OpenEdge® Development:
Messaging and ESB
Copyright

© 2017 Progress Software Corporation and/or its subsidiaries or affiliates. All rights reserved.

These materials and all Progress® software products are copyrighted and all rights are reserved by Progress Software Corporation. The information in these materials is subject to change without notice, and Progress Software Corporation assumes no responsibility for any errors that may appear therein. The references in these materials to specific platforms supported are subject to change.

Business Making Progress, Corticon, DataDirect (and design), DataDirect Cloud, DataDirect Connect, DataDirect Connect64, DataDirect XML Converters, DataDirect XQuery, Deliver More Than Expected, Icenium, Kendo UI, Making Software Work Together, NativeScript, OpenEdge, Powered by Progress, Progress, Progress Software Developers Network, Rollbase, SequeLink, Sitefinity (and Design), SpeedScript, Stylus Studio, TeamPulse, Telerik, Telerik (and Design), Test Studio, and WebSpeed are registered trademarks of Progress Software Corporation or one of its affiliates or subsidiaries in the U.S. and/or other countries. AccelEvent, Analytics360, AppsAlive, AppServer, Arcade, BravePoint, BusinessEdge, DataDirect Spy, DataDirect SupportLink, DevCraft, DigitalFactory, Fiddler, Future Proof, High Performance Integration, JustCode, JustDecompile, JustMock, JustTrace, OpenAccess, ProDataSet, Progress Arcade, Progress Profiles, Progress Results, Progress RFID, Progress Software, ProVision, PSE Pro, SectorAlliance, Sitefinity, SmartBrowser, SmartComponent, SmartDataBrowser, SmartDataObjects, SmartDataView, SmartDialog, SmartFolder, SmartFrame, SmartObjects, SmartPanel, SmartQuery, SmartViewer, SmartWindow, WebClient, and Who Makes Progress are trademarks or service marks of Progress Software Corporation and/or one of its affiliates or subsidiaries in the U.S. and other countries. Java is a registered trademark of Oracle and/or its affiliates. Any other marks contained herein may be trademarks of their respective owners.

Please refer to the Release Notes applicable to the particular Progress product release for any third-party acknowledgements required to be provided in the documentation associated with the Progress product.

The Release Notes can be found in the OpenEdge installation directory and online at: https://community.progress.com/technicalusers/w/openedgegeneral/1329.openedge-product-documentation-overview.aspx.

For the latest documentation updates see OpenEdge Product Documentation on Progress Communities: (https://community.progress.com/technicalusers/w/openedgegeneral/1329.openedge-product-documentation-overview.aspx).

March 2017

Last updated with new content: Release 11.7.0

Updated: 2017/03/24
Table of Contents

Preface .................................................................................................................................17
  Purpose.................................................................................................................................17
  Audience..............................................................................................................................18
  Organization.........................................................................................................................18
  Using ABL documentation..................................................................................................19
    References to ABL compiler and run-time features.........................................................19
    References to ABL data types.........................................................................................19
  Typographical conventions...............................................................................................20
  Examples of syntax descriptions....................................................................................21
    Long syntax descriptions split across lines................................................................22
    Complex syntax descriptions with both required and optional elements...................23
  Example procedures.........................................................................................................23
  OpenEdge messages.........................................................................................................24
    Obtaining more information about OpenEdge messages.............................................24

Chapter 1: OpenEdge Applications in the Sonic Environment...............................27
  Sonic messaging and integration systems........................................................................27
    SonicMQ.........................................................................................................................28
    Sonic ESB.....................................................................................................................28
  OpenEdge Adapters for Sonic integration......................................................................28
    OpenEdge Adapter for SonicMQ..................................................................................29
    OpenEdge Adapter for Sonic ESB...............................................................................30
  Security considerations....................................................................................................31
    Where to find detailed information............................................................................32
  OpenEdge Adapter for generic JMS-compliant messaging system..................................33
    Prerequisites................................................................................................................33

Chapter 2: Understanding the OpenEdge Adapters .................................................35
  Unified domain for JMS sessions....................................................................................35
  Exchanging messages.......................................................................................................36
  Understanding ABL - JMS object model.........................................................................36
    Session objects............................................................................................................37
    Message Consumer objects.........................................................................................37
    Message objects..........................................................................................................37

Chapter 3: Working with the OpenEdge Adapter for SonicMQ...............................39
  Accessing the OpenEdge Adapter for SonicMQ.............................................................39
Chapter 4: Working with the Generic JMS Adapter .................................41

Configuring and administering the Generic JMS Adapter ..............................................................41
   Editing the jmsProvider.properties file..................................................................................42
   Setting your JMS Provider for Generic Adapter Broker..........................................................42
   Updating the classpath for your JMS Provider ......................................................................43
   Configuring the Connection Factory .......................................................................................43
Configuring BrokerConnect ...........................................................................................................46
Non-supported API methods for Generic JMS Adapter .................................................................46

Chapter 5: Introduction to messaging...........................................................................................49

Point-to-Point (PTP) messaging.....................................................................................................49
   Sending and receiving messages...............................................................................................50
   PTP messaging options and features..........................................................................................50
   Building scalable server architecture with PTP queuing ..........................................................51
Publish-and-Subscribe messaging .............................................................................................51
   Sending and receiving messages...............................................................................................51
   Pub/Sub messaging options and features..................................................................................52
   Integrating with the native ABL publish-and-subscribe mechanism ........................................53
Comparing PTP and Pub/Sub messaging ......................................................................................53
Messages and message types........................................................................................................54

Chapter 6: Understanding the Messaging Models.......................................................................57

Using PTP messaging ....................................................................................................................57
   Creating a session procedure and connecting to a SonicMQ Broker ..........................................58
   Sending messages to a PTP queue ..............................................................................................58
   Receiving messages from a PTP queue ........................................................................................58
   Receiving a reply ........................................................................................................................59
   Temporary queues.....................................................................................................................59
   Deleting objects........................................................................................................................59
   Methods unique to Point-to-Point messaging ............................................................................60
Using PUB/SUB messaging ..........................................................................................................60
   Creating a session procedure and connecting to a SonicMQ Broker ..........................................60
   Publishing a message to a Pub/Sub topic ...................................................................................61
   Subscribing to a Pub/Sub topic and receiving messages .............................................................61
   Receiving a reply ........................................................................................................................61
   Durable subscriptions ...............................................................................................................62
   Temporary topic .........................................................................................................................62
   Deleting objects.........................................................................................................................62
   Methods unique to Pub/Sub messaging ....................................................................................62

Chapter 7: Implementing Messaging ...............................................................................................65
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Managing connections and sessions</td>
<td>65</td>
</tr>
<tr>
<td>Creating a JMS session</td>
<td>66</td>
</tr>
<tr>
<td>Deleting a JMS session</td>
<td>67</td>
</tr>
<tr>
<td>Connection options</td>
<td>67</td>
</tr>
<tr>
<td>Managing fail-over support</td>
<td>69</td>
</tr>
<tr>
<td>Setting and getting JMS connection and session attributes</td>
<td>70</td>
</tr>
<tr>
<td>Connecting to the OpenEdge Adapter for SonicMQ</td>
<td>70</td>
</tr>
<tr>
<td>Load balancing</td>
<td>70</td>
</tr>
<tr>
<td>Client persistence</td>
<td>70</td>
</tr>
<tr>
<td>Fault tolerance</td>
<td>71</td>
</tr>
<tr>
<td>Establishing session control</td>
<td>72</td>
</tr>
<tr>
<td>Accessing message delivery parameters</td>
<td>72</td>
</tr>
<tr>
<td>Request/Reply</td>
<td>73</td>
</tr>
<tr>
<td>Message selectors</td>
<td>74</td>
</tr>
<tr>
<td>Externally managed connections</td>
<td>74</td>
</tr>
<tr>
<td>Using serialized connection objects</td>
<td>74</td>
</tr>
<tr>
<td>Finding administered objects in JNDI or proprietary directories</td>
<td>82</td>
</tr>
<tr>
<td>Setting the CLASSPATH</td>
<td>83</td>
</tr>
<tr>
<td>Working with messages</td>
<td>84</td>
</tr>
<tr>
<td>Message life cycle</td>
<td>84</td>
</tr>
<tr>
<td>Creating, populating, and accessing messages</td>
<td>85</td>
</tr>
<tr>
<td>Sending messages to a queue</td>
<td>93</td>
</tr>
<tr>
<td>Publishing messages to a topic</td>
<td>94</td>
</tr>
<tr>
<td>Clearing messages</td>
<td>94</td>
</tr>
<tr>
<td>Deleting messages</td>
<td>94</td>
</tr>
<tr>
<td>Accessing message header properties</td>
<td>94</td>
</tr>
<tr>
<td>Accessing message properties</td>
<td>95</td>
</tr>
<tr>
<td>Consuming messages</td>
<td>96</td>
</tr>
<tr>
<td>Creating a Message Consumer object</td>
<td>96</td>
</tr>
<tr>
<td>Creating a message handler process</td>
<td>97</td>
</tr>
<tr>
<td>Setting reply properties</td>
<td>97</td>
</tr>
<tr>
<td>Receiving messages from a queue</td>
<td>97</td>
</tr>
<tr>
<td>Subscribing to a topic</td>
<td>98</td>
</tr>
<tr>
<td>Terminating the Message Consumer object</td>
<td>98</td>
</tr>
<tr>
<td>Processing messages</td>
<td>98</td>
</tr>
<tr>
<td>Controlling flow of messages</td>
<td>98</td>
</tr>
<tr>
<td>Reusing messages</td>
<td>98</td>
</tr>
<tr>
<td>Message-reception issues</td>
<td>98</td>
</tr>
<tr>
<td>Reply mechanisms</td>
<td>100</td>
</tr>
<tr>
<td>Transaction and recovery procedures</td>
<td>101</td>
</tr>
<tr>
<td>Transacted session</td>
<td>101</td>
</tr>
<tr>
<td>Message acknowledgement, forwarding, and recovery</td>
<td>102</td>
</tr>
<tr>
<td>Error and condition handling</td>
<td>103</td>
</tr>
<tr>
<td>Handling errors</td>
<td>104</td>
</tr>
<tr>
<td>Synchronously reported errors and conditions</td>
<td>104</td>
</tr>
</tbody>
</table>
Appendix A: ABL - JMS API Reference........................................................................161

Session objects...........................................................................................................169
jmssession.p..................................................................................................................170
ptpsession.p..................................................................................................................170
pubsubsession.p............................................................................................................171

