



TIBCO Rendezvous®

C Reference

Version 8.8.0 | February 2025

Contents

Programmer's Checklist	16
Programming Environment	17
Code	17
Compile	17
Link	17
Run	17
Programming Restrictions	18
Include These Header Files	19
Link These Library Files	20
UNIX	20
Microsoft Windows	21
IBM i	24
IPM Library	27
Environment	28
tibrv_Close()	30
tibrv_IsIPM()	32
tibrv_Open()	33
tibrv_SetCodePages()	35
tibrv_SetRVParameters()	37
tibrv_Version()	39

tibrvSecureDaemon_SetDaemonCert()	40
tibrvSecureDaemon_SetUserCertWithKey()	42
tibrvSecureDaemon_SetUserCertWithKeyBin()	44
tibrvSecureDaemon_SetUserNameWithPassword()	46
Messages	47
Message Operations by Functional Group	48
Message Operations in Alphabetical Order	53
Message Ownership and Control	57
Validity of Data Extracted From Messages	58
Scalar Snapshot	58
Pointer Snapshot	59
Deleting Snapshot References	62
Multiple Subscription Snapshots	63
Field Names and Field Identifiers	64
Finding a Field Instance	66
Strings and Character Encodings	67
tibrvLocalData	68
tibrvMsg	71
tibrvMsgDateTime	72
tibrvMsgField	75
tibrvMsg_AddField()	77
Add Scalar	80

Add Array	82
Add Nested Message	84
Add String	86
Add Opaque Byte Sequence	88
Add XML Byte Sequence	90
Add DateTime	92
tibrvMsg_ClearReferences()	94
tibrvMsg_ConvertToString()	95
tibrvMsg_Create()	97
tibrvMsg_CreateCopy()	99
tibrvMsg_CreateFromBytes()	101
tibrvMsg_Destroy()	103
tibrvMsg_Detach()	105
tibrvMsg_Expand()	107
tibrvMsg_GetAsBytes()	108
tibrvMsg_GetAsBytesCopy()	110
tibrvMsg_GetByteSize()	112
tibrvMsg_GetClosure()	114
tibrvMsg_GetCurrentTime()	115
tibrvMsg_GetEvent()	117

<code>tibrvMsg_GetField()</code>	118
Get Scalar	122
Get Array	125
Get Nested Message	128
Get String	130
Get Opaque Byte Sequence	132
Get XML Byte Sequence	134
Get DateTime	136
<code>tibrvMsg_GetFieldByIndex()</code>	138
<code>tibrvMsg_GetFieldInstance()</code>	140
<code>tibrvMsg_GetNumFields()</code>	142
<code>tibrvMsg_GetReplySubject()</code>	143
<code>tibrvMsg_GetSendSubject()</code>	145
<code>tibrvMsg_MarkReferences()</code>	147
<code>tibrvMsg_RemoveField()</code>	149
<code>tibrvMsg_RemoveFieldInstance()</code>	151
<code>tibrvMsg_Reset()</code>	153
<code>tibrvMsg_SetReplySubject()</code>	154
<code>tibrvMsg_SetSendSubject()</code>	156
<code>tibrvMsg_UpdateField()</code>	158

Update Scalar	162
Update Array	165
Update Nested Message	168
Update String	170
Update Opaque Byte Sequence	172
Update XML Byte Sequence	174
Update DateTime	176
Events	178
Operations by Functional Group	179
Operations in Alphabetical Order	181
tibrvEvent	183
tibrvEventCallback	185
tibrvEventOnComplete	187
tibrvEventVectorCallback	191
tibrvEvent_CreateIO()	193
tibrvEvent_CreateListener()	196
tibrvEvent_CreateTimer()	200
tibrvEvent_CreateVectorListener()	203
tibrvEvent_Destroy()	209
tibrvEvent_DestroyEx()	211

<code>tibrvEvent_GetIOSource()</code>	213
<code>tibrvEvent_GetIOType()</code>	214
<code>tibrvEvent_GetListenerSubject()</code>	215
<code>tibrvEvent_GetListenerTransport()</code>	216
<code>tibrvEvent_GetTimerInterval()</code>	217
<code>tibrvEvent_GetType()</code>	218
<code>tibrvEvent_GetQueue()</code>	219
<code>tibrvEvent_ResetTimerInterval()</code>	220
<code>tibrvEventType</code>	222
<code>tibrvIOType</code>	223
Event Queues	224
Operations by Functional Group	225
Operations in Alphabetical Order	227
<code>tibrvQueue</code>	229
<code>tibrvQueueHook</code>	230
<code>tibrvQueueLimitPolicy</code>	232
<code>tibrvQueueOnComplete</code>	234
<code>tibrvQueue_Create()</code>	236
<code>tibrvQueue_Destroy()</code>	238
<code>tibrvQueue_Dispatch()</code>	240

<code>tibrvQueue_GetCount()</code>	241
<code>tibrvQueue_GetHook()</code>	242
<code>tibrvQueue_GetLimitPolicy()</code>	243
<code>tibrvQueue_GetName()</code>	245
<code>tibrvQueue_GetPriority()</code>	247
<code>tibrvQueue_Poll()</code>	248
<code>tibrvQueue_RemoveHook()</code>	249
<code>tibrvQueue_SetHook()</code>	250
<code>tibrvQueue_SetLimitPolicy()</code>	251
<code>tibrvQueue_SetName()</code>	253
<code>tibrvQueue_SetPriority()</code>	254
<code>tibrvQueue_TimedDispatch()</code>	255
Event Queue Groups	257
<code>tibrvQueueGroup</code>	258
<code>tibrvQueueGroup_Add()</code>	259
<code>tibrvQueueGroup_Create()</code>	260
<code>tibrvQueueGroup_Destroy()</code>	261
<code>tibrvQueueGroup_Dispatch()</code>	262
<code>tibrvQueueGroup_Poll()</code>	263
<code>tibrvQueueGroup_Remove()</code>	264

<code>tibrvQueueGroup_TimedDispatch()</code>	265
<code>Dispatcher Thread</code>	267
<code>tibrvDispatchable</code>	268
<code>tibrvDispatcher</code>	269
<code>tibrvDispatcher_Create()</code>	270
<code>tibrvDispatcher_Destroy()</code>	272
<code>tibrvDispatcher_GetName()</code>	273
<code>tibrvDispatcher_SetName()</code>	274
<code>Transport</code>	275
<code>tibrvTransport</code>	277
<code>tibrvTransportBatchMode</code>	279
<code>tibrvTransport_Create()</code>	280
<code>tibrvTransport_CreateInbox()</code>	283
<code>tibrvTransport_Destroy()</code>	285
<code>tibrvTransport_GetDaemon()</code>	287
<code>tibrvTransport_GetDescription()</code>	289
<code>tibrvTransport_GetNetwork()</code>	291
<code>tibrvTransport_GetService()</code>	292
<code>tibrvTransport_RequestReliability()</code>	293
<code>tibrvTransport_Send()</code>	295

<code>tibrvTransport_Sendv()</code>	296
<code>tibrvTransport_SendReply()</code>	298
<code>tibrvTransport_SendRequest()</code>	300
<code>tibrvTransport_SetBatchMode()</code>	302
<code>tibrvTransport_SetBatchSize()</code>	304
<code>tibrvTransport_SetDescription()</code>	306
Virtual Circuits	308
<code>tibrvTransport_CreateAcceptVc()</code>	309
<code>tibrvTransport_CreateConnectVc()</code>	313
<code>tibrvTransport_WaitForVcConnection()</code>	316
Fault Tolerance	318
<code>tibrvft_Version()</code>	321
<code>tibrvftAction</code>	322
<code>tibrvftMember</code>	324
<code>tibrvftMemberCallback</code>	325
<code>tibrvftMemberOnComplete</code>	327
<code>tibrvftMember_Create()</code>	329
<code>tibrvftMember_Destroy()</code>	333
<code>tibrvftMember_GetGroupName()</code>	335
<code>tibrvftMember_GetQueue()</code>	336

tibrvftMember_GetTransport()	337
tibrvftMember_GetWeight()	338
tibrvftMember_SetWeight()	339
tibrvftMonitor	341
tibrvftMonitorCallback	342
tibrvftMonitorOnComplete	344
tibrvftMonitor_Create()	346
tibrvftMonitor_Destroy()	349
tibrvftMonitor_GetGroupName()	351
tibrvftMonitor_GetQueue()	352
tibrvftMonitor_GetTransport()	353
Certified Message Delivery	354
Operations in Alphabetical Order	360
tibrvcm_Version()	364
tibrvcmEvent	365
tibrvcmEventCallback	367
tibrvcmEvent_ConfirmMsg()	369
tibrvcmEvent_CreateListener()	371
tibrvcmEvent_Destroy()	373
tibrvcmEvent_GetListenerSubject()	375

tibrvcmlEvent_GetListenerTransport()	376
tibrvcmlEvent_GetQueue()	377
tibrvcmlEvent_SetExplicitConfirm()	378
tibrvcmlTransport	380
tibrvcmlTransportOnComplete	381
tibrvcmlTransport_AddListener()	383
tibrvcmlTransport_AllowListener()	385
tibrvcmlTransport_Create()	386
tibrvcmlTransport_Destroy()	390
tibrvcmlTransport_DisallowListener()	392
tibrvcmlTransport_ExpireMessages()	394
tibrvcmlTransport_GetDefaultCMTimeLimit()	396
tibrvcmlTransport_GetLedgerName()	398
tibrvcmlTransport_GetName()	399
tibrvcmlTransport_GetRequestOld()	400
tibrvcmlTransport_GetSyncLedger()	401
tibrvcmlTransport_GetTransport()	402
tibrvcmlTransport_RemoveListener()	403
tibrvcmlTransport_RemoveSendState()	405
tibrvcmlTransport_ReviewLedger()	407

tibrvcmReviewCallback	409
tibrvcmTransport_Send()	412
tibrvcmTransport_SendReply()	414
tibrvcmTransport_SendRequest()	416
tibrvcmTransport_SetDefaultCMTimeLimit()	419
tibrvcmTransport_SetPublisherInactivityDiscardInterval()	421
tibrvcmTransport_SyncLedger()	423
tibrvMsg_GetCMSender()	425
tibrvMsg_GetCMSequence()	427
tibrvMsg_GetCMTimeLimit()	429
tibrvMsg_SetCMTimeLimit()	431
Distributed Queues	433
Operations in Alphabetical Order	434
Distributed Queue Overview	435
tibrvcmTransport_CreateDistributedQueue()	436
tibrvcmTransport_GetCompleteTime()	442
tibrvcmTransport_GetUnassignedMessageCount()	443
tibrvcmTransport_GetTaskBacklogLimits()	444
tibrvcmTransport_GetWorkerWeight()	446
tibrvcmTransport_GetWorkerTasks()	447

tibrvcTransport_SetCompleteTime()	448
tibrvcTransport_SetTaskBacklogLimit...()	450
tibrvcTransport_SetWorkerWeight()	452
tibrvcTransport_SetWorkerTasks()	453
Datatypes	455
Wire Format Datatypes	456
C Datatypes	459
Datatype Conversion	461
General Rules	461
Converting to Boolean	462
Status	464
tibrv_status	465
tibrvStatus_GetText()	470
Custom Datatypes	471
Operations by Functional Group	472
Operations in Alphabetical Order	473
Architecture Overview	474
Adding Data	475
Extracting Data	476
Custom Datatype Checklist	477
Convenience Functions	478
Get	478

Add and Update	478
tibrvMsg_SetHandlers()	480
tibrvMsgDataType	482
tibrvMsgData_ByteSize()	484
tibrvMsgData_Converter	485
tibrvMsgData_CopyBytes()	487
tibrvMsgData_Decoder	489
tibrvMsgData_Encoder	491
tibrvMsgData_GetBytes()	494
tibrvMsgData_GetSize()	496
tibrvMsgData_Malloc()	498
tibrvMsgData_SetSize()	500
TIBCO Documentation and Support Services	502
Legal and Third-Party Notices	504

Programmer's Checklist

This section presents important general information for programmers using TIBCO Rendezvous® software.

Programming Environment

When developing or deploying TIBCO Rendezvous® C programs, remember these steps.

Code

- Include the appropriate Rendezvous C header files; see [Include These Header Files](#).
- Include the same header files whether the program communicates through a Rendezvous daemon or using TIBCO Rendezvous® Server In-Process Module (IPM).

Compile

- Compile programs with an ANSI-compliant C compiler.

Link

- Link with the appropriate Rendezvous C library files; see [Link These Library Files](#).

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 functions 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
Communications, Events and Data	
All programs must include this API header file.	
tibrv/tibrv.h	Include this header file for the Rendezvous C API.
Certified Message Delivery and Distributed Queue	
tibrv/cm.h	Include this header file for the Rendezvous certified message delivery and distributed queue C API. Including cm.h automatically includes tibrv.h too.
Fault Tolerance	
tibrv/ft.h	Include this header file for the Rendezvous fault tolerance C API. Including ft.h automatically includes tibrv.h too.
Secure Daemons and OpenSSL	
Programs that connect to secure daemons (rvsd, rvsrd) must include this header file.	
tibrv/sd.h	Include this header file for the Rendezvous secure daemon C API.

Link These Library Files

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

- [UNIX](#)
- [Microsoft Windows](#)
- [IBM i](#)

See Also

[IPM Library](#)[IPM Library](#)[IPM Library](#)

TIBCO Rendezvous for z/OS Installation and Configuration

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
Communications, Data and Event Manager	
All programs must link this library.	
<code>-ltibrv</code>	Programs that connect to ordinary daemons (<code>rxd</code> , <code>rverd</code>) must link using this library flag.
Secure Daemon	

Linker Flag	Description
<code>-ltibrvsd</code> <code>-lssl</code> <code>-lcrypto</code>	Programs that connect to secure daemons (rvsd, rvsrd) must also link using these three flags.

Certified Message Delivery, Fault Tolerance, and Distributed Queues

Programs may also link one or more of these libraries as needed.

<code>-ltibrvcn</code>	<p>Programs that use certified message delivery must link using this library flag.</p> <p>Programs that use distributed queues must link using this library flag.</p>
<code>-ltibrvft</code>	<p>Programs that use fault tolerance features must link using this library flag.</p> <p>Programs that use distributed queues must link using this library flag.</p>
<code>-ltibrvcmq</code>	<p>Programs that use distributed queues 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

Release 8.8.0 supports the following:

- 64-bit Windows client libraries and daemons
- 32-bit Windows, non-secure libraries only

Both DLLs and static libraries are available. We recommend DLLs to ease forward migration.

Library Files for Microsoft Windows

Library File	Description
Communications, Data and Event Manager	
All programs must link <i>only one</i> of these libraries.	
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 libssl.lib	Secure daemon additions. DLL (import library).
libtibrvsd.lib libcrypto.lib libssl.lib	Secure daemon additions. Static library, for use with the /MT compiler option.
libtibrvsdmd.lib libcrypto.lib libssl.lib	Secure daemon additions. Static library, for use with the /MD compiler option.

Library File	Description
--------------	-------------

Certified Message Delivery

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

Programs that use **distributed queues** must link *only one* 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 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.
libtibrvftmdlib	Rendezvous fault tolerance software.

Library File	Description
	Static library, for use with the /MD compiler option.

Distributed Queue

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

IBM i

This section contains information specific to IBM i platforms.

Compile & Link



Note

In the following command lines, the slash character (/) is a literal; it does not mean *either or*.

Procedure

1. Add the TIBCO Rendezvous installation library to the library search path:


```
ADDLIB (RV_installation_library)
```

2. Compile:

```
CRTCMOD MODULE (module_library_name/module_name) SRCFILE (source_library_
name/source_file_name) SRCMBR (source_member) LONGLVL (*EXTENDED) PFROPT
(*STRDONLY) DEFINE(_MULTI_THREADED)
```

3. Link:

```
CRTPGM PGM (program_library/program_name) MODULE (module_library_
name/module_name) BNDSRVPGM (optional_RV_libraries LIBTIBRV)
```

Library Files

Library Files for IBM i

Library File	Description
Communications, Data and Event Manager	
All programs must link this library.	
LIBTIBRV	Rendezvous C library.
Certified Message Delivery	
Programs that use certified message delivery must link this library.	
Programs that use distributed queues must link this library.	
LIBTIBRVCM	Rendezvous certified message delivery software.
Fault Tolerance	
Programs that use fault tolerance must link this library.	
Programs that use distributed queues must link this library.	

Library File	Description
LIBTIBRVFT	Rendezvous fault tolerance software.

Distributed Queue

Programs that use **distributed queues** must link this library.

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

LIBTIBRVC0	Rendezvous distributed queue software.
------------	--

IPM Library

Linking commands are identical whether a C program uses standard Rendezvous communication library or the IPM library. To select between these two libraries, modify the library path environment variable according to [Selecting the Communications Library](#).

Selecting the Communications Library

Library	Instructions
Rendezvous Standard Communications Library	<p>Ensure that the subdirectory <code>TIBCO_HOME/lib</code> appears <i>before</i> <code>TIBCO_HOME/lib/ipm</code> in your library path environment variable.</p> <p>For static linking, you may reference the standard library by an explicit pathname (<code>TIBCO_HOME/lib/library_name</code>).</p>
Rendezvous IPM Communications Library	<p>Ensure that the subdirectory <code>TIBCO_HOME/lib/ipm</code> appears <i>before</i> <code>TIBCO_HOME/lib</code> in your library path environment variable.</p> <p>For static linking, you may (as an alternative) reference the standard library by an explicit pathname (<code>TIBCO_HOME/lib/ipm/library_name</code>).</p>

Existing Rendezvous applications linked against the standard shared library do not require modifications in order to link instead against the IPM library.

Environment

This section describes the functions relating to the global internal machinery upon which Rendezvous software depends.

Function	Description
tibrv_Close()	Stop and destroy Rendezvous internal machinery.
tibrv_Open()	Create and start Rendezvous internal machinery.
tibrv_Version()	Identify the Rendezvous API release number.
Secure Daemon	
tibrvSecureDaemon_SetDaemonCert()	Register trust in a secure daemon.
tibrvSecureDaemon_SetUserCertWithKey()	Register a (PEM) certificate with private key for identification to secure daemons.
tibrvSecureDaemon_SetUserCertWithKeyBin()	Register a (PKCS #12) certificate with private key for identification to secure daemons.
tibrvSecureDaemon_SetUserNameWithPassword()	Register a (PEM) certificate with private key for identification to secure daemons.

Function	Description
IPM	
tibrv_IsIPM()	Test whether IPM is operating.
tibrv_OpenEx()	Create and start Rendezvous internal machinery.
tibrv_SetRVParameters()	Set TIBCO Rendezvous daemon command line parameters for IPM.
EBCDIC	
tibrv_SetCodePages()	Set code pages for automatic string conversion on EBCDIC platforms (namely, IBM i and z/OS).

tibrv_Close()

Function

Declaration

```
tibrv_status tibrv_Close(void);
```

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 and queues are unusable, so programs can no longer dispatch events.
- All transports are unusable, so programs can no longer send outbound messages.

`tibrv_Close()` frees storage associated with events, queues, queue groups, transports and dispatcher threads. However, `tibrv_Close()` does not destroy Rendezvous messages; the program must explicitly destroy messages to reclaim their storage.

Programs can call `tibrv_Close()` in any thread.

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_IsIPM()

Function

Declaration

```
tibrv_bool tibrv_IsIPM(void);
```

Purpose

Test whether IPM is operating.

Remarks

You can use this call to determine whether an application program process has linked the IPM library. You can test that your program dynamically links the correct library. You can program different behavior depending on which library is linked.

TIBRV_TRUE indicates that the program links the IPM library (from the `lib/ipm/` subdirectory).

TIBRV_FALSE indicates that the program links the standard Rendezvous library (from the `lib/` directory).

tibrv_Open()

Function

Declaration

```
tibrv_status tibrv_Open(void);  
tibrv_status tibrv_OpenEx(  
    const char    *pathname);
```

Purpose

Create and 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, queues, and transports are unusable. However, calls that manipulate messages do not require this machinery, and programs may use them before calling `tibrv_Open()`.

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.

IPM

Programs that use IPM can start the Rendezvous machinery either with `tibrv_Open` or with `tibrv_OpenEx`. The extended call is available only with IPM. When IPM is not available, the extended call fails with error status.

The extended call accepts a filepath name, which explicitly specifies a configuration file. IPM reads parameter values from that file.

Parameter	Description
pathname	<p>Programs that use IPM can supply a filepath name, which explicitly specifies a configuration file. IPM reads parameter values from that file.</p> <p>For details, see Configuring IPM in TIBCO Rendezvous Concepts.</p> <p>When IPM is not available, this version of the method fails with error status.</p> <p>Not supported for Visual Basic.</p>

IPM: Specifying a Configuration File

```
char* cfgfile = "/var/tmp/mycfgfile"
tibrv_OpenEx(cfgfile);
```

See Also

[tibrv_Close\(\)](#)

[Configuring IPM in TIBCO Rendezvous Concepts](#)

tibrv_SetCodePages()

Function

Declaration

```
tibrv_status tibrv_SetCodePages(  
    char* host_codepage,  
    char* net_codepage);
```

Purpose

Set code pages for automatic string conversion on EBCDIC platforms (namely, IBM i and z/OS).

String Conversion

Rendezvous software uses the operating system's `iconv()` call to automatically convert strings. This automatic conversion applies to strings within message fields, field names, subject names, and other strings associated with messages (even when they are not strictly inside a message). Conversion occurs only as needed:

- Programs running in EBCDIC environments represent all strings using an EBCDIC code page (called the *host code page*). Before inserting strings into a message object, Rendezvous software converts those strings to an ASCII character set (the *network code page*).
- Conversely, when extracting strings from a message object, Rendezvous software converts those strings to the EBCDIC host code page before presenting the strings to the program.

Remarks

This call sets the host and network code pages for string conversions in EBCDIC environments. On other operating system platforms, this call has no effect, and returns without error.

Call this function when the system code pages differ from the Rendezvous default code pages (see the table of [Default Code Pages](#)). Throughout an enterprise, all sending and receiving programs must use the same code pages.

Both arguments are string names of code pages. To determine valid code page names for your operating system, see documentation from the operating system vendor.

Programs may call this function at most once. The call *must* precede the first call to [tibrv_Open\(\)](#).

Parameter	Description
<code>host_codepage</code>	Set this code page as the native (EBCDIC) character encoding for the host computer.
<code>net_codepage</code>	Set this code page as the ASCII character set for the network.

Default Code Pages

To use a default code page, programs may supply `NULL` for either parameter. Using the default code pages in both parameter positions has the same effect as not calling this function at all.

OS Platform	Default Host Code Page	Default Network Code Page
IBM i	"00000" This value instructs IBM i routines to use the system-defined default code page.	"00819"
z/OS	"IBM-1047"	"ISO8859-1"

See Also

[Strings and Character Encodings](#)

tibrv_SetRVParameters()

Function

Declaration

```
tibrv_status tibrv_SetRVParameters(  
    tibrv_u32      argc,  
    const char**   argv );
```

Purpose

Set TIBCO Rendezvous daemon command line parameters for IPM.

Remarks

The TIBCO Rendezvous daemon process (`rvd`) accepts several command line parameters. When IPM serves the role of the daemon, this function lets you supply those parameters from within the application program.

This call is optional. When this call is present, it *must* precede the call to [tibrv_Open\(\)](#). For interaction semantics, see Parameter Configuration—Precedence and Interaction in TIBCO Rendezvous Concepts.

This call is available only with IPM. When IPM is not available, this call fails with error status.

Parameter	Description
<code>argc</code>	Supply the number of strings in the argument vector.
<code>argv</code>	Supply an array of null-terminated strings. Each string is either a command line parameter name (for example, <code>-logfile</code>) or its value. For details about parameters, see <code>rvd</code> in TIBCO Rendezvous Administration

IPM: Configuring Parameters In Program Code

```
const char* rvParams[] = {"-reliability", "3",  
                          "-reuse-port", "30000"};  
tibrv_SetRVParameters(sizeof(rvParams)/sizeof(char*), rvParams);  
tibrv_Open();
```

See Also

Configuring IPM in TIBCO Rendezvous Concepts

`tibrv_Version()`

Function

Declaration

```
const char* tibrv_Version(void);
```

Purpose

Identify the Rendezvous API release number.

tibrvSecureDaemon_SetDaemonCert()

Function

Declaration

```
tibrv_status tibrvSecureDaemon_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 function identify trustworthy daemons. Programs divulge usernames 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 Daemon Name .
daemonCert	Register this public certificate. The text of this certificate must be in PEM encoding. See also Certificate .

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 [tibrvTransport_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.

`tibrvSecureDaemon_SetUserCertWithKey()`

Function

Declaration

```
tibrv_status tibrvSecureDaemon_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 functions that achieve similar effects:

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

Parameter	Description
<code>userCertWithKey</code>	Register this user certificate with private key. The text of these certificates must be in PEM encoding.
<code>password</code>	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

[tibrvSecureDaemon_SetUserCertWithKeyBin\(\)](#)

`tibrvSecureDaemon_SetUserCertWithKeyBin()`

Function

Declaration

```
tibrv_status tibrvSecureDaemon_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 functions that achieve similar effects:

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

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

[tibrvSecureDaemon_SetUserCertWithKey\(\)](#)

www.rsasecurity.com/rsalabs/pkcs

tibrvSecureDaemon_ SetUserNameWithPassword()

Function

Declaration

```
tibrv_status tibrvSecureDaemon_SetUserNameWithPassword(  
    const char*    userName,  
    const char*    password);
```

Purpose

Register a username 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 username 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.

Messages

Messages are the central unit of information that Rendezvous programs exchange.

This section describes messages and the functions that manipulate them.

Message Operations by Functional Group

Function	Description
Message Life Cycle	
tibrvMsg_Create()	Allocate storage and initialize it as a new message.
tibrvMsg_CreateCopy()	Copy a message.
tibrvMsg_Destroy()	Destroy a message; free the storage that it occupies.
tibrvMsg_Detach()	Detach an inbound message from Rendezvous storage; the program assumes responsibility for destroying the message.
tibrvMsg_Expand()	Enlarge a message by allocating additional storage.
tibrvMsg_Reset()	Clear a message, preparing it for re-use.
Message Attributes	
tibrvMsg_GetNumFields()	Extract the number of fields in a message.
tibrvMsg_GetByteSize()	Return the size of a message (in bytes).
tibrvMsg_ConvertToString()	Format a message as a string.

Function	Description
Address Attributes	
tibrvMsg_GetSendSubject()	Extract the subject from a message.
tibrvMsg_SetSendSubject()	Set the subject for a message.
tibrvMsg_GetReplySubject()	Extract the reply subject from a message.
tibrvMsg_SetReplySubject()	Set the reply subject for a message.
Message Fields	
tibrvMsg_AddField()	Add a field to a message.
	Add Scalar
	Add Array
	Add Nested Message
	Add String
	Add Opaque Byte Sequence
	Add XML Byte Sequence
	Add DateTime

Function	Description
tibrvMsg_GetField()	Get a specified field from a message.
	Get Scalar
	Get Array
	Get Nested Message
	Get String
	Get Opaque Byte Sequence
	Get XML Byte Sequence
	Get DateTime
tibrvMsg_GetFieldByIndex()	Get a field from a message by an index.
tibrvMsg_GetFieldInstance()	Get a specified instance of a field from a message.
tibrvMsg_RemoveField()	Remove a field from a message.
tibrvMsg_RemoveFieldInstance()	Remove a specified instance of a field from a message.

Function	Description
tibrvMsg_UpdateField()	Update a field within a message.
	Update Scalar
	Update Array
	Update Nested Message
	Update String
	Update Opaque Byte Sequence
	Update XML Byte Sequence
	Update DateTime

Archiving Messages as Byte Sequences

tibrvMsg_CreateFromBytes()	Create a new message, and populate it with data.
tibrvMsg_GetAsBytes()	Extract the data from a message as a byte sequence.
tibrvMsg_GetAsBytesCopy()	Extract a copy of the data from a message as a byte sequence.

Deleting Snapshot References

tibrvMsg_ClearReferences()	Clear references in this message.
tibrvMsg_MarkReferences()	Mark references in this message.

Dispatch Attributes

tibrvMsg_GetClosure()	Extract the closure object
---------------------------------------	----------------------------

Function	Description
	corresponding to a (dispatched) message object.
<code>tibrvMsg_GetEvent()</code>	Extract the event associated with a (dispatched) message object.
Utilities	
<code>tibrvMsg_GetCurrentTime()</code>	Get the current time.
Types	
<code>tibrvLocalData</code>	This type is the union of all the datatypes that a Rendezvous message can contain as data in a message field.
<code>tibrvMsg</code>	This type represents inbound or outbound Rendezvous messages.
<code>tibrvMsgDateTime</code>	Represents dates and times.
<code>tibrvMsgField</code>	Fields hold data within messages. Programs manipulate the content of field structs using the public struct accessors of this type.
Custom Datatypes	
<code>tibrvMsg_SetHandlers()</code>	Define a program-specific datatype, by registering functions to transfer it between local format and wire format, and convert it to other datatypes.

Message Operations in Alphabetical Order

Function	Description
tibrvLocalData	This type is the union of all the datatypes that a Rendezvous message can contain as data in a message field.
tibrvMsg	This type represents inbound or outbound Rendezvous messages.
tibrvMsgDateTime	Represents dates and times.
tibrvMsgField	Fields hold data within messages. Programs manipulate the content of field structs using the public struct accessors of this type.
tibrvMsg_AddField()	Add a field to a message.
	Add Scalar
	Add Array
	Add Nested Message
	Add String
	Add Opaque Byte Sequence
	Add XML Byte Sequence
	Add DateTime

Function	Description
tibrvMsg_ClearReferences()	Clear references in this message.
tibrvMsg_ConvertToString()	Format a message as a string.
tibrvMsg_Create()	Allocate storage and initialize it as a new message.
tibrvMsg_CreateCopy()	Copy a message.
tibrvMsg_CreateFromBytes()	Create a new message, and populate it with data.
tibrvMsg_Destroy()	Destroy a message; free the storage that it occupies.
tibrvMsg_Detach()	Detach an inbound message from Rendezvous storage; the program assumes responsibility for destroying the message.
tibrvMsg_Expand()	Enlarge a message by allocating additional storage.
tibrvMsg_GetAsBytes()	Extract the data from a message as a byte sequence.
tibrvMsg_GetAsBytesCopy()	Extract a copy of the data from a message as a byte sequence.
tibrvMsg_GetByteSize()	Return the size of a message (in bytes).
tibrvMsg_GetClosure()	Extract the closure object corresponding to a (dispatched) message object.
tibrvMsg_GetCurrentTime()	Get the current time.

Function	Description
tibrvMsg_GetEvent()	Extract the event associated with a (dispatched) message object.
tibrvMsg_GetField()	Get a specified field from a message. <hr/> Get Scalar <hr/> Get Array <hr/> Get Nested Message <hr/> Get String <hr/> Get Opaque Byte Sequence <hr/> Get XML Byte Sequence <hr/> Get DateTime
tibrvMsg_GetFieldByIndex()	Get a field from a message by an index.
tibrvMsg_GetFieldInstance()	Get a specified instance of a field from a message.
tibrvMsg_GetNumFields()	Extract the number of fields in a message.
tibrvMsg_GetReplySubject()	Extract the reply subject from a message.
tibrvMsg_MarkReferences()	Mark references in this message.
tibrvMsg_GetSendSubject()	Extract the subject from a message.
tibrvMsg_RemoveField()	Remove a field from a message.

Function	Description
tibrvMsg_RemoveFieldInstance()	Remove a specified instance of a field from a message.
tibrvMsg_Reset()	Clear a message, preparing it for re-use.
tibrvMsg_SetHandlers()	Define a program-specific datatype, by registering functions to transfer it between local format and wire format, and convert it to other datatypes.
tibrvMsg_SetReplySubject()	Set the reply subject for a message.
tibrvMsg_SetSendSubject()	Set the subject for a message.
tibrvMsg_UpdateField()	Update a field within a message.
	Update Scalar
	Update Array
	Update Nested Message
	Update String
	Update Opaque Byte Sequence
	Update XML Byte Sequence
	Update DateTime

Message Ownership and Control

When a C program creates a message, the program *owns* that message. The program is responsible for destroying that message when it no longer needs the storage. That is, every *create* message operation must be paired with a *destroy* message operation.

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

Rendezvous software *controls* the storage in which all messages reside—even messages that the program owns. Programs must not directly modify the storage in which a message resides. Programs may change the contents of a message only by using Rendezvous functions that add, remove or update fields.

Validity of Data Extracted From Messages

To extract data values from a message, programs use a set of *get* convenience functions. All of these functions 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.

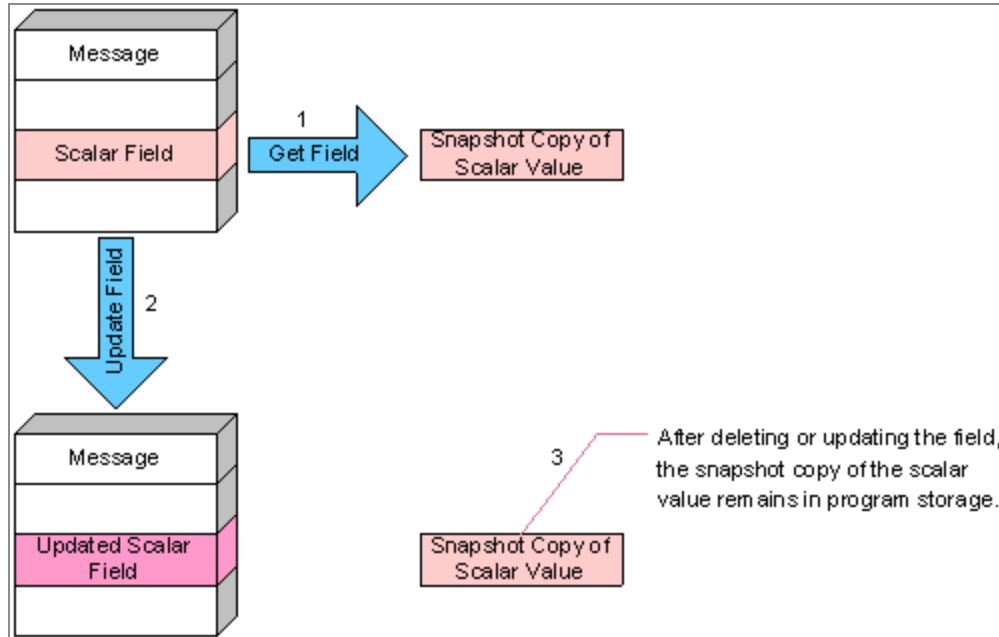
Rendezvous messages implement snapshot semantics using two separate strategies for scalar data and pointer data.

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* function; the *get* function 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

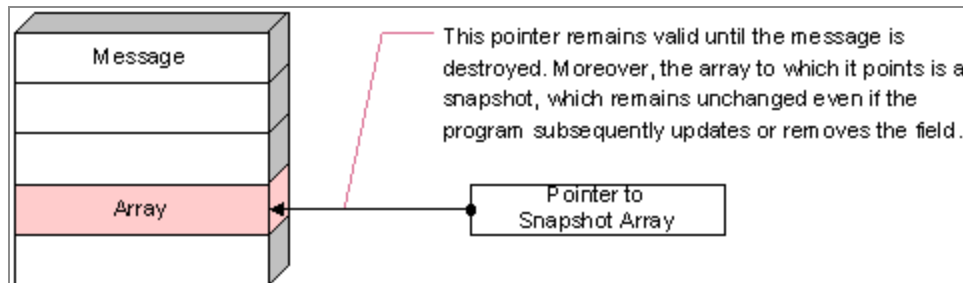


Pointer Snapshot

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

To extract the value of a pointer field, a program declares a typed pointer in program-owned storage, and passes its address to the *get* function; the *get* function copies a *pointer* to the field value into the program's pointer address. The function 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.)

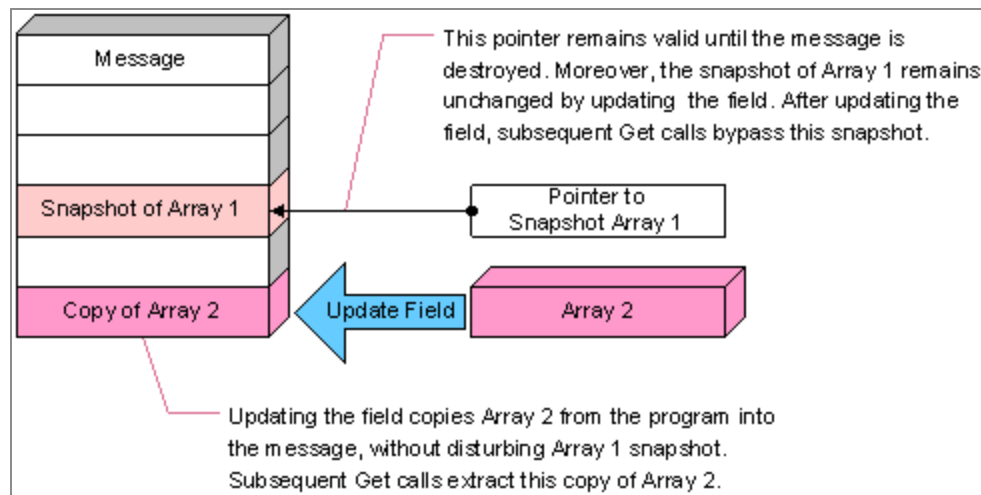
Rendezvous Protects Pointer Snapshots from Changes to the Message

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* function 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 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.

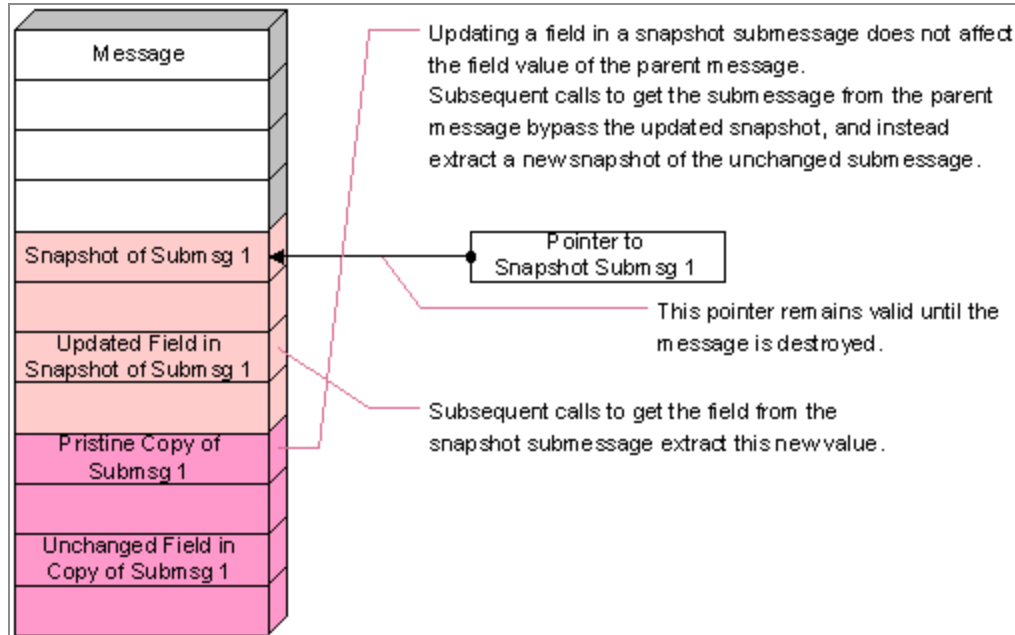
Rendezvous Protects the Message from Changes to Submessage Snapshots

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 functions 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 functions 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 function 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 functions 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 functions 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 function modifies an inbound message (whether detached or not), Rendezvous software still presents the original message content to subsequent callback functions.

Field Names and Field Identifiers

In Rendezvous 5 and earlier releases, programs would specify fields within a message using a field name. In Rendezvous 6 and later releases, 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.

Regular message functions specify fields using a field name. Extended functions specify fields using a combination of a field name and a unique field identifier.

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 functions that *get* message fields (including functions that *update* or *remove* fields, since these functions 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. Extended functions begin here. Regular functions begin at step 3.

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. Regular functions begin here. Extended functions also begin here when the program supplied *zero* as the identifier (supplying zero is equivalent to not supplying a field identifier, so the behavior is equivalent to the corresponding regular function).

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 uses an extended function to add 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 and returns the status code `TIBRV_ID_IN_USE`.

Passing Field Name and Identifiers

The program can pass the field name and identifier to functions in either of two ways:

- All extended functions accept two separate parameters, named `fieldName` and `fieldId`.
- `tibrvMsg_AddField()` and `tibrvMsg_UpdateField()` do *not* have extended functions. Instead, they accept a `field` parameter, which contains these two items as members within a `tibrvMsgField` struct. (The convenience functions that add and update fields *do* have extended functions.)

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 functions find a specific instance by name and number (they do not use field identifiers):

- [tibrvMsg_RemoveFieldInstance\(\)](#).
- [tibrvMsg_GetFieldInstance\(\)](#).

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.

EBCDIC

For information about string encoding in EBCDIC environments, see [tibrv_SetCodePages\(\)](#).

tibrvLocalData

Type


Declaration

