outside, a distributed queue appears as though a single transport object; inside, the group members act in concert to process inbound task messages. Ordinary transports and CM transports can send task messages to the group. Notice that the senders are not group members, and do not do anything special to send messages to a group; rather, they send messages to ordinary subject names. Inside the group, the member acting as scheduler assigns each task message to exactly one of the other members (which act as workers); only that worker processes the task message. Each member uses CM listener objects to receive task messages.

Distributed queues depend upon the certified delivery methods and the fault tolerance methods.

**Note**

We do not recommend sending messages across network boundaries to a distributed queue, nor distributing queue members across network boundaries. However, when crossing network boundaries in either of these ways, you must configure the Rendezvous routing daemons to exchange `_RVCM` and `_RVCMQ` administrative messages. For details, see *Distributed Queues in TIBCO Rendezvous Administration*.

## See Also

Distributed Queue in *TIBCO Rendezvous Concepts*

# TibrvCmQueueTransport

*Class*

## Declaration

```
class TibrvCmQueueTransport : public TibrvCmTransport
    TibrvCmQueueTransport();          // Create empty.
    virtual ~TibrvCmQueueTransport(); // Destroy and reclaim storage.
```

## Purpose

Coordinate a distributed queue for *one-of-n* delivery.

## Remarks

Each [TibrvCmQueueTransport](#) object employs a [TibrvTransport](#) for network communications. The [TibrvCmQueueTransport](#) adds the accounting and coordination mechanisms needed for one-of-n delivery.

Several [TibrvCmQueueTransport](#) objects can employ one [TibrvTransport](#), which also remains available for its own ordinary listeners and for sending ordinary messages.

The constructor creates a hollow object; [TibrvCmQueueTransport::create\(\)](#) makes it operational. The destructor calls the destroy method, unless the C object is already destroyed. Programs must explicitly destroy each [TibrvCmQueueTransport](#) object. Destroying a [TibrvCmQueueTransport](#) invalidates any certified listeners using that transport (while preserving their certified delivery agreements).

All members of a distributed queue must listen to exactly the same set of subjects. See [Enforcing Identical Subscriptions in TIBCO Rendezvous Concepts](#).

Scheduler recovery and task rescheduling are available only when the task message is a certified message (that is, a certified delivery agreement is in effect between the task sender and the distributed queue transport scheduler).

## Disabled Methods

Although [TibrvCmQueueTransport](#) is a subclass of [TibrvCmTransport](#), all methods related to sending messages are disabled in [TibrvCmQueueTransport](#); for a list, see [Disabled](#)

Methods. See also Certified Delivery Behavior in Queue Members in TIBCO Rendezvous Concepts.

Method	Description
<a href="#">TibrvCmQueueTransport::create()</a>	Create a transport as a distributed queue member.
<a href="#">TibrvCmQueueTransport::destroy()</a>	Destroy a distributed queue member object.
<a href="#">TibrvCmQueueTransport::getCompleteTime()</a>	Extract the worker complete time limit of a distributed queue member.
<a href="#">TibrvCmQueueTransport::getUnassignedMessageCount()</a>	Extract the number of unassigned task messages from a distributed queue transport.
<a href="#">TibrvCmQueueTransport::getWorkerWeight()</a>	Extract the worker weight of a distributed queue member.
<a href="#">TibrvCmQueueTransport::getWorkerTasks()</a>	Extract the worker task capacity of a distributed queue member.
<a href="#">TibrvCmQueueTransport::setCompleteTime()</a>	Set the worker complete time limit of a distributed queue member.
<a href="#">TibrvCmQueueTransport::setTaskBacklogLimit...()</a>	Set the scheduler task queue limits of a distributed queue transport.
<a href="#">TibrvCmQueueTransport::setWorkerWeight()</a>	Set the worker weight of a distributed queue member.
<a href="#">TibrvCmQueueTransport::setWorkerTasks()</a>	Set the worker task capacity of a distributed queue member.

## Inherited Methods

Legal Methods

- [TibrvCmTransport::getName\(\)](#)
- [TibrvCmTransport::getTransport\(\)](#)
- [TibrvTransport::destroy\(\)](#)
- [TibrvTransport::isValid\(\)](#)
- [TibrvTransport::getHandle\(\)](#)
- [TibrvTransport::setDescription\(\)](#)

Disabled Methods

- [TibrvCmTransport::addListener\(\)](#)
- [TibrvCmTransport::allowListener\(\)](#)
- [TibrvCmTransport::disallowListener\(\)](#)
- [TibrvCmTransport::getDefaultTimeLimit\(\)](#)
- [TibrvCmTransport::getLedgerName\(\)](#)
- [TibrvCmTransport::getRequestOld\(\)](#)
- [TibrvCmTransport::getSyncLedger\(\)](#)
- [TibrvCmTransport::removeListener\(\)](#)
- [TibrvCmTransport::removeSendState\(\)](#)
- [TibrvCmTransport::reviewLedger\(\)](#)
- [TibrvCmTransport::send\(\)](#)
- [TibrvCmTransport::sendReply\(\)](#)
- [TibrvCmTransport::sendRequest\(\)](#)
- [TibrvCmTransport::setDefaultTimeLimit\(\)](#)
- [TibrvCmTransport::syncLedger\(\)](#)
- [TibrvTransport::createInbox\(\)](#)
- [TibrvTransport::send\(\)](#)
- [TibrvTransport::sendReply\(\)](#)