Methods in the Session objects....................................................................................171
Methods in the Message Consumer objects...............................................................173
Methods in the Message objects..................................................................................174

acknowledgeAndForward procedure............................................................................177
addBytesPart procedure...............................................................................................178

addMessagePart procedure.........................................................................................178
addTextPart procedure................................................................................................179
appendText procedure...............................................................................................180

beginSession procedure............................................................................................180
browseQueue procedure.............................................................................................181
cancelDurableSubscription procedure.......................................................................182
clearBody procedure..................................................................................................182
clearProperties procedure........................................................................................183

commitReceive procedure.........................................................................................183
commitSend procedure...............................................................................................184
createBytesMessage procedure..................................................................................184
createChangeStateConsumer procedure....................................................................185
createDataSetMessage procedure.............................................................................186
createHeaderMessage procedure................................................................................186
createMapMessage procedure....................................................................................186
createMessageConsumer procedure...........................................................................187
createMultipartMessage procedure..........................................................................188
createRejectedMessageConsumer procedure...........................................................188
createStreamMessage procedure..............................................................................189
createTemporaryQueue procedure.............................................................................190
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>createTemporaryTopic procedure</td>
<td>190</td>
</tr>
<tr>
<td>createTempTableMessage procedure</td>
<td>191</td>
</tr>
<tr>
<td>createTextMessage procedure</td>
<td>191</td>
</tr>
<tr>
<td>createXMLMessage procedure</td>
<td>192</td>
</tr>
<tr>
<td>deleteConsumer procedure</td>
<td>192</td>
</tr>
<tr>
<td>deleteMessage procedure</td>
<td>193</td>
</tr>
<tr>
<td>deleteSaxWriter procedure</td>
<td>193</td>
</tr>
<tr>
<td>deleteSession procedure</td>
<td>194</td>
</tr>
<tr>
<td>deleteTemporaryQueue procedure</td>
<td>194</td>
</tr>
<tr>
<td>deleteTemporaryTopic procedure</td>
<td>195</td>
</tr>
<tr>
<td>endOfStream function</td>
<td>195</td>
</tr>
<tr>
<td>getAdapterService function</td>
<td>196</td>
</tr>
<tr>
<td>getApplicationContext function</td>
<td>197</td>
</tr>
<tr>
<td>getBrokerURL function</td>
<td>197</td>
</tr>
<tr>
<td>getBytesCount function</td>
<td>198</td>
</tr>
<tr>
<td>getBytesPartByID function</td>
<td>198</td>
</tr>
<tr>
<td>getBytesPartByIndex function</td>
<td>199</td>
</tr>
<tr>
<td>getBytesToRaw function</td>
<td>200</td>
</tr>
<tr>
<td>getChar function</td>
<td>200</td>
</tr>
<tr>
<td>getCharCount function</td>
<td>201</td>
</tr>
<tr>
<td>getCharProperty function</td>
<td>201</td>
</tr>
<tr>
<td>getClientID function</td>
<td>202</td>
</tr>
<tr>
<td>getClientPersistence function</td>
<td>202</td>
</tr>
<tr>
<td>getClientTransactionBufferSize function</td>
<td>203</td>
</tr>
<tr>
<td>getConnectionID function</td>
<td>204</td>
</tr>
<tr>
<td>getConnectionMetaData function</td>
<td>205</td>
</tr>
<tr>
<td>getConnectionURLs function</td>
<td>205</td>
</tr>
<tr>
<td>getContentType</td>
<td>205</td>
</tr>
<tr>
<td>getDataSet function</td>
<td>206</td>
</tr>
<tr>
<td>getDate function</td>
<td>207</td>
</tr>
<tr>
<td>getDateProperty function</td>
<td>207</td>
</tr>
<tr>
<td>getDateTime function</td>
<td>208</td>
</tr>
<tr>
<td>getDateTimeProperty function</td>
<td>209</td>
</tr>
<tr>
<td>getDateTime-TZ function</td>
<td>209</td>
</tr>
<tr>
<td>getDateTimeTzProperty function</td>
<td>210</td>
</tr>
<tr>
<td>getDecimal function</td>
<td>210</td>
</tr>
<tr>
<td>getDecimalProperty function</td>
<td>211</td>
</tr>
<tr>
<td>getDefaultPersistency function</td>
<td>211</td>
</tr>
<tr>
<td>getDefaultPriority function</td>
<td>212</td>
</tr>
<tr>
<td>getDefaultTimeToLive function</td>
<td>212</td>
</tr>
<tr>
<td>getDestinationName function</td>
<td>213</td>
</tr>
<tr>
<td>getFaultTolerant function</td>
<td>213</td>
</tr>
<tr>
<td>getFaultTolerantReconnectTimeout function</td>
<td>214</td>
</tr>
<tr>
<td>getFlowToDisk function</td>
<td>214</td>
</tr>
<tr>
<td>Function/Variable</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------</td>
<td>------</td>
</tr>
<tr>
<td>reset procedure</td>
<td>265</td>
</tr>
<tr>
<td>requestReply procedure</td>
<td>264</td>
</tr>
<tr>
<td>recover procedure</td>
<td>263</td>
</tr>
<tr>
<td>receiveFromQueue procedure</td>
<td>262</td>
</tr>
<tr>
<td>readLongStringCP function</td>
<td>262</td>
</tr>
<tr>
<td>readLongString function</td>
<td>261</td>
</tr>
<tr>
<td>readLogical function</td>
<td>261</td>
</tr>
<tr>
<td>readInt64 function</td>
<td>260</td>
</tr>
<tr>
<td>readInt function</td>
<td>259</td>
</tr>
<tr>
<td>readDateTime-TZ function</td>
<td>258</td>
</tr>
<tr>
<td>readDateTime function</td>
<td>258</td>
</tr>
<tr>
<td>readChar function</td>
<td>256</td>
</tr>
<tr>
<td>readBytesToRaw procedure</td>
<td>256</td>
</tr>
<tr>
<td>publish procedure</td>
<td>254</td>
</tr>
<tr>
<td>moveToNext procedure</td>
<td>253</td>
</tr>
<tr>
<td>getReusedMessage function</td>
<td>250</td>
</tr>
<tr>
<td>getSelectorAtBroker function</td>
<td>250</td>
</tr>
<tr>
<td>getShutdownWaitFor function</td>
<td>249</td>
</tr>
<tr>
<td>getSingleMessageAcknowledgement function</td>
<td>249</td>
</tr>
<tr>
<td>JMS-MAXIMUM-MESSAGES global variable</td>
<td>249</td>
</tr>
<tr>
<td>messageHandler procedure</td>
<td>249</td>
</tr>
<tr>
<td>getUser function</td>
<td>248</td>
</tr>
<tr>
<td>hasReplyTo function</td>
<td>248</td>
</tr>
<tr>
<td>inErrorHandling function</td>
<td>248</td>
</tr>
<tr>
<td>inMessageHandling function</td>
<td>249</td>
</tr>
<tr>
<td>inQueueBrowsing function</td>
<td>250</td>
</tr>
<tr>
<td>inReplyHandling function</td>
<td>250</td>
</tr>
<tr>
<td>isFaultTolerant function</td>
<td>251</td>
</tr>
<tr>
<td>isMessagePart function</td>
<td>251</td>
</tr>
<tr>
<td>getX-Document function</td>
<td>251</td>
</tr>
<tr>
<td>getTransactedSend function</td>
<td>247</td>
</tr>
<tr>
<td>getTransactedReceive function</td>
<td>247</td>
</tr>
<tr>
<td>getTextSegment function</td>
<td>246</td>
</tr>
<tr>
<td>getTextPartByIndex function</td>
<td>246</td>
</tr>
<tr>
<td>getTextPartByID function</td>
<td>246</td>
</tr>
<tr>
<td>getTransactedSend function</td>
<td>247</td>
</tr>
<tr>
<td>getTransactedReceive function</td>
<td>247</td>
</tr>
<tr>
<td>getText function</td>
<td>245</td>
</tr>
<tr>
<td>getTextPartByID function</td>
<td>245</td>
</tr>
<tr>
<td>getTextPartByIndex function</td>
<td>246</td>
</tr>
<tr>
<td>getTextSegment function</td>
<td>246</td>
</tr>
<tr>
<td>getSession function</td>
<td>241</td>
</tr>
<tr>
<td>getTempTable function</td>
<td>244</td>
</tr>
<tr>
<td>getShutdownWaitFor function</td>
<td>243</td>
</tr>
<tr>
<td>getSequential function</td>
<td>242</td>
</tr>
<tr>
<td>getSelectorAtBroker function</td>
<td>242</td>
</tr>
<tr>
<td>getHoldoff function</td>
<td>241</td>
</tr>
<tr>
<td>getReplyTimeToLive function</td>
<td>239</td>
</tr>
<tr>
<td>getReplyToDestinationType function</td>
<td>240</td>
</tr>
<tr>
<td>getReusedMessage function</td>
<td>240</td>
</tr>
<tr>
<td>getSaxWriter function</td>
<td>241</td>
</tr>
<tr>
<td>getSession function</td>
<td>241</td>
</tr>
<tr>
<td>getSequential function</td>
<td>242</td>
</tr>
<tr>
<td>getHoldoff function</td>
<td>241</td>
</tr>
<tr>
<td>getReplyTimeToLive function</td>
<td>239</td>
</tr>
<tr>
<td>getReplyToDestinationType function</td>
<td>240</td>
</tr>
<tr>
<td>getReusedMessage function</td>
<td>240</td>
</tr>
<tr>
<td>getSaxWriter function</td>
<td>241</td>
</tr>
<tr>
<td>getSession function</td>
<td>241</td>
</tr>
<tr>
<td>Procedure Name</td>
<td>Page</td>
</tr>
<tr>
<td>-----------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>rollbackSend procedure</td>
<td>266</td>
</tr>
<tr>
<td>sendToQueue procedure</td>
<td>266</td>
</tr>
<tr>
<td>setAdapterService procedure</td>
<td>267</td>
</tr>
<tr>
<td>setApplicationContext procedure</td>
<td>268</td>
</tr>
<tr>
<td>setBoolean procedure</td>
<td>268</td>
</tr>
<tr>
<td>setBooleanProperty procedure</td>
<td>269</td>
</tr>
<tr>
<td>setBrokerURL procedure</td>
<td>270</td>
</tr>
<tr>
<td>setByte procedure</td>
<td>270</td>
</tr>
<tr>
<td>setByteProperty procedure</td>
<td>271</td>
</tr>
<tr>
<td>setBytesFromRaw procedure</td>
<td>271</td>
</tr>
<tr>
<td>setChar procedure</td>
<td>272</td>
</tr>
<tr>
<td>setClientID procedure</td>
<td>273</td>
</tr>
<tr>
<td>setClientPersistence procedure</td>
<td>273</td>
</tr>
<tr>
<td>setClientTransactionBufferSize procedure</td>
<td>274</td>
</tr>
<tr>
<td>setConnectID procedure</td>
<td>275</td>
</tr>
<tr>
<td>setConnectionFile procedure</td>
<td>275</td>
</tr>
<tr>
<td>setConnectionURLs procedure</td>
<td>276</td>
</tr>
<tr>
<td>dataSet procedure</td>
<td>277</td>
</tr>
<tr>
<td>setDate procedure</td>
<td>277</td>
</tr>
<tr>
<td>setDateProperty procedure</td>
<td>278</td>
</tr>
<tr>
<td>setDateTime procedure</td>
<td>279</td>
</tr>
<tr>
<td>setDateTimeProperty procedure</td>
<td>280</td>
</tr>
<tr>
<td>setDateTime-TZ procedure</td>
<td>280</td>
</tr>
<tr>
<td>setDateTimeTzProperty procedure</td>
<td>281</td>
</tr>
<tr>
<td>setDefaultPersistency procedure</td>
<td>281</td>
</tr>
<tr>
<td>setDefaultPriority procedure</td>
<td>282</td>
</tr>
<tr>
<td>setDefaultTimeToLive procedure</td>
<td>283</td>
</tr>
<tr>
<td>setDouble procedure</td>
<td>283</td>
</tr>
<tr>
<td>setDoubleProperty procedure</td>
<td>284</td>
</tr>
<tr>
<td>setErrorHandler procedure</td>
<td>284</td>
</tr>
<tr>
<td>setFaultTolerant procedure</td>
<td>285</td>
</tr>
<tr>
<td>setFaultTolerantReconnectTimeout procedure</td>
<td>286</td>
</tr>
<tr>
<td>setFloat procedure</td>
<td>287</td>
</tr>
<tr>
<td>setFloatProperty procedure</td>
<td>287</td>
</tr>
<tr>
<td>setFlowToDisk procedure</td>
<td>287</td>
</tr>
<tr>
<td>setInt procedure</td>
<td>288</td>
</tr>
<tr>
<td>setIntProperty procedure</td>
<td>289</td>
</tr>
<tr>
<td>setInt64 procedure</td>
<td>289</td>
</tr>
<tr>
<td>setInt64Property procedure</td>
<td>290</td>
</tr>
<tr>
<td>setInitialConnectionTimeout procedure</td>
<td>290</td>
</tr>
<tr>
<td>setJMSCorrelationID procedure</td>
<td>291</td>
</tr>
<tr>
<td>setJMSCorrelationIDAsBytes procedure</td>
<td>292</td>
</tr>
<tr>
<td>setJMSReplyTo procedure</td>
<td>292</td>
</tr>
<tr>
<td>setJMSServerName procedure</td>
<td>293</td>
</tr>
<tr>
<td>setJMSType procedure</td>
<td>293</td>
</tr>
<tr>
<td>setJMSCorrelationIDAsBytes procedure</td>
<td>292</td>
</tr>
<tr>
<td>setJMSReplyTo procedure</td>
<td>292</td>
</tr>
<tr>
<td>setJMSServerName procedure</td>
<td>293</td>
</tr>
<tr>
<td>setJMSType procedure</td>
<td>293</td>
</tr>
<tr>
<td>Procedure Name</td>
<td>Page</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>-------</td>
</tr>
<tr>
<td>setLoadBalancing procedure</td>
<td>294</td>
</tr>
<tr>
<td>setLocalStoreDirectory procedure</td>
<td>294</td>
</tr>
<tr>
<td>setLocalStoreSize procedure</td>
<td>295</td>
</tr>
<tr>
<td>setLocalStoreWaitTime procedure</td>
<td>296</td>
</tr>
<tr>
<td>setLong procedure</td>
<td>296</td>
</tr>
<tr>
<td>setLongProperty procedure</td>
<td>297</td>
</tr>
<tr>
<td>setLongString procedure</td>
<td>298</td>
</tr>
<tr>
<td>setLongText procedure</td>
<td>298</td>
</tr>
<tr>
<td>setMemptr procedure</td>
<td>299</td>
</tr>
<tr>
<td>setNoAcknowledge procedure</td>
<td>300</td>
</tr>
<tr>
<td>setNoErrorDisplay procedure</td>
<td>300</td>
</tr>
<tr>
<td>setPassword procedure</td>
<td>301</td>
</tr>
<tr>
<td>setPingInterval procedure</td>
<td>301</td>
</tr>
<tr>
<td>setPrefetchCount procedure</td>
<td>302</td>
</tr>
<tr>
<td>setPrefetchThreshold procedure</td>
<td>303</td>
</tr>
<tr>
<td>setReconnectInterval procedure</td>
<td>304</td>
</tr>
<tr>
<td>setReconnectTimeout procedure</td>
<td>304</td>
</tr>
<tr>
<td>setReplyAutoDelete procedure</td>
<td>305</td>
</tr>
<tr>
<td>setReplyPersistency procedure</td>
<td>306</td>
</tr>
<tr>
<td>setReplyPriority procedure</td>
<td>306</td>
</tr>
<tr>
<td>setReplyTimeToLive procedure</td>
<td>307</td>
</tr>
<tr>
<td>setReplyToDestinationType procedure</td>
<td>308</td>
</tr>
<tr>
<td>setReuseMessage procedure</td>
<td>308</td>
</tr>
<tr>
<td>setSaxReader procedure</td>
<td>309</td>
</tr>
<tr>
<td>setSelectorAtBroker procedure</td>
<td>309</td>
</tr>
<tr>
<td>setSequential procedure</td>
<td>310</td>
</tr>
<tr>
<td>setShort procedure</td>
<td>310</td>
</tr>
<tr>
<td>setShortProperty procedure</td>
<td>311</td>
</tr>
<tr>
<td>setShutdownWaitFor procedure</td>
<td>312</td>
</tr>
<tr>
<td>setSingleMessageAcknowledgement procedure</td>
<td>312</td>
</tr>
<tr>
<td>setString procedure</td>
<td>313</td>
</tr>
<tr>
<td>setStringProperty procedure</td>
<td>313</td>
</tr>
<tr>
<td>setTempTable procedure</td>
<td>314</td>
</tr>
<tr>
<td>setText procedure</td>
<td>315</td>
</tr>
<tr>
<td>setTransactedReceive procedure</td>
<td>315</td>
</tr>
<tr>
<td>setTransactedSend procedure</td>
<td>316</td>
</tr>
<tr>
<td>setUser procedure</td>
<td>316</td>
</tr>
<tr>
<td>setX-Document procedure</td>
<td>317</td>
</tr>
<tr>
<td>startReceiveMessages procedure</td>
<td>317</td>
</tr>
<tr>
<td>stopReceiveMessages procedure</td>
<td>318</td>
</tr>
<tr>
<td>subscribe procedure</td>
<td>319</td>
</tr>
<tr>
<td>waitForMessages procedure</td>
<td>320</td>
</tr>
<tr>
<td>writeBoolean procedure</td>
<td>321</td>
</tr>
<tr>
<td>writeByte procedure</td>
<td>321</td>
</tr>
<tr>
<td>writeBytesFromRaw procedure</td>
<td>322</td>
</tr>
</tbody>
</table>
Preface