```
typedef union {  
    tibrvMsg          msg;  
    char*             str;  
    void*             buf;  
    void*             array;  
    tibrv_bool        boolean;  
    tibrv_i8           i8;  
    tibrv_u8           u8;  
    tibrv_i16          i16;  
    tibrv_u16          u16;  
    tibrv_i32          i32;  
    tibrv_u32          u32;  
    tibrv_i64          i64;  
    tibrv_u64          u64;  
    tibrv_f32          f32;  
    tibrv_f64          f64;  
    tibrv_ipport16     ipport16;  
    tibrv_ipaddr32     ipaddr32;  
    tibrvMsgDateTime   date;  
} tibrvLocalData;
```

Purpose

This type is the union of all the datatypes that a Rendezvous message can contain as data in a message field.

Accessor	Description
msg	Rendezvous message
str	character string; NULL-terminated; encoding depends on ISO locale

Accessor	Description
buf	opaque buffer, or XML data buffer (<i>uncompressed</i>)
 Warning	Do not use this field to access data that requires memory alignment; it does not necessarily preserve alignment. Instead, see Add Opaque Byte Sequence , Get Opaque Byte Sequence , or Update Opaque Byte Sequence .
array	array (any valid element type)
boolean	boolean
i8	8-bit integer
u8	8-bit unsigned integer
i16	16-bit integer
u16	16-bit unsigned integer
i32	32-bit integer
u32	32-bit unsigned integer
i64	64-bit integer
u64	64-bit unsigned integer
f32	32-bit floating point number
f64	64-bit floating point number
ipport16	2-byte IP port, in network byte order
ipaddr32	4-byte IP address, in network byte order
date	Rendezvous date-time value; see tibrvMsgDateTime

See Also

[Wire Format Datatypes](#)

[C Datatypes](#)

[Datatype Conversion](#)

`tibrvMsg`

Type

Declaration

```
typedef void* tibrvMsg;
```

Purpose

This type represents inbound or outbound Rendezvous messages.

Remarks

Programs manipulate messages using the calls in this section.

tibrvMsgDateTime

Type

Declaration

```
typedef struct
{
    tibrv_i64    sec;
    tibrv_u32    nsec;
} tibrvMsgDateTime;
```

Purpose

Represents dates and times.

Accessor	Description
sec	Signed 64-bit integer representing seconds.
nsec	Unsigned 32-bit integer representing nanoseconds <i>after the seconds value</i> . This value is always non-negative, between zero and 999999999. 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 before midnight of December 31, 1969 is - 1 seconds plus 500,000,000 nanoseconds.

Converting Dates 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.

Representations

Rendezvous software represents time values in two ways—one within 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 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 programs— <code>TIBRVMSG_DATETIME_SEC_MAX</code> and <code>TIBRVMSG_DATETIME_SEC_MIN</code>.</p>								
	<table> <tr> <td>range in years</td><td>292,471,208,677</td></tr> <tr> <td>range in seconds</td><td>9,223,372,036,854,775,807</td></tr> <tr> <td>restricted range in seconds</td><td>549,755,813,887</td></tr> <tr> <td>restricted range in millisecs</td><td>549,755,813,887,000</td></tr> </table>	range in years	292,471,208,677	range in seconds	9,223,372,036,854,775,807	restricted range in seconds	549,755,813,887	restricted range in millisecs	549,755,813,887,000
range in years	292,471,208,677								
range in seconds	9,223,372,036,854,775,807								
restricted range in seconds	549,755,813,887								
restricted range in millisecs	549,755,813,887,000								

Representation	Details
Rendezvous wire format	Seconds as a 40-bit signed integer, plus microseconds as a 24-bit unsigned integer.
	range in years 17,432
	range in seconds 549,755,813,887
	range in milliseconds 549,755,813,887,000

See Also

[Add DateTime](#)

[tibrvMsg_ConvertToString\(\)](#)

[tibrvMsg_GetCurrentTime\(\)](#)

[Get DateTime](#)

[Update DateTime](#)

tibrvMsgField

Type

Declaration

```
typedef struct
{
    char*          name;
    tibrv_u32      size;
    tibrv_u32      count;
    tibrvLocalData data;
    tibrv_u16      id;
    tibrv_u8       type;
} tibrvMsgField;
```

Purpose

Fields hold data within messages. Programs manipulate the content of field structs using the public struct accessors of this type.

Accessor	Description
name	The field's name. NULL signifies the empty string as the field name. Field name strings use the ISO 8859-1 character encoding.
size	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 bytes of data.
count	The number of values in an array field. For array types, count is the number of array elements.

Accessor	Description
	<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, opaque byte sequences and XML data, 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	The field's data value.
id	The field's identifier. Identifiers are optional, but must be unique within each message.
type	A Rendezvous datatype token denoting the type of the field's data.

tibrvMsg_AddField()

Function

Declaration

```
tibrv_status tibrvMsg_AddField(  
    tibrvMsg      message,  
    tibrvMsgField* field);
```

Purpose

Add a field to a message.

Remarks

This function copies the data into the new message field. All related convenience functions behave similarly.

Parameter	Description
message	Add the new field to this message.
field	Add this field to the message.

Adding Fields to a Nested Message

Earlier releases of Rendezvous software allowed programs to append fields to a nested submessage under certain conditions. Starting with release 6, Rendezvous software no longer supports this special case convenience. Instead, programs must use this three-step process:

Procedure

1. Extract the nested submessage, using `tibrvMsg_getMsg()`.

2. Add the new fields to the extracted submessage, using type-specific convenience functions or this function. 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, using `TibrvMsg_UpdateMsg()`.

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 zeroth 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 Functions

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

- Code readability.

- Type checking.
- Accept constants and literals directly.

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

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

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

Extended Functions

Each convenience function has two forms.

- The usual form specifies the field to add using a field name.
- The extended form specifies the field to add using a field name and a field identifier.

For example, the function `tibrvMsg_AddI32()` is paired with the extended form `tibrvMsg_AddI32Ex()`.

Field identifiers have type `tibrv_u16`. 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.

For more information, see [Adding a New Field](#).

Add Scalar

Convenience Functions

Declaration

```
tibrv_status  
tibrvMsg_AddScalar_type(  
    tibrvMsg      message,  
    const char*   fieldName,  
    scalar_type   value);  
tibrv_status  
tibrvMsg_AddScalar_typeEx(  
    tibrvMsg      message,  
    const char*   fieldName,  
    scalar_type   value,  
    tibrv_i16     fieldId);
```

Purpose

Add a field containing a scalar value.

Function Name	Field Value Type	Type Description
tibrvMsg_Add Bool	tibrv_ bool	boolean
tibrvMsg_Add F32	tibrv_ f32	32-bit floating point
tibrvMsg_Add F64	tibrv_ f64	64-bit floating point
tibrvMsg_Add I8	tibrv_ i8	8-bit integer
tibrvMsg_Add I16	tibrv_ i16	16-bit integer
tibrvMsg_Add I32	tibrv_ i32	32-bit integer

Function Name	Field Value Type	Type Description
tibrvMsg_Add I64	tibrv_ i64	64-bit integer
tibrvMsg_Add U8	tibrv_ u8	8-bit unsigned integer
tibrvMsg_Add U16	tibrv_ u16	16-bit unsigned integer
tibrvMsg_Add U32	tibrv_ u32	32-bit unsigned integer
tibrvMsg_Add U64	tibrv_ u64	64-bit unsigned integer
tibrvMsg_Add IPAddr32	tibrv_ ipaddr32	4-byte IP address
tibrvMsg_Add IPPort16	tibrv_ ipport16	2-byte IP port

Parameter	Description
message	Add the new field to this message.
fieldName	Create the new field with this name.
value	<p>Add a new field with this value (which may be a literal or stored in a variable).</p> <p>The convenience function must correspond to the datatype of this value. We recommend casting the data to match the convenience function.</p> <p>The function copies the value into the new message 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 Array

Convenience Functions

Declaration

```
tibrv_status
tibrvMsg_Addelement_typeArray(
    tibrvMsg      message,
    const char*   fieldName,
    const element_type* value,
    tibrv_u32     numElements);
tibrv_status
tibrvMsg_Addelement_typeArrayEx(
    tibrvMsg      message,
    const char*   fieldName,
    const element_type* value,
    tibrv_u32     numElements,
    tibrv_u16     fieldId);
```

Purpose

Add a field containing an array value.

Function Name	Element Type	Type Description
tibrvMsg_AddF32Array	tibrv_f32	32-bit floating point array
tibrvMsg_AddF64Array	tibrv_f64	64-bit floating point array
tibrvMsg_AddI8Array	tibrv_i8	8-bit integer array
tibrvMsg_AddI16Array	tibrv_i16	16-bit integer array
tibrvMsg_AddI32Array	tibrv_i32	32-bit integer array

Function Name	Element Type	Type Description
<code>tibrvMsg_AddI64Array</code>	<code>tibrv_i64</code>	64-bit integer array
<code>tibrvMsg_AddU8Array</code>	<code>tibrv_u8</code>	8-bit unsigned integer array
<code>tibrvMsg_AddU16Array</code>	<code>tibrv_u16</code>	16-bit unsigned integer array
<code>tibrvMsg_AddU32Array</code>	<code>tibrv_u32</code>	32-bit unsigned integer array
<code>tibrvMsg_AddU64Array</code>	<code>tibrv_u64</code>	64-bit unsigned integer array
<code>tibrvMsg_AddMsgArray</code>	<code>tibrv_Msg</code>	message array
<code>tibrvMsg_AddStringArray</code>	<code>char*</code>	string array

Parameter	Description
<code>message</code>	Add the new field to this message.
<code>fieldName</code>	Create the new field with this name.
<code>value</code>	<p>Add a new field that contains this array.</p> <p>The convenience function must correspond to the datatype of this value.</p> <p>The function <i>copies</i> the array into the new message field.</p>
<code>numElements</code>	When adding an array type, the program supplies the count of array elements in this parameter.
<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 Nested Message

Convenience Function

Declaration

```
tibrv_status tibrvMsg_AddMsg(  
    tibrvMsg      message,  
    const char*   fieldName,  
    tibrvMsg      value);  
tibrv_status tibrvMsg_AddMsgEx(  
    tibrvMsg      message,  
    const char*   fieldName,  
    tibrvMsg      value,  
    tibrv_u16     fieldId);
```

Purpose

Add a field containing a nested submessage.

Remarks

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

Parameter	Description
<i>message</i>	Add the new field to this message.
<i>fieldName</i>	Create the new field with this name.
<i>value</i>	Add a new field that contains this submessage. The function <i>copies</i> the data into the new field.
<i>fieldId</i>	Create the new field with this identifier. Zero is a special

Parameter	Description
	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 Function

Declaration

```
tibrv_status tibrvMsg_AddString(  
    tibrvMsg      message,  
    const char*   fieldName,  
    const char*   value);  
tibrv_status tibrvMsg_AddStringEx(  
    tibrvMsg      message,  
    const char*   fieldName,  
    const char*   value,  
    tibrv_u16     fieldId);
```

Purpose

Add a field containing a string.

Remarks

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

Parameter	Description
message	Add the new field to this message.
fieldName	Create the new field with this name.
value	Add a new field that contains this string (which may be a literal or stored in a variable). The function <i>copies</i> the data into the new field.

Parameter	Description
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 Function

Declaration

```
tibrv_status tibrvMsg_AddOpaque(  
    tibrvMsg      message,  
    const char*   fieldName,  
    const void*   value,  
    tibrv_u32     size);  
tibrv_status tibrvMsg_AddOpaqueEx(  
    tibrvMsg      message,  
    const char*   fieldName,  
    const void*   value,  
    tibrv_u32     size,  
    tibrv_u16     fieldId);
```

Purpose

Add a field containing an opaque byte sequence.

Parameter	Description
message	Add the new field to this message.
fieldName	Create the new field with this name.
value	Add a new field that contains this opaque buffer. The function <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

Parameter	Description
	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 Function

Declaration

```
tibrv_status tibrvMsg_AddXml(  
    tibrvMsg      message,  
    const char*   fieldName,  
    const void*   value,  
    tibrv_u32     size);  
tibrv_status tibrvMsg_AddXmlEx(  
    tibrvMsg      message,  
    const char*   fieldName,  
    const void*   value,  
    tibrv_u32     size,  
    tibrv_u16     fieldId);
```

Purpose

Add a field containing XML data.

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
message	Add the new field to this message.
fieldName	Create the new field with this name.
value	Add a new field that contains the XML data in this buffer. The function <i>copies</i> the data into the new message field.

Parameter	Description
size	When adding XML data, the program supplies the size (in bytes) of the data in this parameter.
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 DateTime

Convenience Function

Declaration

```
tibrv_status tibrvMsg_AddDateTime(  
    tibrvMsg          message,  
    const char*       fieldName,  
    const tibrvMsgDateTime* value);  
tibrv_status tibrvMsg_AddDateTimeEx(  
    tibrvMsg          message,  
    const char*       fieldName,  
    const tibrvMsgDateTime* value,  
    tibrv_u16         fieldId);
```

Purpose

Add a field containing a Rendezvous datetime value.

Parameter	Description
message	Add the new field to this message.
fieldName	Create the new field with this name.
value	Add a new field that contains this datetime value. The function <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 fragment 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>
tibrv_status
addAsTimeT(tibrvMsg msg,
           const char * field,
           time_t value)
{   tibrvMsgDateTime d;
    d.nsec = 0;
    d.sec = (tibrv_i64) value;
    return tibrvMsg_AddDateTime(msg, field, &d); }
```

See Also

[tibrvMsgDateTime](#)

`tibrvMsg_ClearReferences()`

Function

Declaration

```
tibrv_status tibrvMsg_ClearReferences(  
    tibrvMsg message);
```

Purpose

Clear references in this message.

Remarks

This function clears references back to the most recent mark.

For a description and example, see [tibrvMsg_MarkReferences\(\)](#).

Parameter	Description
message	Clear references in this message.

See Also

[Validity of Data Extracted From Messages](#)

[Deleting Snapshot References](#)

[tibrvMsg_MarkReferences\(\)](#)

tibrvMsg_ConvertToString()

Function

Declaration

```
tibrv_status tibrvMsg_ConvertToString(  
    tibrvMsg      message,  
    const char**  string);
```

Purpose

Format a message as a string.

Remarks

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

For most datatypes, this function formats the full value of the field to the output string; however, these two types are exceptions:

<code>TIBRVMSG_OPAQUE</code>	This function abbreviates the value of an opaque field; for example, [472 opaque bytes].
<code>TIBRVMSG_XML</code>	This function abbreviates the value of an XML field; for example, [XML document: 472 bytes]. The size measures <i>uncompressed</i> data.

This function formats `TIBRVMSG_IPADDR32` fields as four dot-separated decimal integers.

This function formats `TIBRVMSG_IPPORT16` fields as one decimal integer.

For printing `tibrvMsgDateTime` values, see [Converting Dates to Strings](#).

Parameter	Description
<code>message</code>	Format this message as a string.
<code>string</code>	The program supplies a location in this parameter; the function stores a pointer to the string in that location. The function allocates storage for the string itself as part of the message; the string pointer remains valid until the message is destroyed. The program must not modify the string.

tibrvMsg_Create()

Function

Declaration

```
tibrv_status tibrvMsg_Create(  
    tibrvMsg*    message);  
tibrv_status tibrvMsg_CreateEx(  
    tibrvMsg*    message,  
    tibrv_u32    initialStorage);
```

Purpose

Allocate storage and initialize it as a new message.

Remarks

This function allocates storage for the new message. A program owns the messages that it creates, and must destroy them to reclaim the storage. That is, each call to this function must be paired with a call to [tibrvMsg_Destroy\(\)](#).

Programs can add address information to the message using [tibrvMsg_SetSendSubject\(\)](#), and [tibrvMsg_SetReplySubject\(\)](#).

Message Size

When adding data to a message would overflow the allocated space, the message automatically expands by allocating additional storage. However, when creating messages that contain many small fields, you can optimize program performance by pre-allocating sufficient storage initially.

Parameter	Description
<code>message</code>	The function stores a pointer to the new message in this location.
<code>initialStorage</code>	Initial space (in bytes) to allocate for the message.

See Also

[tibrvMsg_Destroy\(\)](#)

[tibrvMsg_Expand\(\)](#)

[tibrvMsg_SetReplySubject\(\)](#)

[tibrvMsg_SetSendSubject\(\)](#)

`tibrvMsg_CreateCopy()`

Function

Declaration

```
tibrv_status tibrvMsg_CreateCopy(  
    const tibrvMsg message  
    tibrvMsg*      copy);
```

Purpose

Copy a message.

Remarks

The copy is completely independent of the original message. Pointer data in fields are independent copies of the values.

This function copies *only* the data within the fields of the message. The copy does *not* include any supplementary information—for example, address information or certified delivery information. (Programs can explicitly add supplementary information to the copy using functions such as [tibrvMsg_SetSendSubject\(\)](#).)

This function allocates the storage for the copy. The duration of the copy is independent of the original message.

A program owns the messages that it creates, and must destroy them to reclaim the storage. That is, each call to this function must be paired with a call to [tibrvMsg_Destroy\(\)](#).

Parameter	Description
<code>message</code>	Copy this message.
<code>copy</code>	The function stores a pointer to the new copy in this location.

See Also

[tibrvMsg_Destroy\(\)](#)

[tibrvMsg_SetReplySubject\(\)](#)

[tibrvMsg_SetSendSubject\(\)](#)

tibrvMsg_CreateFromBytes()

Function

Declaration

```
tibrv_status tibrvMsg_CreateFromBytes(  
    tibrvMsg*      message,  
    const void*    bytes);
```

Purpose

Create a new message, and populate it with data.

Remarks

This function allocates storage for the new message. The program owns the message, and must destroy it to reclaim the storage. That is, each call to this function must be paired with a call to [tibrvMsg_Destroy\(\)](#).

This function copies the bytes into the new message.

Programs can add address information to the message using [tibrvMsg_SetSendSubject\(\)](#), and [tibrvMsg_ConvertToString\(\)](#).

Parameter	Description
message	The program supplies a location for the new message. The function creates a new message and stores a pointer to it in that location.
bytes	Fill the new message with this data. This data buffer represents the message in Rendezvous wire format, as produced by tibrvMsg_GetAsBytes() or tibrvMsg_GetAsBytesCopy() .

See Also

[tibrvMsg_GetAsBytes\(\)](#)

[tibrvMsg_GetAsBytesCopy\(\)](#)

tibrvMsg_Destroy()

Function

Declaration

```
tibrv_status tibrvMsg_Destroy(  
    tibrvMsg message);
```

Purpose

Destroy a message; free the storage that it occupies.

Remarks

A program owns all messages that it creates or detaches, and must destroy them to reclaim the storage. That is, each call to `tibrvMsg_Create()` or `tibrvMsg_Detach()` must be paired with a call to this function.

This function frees the storage associated with the message and its snapshot data. In most operating environments it is illegal to access freed storage.

Destroying a message invalidates all pointer data (such as strings, arrays, undetached submessages, byte sequences, or XML data) previously extracted from any fields of the message.

However, destroying a parent message does not affect *detached* submessages of the parent.

It is safe for a program to exit without first destroying all messages.

Do Not Destroy Messages that Rendezvous Software Owns



Warning

Rendezvous software owns all inbound messages (unless explicitly detached). It automatically destroys its messages as the program exits the context of the callback function. It is illegal for your program to destroy a message that it does not own.

Parameter	Description
<code>message</code>	Destroy this message, freeing its storage.

See Also

[tibrvMsg_Create\(\)](#)

[tibrvMsg_Detach\(\)](#)

[tibrvEventCallback](#)

tibrvMsg_Detach()

Function

Declaration

```
tibrv_status tibrvMsg_Detach(  
    tibrvMsg message);
```

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 function; Rendezvous software destroys such messages when the callback function 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 can use this function 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.



Warning

Programs may modify messages *only* with functions 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 Messages](#).

Parameter	Description
<code>message</code>	Detach this message.

See Also

[tibrvMsg_Destroy\(\)](#)

[tibrvEventCallback](#)

[tibrvcnEventCallback](#)

tibrvMsg_Expand()

Function

Declaration

```
tibrv_status tibrvMsg_Expand(  
    tibrvMsg      message,  
    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 function 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 function returns the error code [TIBRV_NO_MEMORY](#).

Parameter	Description
<code>message</code>	Expand this message.
<code>additionalStorage</code>	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 function returns.

tibrvMsg_GetAsBytes()

Function

Declaration

```
tibrv_status tibrvMsg_GetAsBytes(  
    tibrvMsg      message,  
    const void**  bytePtr);
```

Purpose

Extract the data from a message as a byte sequence.

Remarks

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

The storage for the byte sequence is associated with the message, and persists until the message is destroyed. Programs *must not* modify the resulting byte sequence, which remains part of the message object. To make a copy that your program can modify, use [tibrvMsg_GetAsBytesCopy\(\)](#).

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
message	Extract the data bytes from this message.
bytePtr	The program supplies a location. The function stores a pointer to the byte sequence in that location.

See Also

[tibrvMsg_GetAsBytesCopy\(\)](#)

[tibrvMsg_GetByteSize\(\)](#)

[tibrvMsg_CreateFromBytes\(\)](#)

tibrvMsg_GetAsBytesCopy()

Function

Declaration

```
tibrv_status tibrvMsg_GetAsBytesCopy(  
    tibrvMsg      message,  
    void*         buf,  
    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 function 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
message	Extract the data from this message.
buf	The program supplies a buffer. The function copies the byte sequence into that buffer.

Parameter	Description
<code>byteSize</code>	The size of the buffer.

See Also

[tibrvMsg_GetAsBytes\(\)](#)

[tibrvMsg_GetByteSize\(\)](#)

[tibrvMsg_CreateFromBytes\(\)](#)

tibrvMsg_GetByteSize()

Function

Declaration

```
tibrv_status tibrvMsg_GetByteSize(  
    tibrvMsg      message,  
    tibrv_u32*    byteSize);
```

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 function 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
message	Return the size of this message.
byteSize	The program supplies a location. The function stores the message size (in bytes) in that location.

See Also

[tibrvMsg_GetAsBytes\(\)](#)

[tibrvMsg_GetAsBytesCopy\(\)](#)

tibrvMsg_GetClosure()

Function

Declaration

```
tibrv_status tibrvMsg_GetClosure(  
    tibrvMsg      message,  
    void**        closure);
```

Purpose

Extract the closure object corresponding to a (dispatched) message object.

Parameter	Description
message	Extract the closure corresponding to this message object.
closure	The program supplies a location. The function stores the closure object in that location.

Remarks

Dispatch associates the message with a listener event. This call gets the closure from that listener event.

This call is valid only for an inbound message that has already been dispatched to a listener event. Error status `TIBRV_INVALID_MSG` indicates that the message is not associated with a listener event.

See Also

[tibrvMsg_GetEvent\(\)](#)

[tibrvEventVectorCallback](#)

`tibrvMsg_GetCurrentTime()`

Function

Declaration

```
tibrv_status tibrvMsg_GetCurrentTime(  
    tibrvMsgDateTime*    dateTime);  
tibrv_status tibrvMsg_GetCurrentTimeString(  
    char*    local,  
    char*    gmt);
```

Purpose

Get the current time.

Remarks

The first function produces a date-time object; for details, see [tibrvMsgDateTime](#).

The second function produces printable strings.

Both functions are thread-safe.

Parameter	Description
<code>dateTime</code>	The program supplies a location. The function stores the current time in that location.
<code>local</code>	<p>If this argument is non-NULL, then it is the location of a string. The function stores a string representing the current time (in the local time zone) in that location.</p> <p>If this argument is NULL, then the function leaves it unchanged.</p>

Parameter	Description
<code>gmt</code>	<p>If this argument is non-NULL, then it is the location of a string. The function uses tibrvMsg_ConvertToString() to produce a string representing the current time (Zulu time), and stores the string in that location. For details, see Converting Dates to Strings.</p> <p>If this argument is NULL, then the function leaves it unchanged.</p>

See Also

[tibrvMsgDateTime](#)

tibrvMsg_GetEvent()

Function

Declaration

```
tibrv_status tibrvMsg_GetEvent(  
    tibrvMsg      message,  
    tibrvEvent*   event);
```

Purpose

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

Parameter	Description
message	Extract the event associated with this message object.
event	The program supplies a location. The function stores the event object in that location.

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. Error status `TIBRV_INVALID_MSG` indicates that the message is not associated with a listener event.

See Also

[tibrvMsg_GetClosure\(\)](#)

[tibrvEventVectorCallback](#)

tibrvMsg_GetField()

Function

Declaration

```
tibrv_status tibrvMsg_GetField(  
    tibrvMsg      message,  
    const char*   fieldName,  
    tibrvMsgField* field);  
tibrv_status tibrvMsg_GetFieldEx(  
    tibrvMsg      message,  
    const char*   fieldName,  
    tibrvMsgField* field,  
    tibrv_u16     fieldId);
```

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 function takes a snapshot of the field, and stores that information in the `field` argument, overwriting the struct supplied as an argument.

The function copies scalar data into the program's field struct. Pointer data (such as strings, arrays, submessages, opaque byte sequences, or XML 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.

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

Parameter	Description
<code>message</code>	Get the specified field of this message.
<code>fieldName</code>	Get a field with this name.
<code>field</code>	The program supplies a pointer to a <code>tibrvMsgField</code> struct; the function overwrites the contents of the struct with a snapshot of the information from the message field.
<code>fieldId</code>	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 function, and related functions 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. Extended functions begin here. Regular functions begin at step 3.
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. Regular functions begin here. Extended functions also begin here when the program supplied *zero* as the identifier (supplying zero is equivalent to not supplying a field identifier, so the behavior is equivalent to the corresponding regular function).

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

Earlier releases of Rendezvous software allowed programs to get fields from a nested submessage by concatenating field names. Starting with release 6, Rendezvous software no longer supports this special case convenience. Instead, programs must separately extract the nested submessage using `tibrvMsg_GetMsg()`, and then get the desired fields from the submessage.

Convenience Functions

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

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

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

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

These categories of type-specific convenience functions 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](#).

Extended Functions

Each convenience function has two forms.

- The usual form specifies the field to get using a field name.
- The extended form specifies the field to get using a field name and a field identifier.

For example, the function `tibrvMsg_GetI32()` is paired with the extended form `tibrvMsg_GetI32Ex()`.

The field identifier has type `tibrv_u16`. Zero is a special value that signifies no field identifier. All non-zero field identifiers must be unique within each message.

For details, see [Field Names and Field Identifiers](#).

See Also

[Field Names and Field Identifiers](#)

[Get Scalar](#)

[tibrvMsgField](#)

Get Scalar

Convenience Functions

Declaration

```
tibrv_status
tibrvMsg_Getscalar_type(
    tibrvMsg      message,
    const char*   fieldName,
    scalar_type*  value);
tibrv_status
tibrvMsg_Getscalar_typeEx(
    tibrvMsg      message,
    const char*   fieldName,
    scalar_type*  value,
    tibrv_u16     fieldId);
```

Purpose

Get the value of a field as a scalar value.

Remarks

Each convenience function 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 function, then the function attempts to convert the data (see [Datatype Conversion](#)). If conversion is not possible, the function returns `TIBRV_CONVERSION_FAILED`.