### Inherited Methods

[TibrvTransport::sendRequest\(\)](#)

### Related Classes

[TibrvTransport](#)

[TibrvNetTransport](#)

[TibrvCmTransport](#)



# TibrvCmQueueTransport::create()

*Method*

## Declaration

```
TibrvStatus create(  
    TibrvTransport* transport,  
    const char* cmName,  
    tibrv_u32 workerWeight = TIBRVCM_DEFAULT_WORKER_WEIGHT,  
    tibrv_u32 workerTasks = TIBRVCM_DEFAULT_WORKER_TASKS,  
    tibrv_u16 schedulerWeight = TIBRVCM_DEFAULT_SCHEDULER_WEIGHT,  
    tibrv_f64 schedulerHeartbeat = TIBRVCM_DEFAULT_SCHEDULER_HB,  
    tibrv_f64 schedulerActivation = TIBRVCM_DEFAULT_SCHEDULER_ACTIVE );
```

## Purpose

Create a transport as a distributed queue member.


## Remarks

This method creates a new C distributed queue member, and stores it in the C++ object.

The new [TibrvCmQueueTransport](#) must employ a valid [TibrvTransport](#) for network communications.

Parameter	Description
transport	<p>The new <a href="#">TibrvCmQueueTransport</a> employs this <a href="#">TibrvTransport</a> object for network communications.</p> <p>Destroying the <a href="#">TibrvCmQueueTransport</a> does not affect this transport.</p>
cmName	<p>Bind this reusable name to the new transport object, which becomes a member of the distributed queue with this name.</p> <p>The name must be non-NULL, and conform to the syntax rules for</p>

Parameter	Description
	<p>Rendezvous subject names. It cannot begin with reserved tokens. It cannot be a non-reusable name generated by a call to <code>TibrvCmTransport::create()</code>. It cannot be the empty string.</p> <p>For more information, see Reusable Names in TIBCO Rendezvous Concepts.</p>
<code>workerWeight</code>	<p>When the scheduler receives a task, it assigns the task to the available worker with the greatest worker weight.</p> <p>A worker is considered available unless either of these conditions are true:</p> <ul style="list-style-type: none"> <li>• The pending tasks assigned to the worker member exceed its task capacity.</li> <li>• The worker is also the scheduler. (The scheduler assigns tasks to its own worker role only when no other workers are available.)</li> </ul> <p>When omitted, the default value is 1.</p>
<code>workerTasks</code>	<p>Task capacity is the maximum number of tasks that a worker can accept. When the number of accepted tasks reaches this maximum, the worker cannot accept additional tasks until it completes one or more of them.</p> <p>When the scheduler receives a task, it assigns the task to the worker with the greatest worker weight—unless the pending tasks assigned to that worker exceed its task capacity. When the preferred worker has too many tasks, the scheduler assigns the new inbound task to the worker with the next greatest worker weight.</p> <p>The value must be a non-negative integer. When omitted, the default value is 1.</p> <p>Zero is a special value, indicating that this distributed queue member is a dedicated scheduler (that is, it never accepts tasks).</p>

Parameter	Description
	 <p><b>Warning</b></p> <p>Tuning task capacity to compensate for communication time lag is more complicated than it might seem. Before setting this value to anything other than 1, see Task Capacity in TIBCO Rendezvous Concepts.</p>
schedulerWeight	<p>Weight represents the ability of this member to fulfill the role of scheduler, relative to other members with the same name. Cooperating members use relative scheduler weight values to elect one member as the scheduler; members with higher scheduler weight take precedence.</p> <p>When omitted, the default value is 1.</p> <p>Acceptable values range from 0 to 65535. Zero is a special value, indicating that the member can never be the scheduler. For more information, see Rank and Weight in TIBCO Rendezvous Concepts.</p>
schedulerHeartbeat	<p>The scheduler sends heartbeat messages at this interval (in seconds).</p> <p>All <a href="#">TibrvCmQueueTransport</a> objects with the same name must specify the same value for this parameter. The value must be strictly positive. To determine the correct value, see Step 4: Choose the Intervals in TIBCO Rendezvous Concepts.</p> <p>When omitted, the default value is 1.0.</p>
schedulerActivation	<p>When the heartbeat signal from the scheduler has been silent for this interval (in seconds), the cooperating member with the greatest scheduler weight takes its place as the new scheduler.</p> <p>All <a href="#">TibrvCmQueueTransport</a> objects with the same name must specify the same value for this parameter. The value must be strictly positive. To determine the correct value, see Step 4: Choose the Intervals in TIBCO Rendezvous Concepts.</p> <p>When omitted, the default value is 3.5.</p>

Constant	Value
TIBRVCM_DEFAULT_COMPLETE_TIME	0
TIBRVCM_DEFAULT_WORKER_WEIGHT	1
TIBRVCM_DEFAULT_WORKER_TASKS	1
TIBRVCM_DEFAULT_SCHEDULER_WEIGHT	1
TIBRVCM_DEFAULT_SCHEDULER_HB	1.0
TIBRVCM_DEFAULT_SCHEDULER_ACTIVE	3.5

## See Also

[TibrvCmQueueTransport::destroy\(\)](#)

Distributed Queue, in TIBCO Rendezvous Concepts

# TibrvCmQueueTransport::destroy()

*Method*

## Declaration

```
TibrvStatus destroy();
```

## Purpose

Destroy a distributed queue member object.