For details, see the following topics:

- Purpose
- Audience
- Organization
- Using ABL documentation
- Typographical conventions
- Examples of syntax descriptions
- Example procedures
- OpenEdge messages

Purpose

This manual provides overview information and programming guidelines for the OpenEdge® Adapters for SonicMQ® and Sonic ESB®. The SonicMQ Adapter and the Sonic ESB Adapter enable ABL (Advanced Business Language) applications to participate in the Sonic messaging and application integration environment.

It also discusses the generic Java Message Service (JMS) adapter that you can use for JMS-compliant messaging systems IBM WebSphereMQ and ActiveMQ other than SonicMQ and the Sonic ESB.
Audience

This manual is primarily intended for programmers interested in developing services and clients for use in the Sonic environment. Knowledge of ABL programming concepts and techniques is assumed, and a fundamental understanding of Web services technology and Sonic is desirable.

This manual also includes general discussion of the Sonic environment and its relationship to the OpenEdge product suite, which may be of interest to a broader audience including architects, system administrators, and others.

Organization

OpenEdge Applications in the Sonic Environment on page 27
Describes the SonicMQ Adapter and the Sonic ESB Adapter, and provides a brief overview of how the OpenEdge Adapter products function in the Sonic environment.

Working with the OpenEdge Adapter for SonicMQ on page 39
Explains basic concepts and general considerations for integrating OpenEdge applications with the JMS messaging service provided by SonicMQ.

Understanding the Messaging Models on page 57
Explains the Point-to-Point (PTP) and Publish/Subscribe (Pub/Sub) JMS messaging models.

Implementing Messaging on page 65
Describes how an OpenEdge client exchanges messages using the JMS messaging models.

Programming for the OpenEdge Adapter for SonicMQ with the ABL - JMS API on page 109
Provides instructions for using the ABL–JMS API to program applications for the SonicMQ environment.

Guidelines for Using and Programming for the OpenEdge Adapter for Sonic ESB on page 133
Provides general programming guidelines for the OpenEdge Adapter for Sonic ESB using the Native Invocation methodology and the Web Service Invocation methodology. The section also compares the Web Services Invocation methodology of the Sonic ESB Adapter with the OpenEdge Web Services Adapter (WSA). Because Web Services Invocation methodology is identical to the WSA with respect to application development, programmers should refer to WSA documentation, as indicated in this section, for detailed information.

ABL - JMS API Reference on page 161
Provides an alphabetical API reference for the OpenEdge Adapter for SonicMQ.

Messaging Examples on page 333
Provides ABL code examples of Pub/Sub and PTP messaging, and a sample application.

Sample Native Invocation ESB process on page 361
Provides an example of creating an ESB process using the Native Invocation methodology of the OpenEdge Adapter for Sonic ESB.
Using ABL documentation

OpenEdge provides a special purpose programming language for building business applications. In the documentation, the formal name for this language is ABL (Advanced Business Language). With few exceptions, all keywords of the language appear in all UPPERCASE, using a font that is appropriate to the context. All other alphabetic language content appears in mixed case.

For the latest documentation updates see the OpenEdge Product Documentation Overview page on Progress Communities:


References to ABL compiler and run-time features

ABL is both a compiled and an interpreted language that executes in a run-time engine. The documentation refers to this run-time engine as the ABL Virtual Machine (AVM). When the documentation refers to ABL source code compilation, it specifies ABL or the compiler as the actor that manages compile-time features of the language. When the documentation refers to run-time behavior in an executing ABL program, it specifies the AVM as the actor that manages the specified run-time behavior in the program.

For example, these sentences refer to the ABL compiler's allowance for parameter passing and the AVM's possible response to that parameter passing at run time: "ABL allows you to pass a dynamic temp-table handle as a static temp-table parameter of a method. However, if at run time the passed dynamic temp-table schema does not match the schema of the static temp-table parameter, the AVM raises an error." The following sentence refers to run-time actions that the AVM can perform using a particular ABL feature: "The ABL socket object handle allows the AVM to connect with other ABL and non-ABL sessions using TCP/IP sockets."