Function Name	Type	Type Description
tibrvMsg_Get Bool	tibrv_ bool	boolean
tibrvMsg_Get F32	tibrv_ f32	32-bit floating point

Function Name	Type	Type Description
<code>tibrvMsg_GetF64</code>	<code>tibrv_f64</code>	64-bit floating point
<code>tibrvMsg_GetI8</code>	<code>tibrv_i8</code>	8-bit integer
<code>tibrvMsg_GetI16</code>	<code>tibrv_i16</code>	16-bit integer
<code>tibrvMsg_GetI32</code>	<code>tibrv_i32</code>	32-bit integer
<code>tibrvMsg_GetI64</code>	<code>tibrv_i64</code>	64-bit integer
<code>tibrvMsg_GetU8</code>	<code>tibrv_u8</code>	8-bit unsigned integer
<code>tibrvMsg_GetU16</code>	<code>tibrv_u16</code>	16-bit unsigned integer
<code>tibrvMsg_GetU32</code>	<code>tibrv_u32</code>	32-bit unsigned integer
<code>tibrvMsg_GetU64</code>	<code>tibrv_u64</code>	64-bit unsigned integer
<code>tibrvMsg_GetIPAddr32</code>	<code>tibrv_ipaddr32</code>	4-byte IP address
<code>tibrvMsg_GetIPPort16</code>	<code>tibrv_ipport16</code>	2-byte IP port

Parameter	Description
<code>message</code>	Get the specified field of this message.
<code>fieldName</code>	Get a field with this name.
<code>value</code>	When extracting a scalar type, the program supplies a location in this parameter, and the function <i>copies</i> the scalar value of the field to that location.
<code>fieldId</code>	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 Messages](#)

[Field Names and Field Identifiers](#)

Get Array

Convenience Functions

Declaration

```
tibrv_status tibrvMsg_Getelement_typeArray(  
    tibrvMsg      message,  
    const char*   fieldName,  
    element_type** value,  
    tibrv_u32*    numElementsAddr)  
tibrv_status tibrvMsg_Getelement_typeArrayEx(  
    tibrvMsg      message,  
    const char*   fieldName,  
    element_type** value,  
    tibrv_u32*    numElementsAddr,  
    tibrv_u16     fieldId);
```

Purpose

Get the value of a field as an array.

Remarks

Each convenience function 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 function, then the function attempts to convert the data (see [Datatype Conversion](#)). If conversion is not possible, the function 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 functions produce values that are *read-only snapshots* of the field data (see [Pointer Snapshot](#)). Programs must not modify elements of the value array.

Function Name	Element Type	Type Description
tibrvMsg_Get F32Array	tibrv_ f32	32-bit floating point array
tibrvMsg_Get F64Array	tibrv_ i64	64-bit floating point array
tibrvMsg_Get I8Array	tibrv_ i8	8-bit integer array
tibrvMsg_Get I16Array	tibrv_ i16	16-bit integer array
tibrvMsg_Get I32Array	tibrv_ i32	32-bit integer array
tibrvMsg_Get I64Array	tibrv_ i64	64-bit integer array
tibrvMsg_Get U8Array	tibrv_ u8	8-bit unsigned integer array
tibrvMsg_Get U16Array	tibrv_ u16	16-bit unsigned integer array
tibrvMsg_Get U32Array	tibrv_ u32	32-bit unsigned integer array
tibrvMsg_Get U64Array	tibrv_ u64	64-bit unsigned integer array
tibrvMsg_Get MsgArray	tibrv_ Msg	message array
tibrvMsg_Get StringArray	char*	string array

Parameter	Description
message	Get the specified field of this message.
fieldName	Get a field with this name.
value	When extracting an array type, the program supplies a location in this parameter, and the function stores a <i>pointer</i> to the array in that location.
numElementsAddr	When extracting an array type, the program

Parameter	Description
	supplies a location in this parameter, and the function stores the number of array elements in that location.
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 Messages](#)

[Field Names and Field Identifiers](#)

Get Nested Message

Convenience Function

Declaration

```
tibrv_status tibrvMsg_GetMsg(  
    tibrvMsg      message,  
    const char*   fieldName,  
    tibrvMsg*     subMessage);  
tibrv_status tibrvMsg_GetMsgEx(  
    tibrvMsg      message,  
    const char*   fieldName,  
    tibrvMsg*     subMessage,  
    tibrv_u16     fieldId);
```

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 function 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 function 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
message	Get the specified field of this message.
fieldName	Get a field with this name.
subMessage	The program supplies a location in this parameter, and the function stores a <i>pointer</i> to the submessage in that location.
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 Messages](#)

[Field Names and Field Identifiers](#)

[tibrvMsg_Detach\(\)](#)

Get String

Convenience Function

Declaration

```
tibrv_status tibrvMsg_GetString(  
    tibrvMsg      message,  
    const char*   fieldName,  
    const char**  value);  
tibrv_status tibrvMsg_GetStringEx(  
    tibrvMsg      message,  
    const char*   fieldName,  
    const char**  value,  
    tibrv_u16     fieldId);
```

Purpose

Get the value of a field as a character string.

Remarks

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

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

<code>TIBRVMSG_OPAQUE</code>	This function abbreviates the value of an opaque field; for example, [472 opaque bytes].
<code>TIBRVMSG_XML</code>	This function abbreviates the value of an XML field; for example, [XML document: 472 bytes]. The size measures <i>uncompressed</i> 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 function produces values that are *read-only snapshots* of the field data (see [Pointer Snapshot](#)). Programs must not modify the value string.

Parameter	Description
message	Get the specified field of this message.
fieldName	Get a field with this name.
value	The program supplies a location in this parameter, and the function stores a <i>pointer</i> to the field value in that location.
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 Messages](#)

[Field Names and Field Identifiers](#)

[Datatype Conversion](#)

Get Opaque Byte Sequence

Convenience Function

Declaration

```
tibrv_status tibrvMsg_GetOpaque(  
    tibrvMsg      message,  
    const char*   fieldName,  
    const void**  value,  
    tibrv_u32*    length);  
tibrv_status tibrvMsg_GetOpaqueEx(  
    tibrvMsg      message,  
    const char*   fieldName,  
    const void**  value,  
    tibrv_u32*    length,  
    tibrv_u16     fieldId);
```

Purpose

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

Remarks

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

Parameter	Description
message	Get the specified field of this message.
fieldName	Get a field with this name.
length	The program supplies a location in this parameter, and the function stores the length of the opaque byte sequence in that location.
value	The program supplies a location in this parameter, and the function stores a <i>pointer</i> to the field value in that location.
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 Messages](#)

[Field Names and Field Identifiers](#)

Get XML Byte Sequence

Convenience Function

Declaration

```
tibrv_status tibrvMsg_GetXml(  
    tibrvMsg      message,  
    const char*   fieldName,  
    const void**  value,  
    tibrv_u32*    length);  
tibrv_status tibrvMsg_GetXmlEx(  
    tibrvMsg      message,  
    const char*   fieldName,  
    const void**  value,  
    tibrv_u32*    length,  
    tibrv_u16     fieldId);
```

Purpose

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

Remarks

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

Parameter	Description
message	Get the specified field of this message.
fieldName	Get a field with this name.
length	The program supplies a location in this parameter, and the function stores the length of the XML byte sequence in that location.
value	The program supplies a location in this parameter, and the function stores a <i>pointer</i> to the field value in that location.
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 Messages](#)

[Field Names and Field Identifiers](#)

Get DateTime

Convenience Function

Declaration

```
tibrv_status tibrvMsg_GetDateTime(  
    tibrvMsg          message,  
    const char*       fieldName,  
    tibrvMsgDateTime* value);  
tibrv_status tibrvMsg_GetDateTimeEx(  
    tibrvMsg          message,  
    const char*       fieldName,  
    tibrvMsgDateTime* value,  
    tibrv_u16         fieldId);
```

Purpose

Get the value of a field as a Rendezvous datetime value.

Remarks

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

Parameter	Description
message	Get the specified field of this message.

Parameter	Description
fieldName	Get a field with this name.
value	The program supplies a location in this parameter, and the function stores a <i>pointer</i> to the field value in that location.
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>
tibrv_status
getAsTimeT(tibrvMsg msg,
           const char * field,
           time_t * value)
{
    tibrvMsgDateTime d;
    tibrv_status err;
    err = tibrvMsg_GetDateTime(msg, field, &d);
    if( err != TIBRV_OK )
        return err;
    if( d.sec > INT_MAX || d.sec < INT_MIN )
        return TIBRV_CONVERSION_FAILED;
    *value = (time_t) d.sec;
    return TIBRV_OK;
}
```

See Also

[Validity of Data Extracted From Messages](#)

[Field Names and Field Identifiers](#)

tibrvMsg_GetFieldByIndex()

Function

Declaration

```
tibrv_status tibrvMsg_GetFieldByIndex(  
    tibrvMsg          message,  
    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 function copies scalar data into the program's field struct. 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
message	Get the field from this message.
field	The program supplies a pointer to a tibrvMsgField struct; the function overwrites the contents of the struct, using information from the message field.
fieldIndex	Get the field with this index. Zero specifies the first field.

See Also

[tibrvMsg_GetField\(\)](#)

tibrvMsg_GetFieldInstance()

Function

Declaration

```
tibrv_status tibrvMsg_GetFieldInstance(  
    tibrvMsg          message,  
    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 *i*th field named *foo*). Programs can use this function in a loop that examines every field with a specified name.

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

The function copies scalar data into the program's field struct. 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 function 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 function `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
<code>message</code>	Get the specified field instance of this message.
<code>fieldName</code>	Get an instance of the field with this name. NULL specifies the empty string as the field name.
<code>fieldAddr</code>	The program supplies a pointer to a tibrvMsgField struct; the function overwrites the contents of the struct, using information from the message field.
<code>instance</code>	Get this instance of the specified field name. The argument 1 denotes the first instance of the named field.

See Also

[tibrvMsg_GetField\(\)](#)

tibrvMsg_GetNumFields()

Function

Declaration

```
tibrv_status tibrvMsg_GetNumFields(  
    tibrvMsg      message,  
    tibrv_u32*    numFields);
```

Purpose

Extract the number of fields in a message.

Remarks

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

Parameter	Description
message	Extract the number of fields in this message.
numFields	The program supplies a location in this parameter; the function stores the number of fields in that location.

tibrvMsg_GetReplySubject()

Function

Declaration

```
tibrv_status tibrvMsg_GetReplySubject(  
    tibrvMsg      message,  
    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.

Programs must not modify the storage in which this reply subject string resides.

Parameter	Description
message	Get the subject of this message.
replySubject	<p>The function stores a pointer to the reply subject string in this location.</p> <p>The function 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 function that</p>

Parameter	Description
	determines the scope of an inbound message. For more information, see Pointer Snapshot , and Deleting Snapshot References .

See Also

[tibrvMsg_SetSendSubject\(\)](#)

Supplementary Information for Messages in TIBCO Rendezvous Concepts

tibrvMsg_GetSendSubject()

Function

Declaration

```
tibrv_status tibrvMsg_GetSendSubject(  
    tibrvMsg      message,  
    const char**  subject);
```

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.

Programs must not modify the storage in which this subject string resides.

Parameter	Description
message	Get the subject of this message.
subject	<p>The function stores a pointer to the subject string in this location.</p> <p>The function 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 function that determines the scope of an inbound message. For more</p>

Parameter	Description
	information, see Pointer Snapshot , and Deleting Snapshot References .

See Also

[tibrvMsg_SetSendSubject\(\)](#)

Supplementary Information for Messages in TIBCO Rendezvous Concepts

tibrvMsg_MarkReferences()

Function

Declaration

```
tibrv_status tibrvMsg_MarkReferences(  
    tibrvMsg message);
```

Purpose

Mark references in this message.

Parameter	Description
message	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 function 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 template message, getting and updating 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 myTimerCB(  
    tibrvEvent myTimer,  
    tibrvMsg empty,
```

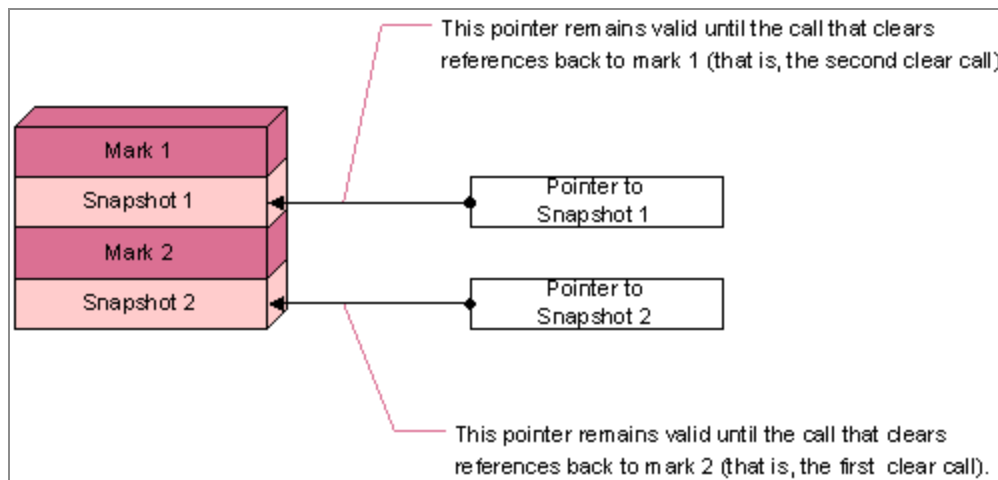
```

void* closure_msg_arg)
{
    tibrvMsg msg = (tibrvMsg)closure_msg_arg;
    tibrvMsg submsg;
    tibrvMsg_MarkReferences(msg);
    tibrvMsg_GetMsg(msg, "foo", &submsg);
    ...
    tibrvMsg_ClearReferences(msg);
    tibrvMsg_SetSendSubject(msg, some_subject);
    tibrvTransport_Send(myTransport, 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 Messages](#)

[Deleting Snapshot References](#)

[tibrvMsg_ClearReferences\(\)](#)

tibrvMsg_RemoveField()

Function

Declaration

```
tibrv_status tibrvMsg_RemoveField(  
    tibrvMsg      message,  
    const char*   fieldName);  
tibrv_status tibrvMsg_RemoveFieldEx(  
    tibrvMsg      message,  
    const char*   fieldName,  
    tibrv_u16     fieldId);
```

Purpose

Remove a field from a message.

Remarks

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

Parameter	Description
message	Remove the specified field from this message.
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 function uses this algorithm to find and remove a field within a message, as specified by a field identifier and a field name.

Procedure

1. The extended function begins here. The regular function begins at 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. (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.
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. The regular function begins here. The extended function also begins here when the program supplied *zero* as the identifier (supplying zero is equivalent to not supplying a field identifier, so the behavior is equivalent to the regular function).
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.

`tibrvMsg_RemoveFieldInstance()`

Function

Declaration

```
tibrv_status tibrvMsg_RemoveFieldInstance(  
    tibrvMsg      message,  
    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 *i*th field named *foo*). Programs can use this function 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 function returns `TIBRV_NOT_FOUND`.

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

Parameter	Description
<code>message</code>	Remove the specified field from this message.
<code>fieldName</code>	Remove the field with this name.
<code>instance</code>	Remove this instance of the field. The argument 1 specifies the first instance of the named field.

See Also

[tibrvMsg_RemoveField\(\)](#)

tibrvMsg_Reset()

Function

Declaration

```
tibrv_status tibrvMsg_Reset(  
    tibrvMsg message);
```

Purpose

Clear a message, preparing it for re-use.

Remarks

This function is the equivalent of [tibrvMsg_Destroy\(\)](#), followed immediately by [tibrvMsg_Create\(\)](#)—except that the storage is not freed, but rather re-used.

When this function 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, opaque byte sequences or XML data) previously extracted from fields of the old message are invalid.

Parameter	Description
message	Reset this message.

See Also

[tibrvMsg_Create\(\)](#)

[tibrvMsg_Destroy\(\)](#)

tibrvMsg_SetReplySubject()

Function

Declaration

```
tibrv_status tibrvMsg_SetReplySubject(  
    tibrvMsg      message,  
    const char*   replySubject);
```

Purpose

Set the reply subject for a message.

Remarks

Recipients of a message can use its reply subject as the address for return messages.

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
message	Set the reply subject of this message.
replySubject	Use this string as the new reply subject. The function copies this string to the message. The empty string is <i>not</i> a legal subject name.

Subject Name Length

The constant TIBRV_SUBJECT_MAX determines the longest possible subject name.

See Also

[tibrvMsg_GetReplySubject\(\)](#)

Supplementary Information for Messages in TIBCO Rendezvous Concepts

tibrvMsg_SetSendSubject()

Function

Declaration

```
tibrv_status tibrvMsg_SetSendSubject(  
    tibrvMsg      message,  
    const char*   subject);
```

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
message	Set the subject of this message.
subject	Use this string as the new subject. The function copies this string to the message. The empty string is <i>not</i> a legal subject name.

Subject Name Length

The constant TIBRV_SUBJECT_MAX determines the longest possible subject name.

See Also

[tibrvMsg_GetSendSubject\(\)](#)

Supplementary Information for Messages in TIBCO Rendezvous Concepts

`tibrvMsg_UpdateField()`

Function

Declaration

```
tibrv_status tibrvMsg_UpdateField(  
    tibrvMsg      message,  
    tibrvMsgField* field);
```

Purpose

Update a field within a message.

For most programs, we recommend type-specific convenience functions instead of this generic function. 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 function 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 function adds the field to the message. Update convenience functions 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 function 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, opaque byte sequences or XML 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.

Parameter	Description
<code>message</code>	Update this message.
<code>field</code>	Update the existing message field using this field.

Field Search Algorithm

This function, and related functions 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. Extended functions begin here. Regular functions begin at step 3.
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. Regular functions begin here. Extended functions also begin here when the program supplied *zero* as the identifier (supplying zero is equivalent to not supplying a field identifier, so the behavior is equivalent to the corresponding regular function).
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 Functions

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

- Code readability.
- Type checking.

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

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

Extended Functions

Each convenience function has two forms.

- The usual form specifies the field to update using a field name.
- The extended form specifies the field to update using a field name and a field identifier.

For example, the function `tibrvMsg_UpdateI32()` is paired with the extended form `tibrvMsg_UpdateI32Ex()`.

The field identifier has type `tibrv_u16`. 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.

For details, see [Field Names and Field Identifiers](#).

Update Scalar

Convenience Functions

Declaration

```
tibrv_status tibrvMsg_Updatescalar_type(
    tibrvMsg      message,
    const char*   fieldName,
    scalar_type   value);
tibrv_status tibrvMsg_Updatescalar_typeEx(
    tibrvMsg      message,
    const char*   fieldName,
    scalar_type   value,
    tibrv_u16     fieldId);
```

Purpose

Update a field containing a scalar value.

Remarks

Each convenience function 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.

Function Name	Value Type	Type Description
tibrvMsg_Update Bool	tibrv_ bool	boolean scalar
tibrvMsg_Update I8	tibrv_ i8	8-bit integer
tibrvMsg_Update U8	tibrv_ u8	8-bit unsigned integer

Function Name	Value Type	Type Description
<code>tibrvMsg_UpdateI16</code>	<code>tibrv_i16</code>	16-bit integer
<code>tibrvMsg_UpdateU16</code>	<code>tibrv_u16</code>	16-bit unsigned integer
<code>tibrvMsg_UpdateI32</code>	<code>tibrv_i32</code>	32-bit integer
<code>tibrvMsg_UpdateU32</code>	<code>tibrv_u32</code>	32-bit unsigned integer
<code>tibrvMsg_UpdateI64</code>	<code>tibrv_i64</code>	64-bit integer
<code>tibrvMsg_UpdateU64</code>	<code>tibrv_u64</code>	64-bit unsigned integer
<code>tibrvMsg_UpdateF32</code>	<code>tibrv_f32</code>	32-bit floating point
<code>tibrvMsg_UpdateF64</code>	<code>tibrv_f64</code>	64-bit floating point
<code>tibrvMsg_UpdateIPAddr32</code>	<code>tibrv_ipaddr32</code>	4-byte IP address
<code>tibrvMsg_UpdateIPPort16</code>	<code>tibrv_ipport16</code>	2-byte IP port

Parameter	Description
<code>message</code>	Update the specified field of this message.
<code>fieldName</code>	Update a field with this name.
<code>value</code>	<p>Update the message field to this value (which may be a literal or stored in a variable).</p> <p>The function copies the value into the new message field.</p>
<code>fieldId</code>	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 Functions

Declaration

```
tibrv_status tibrvMsg_Updateelement_typeArray(
    tibrvMsg      message,
    const char*    fieldName,
    const element_type* value,
    tibrv_u32      numElements);
tibrv_status tibrvMsg_Updateelement_typeArrayEx(
    tibrvMsg      message,
    const char*    fieldName,
    const element_type* value,
    tibrv_u32      numElements,
    tibrv_u16      fieldId);
```

Purpose

Update a field containing an array value.

Remarks

Each convenience function 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](#).)

Function Name	Element Type	Type Description
tibrvMsg_Update I8Array	tibrv_ i8	8-bit integer array
tibrvMsg_Update U8Array	tibrv_ u8	8-bit unsigned integer array
tibrvMsg_Update I16Array	tibrv_ i16	16-bit integer array
tibrvMsg_Update U16Array	tibrv_ u16	16-bit unsigned integer array
tibrvMsg_Update I32Array	tibrv_ i32	32-bit integer array
tibrvMsg_Update U32Array	tibrv_ u32	32-bit unsigned integer array
tibrvMsg_Update I64Array	tibrv_ i64	64-bit integer array
tibrvMsg_Update U64Array	tibrv_ u64	64-bit unsigned integer array
tibrvMsg_Update F32Array	tibrv_ f32	32-bit floating point array
tibrvMsg_Update F64Array	tibrv_ f64	64-bit floating point array
tibrvMsg_Update MsgArray	tibrv_ Msg	message array
tibrvMsg_Update StringArray	char*	string array

Parameter	Description
message	Update the specified field of this message.
fieldName	Update a field with this name.
value	Update the message field to this array value. The function <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.

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 Nested Message

Convenience Function

Declaration

```
tibrv_status tibrvMsg_UpdateMsg(  
    tibrvMsg      message,  
    const char*   fieldName,  
    tibrvMsg      value);  
tibrv_status tibrvMsg_UpdateMsgEx(  
    tibrvMsg      message,  
    const char*   fieldName,  
    tibrvMsg      value,  
    tibrv_u16     fieldId);
```

Purpose

Update a field containing a nested submessage.

Remarks

This convenience function 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.

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

Parameter	Description
message	Update the specified field of this message.
fieldName	Update a field with this name.

Parameter	Description
value	Update the message field to this value. The function <i>copies</i> the new value into the 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 String

Convenience Function

Declaration

```
tibrv_status tibrvMsg_UpdateString(  
    tibrvMsg      message,  
    const char*   fieldName,  
    const char*   value);  
tibrv_status tibrvMsg_UpdateStringEx(  
    tibrvMsg      message,  
    const char*   fieldName,  
    const char*   value,  
    tibrv_u16     fieldId);
```

Purpose

Update a field containing a character string.

Remarks

This convenience function 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
message	Update the specified field of this message.

Parameter	Description
fieldName	Update a field with this name.
value	<p>Update the message field to this value (which may be a literal or stored in a variable).</p> <p>The function <i>copies</i> the new value into the existing field.</p>
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 Function

Declaration

```
tibrv_status tibrvMsg_UpdateOpaque(  
    tibrvMsg      message,  
    const char*   fieldName,  
    const void*   value,  
    tibrv_u32     size);  
tibrv_status tibrvMsg_UpdateOpaqueEx(  
    tibrvMsg      message,  
    const char*   fieldName,  
    const void*   value,  
    tibrv_u32     size,  
    tibrv_u16     fieldId);
```

Purpose

Update a field containing an opaque byte sequence.

Remarks

This convenience function 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
message	Update the specified field of this message.

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

Convenience Function

Declaration

```
tibrv_status tibrvMsg_UpdateXml(  
    tibrvMsg      message,  
    const char*   fieldName,  
    const void*   value,  
    tibrv_u32     size);  
tibrv_status tibrvMsg_UpdateXmlEx(  
    tibrvMsg      message,  
    const char*   fieldName,  
    const void*   value,  
    tibrv_u32     size,  
    tibrv_u16     fieldId);
```

Purpose

Update a field containing an XML byte sequence.

Remarks

This convenience function 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
message	Update the specified field of this message.
fieldName	Update a field with this name.
value	Update the message field to this value. The function <i>copies</i> the new value into the existing field.
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 Function

Declaration

```
tibrv_status tibrvMsg_UpdateDateTime(  
    tibrvMsg          message,  
    const char*       fieldName,  
    const tibrvMsgDateTime* value);  
tibrv_status tibrvMsg_UpdateDateTimeEx(  
    tibrvMsg          message,  
    const char*       fieldName,  
    const tibrvMsgDateTime* value,  
    tibrv_u16         fieldId);
```

Purpose

Update a field containing a datetime value.

Remarks

This convenience function 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.

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
message	Update the specified field of this message.

Parameter	Description
fieldName	Update a field with this name.
value	Update the message field to this value. The function <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](#)

Events

Programs can express interest in events of three kinds—inbound messages, timers, and I/O conditions. When an event occurs, it triggers a program callback function to process the event.

This section presents functions and types associated with event interest and event processing.

Operations by Functional Group

Function or Type	Description
Event Interest: Create and Destroy	
tibrvEvent_CreateListener()	Listen for inbound messages.
tibrvEvent_CreateVectorListener()	Listen for inbound messages, and receive them in a vector.
tibrvEvent_CreateTimer()	Start a timer.
tibrvEvent_CreateIO()	Wait for specified I/O situations to occur.
tibrvEvent_Destroy()	Destroy an event, canceling interest.
tibrvEvent_DestroyEx()	Destroy an event, and run a completion function when all of the destroyed event's callback functions are complete.
Event Accessors	
tibrvEvent_GetIOSource()	Extract the source (socket ID) from an I/O event object.
tibrvEvent_GetIOType()	Extract the I/O type from an I/O event object.
tibrvEvent_GetListenerSubject()	Extract the subject from a listener event object.
tibrvEvent_GetListenerTransport()	Extract the transport from a listener event object.

Function or Type	Description
tibrvEvent_GetTimerInterval()	Extract the interval from a timer event object.
tibrvEvent_GetType()	Extract the type of an event object.
tibrvEvent_GetQueue()	Extract the queue of an event object.
tibrvEvent_ResetTimerInterval()	Reset the interval of a timer event object.
Types Related to Events	
tibrvEvent	Each call to a Rendezvous event creation function results in a new event object, which represents your program's interest in a class of events. Rendezvous software uses the same event object to signal each occurrence of such an event.
tibrvEventCallback	Programs define functions of this type to process events.
tibrvEventVectorCallback	Programs define functions of this type to process message vector events.
tibrvEventOnComplete	A program can destroy an event object even when its callback function is running in one or more threads. Multi-threaded programs can define functions of this type to discover when all callback functions in progress have completed.
tibrvEventType	Distinguish event objects as listeners, timers, or I/O interest.
tibrvIOType	Distinguish event interest in I/O conditions.

Operations in Alphabetical Order

Function or Type	Description
tibrvEvent	Each call to a Rendezvous event creation function results in a new event object, which represents your program's interest in a class of events. Rendezvous software uses the same event object to signal each occurrence of such an event.
tibrvEventCallback	Programs define functions of this type to process events.
tibrvEventOnComplete	A program can destroy an event object even when its callback function is running in one or more threads. Multi-threaded programs can define functions of this type to discover when all callback functions in progress have completed.
tibrvEventVectorCallback	Programs define functions of this type to process message vector events.
tibrvEvent_CreateIO()	Wait for specified I/O situations to occur.
tibrvEvent_CreateListener()	Listen for inbound messages.
tibrvEvent_CreateTimer()	Start a timer.
tibrvEvent_CreateVectorListener()	Listen for inbound messages, and receive them in a vector.
tibrvEvent_Destroy()	Destroy an event, canceling interest.
tibrvEvent_DestroyEx()	Destroy an event, and run a completion

Function or Type	Description
	function when all of the destroyed event's callback functions are complete.
<code>tibrvEvent_GetIOSource()</code>	Extract the source (socket ID) from an I/O event object.
<code>tibrvEvent_GetIOType()</code>	Extract the I/O type from an I/O event object.
<code>tibrvEvent_GetListenerSubject()</code>	Extract the subject from a listener event object.
<code>tibrvEvent_GetListenerTransport()</code>	Extract the transport from a listener event object.
<code>tibrvEvent_GetTimerInterval()</code>	Extract the interval from a timer event object.
<code>tibrvEvent_GetType()</code>	Extract the type of an event object.
<code>tibrvEvent_GetQueue()</code>	Extract the queue of an event object.
<code>tibrvEventType</code>	Distinguish event objects as listeners, timers, or I/O interest.
<code>tibrvIOType</code>	Distinguish event interest in I/O conditions.

tibrvEvent

Type

Declaration

```
typedef tibrvId tibrvEvent;
```

Purpose

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

Remarks

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

Programs use the event object as a token; they cannot view or modify the object, except using accessor functions.

Programs must explicitly destroy each event object. Destroying an event object cancels the program's interest in that event, and frees its storage.

See Also

[tibrvEvent_Destroy\(\)](#)

Event Creation Functions

[tibrvEvent_CreateIO\(\)](#)

[tibrvEvent_CreateListener\(\)](#)

[tibrvEvent_CreateVectorListener\(\)](#)

[tibrvEvent_CreateTimer\(\)](#)

Event Accessor Functions

[tibrvEvent_GetIOSource\(\)](#)

[tibrvEvent_GetIOType\(\)](#)

[tibrvEvent_GetListenerSubject\(\)](#)

[tibrvEvent_GetListenerTransport\(\)](#)

[tibrvEvent_GetTimerInterval\(\)](#)

[tibrvEvent_GetType\(\)](#)

[tibrvEvent_GetQueue\(\)](#)

tibrvEventCallback

Function Type

Declaration

```
typedef void (*tibrvEventCallback) (  
    tibrvEvent    event,  
    tibrvMsg      message,  
    void*         closure);
```

Purpose

Programs define functions of this type to process events.

Remarks

A single callback function type spans listener, timer and I/O events.

Parameter	Description
event	This parameter receives the event object. This object is identical to the object that the program created to express event interest.
message	<p>When the event object is a listener, the callback receives the inbound message in this parameter. If the inbound message is empty, this parameter receives a message that has no fields.</p> <p>When the event object is a timer or I/O event, this parameter receives NULL.</p>
closure	This parameter receives the closure data, which the program supplied in the call that created the event object.

See Also

[tibrvEvent_CreateIO\(\)](#)

[tibrvEvent_CreateListener\(\)](#)

[tibrvEvent_CreateTimer\(\)](#)

tibrvEventOnComplete

Function Type

Declaration

```
typedef void (*tibrvEventOnComplete) (  
    tibrvEvent    destroyedEvent,  
    void*         closure);
```

Purpose

A program can destroy an event object even when its callback function is running in one or more threads. Multi-threaded programs can define functions of this type to discover when all callback functions 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 function runs, the event is already destroyed; this function cannot use the event object in Rendezvous calls.</p>
closure	<p>This parameter receives the closure data, which the program supplied in the call that created the event object.</p>

Remarks

This type of function is important in two situations:

- An event callback function calls `tibrvEvent_DestroyEx()` to destroy its event, and the program must do additional processing *after* the rest of the callback function has completed.
- Several threads dispatch an event (so the event callback function can be running in several threads) and the program must do additional processing after the callback function has completed *in all threads*.

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

`tibrvEvent_DestroyEx()` accepts a *completion function* argument of type `tibrvEventOnComplete`. Rendezvous software ensures that the completion function runs when the last callback-in-progress has completed.

Timing and Context

This completion function can run in two situations:

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

Notice that in this situation, the completion function runs in the program context, instead of the usual context of a callback function. 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 function code.

To protect against this type of deadlock, programmers can use a straightforward thought-experiment as a preventive test. Expand the completion function code immediately after the call to `tibrvEvent_DestroyEx()`—as it would run when the destroyed event's callback function 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
...
tibrvEvent_DestroyEx()
expand completion function 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 Functions are in Progress

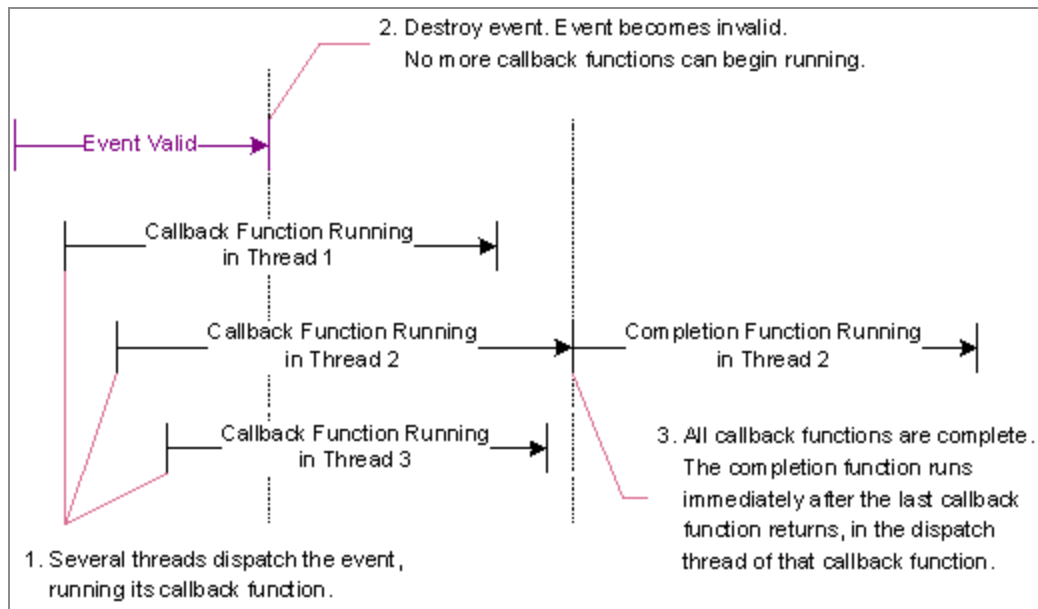
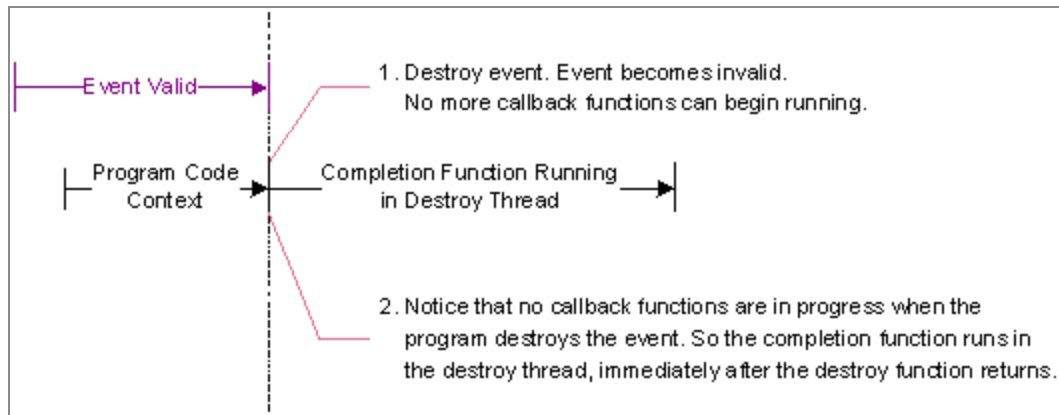


Figure 7: Completion when Callback Functions are Not in Progress



See Also

[tibrvEvent_CreateIO\(\)](#)

[tibrvEvent_CreateListener\(\)](#)

[tibrvEvent_CreateTimer\(\)](#)

[tibrvEvent_DestroyEx\(\)](#)

tibrvEventVectorCallback

Function Type

Declaration