## Remarks

Destroying a [TibrvCmQueueTransport](#) object removes the program from the distributed queue group.

## See Also

[TibrvCmQueueTransport::create\(\)](#)

# TibrvCmQueueTransport::getCompleteTime()

*Method*

## Declaration

```
TibrvStatus getCompleteTime(  
    tibrv_f64& completeTime) const;
```

## Purpose

Extract the worker complete time limit of a distributed queue member.

Parameter	Description
completeTime	The program supplies a variable, and the method stores the worker complete time in that variable.

## See Also

Distributed Queue, in TIBCO Rendezvous Concepts

[TibrvCmQueueTransport::setCompleteTime\(\)](#)

# TibrvCmQueueTransport::getUnassignedMessageCount()

*Function*

## Declaration

```
TibrvStatus getUnassignedMessageCount(  
    tibrv_u32& msgCount) const;
```

## Purpose

Extract the number of unassigned task messages from a distributed queue transport.

## Remarks

An unassigned task message is a message received by the scheduler, but not yet assigned to any worker in the distributed queue.

This call produces a valid count only within a scheduler process. Within a worker process, this call always produces zero.

Parameter	Description
msgCount	The program supplies a variable, and the function stores the unassigned task count in that variable.

# TibrvCmQueueTransport::getWorkerWeight()

*Method*

## Declaration

```
TibrvStatus getWorkerWeight(  
    tibrv_u32& workerWeight) const;
```

## Purpose

Extract the worker weight of a distributed queue member.

Parameter	Description
workerWeight	The program supplies a variable, and the method stores the worker weight in that variable.

## See Also

Distributed Queue, in TIBCO Rendezvous Concepts

[TibrvCmQueueTransport::create\(\)](#)

[TibrvCmQueueTransport::setWorkerWeight\(\)](#)



# TibrvCmQueueTransport::getWorkerTasks()

*Method*

## Declaration

```
TibrvStatus getWorkerTasks(  
    tibrv_u32& workerTasks) const;
```

## Purpose

Extract the worker task capacity of a distributed queue member.

Parameter	Description
workerTasks	The program supplies a variable, and the method stores the worker task capacity in that variable.

## See Also

Distributed Queue, in TIBCO Rendezvous Concepts

[TibrvCmQueueTransport::create\(\)](#)

[TibrvCmQueueTransport::setWorkerTasks\(\)](#)

# TibrvCmQueueTransport::setCompleteTime()

*Method*

## Declaration

```
TibrvStatus setCompleteTime(  
    tibrv_f64 completeTime);
```

## Purpose

Set the worker complete time limit of a distributed queue member.

## Remarks

If the complete time is non-zero, the scheduler waits for a worker member to complete an assigned task. If the complete time elapses before the scheduler receives completion from the worker member, the scheduler reassigns the task to another worker member.

Zero is a special value, which specifies no limit on the completion time—that is, the scheduler does not set a timer, and does not reassign tasks when task completion is lacking. All members implicitly begin with a default complete time value of zero; programs can change this parameter using this method.

Parameter	Description
completeTime	Use this complete time (in seconds). The time must be non-negative.

## See Also

Distributed Queue, in TIBCO Rendezvous Concepts

[TibrvCmQueueTransport::getCompleteTime\(\)](#)

# TibrvCmQueueTransport::setTaskBacklogLimit...()

*Method*

## Declaration

```
TibrvStatus setTaskBacklogLimitInBytes(  
    tibrv_u32 byteLimit);  
TibrvStatus setTaskBacklogLimitInMessages(  
    tibrv_u32 msgLimit);
```

## Purpose

Set the scheduler task queue limits of a distributed queue transport.

## Remarks

The scheduler stores tasks in a queue. These properties limit the maximum size of that queue—by number of bytes or number of messages (or both). When no value is set for these properties, the default is no limit.

When the task messages in the queue exceed either of these limits, Rendezvous software deletes new inbound task messages.

Programs may call each of these methods at most once. Those calls must occur before the transport assumes the scheduler role; after a transport acts as a scheduler, these values are fixed, and subsequent attempts to change them result in status code [TIBRV\\_NOT\\_PERMITTED](#).

Parameter	Description
byteLimit	Use this size limit (in bytes).  Zero is a special value, indicating no size limit.
msgLimit	Use this message limit (number of messages).

Parameter	Description
	Zero is a special value, indicating no limit on the number of messages.

## See Also

Distributed Queue, in TIBCO Rendezvous Concepts

# TibrvCmQueueTransport::setWorkerWeight()

*Method*

## Declaration

```
TibrvStatus setWorkerWeight(  
    tibrv_u32 workerWeight);
```

## Purpose

Set the worker weight of a distributed queue member.

## Remarks

Relative worker weights assist the scheduler in assigning tasks. When the scheduler receives a task, it assigns the task to the available worker with the greatest worker weight.

The default worker weight is 1; programs can set this parameter at creation using [TibrvCmQueueTransport::create\(\)](#), or change it dynamically using this method.

Parameter	Description
workerWeight	Use this worker weight.