References to ABL data types

ABL provides built-in data types, built-in class data types, and user-defined class data types. References to built-in data types follow these rules:

• Like most other keywords, references to specific built-in data types appear in all UPPERCASE, using a font that is appropriate to the context. No uppercase reference ever includes or implies any data type other than itself.
• Wherever integer appears, this is a reference to the INTEGER or INT64 data type.
• Wherever character appears, this is a reference to the CHARACTER, LONGCHAR, or CLOB data type.
• Wherever decimal appears, this is a reference to the DECIMAL data type.
• Wherever numeric appears, this is a reference to the INTEGER, INT64, or DECIMAL data type.

References to built-in class data types appear in mixed case with initial caps, for example, Progress.Lang.Object. References to user-defined class data types appear in mixed case, as specified for a given application example.
## Typographical conventions

This documentation uses the following typographical and syntax conventions:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bold</strong></td>
<td>Bold typeface indicates commands or characters the user types, provides emphasis, or the names of user interface elements.</td>
</tr>
<tr>
<td><em>Italic</em></td>
<td>Italic typeface indicates the title of a document, or signifies new terms.</td>
</tr>
<tr>
<td><strong>SMALL, BOLD CAPITAL LETTERS</strong></td>
<td>Small, bold capital letters indicate OpenEdge key functions and generic keyboard keys; for example, GET and CTRL.</td>
</tr>
<tr>
<td>KEY1+KEY2</td>
<td>A plus sign between key names indicates a <strong>simultaneous</strong> key sequence: you press and hold down the first key while pressing the second key. For example, CTRL+X.</td>
</tr>
<tr>
<td>KEY1 KEY2</td>
<td>A space between key names indicates a <strong>sequential</strong> key sequence: you press and release the first key, then press another key. For example, ESCAPE H.</td>
</tr>
</tbody>
</table>

**Syntax:**

<table>
<thead>
<tr>
<th>Convention</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fixed width</strong></td>
<td>A fixed-width font is used in syntax, code examples, system output, and file names.</td>
</tr>
<tr>
<td><strong>Fixed-width italics</strong></td>
<td>Fixed-width italics indicate variables in syntax.</td>
</tr>
<tr>
<td><strong>Fixed-width bold</strong></td>
<td>Fixed-width bold italic indicates variables in syntax with special emphasis.</td>
</tr>
<tr>
<td><strong>UPPERCASE fixed width</strong></td>
<td>ABL keywords in syntax and code examples are almost always shown in upper case. Although shown in uppercase, you can type ABL keywords in either uppercase or lowercase in a procedure or class.</td>
</tr>
<tr>
<td><strong>Period (.) or colon (:)</strong></td>
<td>All statements except DO, FOR, FUNCTION, PROCEDURE, and REPEAT end with a period. DO, FOR, FUNCTION, PROCEDURE, and REPEAT statements can end with either a period or a colon.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Large brackets indicate the items within them are optional.</td>
</tr>
<tr>
<td>[]</td>
<td>Small brackets are part of ABL.</td>
</tr>
<tr>
<td>{ }</td>
<td>Large braces indicate the items within them are required. They are used to simplify complex syntax diagrams.</td>
</tr>
<tr>
<td>{}</td>
<td>Small braces are part of ABL. For example, a called external procedure must use braces when referencing arguments passed by a calling procedure.</td>
</tr>
</tbody>
</table>
Examples of syntax descriptions

In this example, ACCUM is a keyword, and aggregate and expression are variables:

**Syntax**

```
ACCUM aggregate expression
```

FOR is one of the statements that can end with either a period or a colon, as in this example:

```
FOR EACH Customer NO-LOCK:
  DISPLAY Customer.Name.
END.
```

In this example, STREAM stream, UNLESS-HIDDEN, and NO-ERROR are optional:

**Syntax**

```
DISPLAY [ STREAM stream ] [ UNLESS-HIDDEN ] [ NO-ERROR ]
```

In this example, the outer (small) brackets are part of the language, and the inner (large) brackets denote an optional item:

**Syntax**

```
INITIAL [ constant [ , constant ] ]
```

A called external procedure must use braces when referencing compile-time arguments passed by a calling procedure, as shown in this example:

**Syntax**

```
{ &argument-name }
```
In this example, \texttt{EACH}, \texttt{FIRST}, and \texttt{LAST} are optional, but you can choose only one of them:

**Syntax**

\begin{verbatim}
PRESELECT [ EACH | FIRST | LAST ] record-phrase
\end{verbatim}

In this example, you must include two expressions, and optionally you can include more. Multiple expressions are separated by commas:

**Syntax**

\begin{verbatim}
MAXIMUM ( expression , expression [ , expression ] ... )
\end{verbatim}

In this example, you must specify \texttt{MESSAGE} and at least one \texttt{expression} or \texttt{SKIP [( n )]}, and any number of additional \texttt{expression} or \texttt{SKIP [( n )]} is allowed:

**Syntax**

\begin{verbatim}
MESSAGE { expression | SKIP [( n )] } ...
\end{verbatim}

In this example, you must specify \{\texttt{include-file}, then optionally any number of \texttt{argument} or \texttt{&argument-name = "argument-value"}, and then terminate with \}:

**Syntax**

\begin{verbatim}
{ include-file
  [ argument | &argument-name = "argument-value" ] ... }
\end{verbatim}

**Long syntax descriptions split across lines**

Some syntax descriptions are too long to fit on one line. When syntax descriptions are split across multiple lines, groups of optional and groups of required items are kept together in the required order.

In this example, \texttt{WITH} is followed by six optional items:

**Syntax**

\begin{verbatim}
WITH [ ACCUM max-length ] [ expression DOWN ]
  [ CENTERED ] [ n COLUMNS ] [ SIDE-LABELS ]
  [ STREAM-IO ]
\end{verbatim}
Complex syntax descriptions with both required and optional elements

Some syntax descriptions are too complex to distinguish required and optional elements by bracketing only the optional elements. For such syntax, the descriptions include both braces (for required elements) and brackets (for optional elements).

In this example, ASSIGN requires either one or more field entries or one record. Options available with field or record are grouped with braces and brackets:

Syntax

\[
\text{ASSIGN} \{ [\text{FRAME frame}] \{ \text{field} [= \text{expression}] \} \ \\
\text{WHEN expression} \} \ldots \\
\text{record} [\text{EXCEPT field} \ldots ] \}
\]

Example procedures

OpenEdge documentation may provide example code that illustrates syntax and concepts. You can access many of the example files, and details for installing them, from the following locations:

- A self-extracting Documentation and Samples file available on the OpenEdge download page of the Progress Software Download Center
- The OpenEdge Product Documentation Overview page on Progress Communities:


Once installed, you can locate the example files in the following paths under the OpenEdge Documentation and Samples installation directory:

<table>
<thead>
<tr>
<th>This directory . . .</th>
<th>Contains examples for the following documents . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>src/prodoc/dotnetobjects</td>
<td>OpenEdge Development: GUI for .NET Programming</td>
</tr>
<tr>
<td>src/prodoc/dynamics</td>
<td>The Progress Dynamics documentation</td>
</tr>
<tr>
<td>src/prodoc/getstartoop</td>
<td>OpenEdge Development: Object-oriented Programming</td>
</tr>
<tr>
<td>src/prodoc/handbook</td>
<td>OpenEdge Getting Started: ABL Essentials</td>
</tr>
<tr>
<td>src/prodoc/interfaces</td>
<td>OpenEdge Development: Programming Interfaces</td>
</tr>
<tr>
<td>src/prodoc/json</td>
<td>OpenEdge Development: Working with JSON</td>
</tr>
<tr>
<td>src/prodoc/langref</td>
<td>OpenEdge Development: ABL Reference</td>
</tr>
</tbody>
</table>
OpenEdge messages

OpenEdge displays several types of messages to inform you of routine and unusual occurrences:

- **Execution messages** inform you of errors encountered while OpenEdge is running a procedure; for example, if OpenEdge cannot find a record with a specified index field value.

- **Compile messages** inform you of errors found while OpenEdge is reading and analyzing a procedure before running it; for example, if a procedure references a table name that is not defined in the database.

- **Startup messages** inform you of unusual conditions detected while OpenEdge is getting ready to execute; for example, if you entered an invalid startup parameter.

After displaying a message, OpenEdge proceeds in one of several ways:

- Continues execution, subject to the error-processing actions that you specify or that are assumed as part of the procedure. This is the most common action taken after execution messages.

- Returns to the Procedure Editor, so you can correct an error in a procedure. This is the usual action taken after compiler messages.

- Halts processing of a procedure and returns immediately to the Procedure Editor. This does not happen often.

- Terminates the current session.

OpenEdge messages end with a message number in parentheses. In this example, the message number is 200:

```
** Unknown table name table. (200)
```

If you encounter an error that terminates OpenEdge, note the message number before restarting.

Obtaining more information about OpenEdge messages

In Windows platforms, use OpenEdge online help to obtain more information about OpenEdge messages. Many OpenEdge tools include the following Help menu options to provide information about messages:
Choose Help > Recent Messages to display detailed descriptions of the most recent OpenEdge message and all other messages returned in the current session.

Choose Help > Messages and then type the message number to display a description of a specific OpenEdge message.

In the Procedure Editor, press the HELP key or F1.

On UNIX platforms, use the OpenEdge pro command to start a single-user mode character OpenEdge client session and view a brief description of a message by providing its number.

To use the pro command to obtain a message description by message number:

1. Start the Procedure Editor:

   OpenEdge-instal1-dir/bin/pro

2. Press F3 to access the menu bar, then choose Help > Messages.

3. Type the message number and press ENTER. Details about that message number appear.

4. Press F4 to close the message, press F3 to access the Procedure Editor menu, and choose File > Exit.
OpenEdge Applications in the Sonic Environment

This section briefly describes the Sonic messaging environment and provides an overview of the available options for integrating OpenEdge® applications in such an environment.

For information on the architecture of the OpenEdge Adapters for Sonic ESB® and SonicMQ®, see OpenEdge Getting Started: Application and Integration Services.

For information on configuring and managing the OpenEdge Adapters, see OpenEdge Application Server: Administration.

For details, see the following topics:

• Sonic messaging and integration systems
• OpenEdge Adapters for Sonic integration
• OpenEdge Adapter for generic JMS-compliant messaging system

Sonic messaging and integration systems

The sections that follow describe the two Sonic products for which OpenEdge adapters are available: SonicMQ and Sonic ESB. For complete information about their messaging and application integration functions, refer to the electronic documentation that is installed with each of these products.
SonicMQ

SonicMQ is a message-oriented middleware (MOM) system that allows diverse applications to communicate through Java™ Message Service (JMS) messages in a distributed enterprise system. JMS is an industry-standard messaging API, a standard that defines a full set of messaging capability. The SonicMQ JMS implementation offers exceptional reliability, performance, scalability, availability, and security, as well as powerful XML-processing capabilities.