```
typedef void (*tibrvEventVectorCallback) (  
    tibrvMsg      messages[],  
    tibrv_u32     numMessages);
```

Purpose

Programs define functions of this type to process message vector events.

Remarks

In the simplest arrangement, your callback function processes the messages in the array. When the callback function 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 [tibrvEventCallback](#), this vector callback does not receive the listener event and the closure object as arguments. You can use [tibrvMsg_GetEvent\(\)](#) and [tibrvMsg_GetClosure\(\)](#) to get them from the individual message objects.

Parameter	Description
<code>messages</code>	The callback receives an array of inbound messages in this parameter.

Parameter	Description
<code>numMessages</code>	This parameter receives the number of messages in the array.

See Also

[tibrvMsg_Detach\(\)](#)

[tibrvMsg_GetClosure\(\)](#)

[tibrvMsg_GetEvent\(\)](#)

[tibrvEvent_CreateVectorListener\(\)](#)

tibrvEvent_CreateIO()

Function

Declaration

```
tibrv_status tibrvEvent_CreateIO(  
    tibrvEvent*      event,  
    tibrvQueue       queue,  
    tibrvEventCallback callback,  
    tibrv_i32        socketId,  
    tibrvIOType      ioType,  
    const void*      closure);
```

Purpose

Wait for specified I/O situations to occur.

Parameter	Description
event	<p>For each I/O occurrence, place this event object on the event queue.</p> <p>The program supplies a location, and the function stores the new event object in that location.</p> <p>The event object remains valid until the program explicitly destroys it.</p>
queue	<p>For each I/O occurrence, place the event on this event queue.</p>
callback	<p>On dispatch, process the event with this callback function.</p>
socketID	<p>Wait for I/O occurrences on this socket.</p>

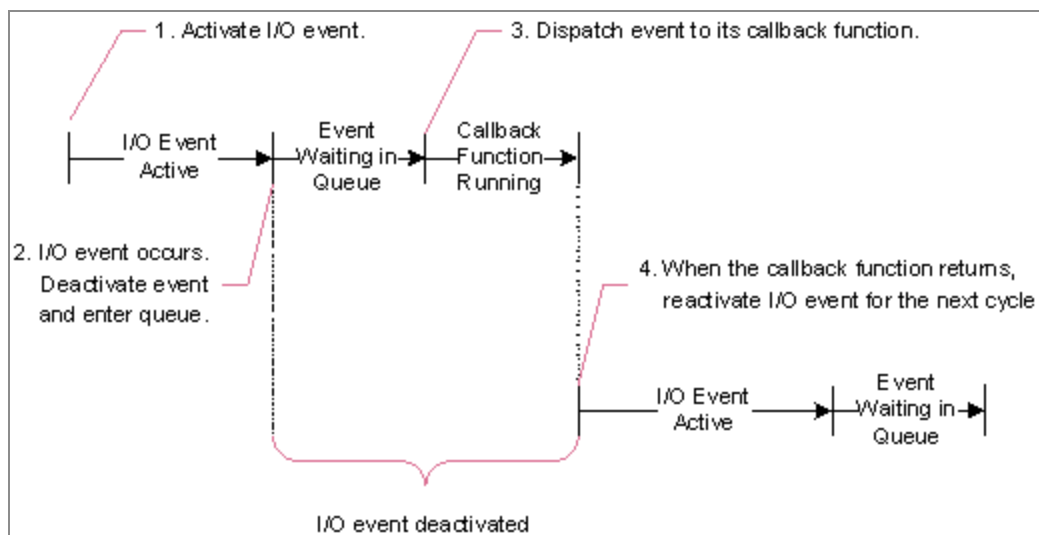
Parameter	Description
<code>ioType</code>	Wait for I/O occurrences of this type. See tibrvIOType .
<code>closure</code>	Store this closure data in the event object.

Activation and Dispatch

This function 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 function to process the event. When the callback function 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 function returns. Consequently, an I/O object can cause at most one event at a time.

Figure 8: 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.

See Also

[tibrvIOType](#)

[tibrvQueue](#)

tibrvEvent_CreateListener()

Function

Declaration

```
tibrv_status tibrvEvent_CreateListener(  
    tibrvEvent*      event,  
    tibrvQueue       queue,  
    tibrvEventCallback callback,  
    tibrvTransport   transport,  
    const char*      subject,  
    const void*      closure);
```

Purpose

Listen for inbound messages.

Parameter	Description
event	<p>For each inbound message, place this event on the event queue.</p> <p>The program supplies a location, and the function stores the new event in that location.</p> <p>The event object remains valid until the program explicitly destroys it.</p>
queue	<p>For each inbound message, place the event on this event queue.</p>
callback	<p>On dispatch, process the event with this callback function.</p>
transport	<p>Listen for inbound messages on this transport.</p>

Parameter	Description
<code>subject</code>	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.
<code>closure</code>	Store this closure data in the event object.

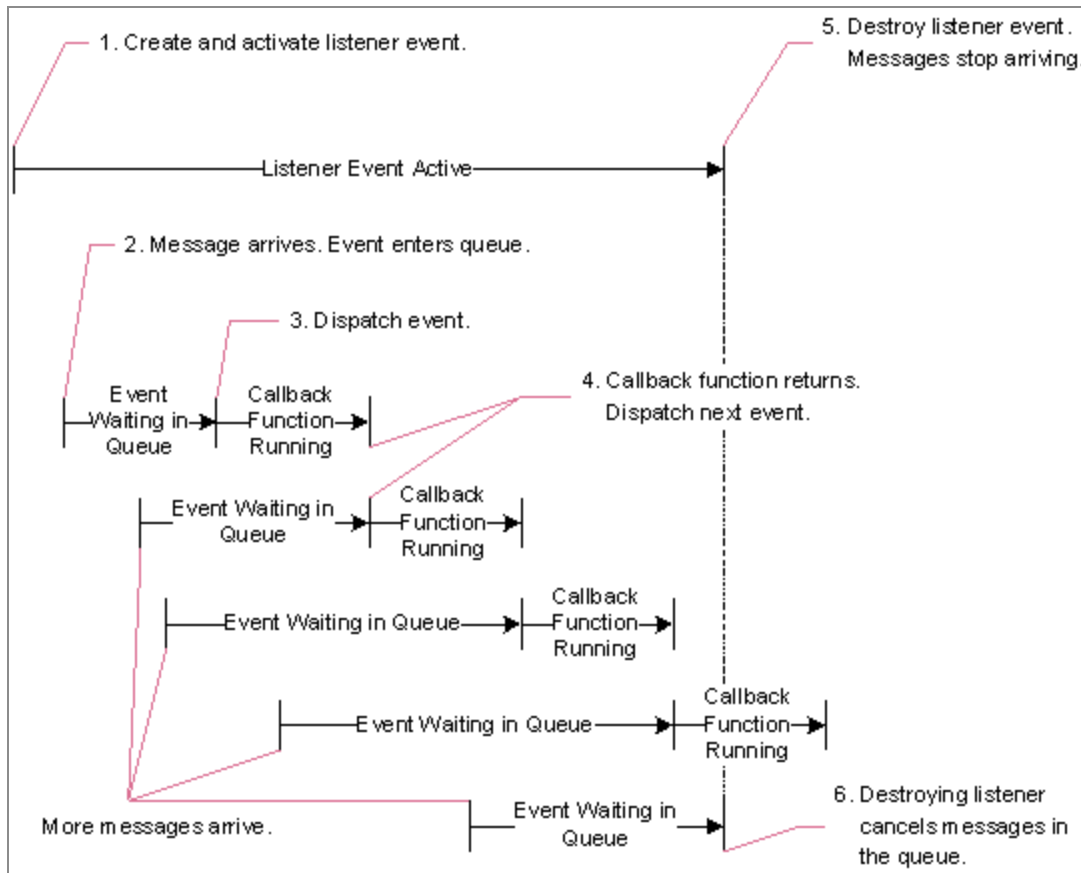
Activation and Dispatch

Inbound messages on the transport that match the subject trigger the event.

This function creates a 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 function 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 also [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 function is processing.

Figure 9: Listener Activation and Dispatch



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

Listening for Advisory Messages

Use this function 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 `tibrvEvent_CreateListener()` 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

[tibrvEvent_GetListenerSubject\(\)](#)

[tibrvTransport_CreateInbox\(\)](#)

tibrvEvent_CreateTimer()

Function

Declaration

```
tibrv_status tibrvEvent_CreateTimer(  
    tibrvEvent*      event,  
    tibrvQueue       queue,  
    tibrvEventCallback callback,  
    tibrv_f64        interval,  
    const void*      closure);
```

Purpose

Start a timer.

Parameter	Description
event	<p>At each time interval, place this event on the event queue.</p> <p>The program supplies a location, and the function stores the new event object in that location.</p> <p>The event object remains valid until the program explicitly destroys it.</p>
queue	<p>At each time interval, place the event on this event queue.</p>
callback	<p>On dispatch, process the event with this callback function.</p>
interval	<p>The timer triggers its callback function at this repeating interval (in seconds).</p>

Parameter	Description
<code>closure</code>	Store this closure data in the event object.

Remarks

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

This function creates a 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 function to process the timer event. 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.

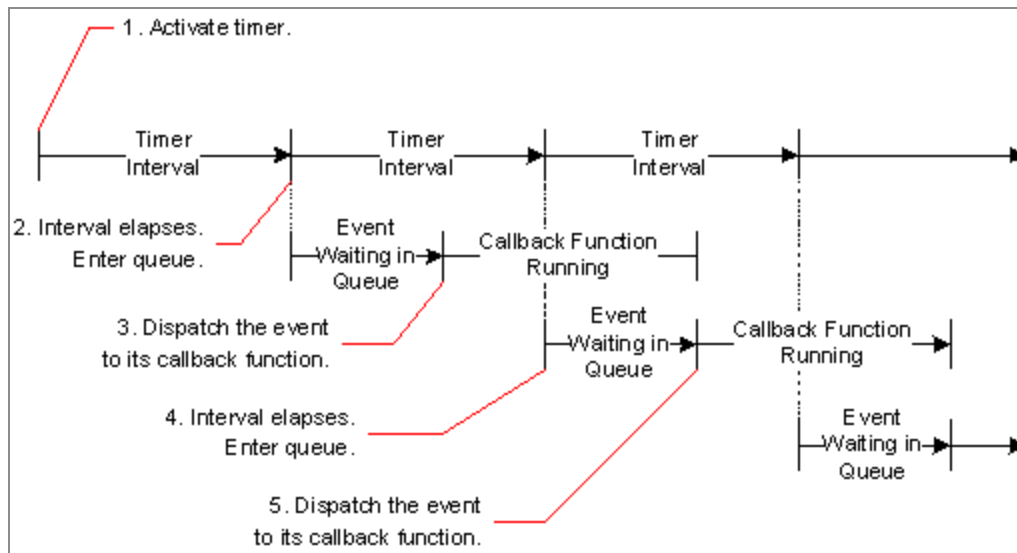
[Timer Activation and Dispatch](#) illustrates a sequence of timer intervals. The number of elapsed timer intervals directly determines the number of event callbacks.

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 functions 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 10: 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*.

See Also

[tibrvEvent_Destroy\(\)](#)

[tibrvEvent_ResetTimerInterval\(\)](#)

Timer Event Semantics in TIBCO Rendezvous Concepts

tibrvEvent_CreateVectorListener()

Function

Declaration

```
tibrv_status tibrvEvent_CreateVectorListener(  
    tibrvEvent*      event,  
    tibrvQueue       queue,  
    tibrvEventVectorCallback callback,  
    tibrvTransport   transport,  
    const char*      subject,  
    const void*      closure);
```

Purpose

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

Parameter	Description
event	<p>For each vector of inbound messages, place this event on the event queue.</p> <p>The program supplies a location, and the function stores the new event in that location.</p> <p>The event object remains valid until the program explicitly destroys it.</p>
queue	<p>For each vector of inbound messages, place the event on this event queue.</p>
callback	<p>On dispatch, process the event with this callback function.</p>
transport	<p>Listen for inbound messages on this transport.</p>

Parameter	Description
<code>subject</code>	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.
<code>closure</code>	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 function 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 function 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; see also [tibrvEvent_Destroy\(\)](#).

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 function (though they need not match the same listener).

From these properties we can derive further inferences:

- If two vector listeners use the same callback function, 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 functions, then the dispatcher cannot group them into the same vector.

Vector Listeners: Same Callback

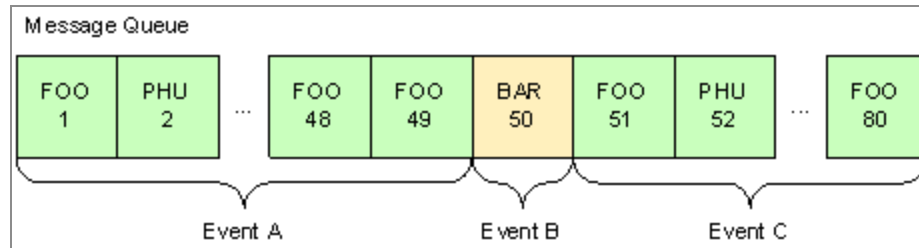
Two vector listeners, F and P, listen on subjects F00 and PHU, respectively. Both F and P designate the same queue, Q1, and the same callback function, C1, to process their messages. In this situation, the dispatcher for Q1 can group messages on subjects F00 and PHU into the same vector (as long as the 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 function, C2 to process its messages. In this situation, the dispatcher for Q1 must group messages on subject BAR separately from messages on subjects F00 and PHU.

Suppose the Q1 contains 49 messages with subjects F00 or PHU, then 1 message with subject BAR, then 30 more messages with subjects F00 and s. [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 F00 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 F00 again, it closes the BAR vector after only one message, and starts a third vector for F00.

Figure 11: 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 function 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

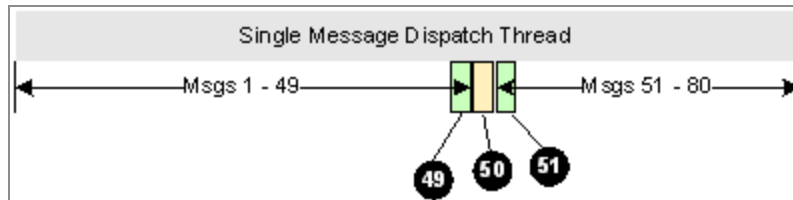
The simplest callback (from the programmer's perspective) processes the messages within a vector in order (that is, the order that dispatcher moves them from the queue into the vector, which mirrors the order in which the messages arrive in the queue). Nonetheless you could program a callback that processes 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 12: Vector Listener Callbacks in a Single Dispatch Thread

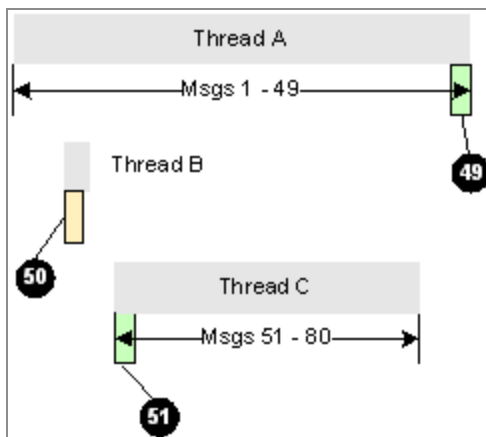


Vector Listeners: Processing Message Vectors in Separate Threads

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 13: 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 F00 callback (in thread A) processes message 49. Furthermore, it is even possible for the F00 callback (in thread C) to process message 51 before the F00 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

[tibrvEventVectorCallback](#)

tibrvEvent_Destroy()

Function

Declaration

```
tibrv_status tibrvEvent_Destroy(  
    tibrvEvent    event);
```

Purpose

Destroy an event, canceling interest.

Remarks

Destroying an event object cancels interest in the specified event. Upon return from [tibrvEvent_Destroy\(\)](#), the destroyed event is no longer dispatched.

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

Destroying event interest invalidates the event object; passing the event object as an argument to subsequent API calls produces an error.

Parameter	Description
event	Cancel interest in this event.

See Also

[tibrvEventOnComplete](#)

[tibrvEvent_CreateIO\(\)](#)

[tibrvEvent_CreateListener\(\)](#)

[tibrvEvent_CreateTimer\(\)](#)

`tibrvEvent_DestroyEx()`

tibrvEvent_DestroyEx()

Function

Declaration

```
tibrv_status tibrvEvent_DestroyEx(  
    tibrvEvent      event,  
    tibrvEventOnComplete completionFunction);
```

Purpose

Destroy an event, and run a completion function when all of the destroyed event's callback functions are complete.

Remarks

Destroying an event object cancels interest in the specified event. Upon return from [tibrvEvent_DestroyEx\(\)](#), the destroyed event's callback function is no longer dispatched.

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

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

Destroying event interest invalidates the event object; passing the event object as an argument to subsequent API calls produces an error.

Parameter	Description
event	Cancel interest in this event.

Parameter	Description
<code>completionFunction</code>	<p>Rendezvous software runs this function immediately after all instances of the event's callback function have completed. If the event's callback function is not running when the event is destroyed, the destroy call runs it before returning.</p> <p>If this parameter is <code>NULL</code>, tibrvEvent_DestroyEx() does not run a completion function; instead, its behavior is the same as tibrvEvent_Destroy().</p>

See Also

[tibrvEventOnComplete](#)

[tibrvEvent_CreateIO\(\)](#)

[tibrvEvent_CreateListener\(\)](#)

[tibrvEvent_Destroy\(\)](#)

`tibrvEvent_GetIOSource()`

Function

Declaration

```
tibrv_status tibrvEvent_GetIOSource(  
    tibrvEvent    event,  
    tibrv_i32*    source);
```

Purpose

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

Parameter	Description
event	Extract the source from this I/O event object.
source	The program supplies a location. The function stores the source of the I/O event object in that location.

See Also

[tibrvEvent_CreateIO\(\)](#)

`tibrvEvent_GetIOType()`

Function

Declaration

```
tibrv_status tibrvEvent_GetIOType(  
    tibrvEvent      event,  
    tibrvIOType*    ioType);
```

Purpose

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

Parameter	Description
<code>event</code>	Extract the I/O type from this I/O event object.
<code>ioType</code>	The program supplies a location. The function stores the I/O type in that location.

See Also

[`tibrvEvent_CreateIO\(\)`](#)

[`tibrvIOType`](#)

`tibrvEvent_GetListenerSubject()`

Function

Declaration

```
tibrv_status tibrvEvent_GetListenerSubject(  
    tibrvEvent      event,  
    const char**    subject);
```

Purpose

Extract the subject from a listener event object.

Parameter	Description
event	Extract the subject from this listener.
subject	The program supplies a location. The function stores the subject of the listener event object in that location.

See Also

[`tibrvEvent_CreateListener\(\)`](#)

`tibrvEvent_GetListenerTransport()`

Function

Declaration

```
tibrv_status tibrvEvent_GetListenerTransport(  
    tibrvEvent      event,  
    tibrvTransport* transport);
```

Purpose

Extract the transport from a listener event object.

Parameter	Description
<code>event</code>	Extract the transport from this listener.
<code>transport</code>	The program supplies a location. The function stores the transport of the listener event object in that location.

See Also

[`tibrvEvent_CreateListener\(\)`](#)

`tibrvEvent_GetTimerInterval()`

Function

Declaration

```
tibrv_status tibrvEvent_GetTimerInterval(  
    tibrvEvent    event,  
    tibrv_f64*    interval);
```

Purpose

Extract the interval from a timer event object.

Parameter	Description
<code>event</code>	Extract the interval from this timer.
<code>interval</code>	The program supplies a location. The function stores the interval of the timer event object in that location.

See Also

[`tibrvEvent_CreateTimer\(\)`](#)

[`tibrvEvent_ResetTimerInterval\(\)`](#)

tibrvEvent_GetType()

Function

Declaration

```
tibrv_status tibrvEvent_GetType(  
    tibrvEvent      event,  
    tibrvEventType* type);
```

Purpose

Extract the type of an event object.

Parameter	Description
event	Extract the event type from this event object.
type	The program supplies a location. The function stores the event object's type in that location.

See Also

[tibrvEvent](#)

[tibrvEventType](#)

`tibrvEvent_GetQueue()`

Function

Declaration

```
tibrv_status tibrvEvent_GetQueue(  
    tibrvEvent    event,  
    tibrvQueue*   queue);
```

Purpose

Extract the queue of an event object.

Parameter	Description
event	Extract the event queue from this event object.
queue	The program supplies a location. The function stores the event object's queue in that location.

See Also

[tibrvEvent](#)

[tibrvQueue](#)

tibrvEvent_ResetTimerInterval()

Function

Declaration

```
tibrv_status tibrvEvent_ResetTimerInterval(  
    tibrvEvent    event,  
    tibrv_f64     newInterval);
```

Purpose

Reset the interval of a timer event object.

Remarks

The timer begins counting the new interval immediately.

Parameter	Description
event	Reset the interval of this timer.
newInterval	The timer triggers its callback function 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 function can affect a timer only before or during its interval—but not after its interval has elapsed.

This function 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.)

See Also

[tibrvEvent_CreateTimer\(\)](#)

[tibrvEvent_GetTimerInterval\(\)](#)

tibrvEventType

Type

Declaration

```
typedef tibrv_u32    tibrvEventType;
```

Purpose

Distinguish event objects as listeners, timers, or I/O interest.

Value	Description
TIBRV_LISTEN_EVENT	An event object with this type listens for inbound messages.
TIBRV_TIMER_EVENT	An event object with this type represents is a timer.
TIBRV_IO_EVENT	An event object with this type expresses interest in an I/O condition.

See Also

[tibrvEvent_GetType\(\)](#)

tibrvIOType

Type

Declaration

```
typedef enum {  
    TIBRV_IO_READ,  
    TIBRV_IO_WRITE,  
    TIBRV_IO_EXCEPTION } tibrvIOType;
```

Purpose

Distinguish event interest in I/O conditions.

Value	Description
TIBRV_IO_READ	The socket is now readable.
TIBRV_IO_WRITE	The socket is now write-available.
TIBRV_IO_EXCEPTION	An exceptional condition occurred on the socket. (For example, out-of-band data has arrived.)

See Also

[tibrvEvent_CreateIO\(\)](#)

[tibrvEvent_GetIOType\(\)](#)

Event Queues

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

This section presents functions and types associated with event queues.

Operations by Functional Group

Function	Description
Queue Life Cycle	
tibrvQueue	Specify an event queue.
tibrvQueue_Create()	Create an event queue.
tibrvQueue_Destroy()	Destroy an event queue.
tibrvQueueOnComplete	A program can destroy a queue object even when callback functions from its events are running in one or more threads. Multi-threaded programs can define functions of this type to discover when all event callback functions in progress have completed.
Dispatch	
tibrvQueue_Dispatch()	Dispatch an event; if no event is ready, block.
tibrvQueue_Poll()	Dispatch an event, if possible.
tibrvQueue_TimedDispatch()	Dispatch an event, but if no event is ready to dispatch, limit the time that this call blocks while waiting for an event.
Properties	
tibrvQueueLimitPolicy	Specify a strategy for resolving overflow of queue limit.
tibrvQueue_GetCount()	Extract the number of events in a queue.

Function	Description
tibrvQueue_GetLimitPolicy()	Extract the limit properties of a queue.
tibrvQueue_GetName()	Extract the name of a queue.
tibrvQueue_GetPriority()	Extract the priority of a queue.
tibrvQueue_SetLimitPolicy()	Set the limit properties of a queue.
tibrvQueue_SetName()	Set the name of a queue.
tibrvQueue_SetPriority()	Set the priority of a queue.
External Event Manager Hook	
tibrvQueueHook	Asynchronously notify an external event manager when a Rendezvous event is ready for dispatch.
tibrvQueue_GetHook()	Extract an event queue hook function.
tibrvQueue_RemoveHook()	Remove the event queue hook function from a queue.
tibrvQueue_SetHook()	Register an event queue hook function.

Operations in Alphabetical Order

Function or Type	Description
tibrvQueue	Specify an event queue.
tibrvQueueHook	Asynchronously notify an external event manager when a Rendezvous event is ready for dispatch.
tibrvQueueLimitPolicy	Specify a strategy for resolving overflow of queue limit.
tibrvQueueOnComplete	A program can destroy a queue object even when callback functions from its events are running in one or more threads. Multi-threaded programs can define functions of this type to discover when all event callback functions in progress have completed.
tibrvQueue_Create()	Create an event queue.
tibrvQueue_Destroy()	Destroy an event queue.
tibrvQueue_Dispatch()	Dispatch an event; if no event is ready, block.
tibrvQueue_GetCount()	Extract the number of events in a queue.
tibrvQueue_GetHook()	Extract an event queue hook function.
tibrvQueue_GetLimitPolicy()	Extract the limit properties of a queue.
tibrvQueue_GetName()	Extract the name of a queue.
tibrvQueue_GetPriority()	Extract the priority of a queue.

Function or Type	Description
tibrvQueue_Poll()	Dispatch an event, if possible.
tibrvQueue_RemoveHook()	Remove the event queue hook function from a queue.
tibrvQueue_SetHook()	Register an event queue hook function.
tibrvQueue_SetLimitPolicy()	Set the limit properties of a queue.
tibrvQueue_SetName()	Set the name of a queue.
tibrvQueue_SetPriority()	Set the priority of a queue.
tibrvQueue_TimedDispatch()	Dispatch an event, but if no event is ready to dispatch, limit the time that this call blocks while waiting for an event.

tibrvQueue

Type

Declaration

```
typedef tibrvId tibrvQueue;  
#define TIBRV_DEFAULT_QUEUE
```

Purpose

Specify an event queue.

Default Queue

The constant TIBRV_DEFAULT_QUEUE represents 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 cannot destroy the default queue, except as a side effect of [tibrv_Close\(\)](#). Programs cannot change the parameters of the default queue.

See Also

[tibrvQueue_Create\(\)](#)

tibrvQueueHook

Function Type

Declaration

```
typedef void (*tibrvQueueHook) (  
    tibrvQueue      eventQueue,  
    void*           closure);
```

Purpose

Asynchronously notify an external event manager when a Rendezvous event is ready for dispatch.

Motivation

Some programs need to use existing event managers to dispatch Rendezvous events. Some event managers repeatedly poll queues for events; others require notification that an event is ready (for example, the Microsoft Windows event manager operates this way). This hook function can inform an event manager when an event enters a queue and is ready for dispatch.

You can think of this function as a wake-up call to the event manager. Once awakened, the event manager dispatches Rendezvous events in its usual way, by calling [tibrvQueue_Dispatch\(\)](#) or [tibrvQueue_TimedDispatch\(\)](#).

Remarks

When coding hook functions, remember that hook functions can be called from any thread. In general, hook functions run in the thread that enqueues the event; that thread can be a program thread, or a Rendezvous internal thread.

Procedure

To use this facility, follow these steps:

Procedure

1. Code a queue hook function to post a message to the target event manager when a message is received.
2. Arrange for the target event handler to call [tibrvQueue_Dispatch\(\)](#).
3. Register the queue hook function, using [tibrvQueue_SetHook\(\)](#).

Parameter	Description
<code>eventQueue</code>	This parameter receives the queue on which an event has arrived.
<code>closure</code>	This parameter receives closure data supplied by the program when it registered this hook function.

See Also

[tibrvQueue_Dispatch\(\)](#)

[tibrvQueue_GetHook\(\)](#)

[tibrvQueue_RemoveHook\(\)](#)

[tibrvQueue_SetHook\(\)](#)

[tibrvQueue_TimedDispatch\(\)](#)

tibrvQueueLimitPolicy

Type

Declaration

```
typedef enum {  
    TIBRVQUEUE_DISCARD_NONE,  
    TIBRVQUEUE_DISCARD_NEW,  
    TIBRVQUEUE_DISCARD_FIRST,  
    TIBRVQUEUE_DISCARD_LAST } tibrvQueueLimitPolicy;
```

Purpose

Specify a strategy for resolving overflow of queue limit.

Value	Description
TIBRVQUEUE_DISCARD_NONE	Never discard events; use this policy when a queue has no limit on the number of events it can contain.
TIBRVQUEUE_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).
TIBRVQUEUE_DISCARD_LAST	Discard the last event in the queue (that is, the youngest event in the queue).
TIBRVQUEUE_DISCARD_NEW	Discard the new event (which would otherwise cause the queue to overflow its maximum events limit).

See Also

[tibrvQueue_Create\(\)](#)

[tibrvQueue_GetLimitPolicy\(\)](#)

[tibrvQueue_SetLimitPolicy\(\)](#)

`QUEUE.LIMIT_EXCEEDED` in TIBCO Rendezvous Concepts

`tibrvQueueOnComplete`

Function Type

Declaration

```
typedef void (*tibrvQueueOnComplete) (  
    tibrvQueue    destroyedQueue,  
    void*         closure);
```

Purpose

A program can destroy a queue object even when callback functions from its events are running in one or more threads. Multi-threaded programs can define functions of this type to discover when all event callback functions in progress have completed.

Parameter	Description
<code>destroyedQueue</code>	This parameter receives the queue object. However, by the time this function runs, the queue is already destroyed; this function cannot use the queue object in Rendezvous calls.
<code>closure</code>	This parameter receives the closure data, which the program supplied in the call that destroyed the queue object.

Remarks

This type of function is important when several threads dispatch from the same queue, and the program must do additional processing after the callback functions have completed *in all threads*.

Upon return from `tibrvQueue_DestroyEx()`, the destroyed queue can no longer dispatch events. However, in each thread where an event callback function is already in progress, that callback function does continue to run until complete.

`tibrvQueue_DestroyEx()` accepts a *completion function* argument of type [tibrvQueueOnComplete](#). Rendezvous software ensures that the completion function runs when the last callback-in-progress has completed.

See Also

[tibrvQueue_Destroy\(\)](#)

tibrvQueue_Create()

Function

Declaration

```
tibrv_status tibrvQueue_Create(  
    tibrvQueue* eventQueue);
```

Purpose

Create an event queue.

Parameter	Description
eventQueue	The program supplies a location, and the function stores the address of a new queue in that location. The queue remains valid until the program explicitly destroys it.

Remarks

Upon creation, new queues use these default values.

Property	Default Value	Set Function
limitPolicy	<code>TIBRVQUEUE_DISCARD_NONE</code>	<code>tibrvQueue_SetLimitPolicy()</code>
maxEvents	zero (unlimited)	
discardAmount	zero	

Property	Default Value	Set Function
name	tibrvQueue	tibrvQueue_SetName()
priority	1	tibrvQueue_SetPriority()

tibrvQueue_Destroy()

Function

Declaration

```
tibrv_status tibrvQueue_Destroy(  
    tibrvQueue    eventQueue);  
tibrv_status tibrvQueue_DestroyEx(  
    tibrvQueue    eventQueue,  
    tibrvQueueOnComplete completionFn,  
    void*         closure);
```

Purpose

Destroy an event queue.

Remarks

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

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()`.

Parameter	Description
<code>eventQueue</code>	Destroy this queue.
<code>completionFn</code>	Rendezvous software runs this function immediately after all event callback functions dispatched from the queue have completed. If no event callback functions are running when the queue is destroyed, the destroy call runs the completion function before returning.

Parameter	Description
	If this parameter is NULL, <code>tibrvQueue_DestroyEx()</code> does not run a completion function; instead, its behavior is the same as tibrvQueue_Destroy() .
<code>closure</code>	Pass this closure argument to the completion function.

See Also

[tibrvQueueOnComplete](#)

tibrvQueue_Dispatch()

Macro

Declaration

```
tibrv_status tibrvQueue_Dispatch(  
    tibrvQueue    eventQueue);
```

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.

Parameter	Description
eventQueue	Dispatch an event from the head of this queue.

See Also

[tibrvQueue_Poll\(\)](#)

[tibrvQueue_TimedDispatch\(\)](#)[tibrvQueue_TimedDispatch\(\)](#)

tibrvQueue_GetCount()

Function

Declaration

```
tibrv_status tibrvQueue_GetCount(  
    tibrvQueue    eventQueue,  
    tibrv_u32*    numEvents);
```

Purpose

Extract the number of events in a queue.

Parameter	Description
eventQueue	Extract the current event count of this queue.
numEvents	The program supplies a location, and the function stores (a snapshot of) the event count of the queue in that location.

tibrvQueue_GetHook()

Function

Declaration

```
tibrv_status tibrvQueue_GetHook(  
    tibrvQueue      eventQueue,  
    tibrvQueueHook* eventQueueHook);
```

Purpose

Extract an event queue hook function.

Parameter	Description
eventQueue	Get the hook function from this queue.
eventQueueHook	The program supplies a location. The function stores the hook function in that location.

See Also

[tibrvQueueHook](#)

[tibrvQueue_RemoveHook\(\)](#)

[tibrvQueue_SetHook\(\)](#)

tibrvQueue_GetLimitPolicy()

Function

Declaration

```
tibrv_status tibrvQueue_GetLimitPolicy(  
    tibrvQueue          eventQueue,  
    tibrvQueueLimitPolicy* policy,  
    tibrv_u32*          maxEvents,  
    tibrv_u32*          discardAmount);
```

Purpose

Extract the limit properties of a queue.

Parameter	Description
eventQueue	Extract the limit information from this queue.
policy	<p>Each queue has a policy for discarding events when a new event would cause the queue to exceed its <code>maxEvents</code> limit. For an explanation of the policy values, see tibrvQueueLimitPolicy.</p> <p>The program supplies a location, and the function stores the limit policy of the queue in that location.</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 location, and the function stores the maximum event limit of the queue in that</p>

Parameter	Description
	location.
<code>discardAmount</code>	<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>The program supplies a location, and the function stores the discard amount of the queue in that location.</p>

See Also

[tibrvQueueLimitPolicy](#)

[tibrvQueue_Create\(\)](#)

[tibrvQueue_SetLimitPolicy\(\)](#)

QUEUE.LIMIT_EXCEEDED in TIBCO Rendezvous Concepts

tibrvQueue_GetName()

Function

Declaration

```
tibrv_status tibrvQueue_GetName(  
    tibrvQueue      eventQueue,  
    const char**    queueName);
```

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
<code>eventQueue</code>	Extract the name of this queue.
<code>queueName</code>	<p>The program supplies a location, and the function places in that location a string pointer to the queue name.</p> <p>The program <i>must not</i> modify the string.</p>

See Also

[tibrvQueue_Create\(\)](#)

`tibrvQueue_SetName()`

`tibrvQueue_GetPriority()`

Function

Declaration

```
tibrv_status tibrvQueue_GetPriority(  
    tibrvQueue eventQueue,  
    tibrv_u32* priority);
```

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.

Parameter	Description
<code>eventQueue</code>	Extract the priority of this queue.
<code>priority</code>	The program supplies a location, and the function copies the priority of the queue into that location.

See Also

[tibrvQueue_Create\(\)](#)

[tibrvQueue_SetPriority\(\)](#)

`tibrvQueue_Poll()`

Macro

Declaration

```
tibrv_status tibrvQueue_Poll(  
    tibrvQueue    eventQueue);
```

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`.

Parameter	Description
<code>eventQueue</code>	Dispatch an event from the head of this queue.

See Also

[`tibrvQueue_Dispatch\(\)`](#)

[`tibrvQueue_TimedDispatch\(\)`](#)[`tibrvQueue_TimedDispatch\(\)`](#)

`tibrvQueue_RemoveHook()`

Function

Declaration

```
tibrv_status tibrvQueue_RemoveHook(  
    tibrvQueue    eventQueue);
```

Purpose

Remove the event queue hook function from a queue.

Parameter	Description
<code>eventQueue</code>	Remove the hook function from this queue.

See Also

[tibrvQueueHook](#)

[tibrvQueue_GetHook\(\)](#)

[tibrvQueue_SetHook\(\)](#)

tibrvQueue_SetHook()

Function

Declaration

```
tibrv_status tibrvQueue_SetHook(  
    tibrvQueue      eventQueue,  
    tibrvQueueHook  eventQueueHook,  
    void*           closure);
```

Purpose

Register an event queue hook function.

Parameter	Description
eventQueue	Attach the hook function to this queue.
eventQueueHook	Call this hook function whenever an event arrives on the queue.
closure	Pass this closure data to the hook function.

See Also

[tibrvQueueHook](#)

[tibrvQueue_GetHook\(\)](#)

[tibrvQueue_RemoveHook\(\)](#)

tibrvQueue_SetLimitPolicy()

Function

Declaration

```
tibrv_status tibrvQueue_SetLimitPolicy(  
    tibrvQueue          eventQueue,  
    tibrvQueueLimitPolicy policy,  
    tibrv_u32           maxEvents,  
    tibrv_u32           discardAmount);
```

Purpose

Set the limit properties of a queue.

Remarks

This function 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
eventQueue	Set the limit properties of this queue.
policy	<p>Each queue has a policy for discarding events when a new event would cause the queue to exceed its <code>maxEvents</code> limit. Choose from the values of tibrvQueueLimitPolicy.</p> <p>When <code>maxEvents</code> is zero (unlimited), the policy must be TIBRVQUEUE_DISCARD_NONE.</p> <p>The program supplies a value, and the function sets</p>

Parameter	Description
	the limit policy of the queue to that value.
<code>maxEvents</code>	<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; in this case, the policy must be <code>TIBRVQUEUE_DISCARD_NONE</code>.</p> <p>The program supplies a value, and the function sets the maximum event limit of the queue to that value.</p>
<code>discardAmount</code>	<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 <code>discardAmount</code> is zero, the policy must be <code>TIBRVQUEUE_DISCARD_NONE</code>.</p> <p>The program supplies a value, and the function sets the discard amount of the queue to that value.</p>

See Also

[tibrvQueue_GetLimitPolicy\(\)](#)

QUEUE.LIMIT_EXCEEDED in TIBCO Rendezvous Concepts

tibrvQueue_SetName()

Function

Declaration

```
tibrv_status tibrvQueue_SetName(  
    tibrvQueue    eventQueue,  
    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
<code>eventQueue</code>	Set the name of this queue.
<code>queueName</code>	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()`

Function

Declaration

```
tibrv_status tibrvQueue_SetPriority(  
    tibrvQueue eventQueue,  
    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
<code>eventQueue</code>	Set the priority of this queue.
<code>priority</code>	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\(\)`](#)

[`tibrvQueueGroup_Dispatch\(\)`](#)

tibrvQueue_TimedDispatch()

Function

Declaration

```
tibrv_status tibrvQueue_TimedDispatch(  
    tibrvQueue eventQueue,  
    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
<code>eventQueue</code>	Dispatch the next event from this queue.
<code>timeout</code>	<p>Maximum time (in seconds) that this call can block while waiting for an event to arrive in the queue.</p> <p><code>TIBRV_NO_WAIT</code> (zero) indicates no blocking (immediate timeout).</p> <p><code>TIBRV_WAIT_FOREVER</code> (-1) indicates no timeout.</p>

See Also

[tibrvQueue_Dispatch\(\)](#)

[tibrvQueue_Poll\(\)](#)

Event Queue Groups

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.