## See Also

Distributed Queue, in TIBCO Rendezvous Concepts

[TibrvCmQueueTransport::create\(\)](#)

[TibrvCmQueueTransport::getWorkerWeight\(\)](#)

# TibrvCmQueueTransport::setWorkerTasks()

*Method*

## Declaration

```
TibrvStatus setWorkerTasks(  
    tibrv_u32 workerTasks);
```

## Purpose

Set the worker task capacity of a distributed queue member.

## Remarks

Task capacity is the maximum number of tasks that a worker can accept. When the number of accepted tasks reaches this maximum, the worker cannot accept additional tasks until it completes one or more of them.

When the scheduler receives a task, it assigns the task to the worker with the greatest worker weight—unless the pending tasks assigned to that worker exceed its task capacity. When the preferred worker has too many tasks, the scheduler assigns the new inbound task to the worker with the next greatest worker weight.

The default worker task capacity is 1.

Zero is a special value, indicating that this distributed queue member is a dedicated scheduler (that is, it never accepts tasks).



### Warning

Tuning task capacity to compensate for communication time lag is more complicated than it might seem. Before setting this value to anything other than 1, see Task Capacity in TIBCO Rendezvous Concepts.

Parameter	Description
workerTasks	Use this task capacity. The value must be a non-negative integer.

## See Also

Distributed Queue, in TIBCO Rendezvous Concepts

[TibrvCmQueueTransport::create\(\)](#)

[TibrvCmQueueTransport::getWorkerTasks\(\)](#)

# Datatypes

---

Rendezvous wire format datatypes are a standard, platform-independent convention for types and sizes. A parallel set of C datatypes represents data within programs.

This section summarizes the two sets of datatypes, and the conversions among the various types.



# Wire Format Datatypes

Wire Format Type	Type Description	Notes
<b>Special Types</b>		
TIBRVMSG_MSG	Rendezvous message	
TIBRVMSG_DATETIME	Rendezvous datetime	
TIBRVMSG_OPAQUE	opaque byte sequence	
TIBRVMSG_STRING	ISO 8859-1 character string (also called Latin-1)	NULL-terminated.
TIBRVMSG_XML	XML data (byte sequence)	
<b>Scalar Types</b>		
TIBRVMSG_BOOL	boolean	TIBRV_FALSE, TIBRV_TRUE
TIBRVMSG_I8	8-bit integer	
TIBRVMSG_I16	16-bit integer	
TIBRVMSG_I32	32-bit integer	
TIBRVMSG_I64	64-bit integer	
TIBRVMSG_U8	8-bit unsigned integer	
TIBRVMSG_U16	16-bit unsigned integer	
TIBRVMSG_U32	32-bit unsigned integer	
TIBRVMSG_U64	64-bit unsigned integer	

Wire Format Type	Type Description	Notes
TIBRVMSG_F32	32-bit floating point	
TIBRVMSG_F64	64-bit floating point	
TIBRVMSG_IPADDR32	4-byte IP address	Network byte order. String representation is four-part dot-delimited notation.
TIBRVMSG_IPPORT16	2-byte IP port	Network byte order. String representation is a 16-bit decimal integer.

### Array Types

TIBRVMSG_I8ARRAY	8-bit integer array	The count property of the field reflects the number of elements in the array.
TIBRVMSG_I16ARRAY	16-bit integer array	
TIBRVMSG_I32ARRAY	32-bit integer array	
TIBRVMSG_I64ARRAY	64-bit integer array	
TIBRVMSG_U8ARRAY	8-bit unsigned integer array	
TIBRVMSG_U16ARRAY	16-bit unsigned integer array	
TIBRVMSG_U32ARRAY	32-bit unsigned integer array	
TIBRVMSG_	64-bit unsigned integer array	

Wire Format Type	Type Description	Notes
<b>U64ARRAY</b>		
TIBRVMSG_ <b>F32ARRAY</b>	32-bit floating point array	
TIBRVMSG_ <b>F64ARRAY</b>	64-bit floating point array	

# C Datatypes

C Type	Type Description	Notes
<code>tibrv_bool</code>	boolean	<code>TIBRV_FALSE</code> , <code>TIBRV_TRUE</code>
<code>tibrv_f32</code>	32-bit floating point	
<code>tibrv_f64</code>	64-bit floating point	
<code>tibrv_i8</code>	8-bit integer	
<code>tibrv_i16</code>	16-bit integer	
<code>tibrv_i32</code>	32-bit integer	
<code>tibrv_i64</code>	64-bit integer	
<code>tibrv_u8</code>	8-bit unsigned integer	
<code>tibrv_u16</code>	16-bit unsigned integer	
<code>tibrv_u32</code>	32-bit unsigned integer	
<code>tibrv_u64</code>	64-bit unsigned integer	
<code>tibrv_ipaddr32</code>	4-byte IP address	Stored in network byte order.  String representation is four-part dot-delimited notation.
<code>tibrv_ipport16</code>	2-byte IP port	Stored in network byte order.  String representation is a decimal integer (all 16-bits).
<code>tibrvMsgDateTime</code>	Rendezvous datetime struct	See <a href="#">TibrvMsgDateTime</a> .  See also <a href="#">tibrvMsgDateTime</a> on page 53 in TIBCO Rendezvous C Reference.