SonicMQ provides full support for JMS Unified Domains, as well as the other standard JMS messaging domains, Point-to-Point (PTP) and Publish-and-Subscribe (Pub/Sub). It guarantees message delivery under all conditions. The loosely coupled, asynchronous data exchange mechanism allows for maximum reliability and flexibility.

The SonicMQ architecture relies on message brokers, each of which manages communications and security for one or more local client applications. Brokers can be located anywhere, according to geographical considerations and business needs. The use of broker clusters can provide load balancing and fault tolerance.

In addition to supporting standard JMS messaging, SonicMQ serves as the foundation for the powerful Sonic ESB application integration framework.

Sonic ESB

Sonic ESB provides the means to create automated workflow processes that can include many discrete applications written in diverse languages. Sonic ESB's service-oriented architecture supports processes that are efficient, flexible, and readily scalable. Sonic ESB is an application integration framework that provides high performance, reliability, and security. Highly sophisticated management and XML-editing tools, content-based routing, and an Orchestration Server make Sonic ESB an industry-leading integration framework for enterprise applications.

Sonic ESB's service-based architecture supports the deployment of discrete applications (services) that exchange messages according to sophisticated automated workflow processes, as well as the exposure of deployed applications as industry Web services. Services are typically small applications that perform very specific functions in response to requests from clients or other services. As such, they can be developed to serve the needs of a variety of business processes and can be readily deployed, scaled, and maintained.

SonicMQ is component of Sonic ESB and is installed as part of the Sonic ESB installation process. Sonic ESB services exchange JMS messages over the underlying SonicMQ backbone.

Although much of the power of Sonic ESB derives from its ability to integrate enterprise-wide applications in complex processes, it is also possible to expose any of its services as a standard Web service. Sonic ESB provides full support for Web service hosting, including WSDL generation, and it offers enhanced security and performance as compared with Web services hosted on a standard Web server or Java Servlet Engine (JSE). The same service can function both as a Web service and as a component of an integrated process.

OpenEdge Adapters for Sonic integration

The OpenEdge product suite offers two components that enable OpenEdge applications to participate fully in a Sonic integration environment:

- OpenEdge Adapter for SonicMQ on page 29
- OpenEdge Adapter for Sonic ESB on page 30
OpenEdge Adapter for SonicMQ

The OpenEdge Adapter for SonicMQ makes it possible for Advanced Business Language (ABL) programmers, working with familiar ABL syntax and tools, to write applications that use JMS messaging to send messages to and receive messages from applications written in ABL or other languages. The OpenEdge Adapter for SonicMQ implements a robust ABL–JMS API that provides access to almost all JMS messaging methods and functions from OpenEdge client applications. GUI, character-based, AppServer, and WebSpeed applications on all platforms supported for OpenEdge clients can participate in exchanging JMS messages. The OpenEdge Adapter for SonicMQ converts OpenEdge AppServer protocol to Java Message Service (JMS) protocol and vice versa, enabling OpenEdge client applications to send and receive JMS messages in a SonicMQ environment.

An OpenEdge application written to take advantage of the ABL–JMS API can talk with another application without knowing whether it is an OpenEdge or a non-OpenEdge application. Java features are mapped to ABL; for example, Java Enumeration Objects map to comma-separated lists in ABL. In an ABL-to-ABL messaging situation, an application can package ABL data within standard messages, for example, to send a temp table or a ProDataSet.

The ABL–JMS API is strongly integrated with the ABL programming model and style. Applications use the ABL event model. ABL procedures represent the JMS connection, Session objects, and Message objects. The ABL programmer uses the methods in these objects for JMS message delivery, acknowledgment, and recovery. All objects are persistent procedures. The API supports the basic types of JMS messaging, Unified Domain, Point-to-Point (PTP) and Publish and Subscribe (Pub/Sub). OpenEdge applications can extend local publish and subscribe for distributed applications. For information on the OpenEdge Adapter for SonicMQ architecture, see *OpenEdge Getting Started: Application and Integration Services*.

OpenEdge Adapter for SonicMQ configuration and management

The OpenEdge Adapter for SonicMQ allows OpenEdge applications to communicate via JMS Messaging through SonicMQ. The OpenEdge Adapter for SonicMQ consists of three connection modes:

- **OpenEdge Adapter for SonicMQ ClientConnect (ClientConnect)** — ClientConnect is for OpenEdge clients and will run transparently as a background process in conjunction with an OpenEdge client or OpenEdge Application Server agent process, with a single adapter process per client process. The application running on the OpenEdge client handles messaging control. ClientConnect takes little or no configuration.

- **OpenEdge Adapter for SonicMQ ServerConnect (ServerConnect)** — ServerConnect is for OpenEdge Application Servers (WebSpeed and AppServer). With this configuration there is a single adapter process per AppServer process, allowing multiple Application Server agents to connect to this single adapter process. ServerConnect is configured at the server.

- **OpenEdge Adapter for SonicMQ BrokerConnect (BrokerConnect)** — BrokerConnect is for OpenEdge client applications. It runs as a separate server process to handle OpenEdge client requests. BrokerConnect is a Unified Broker product, part of the AppServer administration framework. Thus, you can use the Progress Explorer in Windows, and the command-line tools `adaptconfig` and `adaptman` on all supported platforms, to manage BrokerConnect. You can also edit its properties in the `ubroker.properties` file. No configuration is required within the SonicMQ environment.

**Note:** For BrokerConnect, the OpenEdge installation program creates one instance of the OpenEdge Adapter for SonicMQ. In most circumstances, this single adapter instance is sufficient. Although it is possible to create additional instances, there is normally no reason to run multiple OpenEdge Adapter for SonicMQ instances on the same host. Each instance of BrokerConnect runs as a broker process. This process is multi-threaded, with one thread for each active client application; it can connect to any SonicMQ Broker.
OpenEdge Adapter for SonicMQ operation

The following table shows the operation features for each connection mode.

Table 1: Adapter features

<table>
<thead>
<tr>
<th>Feature</th>
<th>BrokerConnect</th>
<th>ClientConnect</th>
<th>ServerConnect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unified Domain</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Client Persistence</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Fault Tolerance</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Message Selectors</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Serialized Connections</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Temporary Queues and Topics</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

To establish a connection and start a session, the client application identifies connection options as an argument to the `ptpsession`, `pubsubsession`, `jmssession` procedure (depending on the chosen JMS messaging domain), specifies the appropriate SonicMQ Broker as an argument to the `setBrokerURL` procedure, and runs the `beginSession` procedure.

Messages are processed when the application is in a `WAIT-FOR` or other IO-blocking state. Non-UI applications, such as AppServer processes or batch processes that cannot use `WAIT FOR`, can use the `waitForMessages` procedure, as can all GUI, character, AppServer, WebSpeed, and batch applications. Applications use the existing ABL error-handling mechanisms to deal with ABL–JMS errors.

After the application finishes executing, it calls `deleteSession` procedure to free adapter resources for use by other clients.

License availability

Your OpenEdge license agreement with Progress Software Corporation might limit the number of concurrent sessions that can exist between clients and the OpenEdge Adapter for SonicMQ. If so, the limit is programmatically enforced, meaning that users may encounter error conditions preventing them from running client applications.

If the OpenEdge Adapter for SonicMQ resources available under your organization's license agreement are not sufficient to meet your usage requirements, please contact your OpenEdge sales representative for information about options for increasing capacity.

OpenEdge Adapter for Sonic ESB

The OpenEdge Adapter for Sonic ESB enables an OpenEdge service hosted on Sonic ESB to be accessed as part of workflow processes managed by Sonic ESB. The OpenEdge Adapter for Sonic ESB supports two methodologies:
• **Native Invocation methodology** — Sonic ESB calls and AppServer application directly. Native Invocation relies on invocation files that are created in your OpenEdge development environment, and are integrated into the workflow process. Native invocation provides simplified exposure of ABL code and a simplified process for mapping ABL parameters to Sonic messages.

• **Web Service methodology** — Sonic ESB calls an AppServer application as a Web service. The OpenEdge Adapter for Sonic ESB converts SOAP messages to AppServer protocol on inbound client requests, and converts outbound AppServer protocol to SOAP messages. Sonic ESB uses Web Service Description Language (WSDL) to make the OpenEdge service available to Web service clients, much as does the WSA in the OpenEdge environment.

**OpenEdge Adapter for Sonic ESB configuration and management**

The OpenEdge Adapter for Sonic ESB is installed through the OpenEdge installation program. Installation registers the OpenEdge Native Invocation and Web Service Invocation type definitions as available application types in Sonic ESB, and configures a default runtime container for deployment installations, and a development container for development installations. It also installs a custom resource editor, which is used to set custom run-time properties and generate Web Services Description Language (WSDL) files for OpenEdge services. For details on configuring and managing the OpenEdge Adapter for Sonic ESB, see *OpenEdge Application Server: Administration*.

**Security considerations**

The OpenEdge implementation of Secure Sockets Layer (SSL) technology enables both BrokerConnect and the OpenEdge Adapter for Sonic ESB to support secure connections between the OpenEdge application and the component functioning as the server. In the case of BrokerConnect, the server component is the adapter itself; in the case of the OpenEdge Adapter for Sonic ESB, the server component is an AppServer.

**Note:** For ClientConnect and ServerConnect, there is no connection to secure over the network.

Security derives from the client authentication of the server's identity via a Public Key Infrastructure (PKI), and a symmetric data encryption system. The security of the keys and digital certificates used by the PKI depends on trust in the certificate authorities (CAs) that issue them. OpenEdge provides default keys for applications that require only encryption/decryption of client/server communications without the need for full client/server authentication. OpenEdge provides tools for managing these keys and the digital certificates for exchanging them. For an overview of OpenEdge security concepts and instructions on the use of these tools, see *OpenEdge Getting Started: Core Business Services*.

**BrokerConnect security**

BrokerConnect provides SSL-based security as the server to an OpenEdge client. Use of secure connections between an application and BrokerConnect requires that the following conditions be satisfied:

• BrokerConnect must be configured to accept SSL connections, and to provide an alias and password for access to the private key/digital certificate used to provide connections to the adapter. For this configuration task, you can use the Progress Explorer (in Windows only) or edit the `ubroker.properties` file. For more information, see *OpenEdge Application Server: Administration*.

**Note:** You can use the `mergeprop` utility installed with OpenEdge to edit the `ubroker.properties` file. For information on using `mergeprop`, see *OpenEdge Getting Started: Installation and Configuration*. 
• The client must have access to a digital (public key) certificate that can authenticate with the digital certificate used by BrokerConnect (the server). For more information, see OpenEdge Getting Started: Core Business Services.