Function or Type	Description
tibrvQueueGroup	Specify an event queue group.
tibrvQueueGroup_Add()	Add an event queue to a queue group.
tibrvQueueGroup_Create()	Create an event queue group.
tibrvQueueGroup_Destroy()	Destroy an event queue group.
tibrvQueueGroup_Dispatch()	Dispatch an event from a queue group; if no event is ready, block.
tibrvQueueGroup_Poll()	Dispatch an event from a queue group, if possible.
tibrvQueueGroup_Remove()	Remove an event queue from a queue group.
tibrvQueueGroup_TimedDispatch()	Dispatch an event, but if no event is ready to dispatch, limit the time that this call blocks while waiting for an event.

`tibrvQueueGroup`

Type

Declaration

```
typedef tibrvId tibrvQueueGroup;
```

Purpose

Specify an event queue group.

See Also

[tibrvQueueGroup_Create\(\)](#)

`tibrvQueueGroup_Add()`

Function

Declaration

```
tibrv_status tibrvQueueGroup_Add(  
    tibrvQueueGroup eventQueueGroup,  
    tibrvQueue      eventQueue);
```

Purpose

Add an event queue to a queue group.

Remarks

If the queue is already in the group, adding it again has no effect, and the call returns [TIBRV_OK](#).

If either the queue or the group is invalid, this function returns an error status.

Parameter	Description
<code>eventQueueGroup</code>	Add an event queue to this queue group.
<code>eventQueue</code>	Add this event queue to a queue group.

See Also

[tibrvQueue](#)

[tibrvQueueGroup_Remove\(\)](#)

`tibrvQueueGroup_Create()`

Function

Declaration

```
tibrv_status tibrvQueueGroup_Create(  
    tibrvQueueGroup* eventQueueGroup);
```

Purpose

Create an event queue group.

Parameter	Description
<code>eventQueueGroup</code>	<p>The program supplies a location, and the function stores the address of a new queue group in that location.</p> <p>The queue group remains valid until the program explicitly destroys it.</p>

See Also

[`tibrvQueueGroup_Destroy\(\)`](#)

`tibrvQueueGroup_Destroy()`

Function

Declaration

```
tibrv_status tibrvQueueGroup_Destroy(  
    tibrvQueueGroup eventQueueGroup);
```

Purpose

Destroy an event queue group.

Remarks

The individual queues in the group continue to exist, even though the group has been destroyed.

Parameter	Description
<code>eventQueueGroup</code>	Destroy this event queue group.

See Also

[`tibrvQueueGroup_Create\(\)`](#)

`tibrvQueueGroup_Dispatch()`

Macro

Declaration

```
tibrv_status tibrvQueueGroup_Dispatch(  
    tibrvQueueGroup eventQueueGroup);
```

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.

Parameter	Description
<code>eventQueueGroup</code>	Dispatch the next event from a queue in this group.

See Also

[`tibrvQueueGroup_Poll\(\)`](#)

[`tibrvQueueGroup_TimedDispatch\(\)`](#)

`tibrvQueueGroup_Poll()`

Macro

Declaration

```
tibrv_status tibrvQueueGroup_Poll(  
    tibrvQueueGroup eventQueueGroup);
```

Purpose

Dispatch an event from a queue group, if possible.

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](#).

Parameter	Description
<code>eventQueueGroup</code>	Dispatch the next event from a queue in this group.

See Also

[tibrvQueueGroup_Dispatch\(\)](#)

[tibrvQueueGroup_TimedDispatch\(\)](#)

`tibrvQueueGroup_Remove()`

Function

Declaration

```
tibrv_status tibrvQueueGroup_Remove(  
    tibrvQueueGroup eventQueueGroup,  
    tibrvQueue      eventQueue);
```

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
<code>eventQueueGroup</code>	Remove an event queue from this queue group.
<code>eventQueue</code>	Remove this event queue from a queue group.

See Also

[tibrvQueue](#)

[tibrvQueueGroup_Add\(\)](#)

`tibrvQueueGroup_TimedDispatch()`

Function

Declaration

```
tibrv_status tibrvQueueGroup_TimedDispatch(  
    tibrvQueueGroup eventQueueGroup,  
    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
<code>eventQueueGroup</code>	Dispatch the next event from a queue in this group.

Parameter	Description
<code>timeout</code>	<p>Maximum time (in seconds) that this call can block while waiting for an event to arrive in the queue group.</p> <p><code>TIBRV_NO_WAIT</code> (zero) indicates no blocking (immediate timeout).</p> <p><code>TIBRV_WAIT_FOREVER</code> (-1) indicates no timeout.</p>

See Also

[tibrvQueue_TimedDispatch\(\)](#)

[tibrvQueueGroup_Dispatch\(\)](#)

[tibrvQueueGroup_Poll\(\)](#)

Dispatcher Thread

Every program must dispatch events. This section describes functions that conveniently create dispatcher threads. Programmers can use this convenience facility, or arrange for event dispatch in other ways.

Function or Type	Description
tibrvDispatchable	Specify an event queue or queue group.
tibrvDispatcher	Specify a dispatcher thread.
tibrvDispatcher_Create()	Create a dispatcher thread.
tibrvDispatcher_Destroy()	Destroy a dispatcher thread.
tibrvDispatcher_GetName()	Extract the name of a dispatcher thread.
tibrvDispatcher_SetName()	Set the name of a dispatcher thread.

`tibrvDispatchable`

Type

Declaration

```
typedef tibrvId tibrvdispatchable;
```

Purpose

Specify an event queue or queue group.

Remarks

This type can refer to either kind of dispatchable object—an event queue or a queue group.

See Also

[tibrvDispatcher_Create\(\)](#)

`tibrvDispatcher`

Type

Declaration

```
typedef tibrvId tibrvdispatcher;
```

Purpose

Specify a dispatcher thread.

See Also

[tibrvDispatcher_Create\(\)](#)

tibrvDispatcher_Create()

Function

Declaration

```
tibrv_status tibrvDispatcher_Create(  
    tibrvDispatcher* dispatcher,  
    tibrvDispatchable dispatchable);  
tibrv_status tibrvDispatcher_CreateEx(  
    tibrvDispatcher* dispatcher,  
    tibrvDispatchable dispatchable,  
    tibrv_f64        idleTimeout);
```

Purpose

Create a dispatcher thread.

Remarks

A dispatcher thread repeatedly dispatches a queue or queue group.

Inside the thread, a loop calls `tibrvQueue_TimedDispatch()` or `tibrvQueueGroup_TimedDispatch()` (as appropriate to the dispatchable argument).

The simple form of this function creates a dispatcher thread that loops indefinitely. The extended function creates a thread that passes the `idleTimeout` argument to the timed dispatch call; 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>dispatcher</code>	The program supplies a location, and the function stores the address of the new dispatcher thread in that location.

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

[`tibrvQueue_TimedDispatch\(\)`](#)

[`tibrvQueueGroup_TimedDispatch\(\)`](#)

[`tibrvDispatchable`](#)

[`tibrvDispatcher_Destroy\(\)`](#)

[`tibrvDispatcher_SetName\(\)`](#)

`DISPATCHER.THREAD_EXITED` in TIBCO Rendezvous Concepts

tibrvDispatcher_Destroy()

Function

Declaration

```
tibrv_status tibrvDispatcher_Destroy(  
    tibrvDispatcher dispatcher);
```

Purpose

Destroy a dispatcher thread.

Parameter	Description
dispatcher	<p>Destroy and exit this dispatcher thread.</p> <p>We do not recommend destroying a dispatcher thread within the same thread (for example, from within a listener callback 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).</p>

See Also

DISPATCHER.THREAD_EXITED in TIBCO Rendezvous Concepts

tibrvDispatcher_GetName()

Function

Declaration

```
tibrv_status tibrvDispatcher_GetName(  
    tibrvDispatcher dispatcher,  
    const char**    dispatchName);
```

Purpose

Extract the name of a dispatcher thread.

Parameter	Description
dispatcher	Get the name of this thread.
dispatchName	The program supplies a location, and the function stores the name of the dispatcher thread in that location.

tibrvDispatcher_SetName()

Function

Declaration

```
tibrv_status tibrvDispatcher_SetName(  
    tibrvDispatcher dispatcher,  
    const char*      dispatchName);
```

Purpose

Set the name of a dispatcher thread.

Parameter	Description
dispatcher	Set the name of this thread.
dispatchName	Use this name as the name of the dispatcher thread.

Transport

Transports manage network connections and send outbound messages.

Function or Type	Description
tibrvTransport	A transport object represents a delivery mechanism for messages.
tibrvTransportBatchMode	Specify the batch mode of a transport.
tibrvTransport_Create()	Create a network transport.
tibrvTransport_CreateInbox()	Create a unique inbox subject name.
tibrvTransport_Destroy()	Destroy a transport.
tibrvTransport_GetDaemon()	Extract the daemon parameter from a transport.
tibrvTransport_GetDescription()	Extract the program description parameter from a transport.
tibrvTransport_GetNetwork()	Extract the network parameter from a transport.
tibrvTransport_GetService()	Extract the service parameter from a transport.
tibrvTransport_RequestReliability()	Request reliability interval (message retention time) for a service.
tibrvTransport_Send()	Send a message.
tibrvTransport_SendReply()	Send a reply message.

Function or Type	Description
tibrvTransport_SendRequest()	Send a request message and wait for a reply.
tibrvTransport_SetBatchMode()	Set the batch mode parameter of a transport.
tibrvTransport_SetBatchSize()	Enable outbound batching of data from IPM, and set the batch size (in bytes).
tibrvTransport_SetDescription()	Set the program description parameter of a transport.

tibrvTransport

Type

Declaration

```
typedef tibrvId tibrvTransport;
```

Purpose

A transport object represents a delivery mechanism for messages.

Remarks

A transport describes a carrier mechanism for messages—whether across a network, among processes on a single computer, or within a process.

A transport also defines the delivery scope of a message—that is, the set of *possible* destinations for the messages it sends.

Programs must explicitly destroy each transport object. Destroying a transport object invalidates subsequent send calls on that transport, invalidates any listeners using that transport, and frees its storage.

Intra-Process Transport

Each process has exactly one intra-process transport; the call [tibrv_Open\(\)](#) automatically creates it, and arranges the constant TIBRV_PROCESS_TRANSPORT to refer to it. Programs must not destroy the intra-process transport.

See Also

[tibrvTransport_Create\(\)](#)

[tibrvTransport_Destroy\(\)](#)

[tibrvTransport_Send\(\)](#)

Transport in TIBCO Rendezvous Concepts

Intra-Process Transport and User Events in TIBCO Rendezvous Concepts

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 <code>rvd</code> 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 <code>rvd</code> in batches—either when its buffer is full, or when a timer interval expires. (Programs cannot adjust the timer interval.)

See Also

[tibrvTransport_SetBatchMode\(\)](#)

Batch Modes for Transports in TIBCO Rendezvous Concepts

tibrvTransport_Create()

Function

Declaration

```
tibrv_status tibrvTransport_Create(  
    tibrvTransport*    transport,  
    const char*        service,  
    const char*        network,  
    const char*        daemon);  
tibrv_status tibrvTransport_CreateLicensed(  
    tibrvTransport*    transport,  
    const char*        service,  
    const char*        network,  
    const char*        daemon,  
    const char*        licenseTicket);
```

Purpose

Create a network transport.

Remarks

These calls create only network transports. The call [tibrv_Open\(\)](#) automatically creates the intra-process transport; programs cannot create additional intra-process transports.

Parameter	Description
transport	<p>The program supplies a location, and the function stores the new transport in that location.</p> <p>The transport remains valid until the program explicitly destroys it.</p>
service	<p>The Rendezvous daemon divides the network into logical partitions. Each network transport communicates on a single service; a transport</p>

Parameter	Description
	<p>can communicate only with other transports on the same service.</p> <p>To communicate over 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>
<code>network</code>	<p>Every network transport communicates with other transports over a 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 Network Parameter in TIBCO Rendezvous Concepts.</p> <p>NULL specifies the primary network interface for the host computer.</p>
<code>daemon</code>	<p>The <code>daemon</code> parameter instructs <code>tibrvTransport_Create()</code> about how and where to find the Rendezvous daemon and establish communication.</p> <p>For details, see Daemon Parameter in TIBCO Rendezvous Concepts.</p> <p>You can specify a daemon on a remote computer. For details, see Remote Daemon in TIBCO Rendezvous Concepts.</p> <p>If you specify a secure daemon, this string must be identical to as the <code>daemonName</code> argument of tibrvSecureDaemon_SetDaemonCert(). See also, Secure Daemon in TIBCO Rendezvous Concepts.</p> <p><code>null</code> 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 network transport must connect to a Rendezvous daemon.

If a Rendezvous daemon process with a corresponding daemon parameter is already running, `tibrvTransport_Create()` connects to it.

If an appropriate Rendezvous local daemon is *not* running, `tibrvTransport_Create()` tries to start it. However, `tibrvTransport_Create()` does not attempt to start a *remote* daemon when none is running.

If `tibrvTransport_Create()` cannot connect to the Rendezvous daemon, it returns the status code `TIBRV_DAEMON_NOT_CONNECTED`, and does not create a 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\(\)](#).

Destroying Transports

Programs must explicitly destroy each transport object. Destroying a transport object invalidates subsequent send calls on that transport, invalidates any listeners using that transport, and frees its storage. See [tibrvTransport_Destroy\(\)](#).

Attempting to use a destroyed transport in any way is an error.

See Also

[tibrvTransport_Destroy\(\)](#)

tibrvTransport_CreateInbox()

Function

Declaration

```
tibrv_status tibrvTransport_CreateInbox(  
    tibrvTransport    transport,  
    char*             subjectString,  
    tibrv_u32          subjectLimit);
```

Purpose

Create a unique inbox subject name.



Warning

This function is the *only* legal way for programs to create inbox subject names.

Remarks

This function 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 function 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 `tibrvEvent_CreateListener()`. The inbox name is only valid for use with the same transport that created it. When calling `tibrvEvent_CreateListener()`, 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.

Programs can overwrite or free the `subjectString` storage at any time.

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.

Parameter	Description
<code>transport</code>	Create an inbox on this transport.
<code>subjectString</code>	The program supplies a string buffer, and the function stores the new inbox subject string in that buffer.
<code>subjectLimit</code>	The number of bytes that the program has allocated to receive the new inbox subject string.

See Also

[`tibrvEvent_CreateListener\(\)`](#)

tibrvTransport_Destroy()

Function

Declaration

```
tibrv_status tibrvTransport_Destroy(  
    tibrvTransport transport);
```

Purpose

Destroy a transport.

Remarks

Programs must explicitly destroy each transport object.

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 all its listeners.
- Subsequent calls that use the destroyed transport return an error status.
- Storage for the transport object is freed.

It is illegal to destroy the intra-process transport (`TIBRV_PROCESS_TRANSPORT`); attempting to destroy it produces the status code [TIBRV_NOT_PERMITTED](#).

Parameter	Description
transport	Destroy this transport.

See Also

[tibrvTransport](#)

[tibrvTransport_Create\(\)](#)

tibrvTransport_GetDaemon()

Function

Declaration

```
tibrv_status tibrvTransport_GetDaemon(  
    tibrvTransport transport,  
    const char**   daemonString);
```

Purpose

Extract the daemon parameter from a transport.

Remarks

This function returns the daemon parameter as a NULL-terminated string.

Parameter	Description
transport	<p>Extract the effective daemon parameter from this transport.</p> <p>It is an error to supply the intra-process transport or virtual circuit transport to this function.</p>
daemonString	<p>The program supplies a location, and the function places in that location a string pointer to the daemon information.</p> <p>The program <i>must not</i> modify nor free the string.</p>

See Also

[tibrvTransport](#)

`tibrvTransport_Create()`

tibrvTransport_GetDescription()

Function

Declaration

```
tibrv_status tibrvTransport_GetDescription(  
    tibrvTransport transport,  
    const char**   description);
```

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.

This function returns the description parameter as a NULL-terminated string.

Parameter	Description
transport	Extract the description parameter from this transport. It is an error to supply the intra-process transport or virtual circuit transport to this function.
description	The program supplies a location, and the function places in that location a string pointer to the description. The program <i>must not</i> modify nor free the string.

See Also

[tibrvTransport](#)

[tibrvTransport_SetDescription\(\)](#)

tibrvTransport_GetNetwork()

Function

Declaration

```
tibrv_status tibrvTransport_GetNetwork(  
    tibrvTransport transport,  
    const char**   networkString);
```

Purpose

Extract the network parameter from a transport.

Remarks

This function returns the network parameter as a NULL-terminated string.

Parameter	Description
transport	Extract the network parameter from this transport. It is an error to supply the intra-process transport or virtual circuit transport to this function.
networkString	The program supplies a location, and the function places in that location a string pointer to the network parameter. The program <i>must not</i> modify the string.

See Also

[tibrvTransport](#)

[tibrvTransport_Create\(\)](#)

tibrvTransport_GetService()

Function

Declaration

```
tibrv_status tibrvTransport_GetService(  
    tibrvTransport transport,  
    const char** serviceString);
```

Purpose

Extract the service parameter from a transport.

Remarks

This function returns the service parameter as a NULL-terminated string.

Parameter	Description
transport	Extract the service parameter from this transport. It is an error to supply the intra-process transport or virtual circuit transport to this function.
serviceString	The program supplies a location, and the function places in that location a string pointer to the service information. The program <i>must not</i> modify the string.

See Also

[tibrvTransport](#)

[tibrvTransport_Create\(\)](#)

tibrvTransport_RequestReliability()

Function

Declaration

```
tibrv_status tibrvTransport_RequestReliability(  
    tibrvTransport    transport,  
    tibrv_f64         reliability);
```

Purpose

Request reliability interval (message retention time) for a service.

Parameter	Description
transport	Request reliability on the service of this transport.
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.

Programs can request reliability only from daemons of release 8.2 or later.

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()`

Function

Declaration

```
tibrv_status tibrvTransport_Send(  
    tibrvTransport transport,  
    tibrvMsg message);
```

Purpose

Send a message.

Parameter	Description
transport	Send the message on this transport.
message	Send this message.

See Also

[tibrvMsg](#)

[tibrvMsg_Create\(\)](#)

[tibrvTransport](#)

tibrvTransport_Sendv()

Function

Declaration

```
tibrv_status tibrvTransport_Sendv(  
    tibrvTransport    transport,  
    tibrvMsg*         messageVector,  
    tibrv_u32          length);
```

Purpose

Send a vector of messages.

Remarks

This function sends a group of messages with one call. In most applications this call is more efficient than a series of send calls on individual messages.



Note

`messageVector` is an array of [tibrvMsg](#) objects. To avoid array out-of-bounds errors, be careful to pass an appropriate `length` (that is, a length that is no larger than the number of [tibrvMsg](#) objects in the array).

Parameter	Description
<code>transport</code>	Send the messages on this transport.
<code>messageVector</code>	Send these messages.
<code>length</code>	Number of messages in the vector.

See Also

[tibrvMsg](#)

[tibrvMsg_Create\(\)](#)

[tibrvTransport](#)

[tibrvTransport_Send\(\)](#)

tibrvTransport_SendReply()

Function

Declaration

```
tibrv_status tibrvTransport_SendReply(  
    tibrvTransport    transport,  
    tibrvMsg          replyMessage,  
    tibrvMsg          requestMessage);
```

Purpose

Send a reply message.

Remarks

This function 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 function overwrites any existing send subject of the reply message with the reply subject of the request message.

Parameter	Description
transport	Send the reply on this transport.
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 function. 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_Create\(\)](#)

[tibrvTransport](#)

tibrvTransport_SendRequest()

Function

Declaration

```
tibrv_status tibrvTransport_SendRequest(
    tibrvTransport transport,
    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
transport	Send the message and receive the reply on this transport.
message	Send this outbound message.
reply	<p>The program supplies a location, and the function stores the inbound reply in that location.</p> <p>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.</p>

Parameter	Description
<code>timeout</code>	Maximum time (in seconds) that this call can block while waiting for a reply. <code>TIBRV_WAIT_FOREVER</code> (-1) indicates no timeout (wait without limit for a reply).

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 function 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, return the status code `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 location specified by the `reply` parameter.
5. Return.

See Also

[tibrvMsg_Create\(\)](#)

[tibrvTransport](#)

`tibrvTransport_SetBatchMode()`

Function

Declaration

```
tibrv_status tibrvTransport_SetBatchMode(  
    tibrvTransport      transport,  
    tibrvTransportBatchMode mode);
```

Purpose

Set the batch mode parameter of a transport.

Remarks

This type of batching is available only with the standard (daemon-based) Rendezvous library. It is not available with the IPM library.

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
<code>transport</code>	Set the batch mode parameter of this transport.
<code>mode</code>	Use this value as the new batch mode.

See Also

[tibrvTransport](#)

[tibrvTransportBatchMode](#)

Batch Modes for Transports in TIBCO Rendezvous Concepts

tibrvTransport_SetBatchSize()

Declaration

```
tibrv_status tibrvTransport_SetBatchSize(  
    tibrvTransport    transport,  
    tibrv_u32         numBytes);
```

Purpose

Enable outbound batching of data from IPM, and set the batch size (in bytes).

Remarks

This type of batching is available only with the IPM library. It is not available with the standard (daemon-based) Rendezvous library.

When the batch size is greater than zero, IPM transfers data to the network in batches. This option can increase throughput, at the cost of higher latency.

When the batch size is zero, IPM transfers data to the network immediately, for lowest latency.

If you do not explicitly set the batch size using this call, then the default behavior disables outbound batching.



Contraindications

Note

These conditions characterize situations in which we do not recommend batching:

- Data latency is *not* acceptable.
 - Batch behavior does *not* produce measurable improvements in the performance of your application.
-

Parameter	Description
<code>transport</code>	Enable (or disable) batching for this transport.
<code>numBytes</code>	Set the batch size (in bytes). Zero is a special value, which disables batching for the transport.

tibrvTransport_SetDescription()

Function

Declaration

```
tibrv_status tibrvTransport_SetDescription(  
    tibrvTransport transport,  
    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.

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
transport	Set the description parameter of this transport. It is an error to supply the intra-process transport or virtual circuit transport to this function.
description	Use this string as the new program description. The function copies this string.

See Also

[tibrvTransport](#)

307 | `tibrvTransport_SetDescription()`

`tibrvTransport_GetDescription()`

Virtual Circuits

Virtual circuits feature Rendezvous communication between two terminals over an exclusive, continuous, monitored connection.

See Also

Virtual Circuits in TIBCO Rendezvous Concepts

Function or Type	Description
tibrvTransport_CreateAcceptVc()	Create a virtual circuit accept object.
tibrvTransport_CreateConnectVc()	Create a virtual circuit connect object.
tibrvTransport_WaitForVcConnection()	Test the connection status of a virtual circuit.

tibrvTransport_CreateAcceptVc()

Function

Declaration

```
tibrv_status tibrvTransport_CreateAcceptVc(  
    tibrvTransport*    vcTransport,  
    const char**       connectSubject,  
    tibrvTransport     transport);
```

Purpose

Create a virtual circuit accept object.

Remarks

A virtual circuit transport can fill the same roles as an ordinary transport. Programs can supply them as arguments to calls that create inbox names, send messages, create listeners and other events.

Parameter	Description
<code>vcTransport</code>	The program supplies a location, and the function stores the new virtual circuit accept transport in that location.
<code>connectSubject</code>	<p>The program supplies a location, and the function 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 <code>tibrvTransport_</code></p>

Parameter	Description
	CreateConnectVc() .
<code>transport</code>	<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.
- [tibrvTransport_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 [tibrvTransport_WaitForVcConnection\(\)](#) with zero as the timeout parameter.

For an explanation, see Testing the New Connection in TIBCO Rendezvous Concepts.

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

Programs must explicitly destroy each virtual circuit transport object. Destroying a transport object precludes subsequent communications on that transport, and frees its storage. Attempting to use a destroyed transport in any way is an error. See [tibrvTransport_Destroy\(\)](#).

Destroying a virtual circuit transport does *not* affect the ordinary transport that the terminal employs.

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.

Disabled Functions

Although virtual circuit transport objects have the same data type as transport objects, some transport functions are disabled for virtual circuit objects.

Transport Functions	
Legal Calls for Virtual Circuit Transports	tibrvTransport_CreateInbox()
	tibrvTransport_Destroy()
	tibrvTransport_Send()
	tibrvTransport_SendReply()
	tibrvTransport_SendRequest()
Disabled Calls for Virtual Circuit Transports	tibrvTransport_GetDaemon()
	tibrvTransport_GetDescription()
	tibrvTransport_GetNetwork()

Transport Functions

[`tibrvTransport_GetService\(\)`](#)

[`tibrvTransport_SetBatchMode\(\)`](#)

[`tibrvTransport_SetDescription\(\)`](#)

See Also

[`tibrvTransport_Destroy\(\)`](#)

[`tibrvTransport_CreateConnectVc\(\)`](#)

[`tibrvTransport_WaitForVcConnection\(\)`](#)

`VC.CONNECTED` in TIBCO Rendezvous Concepts

`VC.DISCONNECTED` in TIBCO Rendezvous Concepts

tibrvTransport_CreateConnectVc()

Function

Declaration

```
tibrv_status tibrvTransport_CreateConnectVc(  
    tibrvTransport*    vcTransport,  
    const char*        connectSubject,  
    tibrvTransport     transport);
```

Purpose

Create a virtual circuit connect object.

Remarks

A virtual circuit transport can fill the same roles as an ordinary transport. Programs can supply them as arguments to calls that create inbox names, send messages, create listeners and other events.

Parameter	Description
<code>vcTransport</code>	The program supplies a location, and the function stores the new virtual circuit connect transport in that location.
<code>connectSubject</code>	<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 <code>tibrvTransport_CreateAcceptVc()</code> creates this subject.</p>
<code>transport</code>	The virtual circuit uses this ordinary transport for communications.

Parameter	Description
	Programs may use this transport for other purposes.
	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).

Test Before Using

Either of two conditions indicate that the connection is ready to use:

- The transport presents the `VC.CONNECTED` advisory.
- [`tibrvTransport_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 [`tibrvTransport_WaitForVcConnection\(\)`](#) with zero as the timeout parameter.

For an explanation, see Testing the New Connection in TIBCO Rendezvous Concepts.

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

Programs must explicitly destroy each virtual circuit transport object. Destroying a transport object precludes subsequent communications on that transport, and frees its

storage. Attempting to use a destroyed transport in any way is an error. See [tibrvTransport_Destroy\(\)](#).

Destroying a virtual circuit transport does *not* affect the ordinary transport that the terminal employs.

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.

See Also

[tibrvTransport_Destroy\(\)](#)

[tibrvTransport_CreateAcceptVc\(\)](#)

[tibrvTransport_WaitForVcConnection\(\)](#)

VC.CONNECTED in TIBCO Rendezvous Concepts

VC.DISCONNECTED in TIBCO Rendezvous Concepts

tibrvTransport_WaitForVcConnection()

Function

Declaration

```
tibrv_status tibrvTransport_WaitForVcConnection(  
    tibrvTransport vcTransport,  
    tibrv_f64      timeout);
```

Purpose

Test the connection status of a virtual circuit.

Remarks

This function produces the same information as the virtual circuit advisory messages—but it produces it synchronously (while advisories are asynchronous). Programs can use this function 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 function to wait until the connection completes.

Parameter	Description
vcTransport	Wait until this virtual circuit transport object has established a connection with its opposite terminal. You may supply either an accept terminal or a connect terminal as an argument.
timeout	This parameter determines the behavior of the call: <ul style="list-style-type: none">• For a quick test of current connection status, supply zero. The call returns immediately, without blocking.

Parameter	Description
	<ul style="list-style-type: none">• 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.• 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.

Status	Description
TIBRV_OK	The connection is complete (ready to use).
TIBRV_TIMEOUT	The connection is not yet complete, but the non-negative time limit for waiting has expired.
TIBRV_VC_NOT_CONNECTED	The connection was formerly complete, but is now irreparably broken.

See Also

[tibrvTransport_CreateAcceptVc\(\)](#)

[tibrvTransport_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 system. Some processes actively fulfill the tasks of the application, while other processes wait in readiness. When one of the active processes fails, another process rapidly assumes active duty.

See Also

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.

Function or Type	Description
tibrvft_Version()	Identify the fault tolerance API release number.
tibrvftAction	Instruct fault tolerance callback functions to react to changing circumstances.
tibrvftMember	Member objects represent program membership in a fault tolerance group.

Function or Type	Description
tibrvftMemberCallback	Process fault tolerance events for a group member.
tibrvftMemberOnComplete	A program can destroy a member object even when its callback function is running in one or more threads. Multi-threaded programs can define functions of this type to discover when all callback functions in progress have completed.
tibrvftMember_Create()	Create a member of a fault tolerance group.
tibrvftMember_Destroy()	TIBRV_TIMEOUT
tibrvftMember_GetGroupName()	Extract the group name of a fault tolerance member.
tibrvftMember_GetQueue()	Extract the event queue of a fault tolerance member.
tibrvftMember_GetTransport()	Extract the transport of a fault tolerance member.
tibrvftMember_GetWeight()	Extract the weight of a fault tolerance member.
tibrvftMember_SetWeight()	Change the weight of a fault tolerance member within its group.
Monitors	
tibrvftMonitor	Monitor objects express interest in fault tolerance events.
tibrvftMonitorCallback	Process fault tolerance events for a monitor.
tibrvftMonitorOnComplete	A program can destroy a monitor object even when its callback function is running in one or

Function or Type	Description
	more threads. Multi-threaded programs can define functions of this type to discover when all callback functions in progress have completed.
<code>tibrvftMonitor_Create()</code>	Monitor a fault tolerance group.
<code>tibrvftMonitor_Destroy()</code>	Stop monitoring a fault tolerance group, and free associated resources.
<code>tibrvftMonitor_GetGroupName()</code>	Extract the group name of a fault tolerance monitor.
<code>tibrvftMonitor_GetQueue()</code>	Extract the event queue of a fault tolerance monitor.
<code>tibrvftMonitor_GetTransport()</code>	Extract the transport of a fault tolerance monitor.

tibrvft_Version()

Function

Declaration