C Type	Type Description	Notes
<code>tibrv_f32*</code>	32-bit floating point array	
<code>tibrv_f64*</code>	64-bit floating point array	
<code>tibrv_i8*</code>	8-bit integer array	
<code>tibrv_i16*</code>	16-bit integer array	
<code>tibrv_i32*</code>	32-bit integer array	
<code>tibrv_i64*</code>	64-bit integer array	
<code>tibrv_u8*</code>	8-bit unsigned integer array	
<code>tibrv_u16*</code>	16-bit unsigned integer array	
<code>tibrv_u32*</code>	32-bit unsigned integer array	
<code>tibrv_u64*</code>	64-bit unsigned integer array	
<code>tibrvMsg</code>	Rendezvous message	See <a href="#">TibrvMsg</a> .  See also <a href="#">tibrvMsg</a> on page 52 in TIBCO Rendezvous C Reference.
<code>char*</code>	character string	ISO 8859-1 (Latin-1) character encoding
<code>void*</code>	opaque byte sequence	

# Datatype Conversion

Rendezvous software converts datatypes in two situations:

- As it translates a message to wire format (when sending a message).
- As it extracts data from a message field.

Convenience methods that *extract* a field from a Rendezvous message automatically decode the field's data to a homologous C type. [Wire Format to C Datatype Conversion Matrix](#) specifies the homologous decodings as well as conversions to other types. See also, [TibrvMsg::getField\(\)](#)[TibrvMsg::getField\(\)](#).

Convenience methods that *add* a field to a Rendezvous message or update an existing field severely restrict type encoding. These methods encode only to homologous types (the solid dots along the diagonal of [Wire Format to C Datatype Conversion Matrix](#) indicate pairs of homologous types).

## General Rules

These general rules govern most conversions.

### Supported

- All wire format types decode to the homologous C datatypes (in *get* calls), and all C datatypes encode to the homologous wire format types (in *add* and *update* calls).
- All wire format numeric scalar types convert to all C numeric scalar types.
- All wire format numeric array types convert to all C numeric array types.

### Caution

- Converting a wire format opaque or XML byte sequence to a C character string creates a printable string, but the string does not capture any of the opaque data bytes (it captures only the number of bytes).
- Converting a wire format signed integer to a C unsigned integer discards the sign.
- Converting a wire format numeric type to a C numeric type with fewer bits risks loss of precision.

- Converting wire format floating point numbers to C integer types discards the fractional part.
- Converting large (out-of-range) wire format floating point numbers to C integers results in the maximum integer of the C target size.

## Not Supported

- Array types do not convert to scalar types.
- Scalar types do not convert to array types.
- C types do not convert to non-homologous wire format types (when adding or updating a field).

## Converting to Boolean

- When converting a string to a boolean, all strings in which the first character is either t or T map to boolean true. All other strings map to boolean false.
- When converting a numeric value to a boolean, zero maps to boolean false. All non-zero numeric values map to boolean true.

Figure 14: Wire Format to C Datatype Conversion Matrix

Get

		C Destination Type																												
		tibrv_bool	tibrv_f32	tibrv_f64	tibrv_i8	tibrv_i16	tibrv_i32	tibrv_i64	tibrv_u8	tibrv_u16	tibrv_u32	tibrv_u64	tibrv_ipaddr32	tibrv_ipport16	tibrvmsgDate Time	tibrv_f32[]	tibrv_f64[]	tibrv_i8[]	tibrv_i16[]	tibrv_i32[]	tibrv_i64[]	tibrv_u8[]	tibrv_u16[]	tibrv_u32[]	tibrv_u64[]	tibrvmsg	void*	char*	tibrvmsg[]	char*
tibrvmsg Source Type	TIBRVMSG_BOOL	●	S	S	S	S	S	S	S	S	S	S															S			
	TIBRVMSG_F32	S	●	N	N	N	N	N	N	N	N	N															S			
	TIBRVMSG_F64	S	N	●	N	N	N	N	N	N	N	N															S			
	TIBRVMSG_I8	S	N	N	●	N	N	N	N	N	N	N															S			
	TIBRVMSG_I16	S	N	N	N	●	N	N	N	N	N	N															S			
	TIBRVMSG_I32	S	N	N	N	N	●	N	N	N	N	N															S			
	TIBRVMSG_I64	S	N	N	N	N	N	●	N	N	N	N															S			
	TIBRVMSG_U8	S	N	N	N	N	N	N	●	N	N	N															S			
	TIBRVMSG_U16	S	N	N	N	N	N	N	N	●	N	N															S			
	TIBRVMSG_U32	S	N	N	N	N	N	N	N	N	●	N															S			
	TIBRVMSG_U64	S	N	N	N	N	N	N	N	N	N	●															S			
	TIBRVMSG_IPADDR32												N		●												S			
	TIBRVMSG_IPPORT16						N			N	N			●													S			
	TIBRVMSG_DATETIME															●											C			
	TIBRVMSG_F32ARRAY															●	N	N	N	N	N	N	N	N	N		S			
	TIBRVMSG_F64ARRAY															N	●	N	N	N	N	N	N	N	N		S			
	TIBRVMSG_I8ARRAY															N	N	●	N	N	N	N	N	N	N		S			
	TIBRVMSG_I16ARRAY															N	N	N	●	N	N	N	N	N	N		S			
	TIBRVMSG_I32ARRAY															N	N	N	N	●	N	N	N	N	N		S			
	TIBRVMSG_I64ARRAY															N	N	N	N	N	●	N	N	N	N		S			
	TIBRVMSG_U8ARRAY															N	N	N	N	N	N	●	N	N	N		S			
	TIBRVMSG_U16ARRAY															N	N	N	N	N	N	N	●	N	N		S			
	TIBRVMSG_U32ARRAY															N	N	N	N	N	N	N	N	●	N		S			
	TIBRVMSG_U64ARRAY															N	N	N	N	N	N	N	N	N	●		S			
	TIBRVMSG_MSG																								●		S			
	TIBRVMSG_OPAQUE																									●	C			
	TIBRVMSG_STRING	P	P	P	P	P	P	P	P	P	P	P	P	P													●			
	TIBRVMSG_XML																									●	C			
	TIBRVMSG_MSGARRAY																											●		
	TIBRVMSG_STRINGARRAY																												●	