• The client application must use a secure protocol to connect to BrokerConnect. See the Managing connections and sessions for details on using SSL-based connection parameters.

In addition to functioning as a server to the OpenEdge client, BrokerConnect is also a client of the SonicMQ Broker. The security of communications between BrokerConnect and the SonicMQ Broker is managed through SonicMQ. For more information, refer to the SonicMQ documentation.

OpenEdge Adapter for Sonic ESB security

An OpenEdge service deployed to the OpenEdge Adapter for Sonic ESB functions as a client of an OpenEdge AppServer. In cases where the AppServer is SSL-enabled, the OpenEdge Adapter for Sonic ESB supports secure communications by providing a secure session tunnel. SSL client options can be specified for each deployed service; these options are described in OpenEdge Application Server: Administration.

Use of secure connections between an ESB-deployed OpenEdge service and an AppServer requires that the following conditions be satisfied:

• The service must be enabled for SSL communications, and it must have access to a digital (public key) certificate that can authenticate with the digital certificate used by the AppServer. For more information, see OpenEdge Getting Started: Core Business Services.

• The AppServer must be configured to accept SSL connections. For this configuration task, you can use the Progress Explorer (in Windows only) or edit the ubroker.properties file. For instructions, see OpenEdge Application Server: Administration.

Note: You can use the mergeprop utility installed with OpenEdge to manually edit the ubroker.properties file. For information on using mergeprop, see OpenEdge Getting Started: Installation and Configuration.

The security of communications between Sonic ESB services and the client applications that call them is provided through the facilities of the Sonic ESB and the underlying SonicMQ. For more information, refer to the Sonic ESB documentation.

Where to find detailed information

The sections that follow provide a concise summary of the available documentation for the OpenEdge Adapter for SonicMQ and the OpenEdge Adapter for Sonic ESB.

OpenEdge Adapter for SonicMQ documentation

Information on the OpenEdge Adapter for SonicMQ is organized as follows:

• Installation — See OpenEdge Getting Started: Installation and Configuration

• Overview and architecture description — See OpenEdge Getting Started: Application and Integration Services

• Programming guidelines — See Working with the OpenEdge Adapter for SonicMQ on page 39, Programming for the OpenEdge Adapter for SonicMQ with the ABL - JMS API on page 109, ABL - JMS API Reference on page 161, and Messaging Examples on page 333
OpenEdge Adapter for SonicESB documentation

Information on the OpenEdge Adapter for Sonic ESB is organized as follows:

- **Installation** — See *OpenEdge Getting Started: Installation and Configuration*
- **Overview and architecture description** — See *OpenEdge Getting Started: Application and Integration Services*
- **Programming guidelines** — See Guidelines for Using and Programming for the OpenEdge Adapter for Sonic ESB on page 133.
  
  For the programming of Web Services and clients for the Web Service Invocation methodology, use the techniques and constructs detailed for the WSA in *OpenEdge Development: Web Services*
- **Management and configuration** — See *OpenEdge Application Server: Administration*

OpenEdge Adapter for generic JMS-compliant messaging system

OpenEdge provides a generic Java Message Service (JMS) adapter for messaging. This generic JMS adapter can operate with any JMS-compliant vendor, for example, SonicMQ, IBM WebSphereMQ, ActiveMQ, and HornetQ.

Prerequisites

Before you can use the generic JMS adapter, you must have configured a JMS provider server and it is ready for use. Ensure that you have access to binaries that are required for the JMS provider.

Make sure the AdminServer and SonicAdapter Broker are installed.

You need to know the values of the following JMS provider properties:

- **PROVIDER_HOST:PROVIDER_PORT**: The host name and port number of the JMS provider server
- **PROVIDER_CONNECTION_FACTORY**: The name of the connection factory of the JMS provider
- **PROVIDER_QUEUE, PROVIDER_TOPIC**: The names of the queues and topics of the JMS provider that are to be used
Understanding the OpenEdge Adapters

The following sections provide general information about the use of the OpenEdge Adapter and how it works:

- Unified domain for JMS sessions on page 35
- Exchanging messages on page 36
- Understanding ABL - JMS object model on page 36

For a more detailed discussion of OpenEdge Adapter for SonicMQ architecture, see *OpenEdge Getting Started: Application and Integration Services*.

For details, see the following topics:

- Unified domain for JMS sessions
- Exchanging messages
- Understanding ABL - JMS object model

**Unified domain for JMS sessions**

Prior to OpenEdge Release 10.1, clients were required to create a JMS session for either PTP or Pub/Sub. In order for a client to use both queues and topics, the client needed to create two separate JMS sessions.

The following ABL code starts a JMS session using queues:

```
RUN jms/ptpsession.p PERSISTENT SET ptpsession (adapterConnection).
```
The following ABL code starts a JMS session using topics:

```
RUN jms/pubsubsession.p PERSISTENT SET pubsubsession (adapterConnection).
```

Currently, clients can use both PTP and Pub/Sub in the same JMS session by using the unified domain model.

The following ABL code starts a JMS session using the unified domain to access both queues and topics in one JMS session object:

```
RUN jms/jmssession.p PERSISTENT SET jmssession (adapterConnection).
```

**Note:** See the [Connection options](#) on page 67 for valid values for `adapterConnection`.

---

## Exchanging messages

The following steps are necessary to allow the exchange of messages between ABL and JMS:

1. The programmer writes an ABL program that connects to the SonicMQ Broker through the OpenEdge Adapter for SonicMQ by creating a persistent *Session object* procedure using `OpenEdge-Install-Directory/jms/jmssession.r`, `OpenEdge-Install-Directory/jms/ptpsession.r`, or `OpenEdge-Install-Directory/jms/pubsubsession.r`.

2. The programmer uses the ABL–JMS API implemented by the *Session objects* to send and receive JMS messages through the OpenEdge client application.

**Note:** The programmer does not have to write any Java or ABL code on the server side. That code is supplied by Progress Software Corporation and installed with the OpenEdge Adapter for SonicMQ and in the client.

---

## Understanding ABL - JMS object model

The *ABL–JMS object model* is a model wherein an OpenEdge application interacts with a JMS messaging broker through ABL objects (persistent procedures) that encapsulate the functionality of JMS sessions and JMS messages. These objects are:

- **Session objects** on page 37
- **Message Consumer objects** on page 37
- **Message objects** on page 37

The OpenEdge application interacts with the SonicMQ Broker through internal procedures, which perform actions, and user-defined function calls to extract values.

The sections that follow describe Session, Message Consumer, and Message objects, how they correspond to ABL procedures, and how to create them.
Session objects

A session is a context for sending and receiving messages. OpenEdge supplies three session procedures that the OpenEdge application uses to interact with JMS. These procedures run persistently to represent a JMS session and its underlying connection:

- jms/jmssession.p — For PTP and Pub/Sub messaging
- jms/ptpsession.p — For PTP messaging
- jms/pubsubsession.p — For Pub/Sub messaging

A single OpenEdge session can have any number of session procedure instances, each of which creates an underlying JMS connection. These session procedures implement internal procedures that return additional ABL objects in the form of ABL persistent procedure handles.

For more information, see Managing connections and sessions on page 65.

Message Consumer objects

The Session object procedures use an internal procedure to create a Message Consumer object. The Message Consumer is a JMS messaging object that performs the following:

- Receives messages from a destination
- Receives asynchronous error messages

The OpenEdge application must set a message handler procedure in a Message Consumer object by implementing an ABL internal procedure with a specific signature.

The OpenEdge Adapter for SonicMQ integrates with ABL event handling. Messages are processed by the Message Consumer when the ABL Virtual Machine (ABL) is in a WAIT–FOR state or other IO-blocking state. While the application is in such a state, all other UI and non-UI events are handled normally. WAIT–FOR can be called explicitly by the ABL code. It can also be called through the waitForMessages procedure in the Session object, which works the same for GUI, character, batch, AppServer, and WebSpeed applications.

For more information, see Consuming messages on page 96.

Message objects

A Message object holds the message or information package being sent. The format of the message is determined by the message type. SonicMQ provides several standard JMS message types, plus the XMLMessage and MultipartMessage type. The Session objects have internal procedures that create the messages and represent the JMS connection, Session objects, and Message objects.

For more information, see Messages and message types on page 54. 
Working with the OpenEdge Adapter for SonicMQ

This section provides an introduction to accessing SonicMQ messaging services from OpenEdge®.

Programming instructions are provided in Programming for the OpenEdge Adapter for SonicMQ with the ABL - JMS API on page 109.

For more details about SonicMQ, see the SonicMQ Programming Guide, which is installed with the SonicMQ product. Information on the Java Message Service specification is available on the Web at www.oracle.com/technetwork/java/docs-136352.html.

For details, see the following topics:

- Accessing the OpenEdge Adapter for SonicMQ

Accessing the OpenEdge Adapter for SonicMQ

To enable access to the SonicMQ Broker from OpenEdge, the following components are required:

- The SonicMQ product from, installed and set up as explained in the Sonic documentation provided with the product
- The OpenEdge Adapter for SonicMQ, installed and set up as detailed in OpenEdge Getting Started: Installation and Configuration

The ABL programmer should be familiar with:

- The ABL event model
• The basic concepts of the JMS model. JMS programming is a plus for accomplishing complex tasks.
Working with the Generic JMS Adapter

To use the generic JMS adapter, you must point OpenEdge to the JMS provider that you are using. To achieve this, OpenEdge must find the connection factory specific to your JMS provider.

Generic JMS Adapter is certified with WebSphereMQ, ActiveMQ, and HornetQ messaging systems.

**Note:** Generic JMS Adapter supports Client Connect, Broker Connect, and Server Connect modes.

For details, see the following topics:

- Configuring and administering the Generic JMS Adapter
- Configuring BrokerConnect
- Non-supported API methods for Generic JMS Adapter

Configuring and administering the Generic JMS Adapter

Before you start configuring the Generic JMS Adapter, make sure that all the prerequisites are met.
Editing the jmsProvider.properties file

The jmsProvider.properties file contains the class names for the provider specific connection factories for JMS client connection.

Open %DLC%\properties\jmsProvider.properties in a text editor, and add the JMS provider’s name and the following properties with the required connection factory classes:

```java
[MyJMSProvider]
javax.jms.ConnectionFactory=
javax.jms.QueueConnectionFactory=
javax.jms.TopicConnectionFactory=
```

**Note:** The jmsProvider.properties file contains the commonly used JMS provider names. You can add your JMS provider’s name and the properties in the same format if it is not listed.