```
const char* tibrvft_Version(void);
```

Purpose

Identify the fault tolerance API release number.

tibrvftAction

Type

Declaration

```
typedef enum
{
    TIBRVFT_PREPARE_TO_ACTIVATE = 1,
    TIBRVFT_ACTIVATE             = 2,
    TIBRVFT_DEACTIVATE           = 3
} tibrvftAction;
```

Purpose

Instruct fault tolerance callback functions to react to changing circumstances.

Remarks

Each token of this enumerated type designates a command to a fault tolerance callback function. The program's callback function 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 function to instruct the program to make itself ready to activate on short notice—so that if the callback function subsequently receives the instruction to activate, it can do so without delay.</p> <p>This token is a hint, indicating that the program might</p>

Token	Description
	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.
TIBRVFT_ACTIVATE	Activate immediately. Rendezvous fault tolerance software passes this token to the callback function to instruct the program to activate.
TIBRVFT_DEACTIVATE	Deactivate immediately. Rendezvous fault tolerance software passes this token to the callback function to instruct the program to deactivate.

See Also

[tibrvftMemberCallback](#)

[tibrvftMember_Create\(\)](#)

tibrvftMember

Type

Declaration

```
typedef tibrvId tibrvftMember;
```

Purpose

Member objects represent program membership in a fault tolerance group.

See Also

[tibrvftMember_Create\(\)](#)

tibrvftMemberCallback

Function Type

Declaration

```
typedef void (*tibrvftMemberCallback) (  
    tibrvftMember    member,  
    const char*      groupName,  
    tibrvftAction     action,  
    void*            closure);
```

Purpose

Process fault tolerance events for a group member.

Remarks

Each member program of a fault tolerance group must define a function of this type. Programs register a member callback function 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 function to activate, deactivate, or prepare to activate the program.

- When the number of active members is less than the active goal, the fault tolerance callback function (in the ranking inactive member process) receives the token `TIBRVFT_ACTIVATE`; the callback function must respond by assuming the duties of an active member.
- When the number of active members exceeds the active goal, the fault tolerance callback function (in any active member that is outranked by another active member) receives the action token `TIBRVFT_DEACTIVATE`; the callback function 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 become lower than the active goal, the fault tolerance callback function (in the ranking inactive member) receives the action

token `TIBRVFT_PREPARE_TO_ACTIVATE`; the callback function 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 function subsequently receives the `TIBRVFT_ACTIVATE` token, it will be ready to activate without delay.

For additional information see Fault Tolerance Callback Actions in TIBCO Rendezvous Concepts.

Parameter	Description
<code>member</code>	This parameter receives the member object.
<code>groupName</code>	This parameter receives a string denoting the name of the fault tolerance group.
<code>action</code>	This parameter receives a value of the enumerated type <code>tibrvftAction</code> , which instructs the callback function to activate, deactivate or prepare to activate.
<code>closure</code>	This parameter receives the closure data, which the program supplied in the call that created the member object.

See Also

[tibrvftAction](#)

[tibrvftMember_Create\(\)](#)

tibrvftMemberOnComplete

Function Type

Declaration

```
typedef void (*tibrvftMemberOnComplete) (  
    tibrvftMember destroyedMember,  
    void* closure);
```

Purpose

A program can destroy a member object even when its callback function is running in one or more threads. Multi-threaded programs can define functions of this type to discover when all callback functions in progress have completed.

Parameter	Description
destroyedMember	<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 function runs, the member is already destroyed; this function cannot use the member object in Rendezvous calls.</p>
closure	<p>This parameter receives the closure data, which the program supplied in the call that created the member object.</p>

Remarks

This type of function is important in two situations:

- Internal fault tolerance callback functions run in several threads (because several threads dispatch the member's event queue), and the program must do additional processing after these callback functions have completed *in all threads*.
- A member callback function calls `tibrvftMember_DestroyEx()` to withdraw from a fault tolerance group, and the program must do additional processing *after* the rest of the callback function has completed.

Upon return from `tibrvftMember_DestroyEx()`, the destroyed member's callback function can no longer begin to run (this is also true of internal callback functions). However, in each thread where a callback function is already in progress, that callback function does continue to run until complete.

`tibrvftMember_DestroyEx()` accepts a *completion function* argument of type [tibrvftMemberOnComplete](#). Rendezvous software ensures that the completion function runs when the last callback-in-progress has completed.

Timing and Context

This information is completely analogous to [tibrvEventOnComplete](#). See that section for important details.

See Also

[tibrvftMember_Create\(\)](#)

`tibrvftMember_DestroyEx()`, see [tibrvftMember_Destroy\(\)](#)

tibrvftMember_Create()

Function

Declaration

```
tibrv_status tibrvftMember_Create(  
    tibrvftMember*      member,  
    tibrvQueue           queue,  
    tibrvftMemberCallback callback,  
    tibrvTransport       transport,  
    const char*          groupName,  
    tibrv_u16            weight,  
    tibrv_u16            activeGoal,  
    tibrv_f64            heartbeatInterval,  
    tibrv_f64            preparationInterval,  
    tibrv_f64            activationInterval,  
    void*                closure);
```

Purpose

Create a member of a fault tolerance group.

Remarks

Upon creating a member object, the program becomes a member of the 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. The function `tibrvftMember_Create()` 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).

Parameter	Description
member	<p>This member object denotes membership in a fault tolerance group.</p> <p>The program supplies a location, and the function stores the new member object in that location.</p> <p>The member object remains valid until the program explicitly destroys it.</p>
queue	Place fault tolerance events for this member on this event queue.
callback	On dispatch, process the event with this callback function.
transport	Use this transport for fault tolerance internal protocol messages (such as heartbeat messages).
groupName	<p>Join the fault tolerant group with this name.</p> <p>The group name must conform to the syntax required for Rendezvous subject names. For details, see Subject Names in TIBCO Rendezvous Concepts.</p>
weight	<p>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.</p> <p>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.</p> <p>For more information, see Rank and Weight in TIBCO Rendezvous Concepts.</p>
activeGoal	Rendezvous fault tolerance software sends callback instructions to maintain this number of active members.

Parameter	Description
	Acceptable values range from 1 to 65535.
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 (TIBRVFT_PREPARE_TO_ACTIVATE) 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 TIBRVFT_PREPARE_TO_ACTIVATE 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 (TIBRVFT_ACTIVATE) 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 (TIBRVFT_ACTIVATE) to the new member.</p> <p>Then interval must be positive. To determine the correct value, see Step 4: Choose the Intervals in TIBCO Rendezvous Concepts.</p>
closure	Store this closure data on the member object.

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.

See Also

[tibrvftAction](#)

[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()

Function

Declaration

```
tibrv_status tibrvftMember_Destroy(  
    tibrvftMember    member);  
tibrv_status tibrvftMember_DestroyEx(  
    tibrvftMember    member,  
    tibrvftMemberOnComplete completionFunction);
```

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.

This function has two effects:

- If this member is active, stop sending the heartbeat signal.
- Reclaim the program storage associated with this member.

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.

Parameter	Description
member	Destroy this member object, and cancel membership in its fault tolerance group.

Parameter	Description
<code>completionFunction</code>	<p>Rendezvous software runs this function immediately after all instances of the member's callback function (and internal callback functions) have completed. If callback functions are not running when the event is destroyed, the destroy call runs it before returning.</p> <p>If this parameter is <code>NULL</code>, <code>tibrvftMember_DestroyEx()</code> does not run a completion function; instead, its behavior is the same as tibrvftMember_Destroy().</p> <p>See tibrvftMemberOnComplete.</p>

Extended Function

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

See Also

[tibrvftMember_Create\(\)](#)

[tibrvftMemberOnComplete](#)

tibrvftMember_GetGroupName()

Function

Declaration

```
tibrv_status tibrvftMember_GetGroupName(  
    tibrvftMember member,  
    const char**  groupName);
```

Purpose

Extract the group name of a fault tolerance member.

Parameter	Description
member	Extract the group name from this member object.
groupName	The program supplies a location, and the function stores the group name in that location.

tibrvftMember_GetQueue()

Function

Declaration

```
tibrv_status tibrvftMember_GetQueue(  
    tibrvftMember    member,  
    tibrvQueue*      queue);
```

Purpose

Extract the event queue of a fault tolerance member.

Parameter	Description
member	Extract the event queue from this member object.
queue	The program supplies a location, and the function stores the queue in that location.

tibrvftMember_GetTransport()

Function

Declaration

```
tibrv_status tibrvftMember_GetTransport(  
    tibrvftMember    member,  
    tibrvTransport*  transport);
```

Purpose

Extract the transport of a fault tolerance member.

Parameter	Description
member	Extract the transport from this member object.
transport	The program supplies a location, and the function stores the transport in that location.

tibrvftMember_GetWeight()

Function

Declaration

```
tibrv_status tibrvftMember_GetWeight(  
    tibrvftMember    member,  
    tibrv_u16*        weight);
```

Purpose

Extract the weight of a fault tolerance member.

Parameter	Description
member	Extract the weight from this member object.
weight	The program supplies a location, and the function stores the weight in that location.

tibrvftMember_SetWeight()

Function

Declaration

```
tibrv_status tibrvftMember_SetWeight(  
    tibrvftMember    member,  
    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
member	Set the weight of this member object.
weight	The new weight value. See weight .

See Also

Adjusting Member Weights in TIBCO Rendezvous Concepts

tibrvftMonitor

Type

Declaration

```
typedef tibrvId tibrvftMonitor;
```

Purpose

Monitor objects express interest in fault tolerance events.

See Also

[tibrvftMonitor_Create\(\)](#)

tibrvftMonitorCallback

Function Type

Declaration

```
typedef void (*tibrvftMonitorCallback) (  
    tibrvftMonitor      monitor,  
    const char*         groupName,  
    tibrv_u32           numActiveMembers,  
    void*               closure);
```

Purpose

Process fault tolerance events for a monitor.

Remarks

A program must define a function of this type as a prerequisite for monitoring a fault tolerance group. Programs register a monitor callback function 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. Programs that do not monitor need not define a monitor callback function.

Parameter	Description
<code>monitor</code>	This parameter receives the monitor object.
<code>groupName</code>	This parameter receives a string denoting the name of the fault tolerance group.

Parameter	Description
numActiveMembers	This parameter receives the number of group members now active.
closure	This parameter receives the closure data, which the program supplied in the call that created the monitor object.

See Also

[tibrvftMonitor_Create\(\)](#)

tibrvftMonitorOnComplete

Function Type

Declaration

```
typedef void (*tibrvftMonitorOnComplete) (  
    tibrvftMember destroyedMonitor,  
    void* closure);
```

Purpose

A program can destroy a monitor object even when its callback function is running in one or more threads. Multi-threaded programs can define functions of this type to discover when all callback functions in progress have completed.

Parameter	Description
destroyedMonitor	<p>This parameter receives the monitor event object. This object is identical to the object that the program created to monitor the fault tolerance group.</p> <p>However, by the time this function runs, the monitor is already destroyed; this function cannot use the monitor object in Rendezvous calls.</p>
closure	<p>This parameter receives the closure data, which the program supplied in the call that created the monitor object.</p>

Remarks

This type of function is important in two situations:

- Internal fault tolerance callback functions run in several threads (because several threads dispatch the monitor's event queue), and the program must do additional processing after these callback functions have completed *in all threads*.
- A monitor callback function calls `tibrvftMonitor_DestroyEx()` to withdraw from a fault tolerance group, and the program must do additional processing *after* the rest of the callback function has completed.

Upon return from `tibrvftMonitor_DestroyEx()`, the destroyed monitor's callback function can no longer begin to run (this is also true of internal callback functions). However, in each thread where a callback function is already in progress, that callback function does continue to run until complete.

`tibrvftMonitor_DestroyEx()` accepts a *completion function* argument of type [tibrvftMonitorOnComplete](#). Rendezvous software ensures that the completion function runs when the last callback-in-progress has completed.

Timing and Context

This information is completely analogous to [tibrvEventOnComplete](#). See that section for important details.

See Also

[tibrvftMonitor_Create\(\)](#)

[tibrvftMonitor_Destroy\(\)](#)

tibrvftMonitor_Create()

Function

Declaration

```
tibrv_status tibrvftMonitor_Create(  
    tibrvftMonitor*    monitor,  
    tibrvQueue         queue,  
    tibrvftMonitorCallback callback,  
    tibrvTransport     transport,  
    const char*        groupName,  
    tibrv_f64          lostInterval,  
    void*              closure);
```

Purpose

Monitor a fault tolerance group.

Remarks

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 function receives the number of active members as an argument.

The group need not have any members at the time of this function call.

Parameter	Description
monitor	This monitor object denotes a passive monitor of a fault tolerance group.

Parameter	Description
	<p>The program supplies a location, and the function stores the new monitor object in that location.</p> <p>The monitor object remains valid until the program explicitly destroys it.</p>
<code>queue</code>	Place events for this monitor on this event queue.
<code>callback</code>	On dispatch, process the event with this callback function.
<code>transport</code>	Listen on this transport for fault tolerance internal protocol messages (such as heartbeat messages).
<code>groupName</code>	<p>Monitor the fault tolerant group with this name.</p> <p>The group name must conform to the syntax required for Rendezvous subject names. For details, see Subject Names in TIBCO Rendezvous Concepts.</p> <p>See also, Group Name.</p>
<code>lostInterval</code>	<p>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.</p> <p>The interval must be positive. To determine the correct value, see Step 4: Choose the Intervals in TIBCO Rendezvous Concepts.</p> <p>See also, Lost Interval.</p>
<code>closure</code>	Store this closure data on 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()

Function

Declaration

```
tibrv_status tibrvftMonitor_Destroy(  
    tibrvftMonitor    monitor);  
tibrv_status tibrvftMonitor_DestroyEx(  
    tibrvftMonitor    monitor,  
    tibrvftMonitorOnComplete    completionFunction);
```

Purpose

Stop monitoring a fault tolerance group, and free associated resources.

Parameter	Description
monitor	Destroy this monitor object.
completionFunction	<p>Rendezvous software runs this function immediately after all instances of the monitor's callback function (and internal callback functions) have completed. If callback functions are not running when the monitor is destroyed, the destroy call runs it before returning.</p> <p>If this parameter is NULL, <code>tibrvftMonitor_DestroyEx()</code> does not run a completion function; instead, its behavior is the same as tibrvftMonitor_Destroy().</p> <p>See tibrvftMonitorOnComplete.</p>

Extended Function

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

tibrvftMonitor_GetGroupName()

Function

Declaration

```
tibrv_status tibrvftMonitor_GetGroupName(  
    tibrvftMonitor    monitor,  
    const char**      groupName);
```

Purpose

Extract the group name of a fault tolerance monitor.

Parameter	Description
monitor	Extract the group name from this monitor object.
groupName	The program supplies a location, and the function stores the group name in that location.

tibrvftMonitor_GetQueue()

Function

Declaration

```
tibrv_status tibrvftMonitor_GetQueue(  
    tibrvftMonitor    monitor,  
    tibrvQueue*       queue);
```

Purpose

Extract the event queue of a fault tolerance monitor.

Parameter	Description
monitor	Extract the event queue from this monitor object.
queue	The program supplies a location, and the function stores the queue in that location.

tibrvftMonitor_GetTransport()

Function

Declaration

```
tibrv_status tibrvftMonitor_GetTransport(  
    tibrvftMonitor    monitor,  
    tibrvTransport*   transport);
```

Purpose

Extract the transport of a fault tolerance monitor.

Parameter	Description
monitor	Extract the transport from this monitor object.
transport	The program supplies a location, and the function stores the transport in that location.

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* listeners. See Distributed Queue in TIBCO Rendezvous Concepts.

Operations by Functional Group

Function	Description
Listening for Messages	
<code>tibrvcmlistener_CreateListener()</code>	Listen for messages that match the subject, and request certified delivery when available.
<code>tibrvcmlistener_Callback</code>	Programs define functions of

Function	Description
	this type to process inbound certified or labeled messages.
<code>tibrvcmlEvent_Destroy()</code>	Destroy a certified delivery listener.
Sending Messages	
<code>tibrvcmlTransport_Send()</code>	Send a labeled message.
<code>tibrvcmlTransport_SendReply()</code>	Send a labeled reply message.
<code>tibrvcmlTransport_SendRequest()</code>	Send a labeled request message and wait for a reply.
Confirmation of Certified Messages	
<code>tibrvcmlEvent_SetExplicitConfirm()</code>	Override automatic confirmation of delivery for this listener.
<code>tibrvcmlEvent_ConfirmMsg()</code>	Explicitly confirm delivery of a certified message.
Registration	
<code>tibrvcmlTransport_AddListener()</code>	Pre-register an anticipated listener.
<code>tibrvcmlTransport_AllowListener()</code>	Invite the named receiver to reinstate certified delivery for its listeners, superseding the effect of any previous disallow calls.
<code>tibrvcmlTransport_DisallowListener()</code>	Cancel certified delivery to all

Function	Description
	listeners at a specific correspondent. Deny subsequent certified delivery registration requests from those listeners.
<code>tibrvcTransport_RemoveListener()</code>	Unregister a specific listener at a specific correspondent, and free associated storage in the sender's ledger.
Ledger	
<code>tibrvcTransport_RemoveSendState()</code>	Reclaim ledger space from obsolete subjects.
<code>tibrvcTransport_ReviewLedger()</code>	Query the ledger for stored items related to a subject name.
<code>tibrvcReviewCallback</code>	Programs define functions of this type to process ledger review summary messages.
<code>tibrvcTransport_SyncLedger()</code>	Synchronize the ledger to its storage medium.
Listener Attributes	
<code>tibrvcEvent_GetListenerSubject()</code>	Extract the subject from a certified delivery listener event object.
<code>tibrvcEvent_GetListenerTransport()</code>	Extract the certified delivery transport from a certified delivery listener event object.

Function	Description
tibrvcmEvent_GetQueue()	Extract the queue of a certified delivery listener object.
tibrvcmEvent_SetExplicitConfirm()	Override automatic confirmation of delivery for this listener.
Transport Life Cycles	
tibrvcmTransport_Create()	Create a transport for certified delivery.
tibrvcmTransport_Destroy()	Destroy a certified delivery transport.
tibrvcmTransportOnComplete	Destroying a CM or CMQ transport involves cleanup operations that can sometimes be lengthy. Programs can define functions of this type to discover when the cleanup has completed.
Transport Attributes	
tibrvcmTransport_GetLedgerName()	Extract the ledger name of a certified delivery transport.
tibrvcmTransport_GetName()	Extract the correspondent name of a certified delivery transport.
tibrvcmTransport_GetRequestOld()	Extract the request old messages flag of a certified delivery transport.
tibrvcmTransport_GetSyncLedger()	Extract the sync ledger flag of

Function	Description
	a certified delivery transport.
<code>tibrvcTransport_GetTransport()</code>	Extract the transport employed by a certified delivery transport.
<code>tibrvcTransport_SetDefaultCMTimeLimit()</code>	Set the default message time limit for all outbound certified messages from a transport.
Message Attributes	
<code>tibrvMsg_GetCMSender()</code>	Extract the correspondent name of the sender from a certified message.
<code>tibrvMsg_GetCMSequence()</code>	Extract the sequence number from a certified message.
<code>tibrvMsg_GetCMTimeLimit()</code>	Extract the message time limit from a certified message.
<code>tibrvMsg_SetCMTimeLimit()</code>	Set the message time limit of a certified message.
<code>tibrvcTransport_SetPublisherInactivityDiscardInterval()</code>	Set a time limit after which a listening CM transport can discard state for inactive CM senders.
Types Related to Certified Delivery	
<code>tibrvcEvent</code>	A certified delivery event object listens for labeled messages and certified messages.

Function	Description
tibrvcmTransport	A certified delivery transport object implements the CM delivery protocol for messages.
tibrvcmEventCallback	Programs define functions of this type to process inbound certified or labeled messages.
tibrvcmReviewCallback	Programs define functions of this type to process ledger review summary messages.
Library Version	
tibrvcm_Version()	Identify the CM API release number.

Operations in Alphabetical Order

Function	Description
tibrvcm_Version()	Identify the CM API release number.
Events (Certified Delivery Listeners)	
tibrvcmEvent	A certified delivery event object listens for labeled messages and certified messages.
tibrvcmEventCallback	Programs define functions of this type to process inbound certified or labeled messages.
tibrvcmEvent_ConfirmMsg()	Explicitly confirm delivery of a certified message.
tibrvcmEvent_CreateListener()	Listen for messages that match the subject, and request certified delivery when available.
tibrvcmEvent_Destroy()	Destroy a certified delivery listener.
tibrvcmEvent_GetListenerSubject()	Extract the subject from a certified delivery listener event object.
tibrvcmEvent_GetListenerTransport()	Extract the certified delivery transport from a certified delivery listener event object.

Function	Description
tibrvcmEvent_GetQueue()	Extract the queue of a certified delivery listener object.
tibrvcmEvent_SetExplicitConfirm()	Override automatic confirmation of delivery for this listener.
Certified Delivery Transports	
tibrvcmTransport	A certified delivery transport object implements the CM delivery protocol for messages.
tibrvcmTransportOnComplete	Destroying a CM or CMQ transport involves cleanup operations that can sometimes be lengthy. Programs can define functions of this type to discover when the cleanup has completed.
tibrvcmTransport_AddListener()	Pre-register an anticipated listener.
tibrvcmTransport_AllowListener()	Invite the named receiver to reinstate certified delivery for its listeners, superseding the effect of any previous disallow calls.
tibrvcmTransport_Create()	Create a transport for certified delivery.
tibrvcmTransport_Destroy()	Destroy a certified delivery transport.

Function	Description
tibrvcmTransport_DisallowListener()	Cancel certified delivery to all listeners at a specific correspondent. Deny subsequent certified delivery registration requests from those listeners.
tibrvcmTransport_ExpireMessages()	Mark specified outbound CM messages as expired.
tibrvcmTransport_GetLedgerName()	Extract the ledger name of a certified delivery transport.
tibrvcmTransport_GetName()	Extract the correspondent name of a certified delivery transport.
tibrvcmTransport_GetRequestOld()	Extract the request old messages flag of a certified delivery transport.
tibrvcmTransport_GetSyncLedger()	Extract the sync ledger flag of a certified delivery transport.
tibrvcmTransport_GetTransport()	Extract the transport employed by a certified delivery transport.
tibrvcmTransport_RemoveSendState()	Reclaim ledger space from obsolete subjects.
tibrvcmTransport_ReviewLedger()	Query the ledger for stored items related to a subject name.
tibrvcmReviewCallback	Programs define functions of this type to process ledger

Function	Description
	review summary messages.
<code>tibrvcMTransport_Send()</code>	Send a labeled message.
<code>tibrvcMTransport_SendReply()</code>	Send a labeled reply message.
<code>tibrvcMTransport_SendRequest()</code>	Send a labeled request message and wait for a reply.
<code>tibrvcMTransport_SetDefaultCMTimeLimit()</code>	Set the default message time limit for all outbound certified messages from a transport.
<code>tibrvcMTransport_SetPublisherInactivityDiscardInterval()</code>	Set a time limit after which a listening CM transport can discard state for inactive CM senders.
<code>tibrvcMTransport_SyncLedger()</code>	Synchronize the ledger to its storage medium.
Certified Messages	
<code>tibrvMsg_GetCMSender()</code>	Extract the correspondent name of the sender from a certified message.
<code>tibrvMsg_GetCMSequence()</code>	Extract the sequence number from a certified message.
<code>tibrvMsg_GetCMTimeLimit()</code>	Extract the message time limit from a certified message.
<code>tibrvMsg_SetCMTimeLimit()</code>	Set the message time limit of a certified message.

tibrvcm_Version()

Function

Declaration

```
const char* tibrvcm_Version(void);
```

Purpose

Identify the CM API release number.

tibrvcmlEvent

Type

Declaration

```
typedef tibrvId tibrvcmlEvent;
```

Purpose

A certified delivery event object listens for labeled messages and certified messages.

Remarks

Each call to the event creation function [tibrvcmlEvent_CreateListener\(\)](#) results in a new certified delivery event object, which represents your program's listening interest in a stream of labeled messages and certified messages. Rendezvous software uses the same event object to signal each occurrence of such an event.

Programs use the event object as a token; they cannot view or modify the object, except using accessor functions.

Programs must explicitly destroy each certified delivery event object using [tibrvcmlEvent_Destroy\(\)](#). Destroying a certified listener object cancels the program's immediate interest in that event, and frees its storage; nonetheless, a parameter to the destroy call determines whether certified delivery agreements continue to persist beyond the destroy call.

See Also

[tibrvcmlEvent_CreateListener\(\)](#)

[tibrvcmlEvent_Destroy\(\)](#)

[tibrvcmlEvent_GetListenerSubject\(\)](#)

[tibrvcmlEvent_GetListenerTransport\(\)](#)

[tibrvcmlEvent_GetQueue\(\)](#)

[tibrvcnEvent_SetExplicitConfirm\(\)](#)

[tibrvcnEvent_ConfirmMsg\(\)](#)

tibrvcmlEventCallback

Function Type

Declaration

```
typedef void (*tibrvcmlEventCallback) (  
    tibrvcmlEvent      cmListener,  
    tibrvcmlMsg        message,  
    void*              closure);
```

Purpose

Programs define functions of this type to process inbound certified or labeled messages.

Remarks

This callback function must also be able to process uncertified and unlabeled messages.

Parameter	Description
cmListener	This parameter receives the listener object. This object is identical to the certified listener object that the program created to begin listening.
message	The callback receives the inbound message in this parameter. If the inbound message is empty, this parameter receives a message that has no fields.
closure	This parameter receives the closure data, which the program supplied in the call that created the event object.

CM Label Information

CM callback functions can use CM label information to discriminate these situations:

- If `tibrvMsg_GetCMSender()` returns status code `TIBRV_NOT_FOUND`, then the message uses the reliable protocol (that is, it was sent from an ordinary transport).
- If `tibrvMsg_GetCMSender()` returns `TIBRV_OK`, then the message uses the certified delivery protocol (that is, it is a labeled message, sent from a CM transport).

Once the CM callback function determines that the message uses the certified delivery protocol, it can discriminate further:

- If `tibrvMsg_GetCMSequence()` returns status code `TIBRV_NOT_FOUND`, then the listener is *not registered* for certified delivery from the sender.
- If `tibrvMsg_GetCMSequence()` returns `TIBRV_OK`, then a certified delivery agreement is in effect for this subject with the sender.

See Also

`tibrvcmlEvent_CreateListener()`

`tibrvMsg_GetCMSender()`

`tibrvMsg_GetCMSequence()`

`tibrvMsg_GetCMTimeLimit()`

tibrvcmlEvent_ConfirmMsg()

Function

Declaration

```
tibrvcml_status tibrvcmlEvent_ConfirmMsg(  
    tibrvcmlEvent      cmListener,  
    tibrvcmlMsg        message);
```

Purpose

Explicitly confirm delivery of a certified message.

Remarks

Use this function only in programs that override automatic confirmation (see [tibrvcmlEvent_SetExplicitConfirm\(\)](#)). The default behavior of certified listeners is to automatically confirm delivery when the callback function returns.

Parameter	Description
cmListener	Confirm receipt by this listener.
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 function. 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 function, but omits the sequence number. In this

case, the receiving program cannot confirm delivery; `tibrvcmlEvent_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

[tibrvcmlEvent](#)

[tibrvcmlEvent_SetExplicitConfirm\(\)](#)

tibrvcmlEvent_CreateListener()

Function

Declaration

```
tibrvcml_status tibrvcmlEvent_CreateListener(  
    tibrvcmlEvent*      cmListener,  
    tibrvcmlQueue       queue,  
    tibrvcmlEventCallback callback,  
    tibrvcmlTransport   cmTransport,  
    const char*         subject,  
    const void*         closure);
```

Purpose

Listen for messages that match the subject, and request certified delivery when available.

Parameter	Description
cmListener	<p>For each inbound message, place this listener event on the event queue.</p> <p>The program supplies a location, and the function stores the new listener object in that location.</p> <p>The event object remains valid until the program explicitly destroys it.</p>
queue	<p>For each inbound message, place the listener event on this event queue.</p>
callback	<p>On dispatch, process the event with this callback function.</p>
cmTransport	<p>Listen for inbound messages on this certified delivery</p>

Parameter	Description
	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 [tibrvcmlEvent_CreateListener\(\)](#) 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

[tibrvcmlEvent](#)

[tibrvcmlTransport_Destroy\(\)](#)

[tibrvEvent_GetListenerSubject\(\)](#)

[tibrvTransport_CreateInbox\(\)](#)

tibrvcmlEvent_Destroy()

Function

Declaration

```
tibrvcml_status tibrvcmlEvent_Destroy(  
    tibrvcmlEvent      cmListener,  
    tibrvcml_bool      cancelAgreements);  
tibrvcml_status tibrvcmlEvent_DestroyEx(  
    tibrvcmlEvent      cmListener,  
    tibrvcml_bool      cancelAgreements,  
    tibrvcmlEventOnComplete completionFunction);
```

Purpose

Destroy a certified delivery listener.

Parameter	Description
cmListener	Destroy this listener.
cancelAgreements	<p>TIBRVCM_CANCEL cancels all certified delivery agreements of this listener; certified senders delete from their ledgers all messages sent to this listener.</p> <p>TIBRVCM_PERSIST leaves all certified delivery agreements in effect, so certified senders continue to store messages.</p>
completionFunction	Rendezvous software runs this function immediately after all instances of the event's callback function have completed. If the event's callback function is not running when the event is destroyed, the destroy call runs it before

Parameter	Description
	returning. If this parameter is <code>NULL</code> , <code>tibrvcmlEvent_DestroyEx()</code> does not run a completion function; instead, its behavior is the same as tibrvcmlEvent_Destroy() . See tibrvEventOnComplete .

Remarks

Destroying event interest invalidates the event object; passing the event object as an argument to subsequent API calls produces an error.

Canceling Agreements

When destroying a 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.

Extended Function

Although [tibrvcmlEvent_Destroy\(\)](#) and `tibrvcmlEvent_DestroyEx()` prevent future dispatch calls from running the destroyed event's callback function, that callback function might be already running in one or more threads that dispatch events from the same queue. In each thread where the callback function is already in progress, that callback function does continue to run until complete.

`tibrvcmlEvent_DestroyEx()` ensures that its `completionFunction` runs when the last callback-in-progress has completed; for important details, see [tibrvEventOnComplete](#).

See Also

[tibrvcmlEvent](#)

`tibrvcmlEvent_GetListenerSubject()`

Function

Declaration

```
tibrvcml_status tibrvcmlEvent_GetListenerSubject(  
    tibrvcmlEvent cmListener,  
    const char**  subject);
```

Purpose

Extract the subject from a certified delivery listener event object.

Parameter	Description
<code>cmListener</code>	Extract the subject from this listener.
<code>subject</code>	The program supplies a location. The function stores the subject of the listener event object in that location.

See Also

[`tibrvcmlEvent`](#)

`tibrvcmlEvent_GetListenerTransport()`

Function

Declaration

```
tibrvcml_status tibrvcmlEvent_GetListenerTransport(  
    tibrvcmlEvent      cmListener,  
    tibrvcmlTransport* transport);
```

Purpose

Extract the certified delivery transport from a certified delivery listener event object.

Parameter	Description
<code>cmListener</code>	Extract the transport from this listener.
<code>transport</code>	The program supplies a location. The function stores the certified delivery transport of the listener event object in that location.

See Also

[`tibrvcmlEvent`](#)

[`tibrvcmlTransport`](#)

tibrvcmlEvent_GetQueue()

Function

Declaration

```
tibrvcml_status tibrvcmlEvent_GetQueue(  
    tibrvcmlEvent    cmListener,  
    tibrvcmlQueue*   queue);
```

Purpose

Extract the queue of a certified delivery listener object.

Parameter	Description
cmListener	Extract the event queue from this event object.
queue	The program supplies a location. The function stores the event object's queue in that location.

See Also

[tibrvcmlEvent](#)

[tibrvcmlQueue](#)

`tibrvcmEvent_SetExplicitConfirm()`

Function

Declaration

```
tibrv_status tibrvcmEvent_SetExplicitConfirm(  
    tibrvcmEvent cmListener)
```

Purpose

Override automatic confirmation of delivery for this listener.

Remarks

The default behavior of certified listeners is to automatically confirm delivery when the callback function returns (see [tibrvcmEventCallback](#)). 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 [tibrvcmEvent_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, reversing the effect of this function.

Parameter	Description
<code>cmListener</code>	Override automatic confirmation for this listener.

See Also

[tibrvcmEvent](#)

379 | `tibrvcmlEvent_SetExplicitConfirm()`

`tibrvcmlEventCallback`

`tibrvcmlEvent_ConfirmMsg()`

tibrvcMTransport

Type

Declaration

```
typedef tibrvId tibrvcMTransport;
```

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 the [tibrvTransport](#), which also remains available for its own ordinary listeners and for sending ordinary messages.

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), and frees the transport's storage.

See Also

[tibrvcMTransport_Create\(\)](#)

[tibrvcMTransport_Destroy\(\)](#)

[tibrvcMTransport_Send\(\)](#)

tibrvcTransportOnComplete

Function Type

Declaration

```
typedef void (*tibrvcTransportOnComplete) (  
    tibrvcTransport    destroyedTransport,  
    void*              closure);
```

Purpose

Destroying a CM or CMQ transport involves cleanup operations that can sometimes be lengthy. Programs can define functions of this type to discover when the cleanup has completed.

Remarks

After calling the extended destroy function, 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	This parameter receives the transport object. However, by the time this function runs, the transport is already destroyed; this function cannot use the transport object in Rendezvous calls.
closure	This parameter receives the closure data, which the program supplied to <code>tibrvcTransport_DestroyEx()</code> .