Key

● Homologous types; conversion always supported; no loss of information

S Supported conversion; always supported

N Numeric conversion; loss of information is possible(without warning)

P Parsed conversion; supported only when sensible and syntactically correct

C Caution; supported, but results sometimes can be misleading

Unsupported conversion



# Status and Errors

---

# TibrvStatus

*Class*

## Declaration

```
class TibrvStatus
{
    TibrvStatus();
        // Construct empty.
    TibrvStatus(tibrv\_status status); // Construct from C status.
    ~TibrvStatus();                // Reclaim storage.
};
```

## Purpose

Encapsulate status codes.

## Remarks

Status codes are the enumerated values of [tibrv\\_status](#).

Method	Description
<a href="#">TibrvStatus::getCode()</a>	Extract the C status code from a status object.
<a href="#">TibrvStatus::getText()</a>	Extract an interpretive string from a status object.

Constant	Description
TIBRV_OK	The method returned without error.
TIBRV_INIT_FAILURE	Cannot create the network transport.
TIBRV_INVALID_TRANSPORT	The transport has been destroyed, or is otherwise unusable.

Constant	Description
TIBRV_INVALID_ARG	An argument is invalid. Check arguments other than messages, subject names, transports, events, queues and queue groups (which have separate status codes).
TIBRV_NOT_INITIALIZED	The method cannot run because the Rendezvous environment is not initialized (open).
TIBRV_ARG_CONFLICT	Two arguments that require a specific relation are in conflict. For example, the upper end of a numeric range is less than the lower end.
TIBRV_SERVICE_NOT_FOUND	<a href="#">TibrvNetTransport::create()</a> cannot match the service name using <code>getservbyname()</code> .
TIBRV_NETWORK_NOT_FOUND	<a href="#">TibrvNetTransport::create()</a> cannot match the network name using <code>getnetbyname()</code> .
TIBRV_DAEMON_NOT_FOUND	<a href="#">TibrvNetTransport::create()</a> cannot match the daemon port number.
TIBRV_NO_MEMORY	The method could not allocate dynamic storage.
TIBRV_INVALID_SUBJECT	The method received a subject name with incorrect syntax.
TIBRV_DAEMON_NOT_CONNECTED	The Rendezvous daemon process (rvd) exited, or was never started. This status indicates that the program cannot start the daemon and connect to it.
TIBRV_VERSION_MISMATCH	The library, header files and Rendezvous daemon are incompatible.
TIBRV_SUBJECT_COLLISION	It is illegal to create two certified listener events on the same CM transport with overlapping subjects.
TIBRV_VC_NOT_CONNECTED	A virtual circuit terminal was once complete, but is now irreparably broken.

Constant	Description
TIBRV_NOT_PERMITTED	<p>The program attempted an illegal operation.</p> <p>For example:</p> <ul style="list-style-type: none"> <li>• Cannot create ledger file.</li> <li>• Cannot confirm an uncertified message (that is, it has no sequence number).</li> </ul>
TIBRV_INVALID_NAME	The field name is too long; see <a href="#">Field Name Length</a> .
TIBRV_INVALID_TYPE	<ol style="list-style-type: none"> <li>1. The field type is not registered.</li> <li>2. Cannot update field to a type that differs from the existing field's type.</li> </ol>
TIBRV_INVALID_SIZE	The explicit size in the field does not match its explicit type.
TIBRV_INVALID_COUNT	The explicit field count does not match its explicit type.
TIBRV_NOT_FOUND	The method could not find the specified field in the message.
TIBRV_ID_IN_USE	Cannot add this field because its identifier is already present in the message; identifiers must be unique.
TIBRV_ID_CONFLICT	After field search by identifier fails, search by name succeeds, but the actual identifier in the field is non-NULL (so it does not match the identifier supplied).
TIBRV_CONVERSION_FAILED	The method found the specified field, but could not convert it to the desired datatype.
TIBRV_RESERVED_HANDLER	The datatype handler number is reserved for Rendezvous internal datatype handlers.
TIBRV_ENCODER_FAILED	The program's datatype encoder failed.