Here is a sample of the jmsProvider.properties file that provides connection factory class names for WebSphereMQ and ActiveMQ.

```java
[WebSphereMQ]
javax.jms.ConnectionFactory=com.ibm.mq.jms.MQConnectionFactory
javax.jms.QueueConnectionFactory=com.ibm.mq.jms.MQQueueConnectionFactory
javax.jms.TopicConnectionFactory=com.ibm.mq.jms.MQTopicConnectionFactory

[ActiveMQ]
javax.jms.ConnectionFactory=org.apache.activemq.ActiveMQConnectionFactory
javax.jms.QueueConnectionFactory=org.apache.activemq.ActiveMQConnectionFactory
javax.jms.TopicConnectionFactory=org.apache.activemq.ActiveMQConnectionFactory
```

**Note:** Do not modify the javax.jms.ConnectionFactory, javax.jms.QueueConnectionFactory, and javax.jms.TopicConnectionFactory property names. The property values for these property names must be specific to the provider.

Setting your JMS Provider for Generic Adapter Broker

Once you have added the JMS provider you want to use, you must set the JMS provider to use the Generic JMS adapter by default.

In the %DLC%\properties path, open the AdminServerPlugins.properties file and update the jvmargs property in the PluginPolicy.Progress.SonicMQ property section as follows:

```java
PluginPolicy.Progress.SonicMQ
pluginclasspath=C:\Progress115\OpenEdge\java\progress.jar,
C:\PROGRA~2\IBM\WEBSPH~1\java\lib\com.ibm.mqjms.jar
classpath=C:\Progress115\OpenEdge\java\progress.jar,
C:\PROGRA~2\IBM\WEBSPH~1\java\lib\com.ibm.mqjms.jar
jvmargs=-DsonicMQExtensions=false -DjmsProvider=WebSphereMQ
```
The `sonicMQExtensions` property indicates whether to use the new adapter. You can set this property as `-DsonicMQExtensions=false` to use the generic JMS adapter as the JMS provider or set as `-DsonicMQExtensions=true` to use SonicMQ as the JMS provider.

**Note:** The value for the `-DjmsProvider` property must match the name of the JMS provider specified in the `jmsProvider.properties` file.

### Updating the classpath for your JMS Provider

To use a JMS provider other than SonicMQ, you have to replace the references to the Sonic client JAR files with your provider JAR files. To do this, open `%DLC%\properties\ AdminServerPlugins.properties` in a text editor and add the location of the client JAR files of your JMS provider to the `pluginclasspath` and the `classpath` properties. In the following example, `com.ibm.mqjms.jar` is the WebSphereMQ JMS client jar that is appended to the `pluginclasspath` and the `classpath` properties.

```plaintext
PluginPolicy.Progress.SonicMQ

pluginclasspath=C:\Progress115\OpenEdge\java\progress.jar,
C:\PROGRA~2\IBM\WEBSPH~1\java\lib\com.ibm.mqjms.jar

classpath=C:\Progress115\OpenEdge\java\progress.jar,
C:\PROGRA~2\IBM\WEBSPH~1\java\lib\com.ibm.mqjms.jar

jvmargs=-DsonicMQExtensions=false -DjmsProvider=WebSphereMQ
```

**Note:** SonicMQ is a string in the section header and `PluginPolicy.Progress.SonicMQ` is the name of the plug-in that OpenEdge provides, not the SonicMQ provider.

The provider JAR files are added to the classpath, and the run-time picks up the provider-specific connection factory objects.

### Configuring the Connection Factory

You can either update the `AdminServerPlugins.properties` file or use the Java Naming and Directory Interface (JNDI) to locate the connection factory. See Updating the classpath for your JMS Provider on page 43 for more details on updating the `AdminServerPlugins.properties` file.

### Using JNDI administered objects

You can locate the connection factory by using JNDI to find the JMS-administered objects in the JNDI namespace.

To prevent class conflict with the `AdminObjectFinder.java` file, the generic JMS adapter packages the class in a different package, `jmsfromABL(not jmsfrom4gl)`. This new class provides the adapter with the configuration to find the JMS-administered objects in the JNDI namespace.

1. Use the following example to create a `jmsfromABL.AdminObjectFinder.java` class file.
package jmsfromABL;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.TopicConnectionFactory;
import javax.jms.QueueConnectionFactory;
import javax.jms.Topic;
import javax.jms.Queue;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NameNotFoundException;
import javax.naming.NamingException;
import java.util.Hashtable;

public class AdminObjectFinder{

    public Context context = null;
    public AdminObjectFinder() throws Exception{
        Hashtable<String, String> env = new Hashtable<>();
        env.put(Context.INITIAL_CONTEXT_FACTORY,
                "com.sun.jndi.fscontext.RefFSContextFactory");
        env.put(Context.PROVIDER_URL, "file:/C:/JNDI");
        env.put(Context.SECURITY_PRINCIPAL, "username");
        env.put(Context.SECURITY_CREDENTIALS, "password");
        context = new InitialContext(env);
    }

    public TopicConnectionFactory getTopicConnectionFactory(String name) throws Exception {
        TopicConnectionFactory factory = null;
        factory = (javax.jms.TopicConnectionFactory)context.lookup(name);
        return factory;
    }

    public QueueConnectionFactory getQueueConnectionFactory(String name) throws Exception {
        QueueConnectionFactory factory = null;
        factory = (javax.jms.QueueConnectionFactory)context.lookup(name);
        return factory;
    }

    public ConnectionFactory getConnectionFactory(String name) throws NamingException {
        return (ConnectionFactory) context.lookup(name);
    }

    public Topic getTopic(String name) throws Exception {
        Topic topic = null;
        Object object = null;
        object = context.lookup(name);
        if (object != null) {
            topic = (javax.jms.Topic) object;
        }
        return topic;
    }

    public Queue getQueue(String name) throws Exception {
        Queue queue = null;
        Object object = null;
        object = context.lookup(name);
        if (object != null) {
            queue = (javax.jms.Queue) object;
        }
        return queue;
    }
}
public Destination getDestination(String name) throws NamingException {
    return (Destination) context.lookup(name);
}

env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.fscontext.RefFSContextFactory")

Specifies the name of the context factory class for WebSphereMQ.

Note: This example assumes that JNDI .bindings file is placed at a physical location in the
FileSystem and thus is referred to as RefFileSystemContextFactory. This .binding file contains
attributes that are used to create a ConnectionFactory object and other JNDI resources. For
WebSphereMQ, the attributes to create connection factory resides in a .bindings file and for
ActiveMQ, in the jndi.properties file.

env.put(Context.PROVIDER_URL, "file:/C:/JNDI")

Specifies the location of the JNDI .bindings file. For WebSphereMQ, the location of .bindings file
and for ActiveMQ, the location of jndi.properties file.

env.put(Context.SECURITY_PRINCIPAL, "username");
env.put(Context.SECURITY_CREDENTIALS, "password")

Specifies the security credentials to access the MQService.

Note:
• The jmsfromABL.AdminObjectFinder name is mandatory in the class file.
• The class and the get...() methods must be declared as a public element.

2. Compile the jmsfromABL.AdminObjectFinder.java class file and place it in a new JAR file.
3. Add the location of this new JAR file to the pluginclasspath and classpath variables in the
AdminServerPlugins.properties file, as in the following example:

[PluginPolicy.Progress.SonicMQ]
pluginclasspath=C:\Progress115\OpenEdge/java/progress.jar,
C:\PROGRA~2\IBM\WEBSPH~1\java\lib\com.ibm.mqjms.jar,
C:\AdminObjectFinder.jar
classpath=C:\Progress115\OpenEdge/java/progress.jar,
C:\PROGRA~2\IBM\WEBSPH~1\java\lib\com.ibm.mqjms.jar,
C:\AdminObjectFinder.jar
jvmargs=-DsonicMQExtensions=false -DjmsProvider=WebSphereMQ
Configuring BrokerConnect

The generic JMS adapter allows you to connect to SonicMQ, ActiveMQ, and WebSphereMQ JMS providers. JMS client connects to the JMS server differently for different JMS and thus needs different inputs as follows:

- SonicMQ connection requires the SonicMQ Broker URL such as `tcp://myhost:2506`
- ActiveMQ connection requires the ActiveMQ provider URL such as `tcp://sampleHost:61616`
- WebSphereMQ connection requires the QueueManager name as it is configured in the JNDI namespace.

As generic JMS adapter behaves as a proxy client for the JMS connection, the ABL API that creates connection with the JMS provider must provide an appropriate input for the provider.

If you wish to wish to migrate your applications from SonicMQ to ActiveMQ or WebSphereMQ, you must modify your ABL-JMS API to point to the appropriate MQ Broker URL as follows:

For WebSphereMQ:

```plaintext
RUN setBrokerURL IN psSessionH (INPUT "SampleQMgr").
```

where `SampleQMgr` is the name of the QueueManager or the connection factory for the queue or the topic that you want to use.

**Note:**

You can create the Connection Factory in the WebSphere MQClient Mode as `ConnectionFactory`, `QueueConnectionFactory`, or `TopicConnectionFactory`. When an ABL client tries to use the Connection Factory that you created, make sure that the ABL Session type and the corresponding `ConnectionFactory` that you are using to connect are compatible as listed below:

- Use `jmsSession` with `ConnectionFactory`, `TopicConnectionFactory` or `QueueConnectionFactory`.
- Use `pubsubSession` with only `TopicConnectionFactory`
- Use `ptpSession` with only `QueueConnectionFactory`  

For ActiveMQ:

```plaintext
RUN setBrokerURL IN psSessionH (INPUT "myhost:61616").
```

where `61616` is the port at which the ActiveMQ Broker URL is listening.

Non-supported API methods for Generic JMS Adapter

The JMS providers other than SonicMQ do not support all the capabilities and methods in this API. Here is a list of capabilities that are not supported:

- FaultTolerant
- Client Persistency
- Load Balancing
Non-supported API methods for Generic JMS Adapter

- Multi-part messages
- Message selector
- Acknowledge And Forward
- Failover
- FlowToDisk

**Note:** For a complete list of all methods in this API, see ABL - JMS API Reference on page 161

The generic JMS adapter does not support the following methods in Session objects:

- createChangeStateConsumer procedure
- createMultipartMessage procedure
- createRejectedMessageConsumer procedure
- getClientPersistence function
- getClientTransactionBufferSize function
- getConnectionURLs function
- getFaultTolerant function
- getFaultTolerantReconnectTimeout function
- getFlowToDisk function
- getInitialConnectionTimeout function
- getLocalStoreDirectory function
- getLocalStoreSize function
- getLocalStoreWaitTime function
- getReconnectInterval function
- getReconnectTimeout function
- getSelectorAtBroker function
- getSequential function
- getSingleMessageAcknowledgement function
- isFaultTolerant function
- setClientPersistence procedure
- setConnectionFile procedure
- setConnectionURLs procedure
- setFaultTolerant procedure
- setFaultTolerantReconnectTimeout procedure
- setFaultTolerantReconnectTimeout procedure
The generic JMS adapter does not support the following method in Message Consumer objects:

- acknowledgeAndForward procedure

The generic JMS adapter does not support the following methods in Message objects:

- addMessagePart procedure
- getBytesPartByID function
- getBytesPartByIndex function
- getContentType function
Introduction to messaging

The OpenEdge Adapter for SonicMQ provides access to the Java™ Messaging Service (JMS) APIs in the Sonic Environment. JMS is used for passing messages between different applications in a distributed environment.