See Also

[tibrvcTransport_Destroy\(\)](#)

`tibrvcMTransport_AddListener()`

Function

Declaration

```
tibrv_status tibrvcMTransport_AddListener(  
    tibrvcMTransport cmTransport,  
    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 sender 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 code `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 function to anticipate listeners; the anticipated listening programs must also require old messages when creating certified delivery transports.

Parameter	Description
<code>cmTransport</code>	Instruct this transport to pre-register the named listeners.
<code>cmName</code>	Anticipate a listener from a correspondent with this reusable name.
<code>subject</code>	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()`

Function

Declaration

```
tibrv_status tibrvcMTransport_AllowListener(  
    tibrvcMTransport cmTransport,  
    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
<code>cmTransport</code>	Instruct this transport to allow the named listeners.
<code>cmName</code>	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

tibrvcTransport_Create()

Function

Declaration

```
tibrv_status tibrvcTransport_Create(  
    tibrvcTransport* cmTransport,  
    tibrvTransport    transport,  
    const char*       cmName,  
    tibrv_bool        requestOld,  
    const char*       ledgerName,  
    tibrv_bool        syncLedge);
```

Purpose

Create a transport for certified delivery.

Remarks

The new certified delivery transport must employ a valid transport for network communications.

Parameter	Description
cmTransport	<p>The program supplies a location, and the function stores the new certified delivery transport in that location.</p> <p>The certified delivery transport remains valid until the program explicitly destroys it.</p>
transport	<p>The new cmTransport employs this transport object for network communications.</p> <p>Destroying the cmTransport does not affect this</p>

Parameter	Description
	<p>tibrvTransport.</p> <p>It is illegal to supply a virtual circuit transport object for this parameter.</p>
<code>cmName</code>	<p>Bind this reusable name to the new <code>cmTransport</code>, so the <code>cmTransport</code> 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 tibrvcMTransport_Create(). It cannot be the empty string.</p> <p>If this argument is NULL, then tibrvcMTransport_Create() generates a unique, non-reusable name for the duration of the transport.</p> <p>For more information, see Name.</p>
<code>requestOld</code>	<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> <i>and</i> <code>cmName</code> is non-NULL, then the new <code>cmTransport</code> requires certified senders to retain unacknowledged messages sent to this persistent correspondent. When the new <code>cmTransport</code> begins listening to the appropriate subjects, the senders can complete delivery. (It is an error to supply <code>TIBRV_TRUE</code> when <code>cmName</code> is NULL.)</p> <p>If this parameter is <code>TIBRV_FALSE</code>, then the new <code>cmTransport</code> does not require certified senders to retain unacknowledged messages. Certified senders may delete those messages from their ledgers.</p>

Parameter	Description
<code>ledgerName</code>	<p>If this argument is non-NULL, then the new <code>cmTransport</code> 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 this argument is NULL, then the new <code>cmTransport</code> uses a process-based ledger.</p> <p>For more information, see Ledger File.</p>
<code>syncLedger</code>	<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>, the operating system writes changes to the storage medium asynchronously.</p>

Name

If `name` is NULL, then [tibrvcMTransport_Create\(\)](#) generates a unique, non-reusable name for the new certified delivery transport.

If `name` 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, supply `TIBRV_FALSE`. CM 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 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 [tibrvcTransport_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

[tibrvcTransport_Destroy\(\)](#)

tibrvcMTransport_Destroy()

Function

Declaration

```
tibrv_status tibrvcMTransport_Destroy(  
    tibrvcMTransport cmTransport);  
tibrv_status tibrvcMTransport_DestroyEx(  
    tibrvcMTransport cmTransport,  
    tibrvcMTransportOnComplete completionFunction,  
    void* closure);
```

Purpose

Destroy a certified delivery transport.

Parameter	Description
cmTransport	Destroy this transport.
completionFunction	<p>Rendezvous software runs this function 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 tibrvcMTransportOnComplete.</p>
closure	Pass this closure data to the completion function.

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.

Distributed Queue

To destroy a distributed queue transport, call `tibrvcTransport_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

[tibrvcTransportOnComplete](#)

[tibrvcTransport_Create\(\)](#)

[tibrvcTransport_CreateDistributedQueue\(\)](#)

`tibrvcMTransport_DisallowListener()`

Function

Declaration

```
tibrvcM_status tibrvcMTransport_DisallowListener(  
    tibrvcMTransport cmTransport,  
    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 function that processes the `REGISTRATION.REQUEST` advisory.

This function 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 `tibrvcmtTransport_DisallowListener()`, call `tibrvcmtTransport_AllowListener()`.

Parameter	Description
<code>cmTransport</code>	Instruct this transport to disallow the named listeners.
<code>cmName</code>	Cancel certified delivery to listeners of the transport with this name.

See Also

[Name](#)

[tibrvcmtTransport_AllowListener\(\)](#)

Disallowing Certified Delivery in TIBCO Rendezvous Concepts

tibrvcMTransport_ExpireMessages()

Function

Declaration

```
tibrv_status tibrvcMTransport_ExpireMessages(  
    tibrvcMTransport    cmTransport,  
    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
cmTransport	Mark messages as expired in the ledger for this CM transport.
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 in TIBCO Rendezvous Concepts

`tibrvcMTransport_GetDefaultCMTimeLimit()`

Function

Declaration

```
tibrv_status tibrvcMTransport_GetDefaultCMTimeLimit(  
    tibrvcMTransport cmTransport,  
    tibrv_f64*       timeLimit);
```

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 [tibrvMsg_SetCMTimeLimit\(\)](#)). 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 function); 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
<code>cmTransport</code>	Set the default message time limit of this transport.
<code>timeLimit</code>	The program supplies a location, and the function stores the default time limit (in whole seconds) in that location.

See Also

[tibrvcMTransport_SetDefaultCMTimeLimit\(\)](#)

`tibrvMsg_SetCMTimeLimit()`

tibrvcmTransport_GetLedgerName()

Function

Declaration

```
tibrv_status tibrvcmTransport_GetLedgerName(  
    tibrvcmTransport cmTransport,  
    const char**      ledgerName);
```

Purpose

Extract the ledger name of a certified delivery transport.

Parameter	Description
cmTransport	Extract the ledger name from this certified delivery transport object.
ledgerName	The program supplies a location, and the function stores the ledger name in that location.

Errors

The error code [TIBRV_ARG_CONFLICT](#) can indicate that the transport does not have a ledger file.

See Also

[Ledger File](#)

[tibrvcmTransport_Create\(\)](#)

`tibrvcTransport_GetName()`

Function

Declaration

```
tibrv_status tibrvcTransport_GetName(  
    tibrvcTransport cmTransport,  
    const char**    cmName);
```

Purpose

Extract the correspondent name of a certified delivery transport.

Parameter	Description
<code>cmTransport</code>	Extract the name from this certified delivery transport object.
<code>cmName</code>	The program supplies a location, and the function stores the correspondent name in that location.

See Also

[Name](#)

[tibrvcTransport_Create\(\)](#)

`tibrvcMTransport_GetRequestOld()`

Function

Declaration

```
tibrvc_status tibrvcMTransport_GetRequestOld(  
    tibrvcMTransport cmTransport,  
    tibrvc_bool*      requestOld);
```

Purpose

Extract the request old messages flag of a certified delivery transport.

Parameter	Description
<code>cmTransport</code>	Extract the request old messages flag from this certified delivery transport object.
<code>requestOld</code>	The program supplies a location, and the function stores the request old messages flag in that location.

See Also

[requestOld](#)

[tibrvcMTransport_Create\(\)](#)

tibrvcmTransport_GetSyncLedger()

Function

Declaration

```
tibrv_status tibrvcmTransport_GetSyncLedger(  
    tibrvcmTransport cmTransport,  
    tibrv_bool*      syncLedger);
```

Purpose

Extract the sync ledger flag of a certified delivery transport.

Parameter	Description
cmTransport	Extract the sync ledger flag from this certified delivery transport object.
syncLedger	The program supplies a location, and the function stores the sync ledger flag in that location.

Errors

The error code [TIBRV_ARG_CONFLICT](#) can indicate that the transport does not have a ledger file.

See Also

[Ledger File](#)

[tibrvcmTransport_Create\(\)](#)

`tibrvcTransport_GetTransport()`

Function

Declaration

```
tibrv_status tibrvcTransport_GetTransport(  
    tibrvcTransport cmTransport,  
    tibrvTransport* transport);
```

Purpose

Extract the transport employed by a certified delivery transport.

Parameter	Description
<code>cmTransport</code>	Extract the transport from this certified delivery transport object.
<code>transport</code>	The program supplies a location, and the function stores the transport in that location.

See Also

[`tibrvcTransport_Create\(\)`](#)

`tibrvcMTransport_RemoveListener()`

Function

Declaration

```
tibrv_status tibrvcMTransport_RemoveListener(  
    tibrvcMTransport cmTransport,  
    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 function 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 function 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 function, 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 function 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 `cmTransport`, then `tibrvcMTransport_RemoveListener()` returns the status code `TIBRV_INVALID_SUBJECT`.

Parameter	Description
<code>cmTransport</code>	Instruct this transport to unregister the specified listeners.
<code>cmName</code>	Cancel certified delivery of the subject to listeners of this correspondent.
<code>subject</code>	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

tibrvcTransport_RemoveSendState()

Function

Declaration

```
tibrv_status tibrvcTransport_RemoveSendState(  
    tibrvcTransport cmTransport,  
    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 function 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.

Remarks



Warning

Do not use this function to clear subjects that are still in use.

As a side-effect, this function 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.

Parameter	Description
<code>cmTransport</code>	Remove send state from the ledger of this certified delivery transport.
<code>subject</code>	Remove send state for this obsolete subject.

See Also

[tibrvcTransport_ReviewLedger\(\)](#)

[tibrvcTransport_Send\(\)](#)

DELIVERY.COMPLETE in TIBCO Rendezvous Concepts

`tibrvcMTransport_ReviewLedger()`

Function

Declaration

```
tibrv_status tibrvcMTransport_ReviewLedger(  
    tibrvcMTransport    cmTransport,  
    tibrvcMReviewCallback callback,  
    const char*         subject,  
    const void*         closure);
```

Purpose

Query the ledger for stored items related to a subject name.

Remarks

The callback function receives one message for each matching subject of outbound messages stored in the ledger. For example, when `F00.*` is the subject, `tibrvcMTransport_ReviewLedger()` calls its callback function separately for each matching subject—once for `F00.BAR`, once for `F00.BAZ`, and once for `F00.BOX`.

However, if the callback function 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 function.

For information about the content and format of the callback messages, see [tibrvcMReviewCallback](#).

Parameter	Description
<code>cmTransport</code>	Review the ledger of this transport.

Parameter	Description
callback	This function receives the review messages.
subject	Query for items related to this subject name. If this subject contains wildcard characters ("*" or ">"), then review all items with matching subject names. The callback function receives a separate message for each matching subject in the ledger.
closure	Pass this closure data to the callback function.

See Also

[tibrvcReviewCallback](#)

tibrvcmReviewCallback

Function Type

Declaration

```
typedef void* (*tibrvcmReviewCallback) (  
    tibrvcmTransport    cmTransport,  
    const char*         subject,  
    tibrvMsg            message,  
    void*               closure);
```

Purpose

Programs define functions of this type to process ledger review summary messages.

Remarks

[tibrvcmTransport_ReviewLedger\(\)](#) calls this callback function once for each matching subject stored in the ledger.

To continue reviewing the ledger, return NULL from this callback function. To stop reviewing the ledger, return non-NULL from this callback function; [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.
message	This parameter receives a summary message describing the delivery status of messages in the ledger. The table in the Message Fields section describes the fields of the summary message.

Parameter	Description
closure	This parameter receives closure data which the program supplied to <code>tibrvcmlTransport_ReviewLedger()</code> .

Message Fields

This callback function receives ledger summary messages with these fields.

Field Name	Description
subject	The subject that this message summarizes. This field has datatype <code>TIBRVMSG_STRING</code> .
seqno_last_sent	The sequence number of the most recent message sent with this subject name. This field has datatype <code>TIBRVMSG_U64</code> .
total_msgs	The total number of messages stored at this subject name. This field has datatype <code>TIBRVMSG_U32</code> .
total_size	The total storage (in bytes) occupied by all messages with this subject name. If the ledger contains several messages with this subject name, then this field sums the storage space over all of them. This field has datatype <code>TIBRVMSG_U64</code> .
listener	Each summary message can contain one or more fields named <code>listener</code> . Each <code>listener</code> field contains a nested submessage with details about a single registered listener. This field has datatype <code>TIBRVMSG_MSG</code> .
listener.name	Within each <code>listener</code> submessage, the <code>name</code> field contains the name of the listener transport.

Field Name	Description
	This field has datatype TIBRVMSG_STRING .
<code>listener.last_confirmed</code>	Within each listener submessage, the <code>last_confirmed</code> field contains the sequence number of the last message for which the listener confirmed delivery. This field has datatype TIBRVMSG_U64 .

See Also

[tibrvcmlTransport_ReviewLedger\(\)](#)

tibrvcMTransport_Send()

Function

Declaration

```
tibrv_status tibrvcMTransport_Send(  
    tibrvcMTransport cmTransport,  
    tibrvMsg message);
```

Purpose

Send a labeled message.

Remarks

This function 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 [tibrvMsg_SetCMTimeLimit\(\)](#). If a time limit is not already set for the outbound message, this function sets it to the transport's default time limit (see [tibrvcMTransport_SetDefaultCMTimeLimit\(\)](#)); if that default is not set for the transport, the default time limit is zero (no time limit).

Parameter	Description
<code>cmTransport</code>	Send a message using this certified delivery transport.
<code>message</code>	Send this message. Wildcard subjects are illegal.

See Also

[tibrvcTransport_SendReply\(\)](#)

[tibrvcTransport_SendRequest\(\)](#)

[tibrvcTransport_SetDefaultCMTimeLimit\(\)](#)

[tibrvMsg_SetCMTimeLimit\(\)](#)

tibrvcMTransport_SendReply()

Function

Declaration

```
tibrv_status tibrvcMTransport_SendReply(  
    tibrvcMTransport    cmTransport,  
    tibrvMsg            replyMessage,  
    tibrvMsg            requestMessage);
```

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 function can send a labeled reply to an ordinary message.

This function automatically registers the requesting CM transport, so the reply message is certified.



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).

Parameter	Description
cmTransport	Send a message using this certified delivery

Parameter	Description
	transport.
replyMessage	Send this <i>outbound</i> reply message.
requestMessage	<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 function produces an error.</p>

See Also

[tibrvcTransport_Send\(\)](#)

[tibrvcTransport_SendRequest\(\)](#)

tibrvcMTransport_SendRequest()


Function

Declaration

```
tibrv_status
tibrvcMTransport_SendRequest(
    tibrvcMTransport    cmTransport,
    tibrvMsg             message);
tibrvMsg*              reply,
tibrv_f64               timeout);
```

Purpose

Send a labeled request message and wait for a reply.

**Warning**

Blocking can Stall Event Dispatch

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
cmTransport	Send a message using this certified delivery transport.
message	Send this request message. Wildcard subjects are illegal.
reply	The program supplies a location, and the function stores the inbound reply in that location. The program need not create the reply message, nor

Parameter	Description
	allocate space for it. However, the program <i>must destroy</i> the reply message, even though it did not create it.
<code>timeout</code>	Maximum time (in seconds) that this call can block while waiting for a reply. <code>TIBRV_WAIT_FOREVER</code> (-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 function operates in several synchronous steps:

Procedure

1. Create a [tibrvcEvent](#) that listens for messages on the reply subject of message.
2. Label and send the outbound message.
3. Block until the listener receives a reply; if the time limit expires before a reply arrives, 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 location specified by the `reply` parameter.
5. Return.

See Also

[tibrvcTransport_Send\(\)](#)

[tibrvcTransport_SendReply\(\)](#)

`tibrvcMTransport_SetDefaultCMTimeLimit()`

Function

Declaration

```
tibrv_status tibrvcMTransport_SetDefaultCMTimeLimit(  
    tibrvcMTransport cmTransport,  
    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 [tibrvMsg_SetCMTimeLimit\(\)](#)). If a time limit is not already set for the outbound message, this function sets it to the transport's default time limit (set with this function); 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
<code>cmTransport</code>	Set the default message time limit of this transport.
<code>timeLimit</code>	Use this time limit (in whole seconds). The time limit must be non-negative.

See Also

[tibrvcMTransport_GetDefaultCMTimeLimit\(\)](#)

420 | `tibrvcTransport_SetDefaultCMTimeLimit()`

`tibrvMsg_SetCMTimeLimit()`

tibrvcmTransport_SetPublisherInactivityDiscardInterval()

Function

Declaration

```
tibrv_status tibrvcmTransport_SetPublisherInactivityDiscardInterval(  
    tibrvcmTransport    cmTransport,  
    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
<code>cmTransport</code>	Set the inactivity discard interval of this transport.
<code>timeout</code>	Use this time limit (in whole seconds). The time limit must be non-negative.

tibrvcTransport_SyncLedger()

Function

Declaration

```
tibrv_status tibrvcTransport_SyncLedger(  
    tibrvcTransport cmTransport);
```

Purpose

Synchronize the ledger to its storage medium.

Remarks

When this function returns, the transport's current state is safely stored in the ledger file.

Transports that use synchronous ledger files need not call this function, since the current state is automatically written to the file system before returning. Transports that use process-based ledger storage need not call this function, since they have no ledger file.

Parameter	Description
cmTransport	Synchronize the ledger file of this certified delivery transport object.

Errors

The error code [TIBRV_INVALID_ARG](#) can indicate that the transport does not have a ledger file.

See Also

[Ledger File](#)

424 | `tibrvcTransport_SyncLedger()`

`tibrvcTransport_Create()`

`tibrvcTransport_GetSyncLedger()`

tibrvMsg_GetCMSender()

Function

Declaration

```
tibrv_status tibrvMsg_GetCMSender(  
    tibrvMsg      message,  
    const char**  name);
```

Purpose

Extract the correspondent name of the sender from a certified message.

Parameter	Description
message	Extract the sender name from this message.
name	The program supplies a location. The function stores the name in that location.

Status

This function returns a status code that discriminates between labeled messages and other messages.

- If the message is from a CM sender, then `tibrvMsg_GetCMSender()` returns the status code `TIBRV_OK` and yields a valid CM correspondent name.
- If the message is *not* from a CM sender, then `tibrvMsg_GetCMSender()` returns the status code `TIBRV_NOT_FOUND`.

See Also

[tibrvcmTransport_Create\(\)](#)

426 | `tibrvMsg_GetCMSender()`

[`tibrvcmTransport_GetName\(\)`](#)

tibrvMsg_GetCMSequence()

Function

Declaration

```
tibrv_status tibrvMsg_GetCMSequence(  
    tibrvMsg      message,  
    tibrv_u64*    sequenceNumber);
```

Purpose

Extract the sequence number from a certified message.

Remarks

Rendezvous certified delivery sending functions automatically generate positive sequence numbers for outbound labeled messages.

Parameter	Description
message	Extract the sequence number from this message.
sequenceNumber	The program supplies a location. The function stores the sequence number in that location.

Status

This function 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 `tibrvMsg_GetCMSequence()` 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 `tibrvMsg_GetCMSequence()` *in the context of a `tibrvcmEventCallback` function* 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 `tibrvMsg_GetCMSequence()` (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\(\)](#)

tibrvMsg_GetCMTimeLimit()

Function

Declaration

```
tibrv_status tibrvMsg_GetCMTimeLimit(  
    tibrvMsg      message,  
    tibrv_f64*    timeLimit);
```

Purpose

Extract the message time limit from a certified message.

Remarks

Programs can explicitly set the message time limit (see [tibrvMsg_SetCMTimeLimit\(\)](#)).

Zero is a special value, indicating no time limit.

If a time limit is not set for a message, this function 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
message	Extract the time limit from this message.
timeLimit	The program supplies a location. The function stores the time limit in that location.

See Also

[tibrvcmTransport_Send\(\)](#)

[tibrvMsg_SetCMTimeLimit\(\)](#)

tibrvMsg_SetCMTimeLimit()

Function

Declaration

```
tibrv_status tibrvMsg_SetCMTimeLimit(  
    tibrvMsg      message,  
    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 function. 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_SetDefaultCMTimeLimit\(\)](#)); 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 function.

Parameter	Description
message	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_GetDefaultCMTimeLimit\(\)](#)

[tibrvcmTransport_SetDefaultCMTimeLimit\(\)](#)

[tibrvMsg_GetCMTimeLimit\(\)](#)

Distributed Queues

Programs can use distributed queues for *one of n* certified delivery to a group of worker processes.

Operations in Alphabetical Order

Function	Description
tibrvcmTransport_CreateDistributedQueue()	Create a distributed queue member.
tibrvcmTransport_GetCompleteTime()	Extract the worker complete time limit of a distributed queue transport.
tibrvcmTransport_GetUnassignedMessageCount()	Extract the number of unassigned task messages from a distributed queue transport.
tibrvcmTransport_GetWorkerWeight()	Extract the worker weight of a distributed queue transport.
tibrvcmTransport_GetWorkerTasks()	Extract the worker task capacity of a distributed queue transport.
tibrvcmTransport_SetCompleteTime()	Set the worker complete time limit of a distributed queue transport.
tibrvcmTransport_SetTaskBacklogLimit...()	Set the scheduler task queue limits of a distributed queue transport.
tibrvcmTransport_SetWorkerWeight()	Set the worker weight of a distributed queue transport.
tibrvcmTransport_SetWorkerTasks()	Set the worker task capacity of a distributed queue transport.

Distributed Queue Overview

A distributed queue is a group of cooperating transport 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 Queues in TIBCO Rendezvous Administration

`tibrvcTransport_CreateDistributedQueue()`

Function

Declaration

```
tibrv_status tibrvcTransport_CreateDistributedQueue(  
    tibrvcTransport*    cmTransport,  
    tibrvTransport      transport,  
    const char*         cmName);  
tibrv_status tibrvcTransport_CreateDistributedQueueEx(  
    tibrvcTransport*    cmTransport,  
    tibrvTransport      transport,  
    const char*         cmName,  
    tibrv_u32           workerWeight,  
    tibrv_u32           workerTasks,  
    tibrv_u16           schedulerWeight,  
    tibrv_f64           schedulerHeartbeat,  
    tibrv_f64           schedulerActivation);
```

Purpose

Create a distributed queue member.


Remarks

The new distributed queue transport must employ a valid transport for network communications.

All members of a distributed queue must listen to exactly the same set of subjects. See [Enforcing Identical Subscriptions in TIBCO Rendezvous Concepts](#).

To destroy a distributed queue transport, call [tibrvcTransport_Destroy\(\)](#).

Parameter	Description
<code>cmTransport</code>	<p>The program supplies a location, and the function stores the new distributed queue transport in that location.</p> <p>The distributed queue transport remains valid until the program explicitly destroys it.</p>
<code>transport</code>	<p>The new distributed <code>cmTransport</code> employs this transport object for network communications.</p> <p>Destroying the distributed <code>cmTransport</code> does not affect this transport. The program must explicitly call <code>tibrvTransport_Destroy()</code> to destroy this <code>tibrvTransport</code> when it is no longer needed.</p>
<code>cmName</code>	<p>Bind this reusable name to the new distributed <code>cmTransport</code>, so the distributed queue transport 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 Rendezvous subject names. It cannot begin with reserved tokens. It cannot be a non-reusable name generated by a call to <code>tibrvcTransport_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"> • The pending tasks assigned to the worker member exceed its task capacity. • The worker is also the scheduler. (The scheduler assigns tasks to its own worker role only when no other workers are available.) <p>Programs can set this parameter using the extended function. The brief form supplies the default value, <code>TIBRVCM_DEFAULT_WORKER_</code></p>

Parameter	Description
	WEIGHT (1).
workerTasks	<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>Programs can set this parameter using the extended function. The value must be a non-negative integer. The brief form supplies the default value, TIBRVCM_DEFAULT_WORKER_TASKS (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> <p> Warning</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 distributed queue transports use relative scheduler weight values to elect one transport as the scheduler; members with higher scheduler weight take precedence.</p> <p>Programs can set this parameter using the extended function. Acceptable values range from 0 to 65535. Zero is a special value, indicating that the member can never be the scheduler. The brief form supplies the default value, TIBRVCM_DEFAULT_SCHEDULER_</p>

Parameter	Description
	<p>WEIGHT (1).</p> <p>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 members 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>Programs can set this parameter using the extended function. The brief form supplies the default value, TIBRVCM_DEFAULT_SCHEDULER_HB (1.0).</p>
schedulerActivation	<p>When the heartbeat signal from the scheduler has been silent for this interval (in seconds), the member with the greatest scheduler weight takes its place as the new scheduler.</p> <p>All members 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>Programs can set this parameter using the extended function. The brief form supplies the default value, TIBRVCM_DEFAULT_SCHEDULER_ACTIVE (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

Constant	Value
<code>TIBRVCM_DEFAULT_SCHEDULER_HB</code>	1.0
<code>TIBRVCM_DEFAULT_SCHEDULER_ACTIVE</code>	3.5

Relationship to CM

Although distributed queue members are a specialized type of CM transport, their behavior is quite different. Distributed queue transports do not support any functions related to sending messages (for a complete list, see the table of disabled functions).

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 Functions

[`tibrvcmtTransport_AddListener\(\)`](#)
[`tibrvcmtTransport_AllowListener\(\)`](#)
[`tibrvcmtTransport_DisallowListener\(\)`](#)
[`tibrvcmtTransport_GetDefaultCMTimeLimit\(\)`](#)
[`tibrvcmtTransport_GetLedgerName\(\)`](#)
[`tibrvcmtTransport_GetRequestOld\(\)`](#)
[`tibrvcmtTransport_GetSyncLedger\(\)`](#)
[`tibrvcmtTransport_RemoveListener\(\)`](#)
[`tibrvcmtTransport_RemoveSendState\(\)`](#)
[`tibrvcmtTransport_ReviewLedger\(\)`](#)
[`tibrvcmtTransport_Send\(\)`](#)
[`tibrvcmtTransport_SendReply\(\)`](#)
[`tibrvcmtTransport_SendRequest\(\)`](#)
[`tibrvcmtTransport_SetDefaultCMTimeLimit\(\)`](#)

Disabled Functions

[tibrvcTransport_SyncLedger\(\)](#)

See Also

[tibrvcTransport_Destroy\(\)](#)

Distributed Queue, in TIBCO Rendezvous Concepts

tibrvcTransport_GetCompleteTime()

Function

Declaration

```
tibrv_status tibrvcTransport_GetCompleteTime(  
    tibrvcTransport cmTransport,  
    tibrv_f64*      completeTime);
```

Purpose

Extract the worker complete time limit of a distributed queue transport.

Remarks

The complete time parameter of the scheduler affects the reassignment of tasks.

Parameter	Description
cmTransport	Get the complete time of this distributed queue transport.
completeTime	The program supplies a location, and the function stores the worker complete time in that location.

See Also

[tibrvcTransport_SetCompleteTime\(\)](#)

Distributed Queue, in TIBCO Rendezvous Concepts

Complete Time, in TIBCO Rendezvous Concepts

tibrvcTransport_GetUnassignedMessageCount()

Function

Declaration

```
tibrv_status tibrvcTransport_GetUnassignedMessageCount(  
    tibrvcTransport cmTransport,  
    tibrv_u32*      msgCount);
```

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
cmTransport	Extract the number of unassigned task messages from this distributed queue transport.
msgCount	The program supplies a location, and the function stores the unassigned task count in that location.

tibrvcMTransport_GetTaskBacklogLimits()

Function

Declaration

```
tibrv_status tibrvcMTransport_GetTaskBacklogLimits(  
  
    tibrvcMTransport    cmTransport,  
  
    tibrv_u32*          limitBySizeInBytes,  
  
    tibrv_u32*          limitByMessages);
```

Purpose

Get the scheduler task queue limits of a distributed queue transport.

Remarks

The scheduler stores tasks in a queue. This property limits the maximum size of that queue—by number of bytes or number of messages (or both). When no value is set for this property, the default is no limit.

When the task messages in the queue exceed either of these limits, Rendezvous software deletes new inbound task messages.

Parameter	Description
cmTransport	Get the size limits of the task queue of this distributed queue transport.

Parameter	Description
<code>limitBySizeInBytes</code>	<p>The program supplies a location, and the function stores the size limit (in bytes) in that location.</p> <p>Zero is a special value, indicating that no size limit has been set.</p>
<code>limitByMessages</code>	<p>The program supplies a location, and the function stores the message limit (number of messages) in that location.</p> <p>Zero is a special value, indicating that no limit has been set for the number of messages.</p>

See Also

[tibrvcTransport_SetTaskBacklogLimit...\(\)](#)

Distributed Queue, in TIBCO Rendezvous Concepts

`tibrvcTransport_GetWorkerWeight()`

Function

Declaration

```
tibrv_status tibrvcTransport_GetWorkerWeight(  
    tibrvcTransport cmTransport,  
    tibrv_u32*      workerWeight);
```

Purpose

Extract the worker weight of a distributed queue transport.

Parameter	Description
<code>cmTransport</code>	Get the worker weight of this distributed queue transport.
<code>workerWeight</code>	The program supplies a location, and the function stores the worker weight in that location.

See Also

Distributed Queue, in TIBCO Rendezvous Concepts

[`tibrvcTransport_CreateDistributedQueue\(\)`](#)

[`tibrvcTransport_SetWorkerWeight\(\)`](#)

`tibrvcTransport_GetWorkerTasks()`

Function

Declaration

```
tibrv_status tibrvcTransport_GetWorkerTasks(  
    tibrvcTransport cmTransport,  
    tibrv_u32*      workerTasks);
```

Purpose

Extract the worker task capacity of a distributed queue transport.

Parameter	Description
<code>cmTransport</code>	Get the task capacity of this distributed queue transport.
<code>workerTasks</code>	The program supplies a location, and the function stores the task capacity in that location.

See Also

Distributed Queue, in TIBCO Rendezvous Concepts

[`tibrvcTransport_CreateDistributedQueue\(\)`](#)

[`tibrvcTransport_SetWorkerTasks\(\)`](#)

tibrvcMTransport_SetCompleteTime()

Function

Declaration

```
tibrv_status tibrvcMTransport_SetCompleteTime(  
    tibrvcMTransport cmTransport,  
    tibrv_f64         completeTime);
```

Purpose

Set the worker complete time limit of a distributed queue transport.

Remarks

The complete time parameter of the scheduler affects the reassignment of tasks:

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 function.

Parameter	Description
cmTransport	Set the complete time of this distributed queue transport.
completeTime	Use this complete time (in seconds). The time must be non-negative.

See Also

[tibrvcTransport_GetCompleteTime\(\)](#)

Distributed Queue in TIBCO Rendezvous Concepts

Complete Time in TIBCO Rendezvous Concepts

tibrvcTransport_SetTaskBacklogLimit...()

Function

Declaration

```
tibrv_status tibrvcTransport_SetTaskBacklogLimitInBytes(  
    tibrvcTransport    cmTransport,  
    tibrv_u32          limitBySizeInBytes);  
tibrv_status tibrvcTransport_SetTaskBacklogLimitInMessages(  
    tibrvcTransport    cmTransport,  
    tibrv_u32          limitByMessages);
```

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 either these properties, the default is no limit.

When the task messages in the queue exceed either of the two limits, Rendezvous software deletes new inbound task messages.

Programs may call each of these functions at most once. Those calls must occur before the transport assumes the scheduler role; after a transport acts as a scheduler, the values are fixed, and subsequent attempts to change them result in status code [TIBRV_NOT_PERMITTED](#).

Parameter	Description
cmTransport	Set the size limit of the task queue of this distributed queue transport.

Parameter	Description
<code>limitBySizeInBytes</code>	Use this size limit (in bytes). Zero is a special value, indicating no size limit.
<code>limitByMessages</code>	Use this message limit (number of messages). Zero is a special value, indicating no limit on the number of messages.

See Also

[tibrvcTransport_GetTaskBacklogLimits\(\)](#)

Distributed Queue in TIBCO Rendezvous Concepts

`tibrvcMTransport_SetWorkerWeight()`

Function

Declaration

```
tibrv_status tibrvcMTransport_SetWorkerWeight(  
    tibrvcMTransport cmTransport,  
    tibrv_u32 workerWeight);
```

Purpose

Set the worker weight of a distributed queue transport.

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 [tibrvcMTransport_CreateDistributedQueue\(\)](#), or change it dynamically using this function.

Parameter	Description
<code>cmTransport</code>	Set the worker weight of this distributed queue transport.
<code>workerWeight</code>	Use this worker weight.

See Also

Distributed Queue, in TIBCO Rendezvous Concepts

[tibrvcMTransport_CreateDistributedQueue\(\)](#)

[tibrvcMTransport_GetWorkerWeight\(\)](#)

tibrvcMTransport_SetWorkerTasks()

Function

Declaration