Constant	Description
TIBRV_DECODER_FAILED	The program's datatype decoder failed.
TIBRV_INVALID_MSG	The method received a message argument that is not a well-formed message; for example, NULL.
TIBRV_INVALID_FIELD	The program supplied an invalid field as an argument.
TIBRV_INVALID_INSTANCE	The program supplied zero as the field instance number (the first instance is number 1).
TIBRV_CORRUPT_MSG	<p>The method detected a corrupt message argument.</p> <p>The most common cause is that the program corrupted storage by accessing the message in two threads simultaneously (without proper locking).</p>
TIBRV_TIMEOUT	<p>A timed dispatch call returned without dispatching an event.</p> <p>A send request call returned without receiving a reply message.</p> <p>A virtual circuit terminal is not yet ready for use.</p>
TIBRV_INTR	Interrupted operation.
TIBRV_INVALID_DISPATCHABLE	The method received an event queue or queue group that has been destroyed, or is otherwise unusable.
TIBRV_INVALID_DISPATCHER	The method received a dispatcher that is invalid or has been destroyed.
TIBRV_INVALID_EVENT	The method received an event that has been destroyed, or is otherwise unusable.
TIBRV_INVALID_CALLBACK	The method received NULL instead of a callback method.
TIBRV_INVALID_QUEUE	The method received a queue that has been destroyed,

Constant	Description
	or is otherwise unusable.
TIBRV_INVALID_QUEUE_GROUP	The method received a queue group that has been destroyed, or is otherwise unusable.
TIBRV_INVALID_TIME_INTERVAL	The method received a negative timer interval.
TIBRV_INVALID_IO_SOURCE	The method received an invalid I/O source (for this operating system).
TIBRV_INVALID_IO_CONDITION	The method received an invalid I/O condition (for this operating system).
TIBRV_SOCKET_LIMIT	The operation failed because of an operating system socket limitation.
TIBRV_OS_ERROR	<a href="#">Tibrv::open()</a> encountered an operating system error.
TIBRV_INSUFFICIENT_BUFFER	The method received a buffer argument that is too small to contain the result.
TIBRV_EOF	End of file.
TIBRV_INVALID_FILE	<ol style="list-style-type: none"> <li>1. Ledger file is not recognizable as such.</li> <li>2. <a href="#">TibrvSdContext:setUserCertWithKey()</a> or <a href="#">TibrvSdContext:setUserCertWithKeyBin()</a> could not complete a certificate file operation; this status code can indicate either disk I/O failure, or invalid certificate data, or an incorrect password.</li> </ol>
TIBRV_FILE_NOT_FOUND	Rendezvous software could not find the specified file.
TIBRV_NOT_FILE_OWNER	<p>The program cannot open the specified file because another program owns it.</p> <p>For example, ledger files are associated with correspondent names.</p>

Constant	Description
TIBRV_IO_FAILED	Cannot write to ledger file.

# TibrvStatus::getCode()

*Method*

## Declaration

```
tibrv_status getCode() const;
```

## Purpose

Extract the C status code from a status object.



# TibrvStatus::getText()

*Method*

## Declaration

```
const char* getText() const;
```

## Purpose

Extract an interpretive string from a status object.

# TIBCO Documentation and Support Services

---

For information about this product, you can read the documentation, contact TIBCO Support, and join TIBCO Community.

## How to Access TIBCO Documentation

Documentation for TIBCO products is available on the [Product Documentation website](#), mainly in HTML and PDF formats.

The [Product Documentation website](#) is updated frequently and is more current than any other documentation included with the product.

## Product-Specific Documentation

The following documentation for this product is available on the [TIBCO Rendezvous® Product Documentation](#) page:

- *TIBCO Rendezvous® Concepts* - Read this book first. It contains basic information about Rendezvous components, principles of operation, programming constructs and techniques, advisory messages, and a glossary. All other books in the documentation set refer to concepts explained in this book.
- *TIBCO Rendezvous® Administration* - Begins with a checklist of action items for system and network administrators. This book describes the mechanics of TIBCO Rendezvous® licensing, network details, plus a chapter for each component of the TIBCO Rendezvous® software suite. Readers should have TIBCO Rendezvous Concepts at hand for reference.
- *TIBCO Rendezvous® Installation* - Includes step-by-step instructions for installing TIBCO Rendezvous® software on various operating system platforms.
- *TIBCO Rendezvous® C Reference* - Detailed descriptions of each data type and function in the TIBCO Rendezvous® C API. Readers should already be familiar with the C programming language, as well as the material in TIBCO Rendezvous Concepts.
- *TIBCO Rendezvous® C++ Reference* - Detailed descriptions of each class and method in the TIBCO Rendezvous® C++ API. The C++ API uses some data types and functions from the C API, so we recommend the TIBCO Rendezvous C Reference as an

additional resource. Readers should already be familiar with the C++ programming language, as well as the material in TIBCO Rendezvous Concepts.

- *TIBCO Rendezvous® .NET Reference* - Detailed descriptions of each class and method in the TIBCO Rendezvous® .NET interface. Readers should already be familiar with either C# or Visual Basic .NET, as well as the material in TIBCO Rendezvous Concepts.
- *TIBCO Rendezvous® Java Reference* - Detailed descriptions of each class and method in the TIBCO Rendezvous® Java language interface. Readers should already be familiar with the Java programming language, as well as the material in TIBCO Rendezvous Concepts.
- *TIBCO Rendezvous® Configuration Tools* - Detailed descriptions of each Java class and method in the TIBCO Rendezvous® configuration API, plus a command line tool that can generate and apply XML documents representing component configurations. Readers should already be familiar with the Java programming language, as well as the material in TIBCO Rendezvous Administration.
- *TIBCO Rendezvous® z/OS Installation and Configuration* - Information about TIBCO Rendezvous® for IBM z/OS systems regarding installation and maintenance. Some information may be also useful for application programmers.
- *TIBCO Rendezvous® Release Notes* - Lists new features, changes in functionality, deprecated features, migration and compatibility information, closed issues and known issues.

To directly access documentation for this product, double-click the following file:

`TIBCO_HOME/release_notes/TIB_rv_8.7.0_docinfo.html`

where `TIBCO_HOME` is the top-level directory in which TIBCO products are installed.

- On Windows, the default `TIBCO_HOME` is `C:\tibco`.
- On UNIX systems, the default `TIBCO_HOME` is `/opt/tibco`.

## How to Contact Support for TIBCO Products

You can contact the Support team in the following ways:

- To access the Support Knowledge Base and getting personalized content about products you are interested in, visit our [product Support website](#).
- To create a Support case, you must have a valid maintenance or support contract with a Cloud Software Group entity. You also need a username and password to log in to the our [product Support website](#). If you do not have a username, you can

request one by clicking **Register** on the website.

## How to Join TIBCO Community

TIBCO Community is the official channel for TIBCO customers, partners, and employee subject matter experts to share and access their collective experience. TIBCO Community offers access to Q&A forums, product wikis, and best practices. It also offers access to extensions, adapters, solution accelerators, and tools that extend and enable customers to gain full value from TIBCO products. In addition, users can submit and vote on feature requests from within the [TIBCO Ideas Portal](#). For a free registration, go to [TIBCO Community](#).

# Legal and Third-Party Notices

---

SOME CLOUD SOFTWARE GROUP, INC. (“CLOUD SG”) SOFTWARE AND CLOUD SERVICES EMBED, BUNDLE, OR OTHERWISE INCLUDE OTHER SOFTWARE, INCLUDING OTHER CLOUD SG SOFTWARE (COLLECTIVELY, “INCLUDED SOFTWARE”). USE OF INCLUDED SOFTWARE IS SOLELY TO ENABLE THE FUNCTIONALITY (OR PROVIDE LIMITED ADD-ON FUNCTIONALITY) OF THE LICENSED CLOUD SG SOFTWARE AND/OR CLOUD SERVICES. THE INCLUDED SOFTWARE IS NOT LICENSED TO BE USED OR ACCESSED BY ANY OTHER CLOUD SG SOFTWARE AND/OR CLOUD SERVICES OR FOR ANY OTHER PURPOSE.

USE OF CLOUD SG SOFTWARE AND CLOUD SERVICES IS SUBJECT TO THE TERMS AND CONDITIONS OF AN AGREEMENT FOUND IN EITHER A SEPARATELY EXECUTED AGREEMENT, OR, IF THERE IS NO SUCH SEPARATE AGREEMENT, THE CLICKWRAP END USER AGREEMENT WHICH IS DISPLAYED WHEN ACCESSING, DOWNLOADING, OR INSTALLING THE SOFTWARE OR CLOUD SERVICES (AND WHICH IS DUPLICATED IN THE LICENSE FILE) OR IF THERE IS NO SUCH LICENSE AGREEMENT OR CLICKWRAP END USER AGREEMENT, THE LICENSE(S) LOCATED IN THE “LICENSE” FILE(S) OF THE SOFTWARE. USE OF THIS DOCUMENT IS SUBJECT TO THOSE SAME TERMS AND CONDITIONS, AND YOUR USE HEREOF SHALL CONSTITUTE ACCEPTANCE OF AND AN AGREEMENT TO BE BOUND BY THE SAME.

This document is subject to U.S. and international copyright laws and treaties. No part of this document may be reproduced in any form without the written authorization of Cloud Software Group, Inc.

TIBCO, the TIBCO logo, the TIBCO O logo, TIB, Information Bus, FTL, eFTL, Rendezvous, and LogLogic are either registered trademarks or trademarks of Cloud Software Group, Inc. in the United States and/or other countries.

Java and all Java based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

All other product and company names and marks mentioned in this document are the property of their respective owners and are mentioned for identification purposes only. You acknowledge that all rights to these third party marks are the exclusive property of their respective owners. Please refer to Cloud SG’s Third Party Trademark Notices (<https://www.cloud.com/legal>) for more information.

This document includes fonts that are licensed under the SIL Open Font License, Version 1.1, which is available at: <https://scripts.sil.org/OFL>

Copyright (c) Paul D. Hunt, with Reserved Font Name Source Sans Pro and Source Code Pro.

Cloud SG software may be available on multiple operating systems. However, not all operating system platforms for a specific software version are released at the same time. See the “readme” file

for the availability of a specific version of Cloud SG software on a specific operating system platform.

THIS DOCUMENT IS PROVIDED “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS DOCUMENT COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THIS DOCUMENT. CLOUD SG MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S), THE PROGRAM(S), AND/OR THE SERVICES DESCRIBED IN THIS DOCUMENT AT ANY TIME WITHOUT NOTICE.

THE CONTENTS OF THIS DOCUMENT MAY BE MODIFIED AND/OR QUALIFIED, DIRECTLY OR INDIRECTLY, BY OTHER DOCUMENTATION WHICH ACCOMPANIES THIS SOFTWARE, INCLUDING BUT NOT LIMITED TO ANY RELEASE NOTES AND "README" FILES.

This and other products of Cloud SG may be covered by registered patents. For details, please refer to the Virtual Patent Marking document located at <https://www.tibco.com/patents>.

Copyright © 1997-2023. Cloud Software Group, Inc. All Rights Reserved.