```
tibrv_status tibrvcMTransport_SetWorkerTasks(  
    tibrvcMTransport cmTransport,  
    tibrv_u32 workerTasks);
```

Purpose

Set the worker task capacity of a distributed queue transport.

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
<code>cmTransport</code>	Set the task capacity of this distributed queue transport.
<code>workerTasks</code>	<p>Use this task capacity. Value must be a non-negative integer.</p> <p>Zero is a special value, indicating that this distributed queue member is a dedicated scheduler (that is, it never accepts tasks).</p>

See Also

Distributed Queue, in TIBCO Rendezvous Concepts

[tibrvcTransport_CreateDistributedQueue\(\)](#)

[tibrvcTransport_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.

See Also

[Custom Datatypes](#)

Wire Format Datatypes

Wire Format Type	Type Description	Notes
Special Types		
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)	
Scalar Types		
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	

Wire Format Type	Type Description	Notes
TIBRVMSG_U64	64-bit unsigned integer	
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_U64ARRAY	64-bit unsigned integer array	

Wire Format Type	Type Description	Notes
TIBRVMSG_F32ARRAY	32-bit floating point array	
TIBRVMSG_F64ARRAY	64-bit floating point array	
TIBRVMSG_MESSAGEARRAY	message array	
TIBRVMSG_STRINGARRAY	string array	

Custom Types

TIBRVMSG_USER_FIRST	First code available for custom datatypes.	For more information, see Custom Datatypes .
TIBRVMSG_USER_LAST	Last code available for custom datatypes.	

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 tibrvMsgDateTime .

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 tibrvMsg .
<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 functions 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 functions that *add* a field to a Rendezvous message or update an existing field severely restrict type encoding. These functions 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 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

Status

Most Rendezvous functions return a status code, indicating either normal return status or a range of error conditions.

Programs must check the status after every Rendezvous function call, and take appropriate action.

Function	Description
Status Codes	
tibrv_status	Enumerated return codes from Rendezvous functions.
tibrvStatus_GetText()	Return the descriptive string corresponding to a status code.

tibrv_status

Type

Purpose

Enumerated return codes from Rendezvous functions.

Constant	Description
TIBRV_OK	The function returned without error.
TIBRV_INIT_FAILURE	Cannot create the network transport.
TIBRV_INVALID_TRANSPORT	The transport has been destroyed, or is otherwise unusable.
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 function 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	tibrvTransport_Create() cannot match the service name using getservbyname() .
TIBRV_NETWORK_NOT_FOUND	tibrvTransport_Create() cannot match the network name using getnetbyname() .
TIBRV_DAEMON_NOT_FOUND	tibrvTransport_Create() cannot match the daemon port number.

Constant	Description
TIBRV_NO_MEMORY	The function could not allocate dynamic storage.
TIBRV_INVALID_SUBJECT	The function 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.
TIBRV_NOT_PERMITTED	<p>The program attempted an illegal operation.</p> <p>For example:</p> <ul style="list-style-type: none"> • Cannot create ledger file. • Cannot confirm an uncertified message (that is, it has no sequence number).
TIBRV_INVALID_NAME	The field name is too long; see Field Name Length .
TIBRV_INVALID_TYPE	<ol style="list-style-type: none"> 1. The field type is not registered. 2. Cannot update field to a type that differs from the existing field's type.
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.

Constant	Description
TIBRV_NOT_FOUND	The function 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 function 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.
TIBRV_DECODER_FAILED	The program's datatype decoder failed.
TIBRV_INVALID_MSG	The function 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 function 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</p>

Constant	Description
	message. A virtual circuit terminal is not yet ready for use.
TIBRV_INTR	Interrupted operation.
TIBRV_INVALID_DISPATCHABLE	The function received an event queue or queue group that has been destroyed, or is otherwise unusable.
TIBRV_INVALID_DISPATCHER	The function received a dispatcher that is invalid or has been destroyed.
TIBRV_INVALID_EVENT	The function received an event that has been destroyed, or is otherwise unusable.
TIBRV_INVALID_CALLBACK	The function received NULL instead of a callback function.
TIBRV_INVALID_QUEUE	The function received a queue that has been destroyed, or is otherwise unusable.
TIBRV_INVALID_QUEUE_GROUP	The function received a queue group that has been destroyed, or is otherwise unusable.
TIBRV_INVALID_TIME_INTERVAL	The function received a negative timer interval.
TIBRV_INVALID_IO_SOURCE	The function received an invalid I/O source (for this operating system).
TIBRV_INVALID_IO_CONDITION	The function 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	tibrv_Open() encountered an operating system error.

Constant	Description
TIBRV_INSUFFICIENT_BUFFER	The function 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">1. Ledger file is not recognizable as such.2. tibrvSecureDaemon_SetUserCertWithKey() or tibrvSecureDaemon_SetUserCertWithKeyBin() 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.
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>
TIBRV_IO_FAILED	Cannot write to ledger file.
TIBRV_IPM_ONLY	The call is not available because IPM is not operating (that is, the call is available only when IPM is operating).

tibrvStatus_GetText()

Function

Declaration

```
const char* tibrvStatus_GetText(  
    tibrv_status status);
```

Purpose

Return the descriptive string corresponding to a status code.

Remarks

Status strings use the ISO 8859-1 character encoding.

Parameter	Description
status	Return the string for this status code.

See Also

[tibrv_status](#)

Custom Datatypes

Rendezvous programs can manipulate custom datatypes by defining and registering functions to translate them. This appendix describes the required functions and the tools you can use to implement them.

For most programs, the standard set of datatypes is sufficient. If your program does not require custom datatypes, you may skip this appendix.

Operations by Functional Group

Function or Type	Description
Handlers	
tibrvMsg_SetHandlers()	Define a program-specific datatype, by registering functions to transfer it between local format and wire format, and convert it to other datatypes.
tibrvMsgData_Converter	Convert a message field to another local datatype.
tibrvMsgData_Decoder	Decode a wire format field to a local datatype.
tibrvMsgData_Encoder	Encode a local format field to wire format.
Storage Type	
tibrvMsgDataType	Enumerate the types of low-level data references.
Utility Functions	
tibrvMsgData_ByteSize()	Calculate the wire buffer size of data from its C size.
tibrvMsgData_CopyBytes()	Copy data into the wire buffer of a message for an encoder.
tibrvMsgData_GetBytes()	Get data pointer and size from a wire buffer for a decoder.
tibrvMsgData_GetSize()	Get the data size from a wire buffer for a decoder.
tibrvMsgData_Malloc()	Allocate process storage for decoders and converters.
tibrvMsgData_SetSize()	Write the size of data into a wire buffer for an encoder.

Operations in Alphabetical Order

Function or Type	Description
tibrvMsg_SetHandlers()	Define a program-specific datatype, by registering functions to transfer it between local format and wire format, and convert it to other datatypes.
tibrvMsgDataType	Enumerate the types of low-level data references.
tibrvMsgData_ByteSize()	Calculate the wire buffer size of data from its C size.
tibrvMsgData_Converter	Convert a message field to another local datatype.
tibrvMsgData_CopyBytes()	Copy data into the wire buffer of a message for an encoder.
tibrvMsgData_Decoder	Decode a wire format field to a local datatype.
tibrvMsgData_Encoder	Encode a local format field to wire format.
tibrvMsgData_GetBytes()	Get data pointer and size from a wire buffer for a decoder.
tibrvMsgData_GetSize()	Get the data size from a wire buffer for a decoder.
tibrvMsgData_Malloc()	Allocate process storage for decoders and converters.
tibrvMsgData_SetSize()	Write the size of data into a wire buffer for an encoder.

Architecture Overview

[Architecture of Message and Data Manipulation](#) explains the stratification of data, field and message operations into four layers.

Architecture of Message and Data Manipulation

Layer	Description
4. Convenience Functions	<p>Type-specific functions manipulate message field data. This layer contains three functions per datatype—add, get and update. For example, see Add String. All convenience functions call the corresponding generic functions of layer 3 for their main functionality.</p> <p>When defining a custom datatype, a program can implement these three functions (optional).</p>
3. Generic Field Functions	<p>Field functions add, get or update a message field. This layer consists of only three functions: tibrvMsg_AddField(), tibrvMsg_GetField(), tibrvMsg_UpdateField(). Field functions call the datatype handler functions of layer 2 whenever data enters or leaves a message.</p>
2. Datatype Handler Functions	<p>Type-specific handler functions translate data between C format and Rendezvous wire format, and convert it to other datatypes. This layer contains three functions per datatype:</p> <ul style="list-style-type: none"> • encoder, of type tibrvMsgData_Encoder • decoder, of type tibrvMsgData_Decoder • converter, of type tibrvMsgData_Converter <p>To define a custom datatype, a program <i>must</i> implement these functions.</p>
1. Utility Functions	<p>Utility functions ease and standardize the implementation of datatype handler functions in layer 2. For example, tibrvMsgData_CopyBytes().</p>

Programs call functions in layer 4 (and sometimes layer 3) to move data in and out of messages. The remaining lower layers are generally invisible to most programs. That is, most programs call functions until layers 3.

However, a program that defines custom datatypes *must* implement the three functions in layer 2. Nevertheless, the program never calls layer 2 functions directly. Instead the program can define optional convenience functions at layer 4, and call them to move the custom datatype in and out of messages.

Adding Data

This narrative illustrates the interaction of functions at all four layers as they cooperate to add data to a message. The narrative for updating data within a message is parallel.

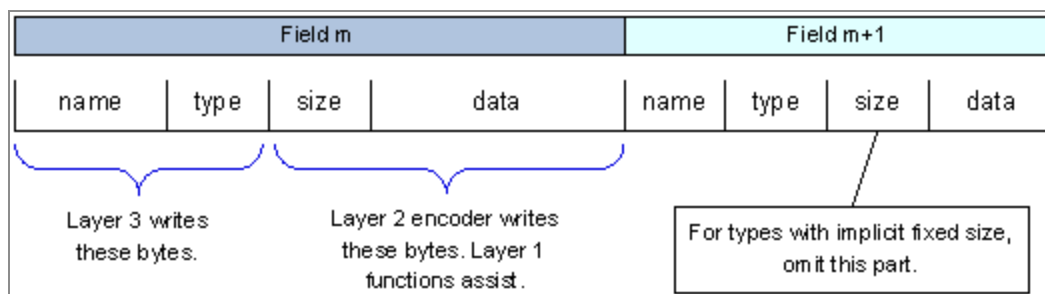
A program begins by calling a layer 4 convenience function to add data. The convenience function creates a field object ([tibrvMsgField](#)) that contains the data, the field name and identifier, a datatype token, and the element count (for arrays only). Then the convenience function calls the layer 3 field function [tibrvMsg_AddField\(\)](#) to add that field to the message.

Among its other tasks, [tibrvMsg_AddField\(\)](#) must write the field into wire buffer storage within the message object; to accomplish this step, it calls the layer 2 encoder function specific to the datatype.

The encoder transforms the data into its wire format, using layer 1 utility functions to copy that data into the message's wire buffer.

Each write operation updates the message's wire buffer pointer to the location where the next write will begin.

Figure 15: Writing Fields into a Message's Wire Buffer



Extracting Data

This narrative illustrates the interaction of functions at all four layers as they cooperate to get data from a message.

A program begins by calling a layer 4 convenience function to get data. The convenience function calls the layer 3 field function `tibrvMsg_GetField()` to find the field in the message.

Among its other tasks, `tibrvMsg_GetField()` must read the data from the message's wire buffer; to accomplish this step, it calls the layer 2 decoder function for the datatype (as specified in the message field).

The decoder uses layer 1 utility functions to extract that data from the message's wire buffer, and transforms the data into its C format.

Returning to layer 3, `tibrvMsg_GetField()` receives the data from the decoder, packaged in the `data` part of a field object (`tibrvMsgField`).

Returning to layer 4, the convenience function receives the field object from `tibrvMsg_GetField()`. The next task is to convert the data from its actual datatype to the target datatype (notice that the program, in effect, explicitly requested a target datatype by calling a particular type-specific convenience function). When the actual data does not already match the target datatype, the convenience function calls down into layer 2, to the converter function associated with the actual datatype.

If legal, the converter transforms the actual data to the target type, and modifies the field object accordingly.

Returning once again to layer 4, the convenience function extracts the modified data from the field object, and passes it back to the calling program.

Custom Datatype Checklist

We recommend modeling your custom datatype implementation on our example code; see the file `examples/usertypes.c`.

To define a custom datatype, a program must do these steps:

Procedure

1. Define the C representation of the new type. This representation is often a C struct.
2. Assign a code number to represent the new type. The new type must be in the range `[TIBRVMSG_USER_FIRST, TIBRVMSG_USER_LAST]`. All other codes are reserved.
3. Define an encoder function to transfer the new type from a field object into a message's wire buffer. See [tibrvMsgData_Encoder](#).
4. Define a decoder function to transfer the new type from a message's wire buffer into a field object. See [tibrvMsgData_Decoder](#).
5. Define a converter function to convert the new type to other datatypes. See [tibrvMsgData_Converter](#).
6. Register the encoder, decoder and converter functions. See [tibrvMsg_SetHandlers\(\)](#).
7. Optional: define convenience functions to add, get and update message fields with data of the new type. See [Convenience Functions](#).

Convenience Functions

When defining custom datatypes, programs have the option to also define convenience functions for the new type. In general, layer 4 convenience functions rely on layer 3 generic field functions to do most of their work, packing or unpacking the field structure as appropriate. You can choose to implement up to three convenience functions:

- *Get* a value of the custom datatype from a message.
- *Add* a value of the custom datatype to a message.
- *Update* an existing field in a message with a new value of the custom datatype.

Get

The get function must fulfill these responsibilities:

- Call `tibrvMsg_GetField()` to find the field in the message.
- Check that `tibrvMsg_GetField()` did not return an error code.
- Check that the `type` of the field matches the target datatype of the convenience function.
 - If the conversion is illegal, return an error code.
 - If it matches, then return, passing back the `data` directly.
 - Otherwise, convert the data to the target datatype. Pass back the converted `data`. (In most cases, converter functions associated with the actual datatype are unable to convert it to a custom type, so either the convenience function must convert it, or declare the conversion illegal.)

Add and Update

The add and update functions must fulfill these responsibilities:

- Validate the integrity of the data.

- Create a field object ([tibrvMsgField](#)) on the stack, and set all six of its parts to describe the data.
- Store it in the message:
 - Add calls [tibrvMsg_AddField\(\)](#) to add the field object to the message.
 - Update calls [tibrvMsg_UpdateField\(\)](#) to update an existing field in the message with the data from the new field object.

tibrvMsg_SetHandlers()

Function

Declaration

```
tibrv_status tibrvMsg_SetHandlers(  
    tibrv_u8          type,  
    tibrvMsgData_Encoder encoder,  
    tibrvMsgData_Decoder decoder,  
    tibrvMsgData_Converter converter);
```

Purpose

Define a program-specific datatype, by registering functions to transfer it between local format and wire format, and convert it to other datatypes.

Remarks

Programs that define custom data handler functions must register them before any message operations involving the custom datatype.

The program's data handler functions must properly address byte order and *endian* issues.

Parameter	Description
type	<p>Use this number as a unique identifier for the new datatype.</p> <p>The type identifier must be in the range [TIBRVMSG_USER_FIRST, TIBRVMSG_USER_LAST]; all other identifiers are reserved.</p>
encoder	<p>This function translates a local format instance of the datatype into wire format. See tibrvMsgData_Encoder.</p>

Parameter	Description
	NULL indicates that the program cannot add or update fields of this datatype.
decoder	This function translates wire format to a local format instance of the datatype. See tibrvMsgData_Decoder . NULL indicates that the program cannot get fields of this datatype.
converter	This function translates a field of this datatype to other datatypes. See tibrvMsgData_Converter . NULL indicates that the program cannot force conversion to another datatype during get and update calls.

See Also

[tibrvMsgData_Converter](#)

[tibrvMsgData_Decoder](#)

[tibrvMsgData_Encoder](#)

tibrvMsgDataType

Type

Declaration

```
typedef enum {  
    tibrvMsgData_Primitive,  
    tibrvMsgData_MallocBlock,  
    tibrvMsgData_SubMessage,  
    tibrvMsgData_WireReference  
} tibrvMsgDataType;
```

Purpose

Enumerate the types of low-level data references.

Remarks

This enumeration specifies the four low-level types of data reference that can be extracted from a message field. Decoders and converters create references to the data that they extract, and must inform the message of the type each time they create a reference. Internal functions use this type information to properly maintain references to the extracted data.

Value	Description
<code>tibrvMsgData_Primitive</code>	<p>Extract primitive types by copying the data into the destination field struct.</p> <p>The predefined set of primitive types is 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, tibrvMsgDateTime.</p>

Value	Description
<code>tibrvMsgData_MallocBlock</code>	<p>Extract this type by allocating a block of storage associated with the message. To allocate the block, programs must call <code>tibrvMsgData_Malloc()</code> rather than <code>malloc</code>.</p> <p>The predefined set of malloc block types includes all arrays. On EBCDIC architectures, it also includes EBCDIC strings.</p>
<code>tibrvMsgData_SubMessage</code>	<p>Extract this type by creating a new <code>tibrvMsg</code> object for the decoded data.</p> <p>The only submessage type is <code>tibrvMsg</code>.</p>
<code>tibrvMsgData_WireReference</code>	<p>Extract this type by copying a pointer into the destination field struct; the pointer refers to a location within the wire buffer of the message.</p> <p>The predefined set of wire reference types includes character strings, opaque byte sequences, and XML data.</p>

See Also

[Wire Format Datatypes](#)

[C Datatypes](#)

[tibrvMsgData_Converter](#)

[tibrvMsgData_Decoder](#)

[tibrvMsgData_Malloc\(\)](#)

tibrvMsgData_ByteSize()

Function

Declaration

```
tibrv_u32 tibrvMsgData_ByteSize(  
    tibrv_u32    content_size);
```

Purpose

Calculate the wire buffer size of data from its C size.

Remarks

Encoders call this layer 1 function to preview the wire size of the data as [tibrvMsgData_CopyBytes\(\)](#) would write it into the message (that is, including extra bytes for size information). Encoders test this wire size against the available space in the message.

Parameter	Description
<code>content_size</code>	The encoder supplies the C size (in bytes) of data.

See Also

[tibrvMsgData_Encoder](#)

[tibrvMsgData_SetSize\(\)](#)

tibrvMsgData_Converter

Function Type

Declaration

```
typedef tibrv_status (*tibrvMsgData_Converter)(  
    tibrvMsgField*      field,  
    tibrv_u8            destination_type,  
    tibrvMsgDataType*   converted_type);
```

Purpose

Convert a message field to another local datatype.

Remarks

Programs define converters for custom datatypes. Layer 2 converter functions augment the [Wire Format to C Datatype Conversion Matrix](#).

Each converter function receives a message field, and replaces information within it.

Each converter must fulfill these responsibilities:

- For disallowed conversions, return the status code [TIBRV_CONVERSION_FAILED](#).
- Convert the field's [data](#) to the `destination_type`, and store the new [data](#) back into the field.
- Set the field's [size](#) part to reflect the new size of the converted data.
- Set the field's [count](#) part to reflect the new element count of the converted data.
- Set the field's [type](#) part to reflect the `destination_type`.
- Store the low-level type of the new reference in `*converted_type`.

**Note**

If your converter implementation allocates process storage, it must call `tibrvMsgData_Malloc()` rather than `malloc`.

Parameter	Description
<code>field</code>	This parameter receives the location of a message field. The converter function modifies that field to effect the conversion.
<code>destination_type</code>	<p>This parameter receives the type identifier representing the datatype of the resulting field. This parameter instructs the converter function to convert the field to this datatype.</p> <p>The parameter can be any of the predefined datatypes listed in Wire Format Datatypes, as well as any custom datatypes that the program defines.</p>
<code>converted_type</code>	This parameter receives a location. The converter function must store the low-level type of the new data reference in that location.

See Also

[tibrvMsgField](#)

[tibrvMsgDataType](#)

[tibrvMsgData_GetBytes\(\)](#)

[tibrvMsgData_GetSize\(\)](#)

tibrvMsgData_CopyBytes()

Function

Declaration

```
tibrv_status tibrvMsgData_CopyBytes(  
    char**      buffer,  
    const void* src,  
    tibrv_u32    size);
```

Purpose

Copy data into the wire buffer of a message for an encoder.

Remarks

This layer 1 function does these steps (see [tibrvMsgData_CopyBytes](#)):

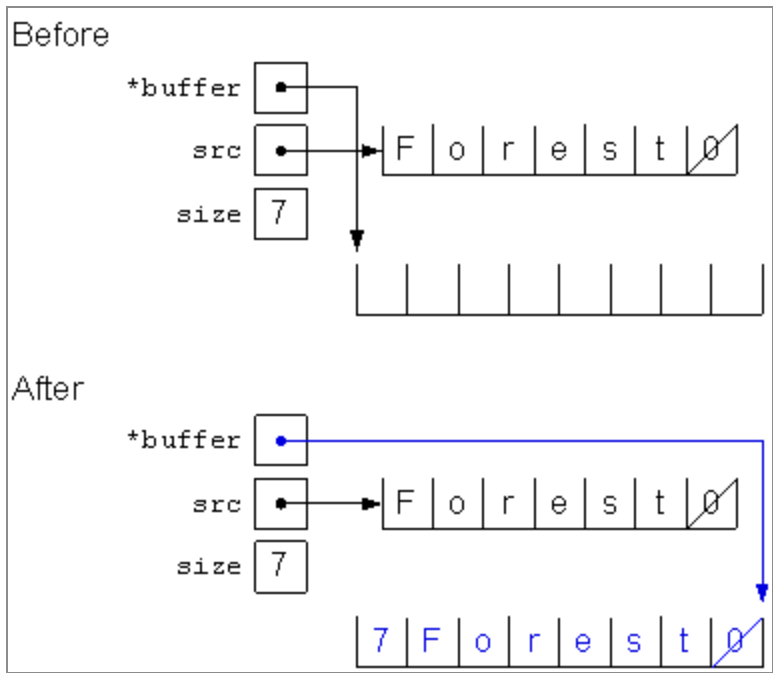
Procedure

1. Compute the wire size of the data from the size parameter.
2. Write the wire size into the wire buffer, and advance `*buffer`. (Now `*buffer` points to the end of the size information, which will be the start of the data.)
3. Copy `size` bytes from `src` into the wire buffer, and advance `*buffer` again. Now `*buffer` points to the end of the copied data.
4. Return.

Parameter	Description
<code>buffer</code>	The encoder supplies the location of an address within the wire buffer of a destination message. This function copies the source data into the destination message, starting at that address.

Parameter	Description
	Before returning, this function advances *buffer to the end of the data that it copied.
src	Copy the data from this source buffer.
size	Number of bytes to copy.

Figure 16: tibrvMsgData_CopyBytes



See Also

[tibrvMsgData_ByteSize\(\)](#)

[tibrvMsgData_Encoder](#)

tibrvMsgData_Decoder

Function Type

Declaration

```
typedef tibrv_status (*tibrvMsgData_Decoder)(  
    char**          wire_buffer,  
    tibrvMsgField*   field,  
    tibrvMsgDataType* decoded_type);
```

Purpose

Decode a wire format field to a local datatype.

Remarks

Programs define decoders for custom datatypes. Layer 2 decoder functions augment the [Wire Format to C Datatype Conversion Matrix](#).

Each decoder must fulfill these responsibilities:

- Set the [size](#), [count](#) and [data](#) parts of the destination field struct. (The layer 3 function sets the [name](#), [id](#) and [type](#).)
- Advance `*wire_buffer` to the end the source data (in the message). [tibrvMsgData_GetBytes\(\)](#) automatically advances this buffer pointer.
- Store the low-level type of the new data reference in `*decoded_type`.
- Check consistency, and properly address byte order and *endian* issues.



Note

If your decoder implementation allocates process storage, it must call [tibrvMsgData_Malloc\(\)](#) rather than `malloc`.

Parameter	Description
<code>wire_buffer</code>	This parameter receives the location of an address within the wire buffer of the source message. The source data starts at that address.
<code>field</code>	This parameter receives the location of a destination field object. The decoder function must set the parts of this field with the source data, its size and count.
<code>decoded_type</code>	This parameter receives a location. The decoder function must store the low-level type of the new data reference in that location.

See Also

[tibrvMsgField](#)

[tibrvMsgDataType](#)

[tibrvMsgData_GetBytes\(\)](#)

tibrvMsgData_Encoder

Function Type

Declaration

```
typedef tibrv_status (*tibrvMsgData_Encoder)(  
    char**                wire_buffer,  
    tibrv_u32             mem_available,  
    tibrvMsgField*        field);
```

Purpose

Encode a local format field to wire format.

Remarks

Programs define encoders for custom datatypes. Layer 2 encoder functions translate custom datatypes into Rendezvous wire format.

Each encoder must fulfill these responsibilities:

- Check that the field contains valid data of the appropriate type.
- Check that the data will fit in the available space; if not, return the error status `TIBRV_NO_MEMORY`.

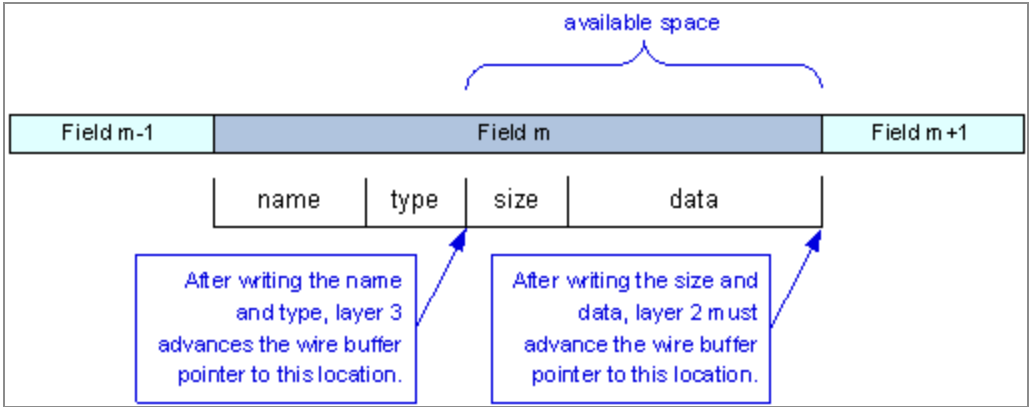
The encoder can use `tibrvMsgData_ByteSize()` to compute the wire size of the data.

- Write the data into wire buffer of the message. Do not overwrite space that is not available.
- Advance `*wire_buffer` to the end of the destination data (in the message). `tibrvMsgData_CopyBytes()` automatically advances this buffer pointer.
- Check consistency, and properly address byte order and *endian* issues.

Note If your encoder implementation allocates process storage, it must call `tibrvMsgData_Malloc()` rather than `malloc`.

Parameter	Description
wire_buffer	<p>This parameter receives the location of an address within the wire buffer of the destination message. The encoder must write the wire-format encoded data to this destination.</p> <p>We strongly recommend using <code>tibrvMsgData_CopyBytes()</code> to transfer the data.</p>
mem_available	<p>This parameter receives the size of the available storage in the message’s wire buffer.</p>
field	<p>This parameter receives a field object, with self-describing data in local format. This source field determines the data to place into the destination wire_buffer.</p>

Figure 17: Advancing the Wire Buffer Pointer



See Also

[tibrvMsgField](#)

[tibrvMsgData_ByteSize\(\)](#)

[tibrvMsgData_CopyBytes\(\)](#)

tibrvMsgData_GetBytes()

Function

Declaration

```
tibrv_status tibrvMsgData_GetBytes(  
    char**      buffer,  
    const void** src,  
    tibrv_u32*   size);
```

Purpose

Get data pointer and size from a wire buffer for a decoder.

Remarks

This layer 1 function helps decoders extract data from a message's wire buffer. It does these steps (see [tibrvMsgData_GetBytes\(\)](#)):

Procedure

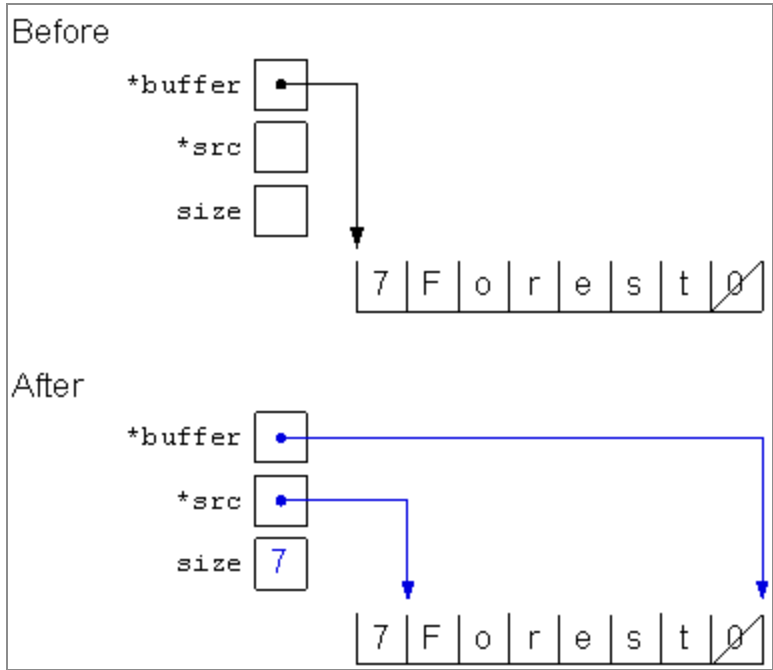
1. Read the size information from the wire buffer (starting at the position `*buffer`) and store it in the `size` parameter.
2. Store the start position of the actual data in the `src` parameter.
3. Advance `*buffer` to point to the end of the actual data.
4. Return.

After [tibrvMsgData_GetBytes\(\)](#) returns, the decoder can obtain the data by copying `*size` bytes from the location `*src`.

Parameter	Description
<code>buffer</code>	The decoder supplies the location of an address within the

Parameter	Description
	wire buffer of a source message. After reading the size information, this function advances *buffer to point to the location at the end of the data.
src	The decoder supplies the location of a buffer pointer. This function stores the address of the actual data in that pointer.
size	The decoder supplies a location. This function stores the size of the actual data in that location.

Figure 18: tibrvMsgData_GetBytes



See Also

[tibrvMsgData_Decoder](#)

tibrvMsgData_GetSize()

Function

Declaration

```
tibrv_status tibrvMsgData_GetSize(  
    char**      buffer,  
    tibrv_u32*   size);
```

Purpose

Get the data size from a wire buffer for a decoder.

Remarks

This layer 1 function helps decoders extract data from a message's wire buffer. It does only part of the work that [tibrvMsgData_GetBytes\(\)](#) does (see [tibrvMsgData_GetSize\(\)](#)):

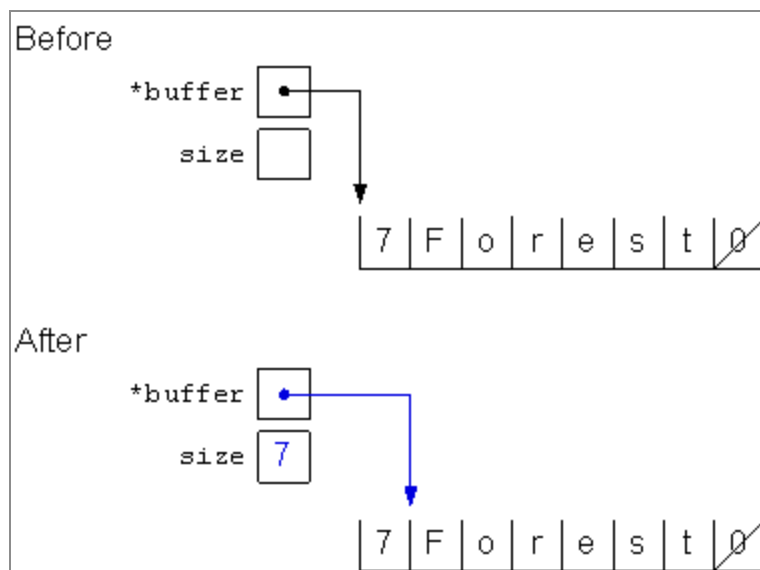
Procedure

1. Read the size information from the wire buffer (starting at the position `*buffer`) and store it in the size parameter.
2. Advance `*buffer` to point to the end of the size information, which is the start of the actual data.
3. Return.

After [tibrvMsgData_GetSize\(\)](#) returns, the decoder can obtain the data by copying `*size` bytes from the location `*buffer`. We recommend using the more complete function [tibrvMsgData_GetBytes\(\)](#); this function gives programmers finer control over pointer advancement (along with the responsibility to advance that pointer appropriately).

Parameter	Description
<code>buffer</code>	The decoder supplies the address of the wire buffer within the message. The function advances <code>*buffer</code> to point to the location at the end of the size (which is the beginning of the data).
<code>size</code>	The decoder supplies a location. This function stores the size of the actual data in that location.

Figure 19: `tibrvMsgData_GetSize`



See Also

[tibrvMsgData_Decoder](#)

[tibrvMsgData_GetBytes\(\)](#)

tibrvMsgData_Malloc()

Function

Declaration

```
void* tibrvMsgData_Malloc(  
    tibrv_u32    size);
```

Purpose

Allocate process storage for decoders and converters.

Remarks

Decoders and converters must use this layer 1 function to allocate storage for extracted data. Do not use ordinary `malloc`.

The function returns a pointer to the storage. The storage remains associated with the message, and persists until the message is destroyed. Programmers need not free this storage; instead, the message automatically frees the storage at the appropriate time.

Decoders and converters that allocate storage using this function must pass back the type [tibrvMsgData_MallocBlock](#).

Parameter	Description
<code>size</code>	Allocate a storage block of this size (in bytes).

See Also

[tibrvMsgDataType](#)

[tibrvMsgData_Converter](#)

[tibrvMsgData_Decoder](#)

`tibrvMsgData_Encoder`

`tibrvMsgData_SetSize()`

Function

Declaration

```
tibrv_status tibrvMsgData_SetSize(  
    char**      buffer,  
    tibrv_u32    size);
```

Purpose

Write the size of data into a wire buffer for an encoder.

Remarks

This layer 1 function helps encoders write size information to a message's wire buffer. It does only part of the work that [tibrvMsgData_CopyBytes\(\)](#) does (see [tibrvMsgData_SetSize\(\)](#)):

Procedure

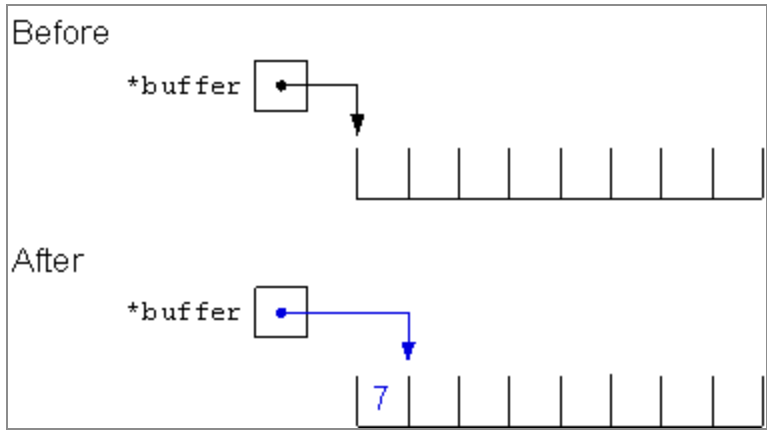
1. Write the size information into the wire buffer, starting at the position `*buffer`.
2. Advance `*buffer` to point to the end of the size information, which will become the start of the actual data.
3. Return.

After [tibrvMsgData_SetSize\(\)](#) returns, the encoder can continue by writing the data starting at the (advanced) position `*buffer`. We recommend using the more complete function [tibrvMsgData_CopyBytes\(\)](#) to write the size and data with one call; when finer control is required, programmers can use this function instead, with corresponding responsibility.

When using this function, the encoder must first use [tibrvMsgData_ByteSize\(\)](#) to measure the wire size from the data length.

Parameter	Description
<code>buffer</code>	The encoder supplies the address of a location within the message's wire buffer. This function writes the <code>size</code> into the destination message, and advances this buffer pointer.
<code>size</code>	Size to copy into the wire buffer.

Figure 20: `tibrvMsgData_SetSize`



See Also

- [tibrvMsgData_ByteSize\(\)](#)
- [tibrvMsgData_CopyBytes\(\)](#)
- [tibrvMsgData_Encoder](#)

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 documentation for this product is available on the [TIBCO Rendezvous® Product Documentation](#) page.

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 [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.cloud.com/legal>.

Copyright © 1997-2025. Cloud Software Group, Inc. All Rights Reserved.