The following sections describe:

• Point-to-Point (PTP) messaging on page 49 (Single sender and receiver of a message)
• Publish-and-Subscribe messaging on page 51 (Single sender and multiple receivers of a message)
• Comparing PTP and Pub/Sub messaging on page 53

For details, see the following topics:

• Point-to-Point (PTP) messaging
• Publish-and-Subscribe messaging
• Comparing PTP and Pub/Sub messaging
• Messages and message types

Point-to-Point (PTP) messaging

Point-to-Point, or PTP, is a domain of JMS messaging in which an application referred to as a sender sends a message to a destination called a queue. Another application, known as a receiver, receives that message from the queue. Messages in a queue are delivered in First-In, First-Out (FIFO) order. Once a message is delivered and acknowledged, the broker removes the message form the queue.
Sending and receiving messages

The following table describes the tasks performed to send and receive messages using PTP messaging.

Table 2: PTP messaging tasks

<table>
<thead>
<tr>
<th>Step</th>
<th>Who</th>
<th>What</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Receiver</td>
<td>Binds to a queue</td>
</tr>
<tr>
<td>2.</td>
<td>Sender</td>
<td>Creates and populates a message</td>
</tr>
<tr>
<td>3.</td>
<td>Sender</td>
<td>Sends the message to the queue</td>
</tr>
<tr>
<td>4.</td>
<td>Message broker</td>
<td>Removes the message from the queue and delivers it to the receiver</td>
</tr>
<tr>
<td>5.</td>
<td>Receiver</td>
<td>Consumes the message</td>
</tr>
<tr>
<td>6.</td>
<td>Receiver</td>
<td>Acknowledges message receipt to the broker</td>
</tr>
<tr>
<td>7.</td>
<td>Message broker</td>
<td>Deletes the message after it is acknowledged</td>
</tr>
</tbody>
</table>

PTP messaging options and features

The following table describes the features of the PTP messaging model.

Table 3: PTP messaging features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message delivery</td>
<td>Ensures a message is delivered only once to a single Message Consumer. The first message received by the broker is the first message delivered. This First In, First Out (FIFO) technique causes subsequent messages to endure until the previous message is consumed. Messages wait for a consumer until the message expires.</td>
</tr>
<tr>
<td>Message persistence</td>
<td>Persist messages on a queue based on the maximum size and threshold for the queue. The message remains even if the broker shuts down.</td>
</tr>
<tr>
<td>Static messaging queues</td>
<td>Messaging queues are created.</td>
</tr>
<tr>
<td>Single Message Consumer</td>
<td>There is only one Message Consumer for a given message. Many consumers can listen or receive on a queue, but only one takes delivery of a specific message.</td>
</tr>
</tbody>
</table>

1 Before starting your session, queues must be defined.
When the message is acknowledged as delivered by the consumer, it is removed from the queue permanently. No one else sees it and no one else receives it.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message acknowledgement</td>
<td>When the message is acknowledged as delivered by the consumer, it is removed from the queue permanently. No one else sees it and no one else receives it.</td>
</tr>
<tr>
<td>Prefetch count and threshold</td>
<td>A receiver can specify how many messages are to be delivered at a time.</td>
</tr>
<tr>
<td>Queue browsing</td>
<td>A receiver can look at the contents of messages on a queue without consuming the messages.</td>
</tr>
</tbody>
</table>

For more information, see Using PTP messaging on page 57.

### Building scalable server architecture with PTP queuing

A typical use of PTP messaging is to build a scalable and reliable server architecture. Both OpenEdge and non-OpenEdge clients send requests to a JMS queue on a broker. OpenEdge servers remove messages from the queue, execute the requests, and reply to the clients. Requests and replies do not get lost in the event of a system failure. Scalability is achieved by providing an increasing number of OpenEdge servers as the number of clients and the rate of requests increases. For an example, see Messaging Examples on page 333.

### Publish-and-Subscribe messaging

In the Publish-and-Subscribe, or Pub/Sub, JMS messaging domain, a message producer is a **publisher** and a **Message Consumer** is a **subscriber**. A publisher sends messages to a destination called a **topic**. Publishers publish messages to topics and subscribers subscribe to topics. A subscriber subscribes to topics and receives all messages published to those topics. All subscribers can consume messages for that topic. An application can be both a publisher and a subscriber, and a single publisher can send a message to multiple subscribers.

### Sending and receiving messages

The following table describes the tasks performed to send and receive messages using PTP messaging.

**Table 4: Pub/Sub messaging tasks**

<table>
<thead>
<tr>
<th>Step</th>
<th>Who</th>
<th>What</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Subscriber</td>
<td>Subscribes to a topic (a subscriber must exist prior to a message being published in order to consume the message)</td>
</tr>
<tr>
<td>2.</td>
<td>Session</td>
<td>Creates and populates a message</td>
</tr>
<tr>
<td>3.</td>
<td>Publisher</td>
<td>Publishes the message to the topic</td>
</tr>
<tr>
<td>4.</td>
<td>Message broker</td>
<td>Delivers the message to the subscribers</td>
</tr>
</tbody>
</table>
Pub/Sub messaging options and features

The following table describes the features of the Pub/Sub messaging model.

Table 5: Pub/Sub messaging features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Topic hierarchy</td>
<td>Topics can be organized into hierarchies</td>
</tr>
<tr>
<td>Guaranteed message delivery</td>
<td>A topic subscription can be durable; message remains when subscriber disconnects</td>
</tr>
<tr>
<td>Request and reply</td>
<td>Ensures the subscribers send a message back to the publisher when a message is received</td>
</tr>
<tr>
<td>Message selector</td>
<td>Filters the messages received from a topic</td>
</tr>
</tbody>
</table>

For more information, see Using PUB/SUB messaging on page 60.
Integrating with the native ABL publish-and-subscribe mechanism

The JMS Pub/Sub model complements ABL publish-and-subscribe syntax (named events) for distributed applications. As shown in the following figure, an ABL program written with the local ABL syntax for publish-and-subscribe can be distributed with the addition of local and remote gateway object modules. Using this model, an ABL programmer can integrate the local application with the SonicMQ functionality without recompiling. Progress Software Corporation recommends this model but does not provide specific software to implement it, except for the sample application files (see Messaging Examples on page 333.

Figure 1: Gateway model

Comparing PTP and Pub/Sub messaging

There are several distinguishing characters between the two message models. The following table compares PTP and Pub/Sub messaging.

Table 6: Comparing features

<table>
<thead>
<tr>
<th>PTP</th>
<th>Pub/Sub</th>
</tr>
</thead>
<tbody>
<tr>
<td>There is one consumer per message.</td>
<td>There are multiple consumers per message.</td>
</tr>
<tr>
<td>The receiver can browse the queue of undelivered messages.</td>
<td>Receivers only see delivered messages.</td>
</tr>
<tr>
<td>The Message broker balances the load of message delivery.</td>
<td>All subscribers receive messages (unless using shared subscription or message selectors).</td>
</tr>
<tr>
<td>The receiver controls the number of messages transferred from the broker.</td>
<td>The broker delivers one message at a time.</td>
</tr>
</tbody>
</table>
Message consumers can use message selectors to filter messages.

Support request/reply.

Queues permit Message Consumers to receive messages sent while disconnected.

## Messages and message types

A *Message* is the package of information sent from a producer to a receiver through the Message Consumer. The following table describes the parts of a message.

<table>
<thead>
<tr>
<th>Part</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header field</td>
<td>A predefined set of names and data</td>
</tr>
<tr>
<td>Property</td>
<td>Message property pairs set by the message producer</td>
</tr>
<tr>
<td>Message body</td>
<td>Message content formatted according to the message type</td>
</tr>
</tbody>
</table>

SonicMQ provides several standard JMS message types, plus the `XMLMessage` and `MultipartMessage` types. The following table lists the SonicMQ message types and content of each.

<table>
<thead>
<tr>
<th>SonicMQ message type</th>
<th>Message body</th>
</tr>
</thead>
<tbody>
<tr>
<td>HeaderMessage</td>
<td>No body—a header-only message that handles bodyless JMS messages</td>
</tr>
<tr>
<td>TextMessage</td>
<td>A standard Java string</td>
</tr>
<tr>
<td>MapMessage</td>
<td>A set of name/value pairs where values are Java primitives</td>
</tr>
<tr>
<td>StreamMessage</td>
<td>A stream of Java primitives</td>
</tr>
<tr>
<td>BytesMessage</td>
<td>A stream of uninterpreted bytes</td>
</tr>
<tr>
<td>MultiPartMessage</td>
<td>Zero or more parts—each part is either arbitrary (character or binary) data or a Sonic message</td>
</tr>
<tr>
<td>XMLMessage</td>
<td>XML tagged text. A SonicMQ extension of the TextMessage</td>
</tr>
</tbody>
</table>
For more information on messages, see the Working with messages.
Understanding the Messaging Models

The JMS messaging models are Point-to-Point (PTP) and Publish/Subscribe (Pub/Sub). PTP allows a message producer to send a message to one Message Consumer. Pub/Sub allows a message producer to send a message that is consumed by multiple consumers.

Programming instructions are provided in Programming for the OpenEdge Adapter for SonicMQ with the ABL - JMS API on page 109.

For details, see the following topics:

- Using PTP messaging
- Using PUB/SUB messaging

Using PTP messaging

PTP messaging allows you to produce a message to be consumed by one receiver. The receiver can specify how many messages to consume at a time, and define a threshold specifying when to consume messages. Also, the receiver can look at the contents of messages on a queue without consuming the messages. Undelivered messages remain on a queue even when there are no receivers. The messages are removed from the queue according to the message’s time-to-live.

The following general steps outline how to exchange messages from an OpenEdge application to a SonicMQ Broker for a PTP session:

**Note:** PTP messaging requires a queue.
1. Create a session procedure and connect to a SonicMQ Broker.
2. Send messages to a PTP queue.
3. Receive messages from a PTP queue.
4. Receive a reply.
5. Delete objects.

Creating a session procedure and connecting to a SonicMQ Broker

The following general steps outline how an OpenEdge application connects to a SonicMQ Broker for a PTP session:

1. The application runs \texttt{jms/jmssession.p} or \texttt{jms/ptpsession.p} persistently to instantiate the appropriate \texttt{Session} object and calls the \texttt{beginSession} procedure on page 180 to start the JMS session.
2. The application uses the handle of the Session object to create and send messages to a queue and to receive messages from a queue.
3. The application calls the \texttt{deleteSession} procedure on page 194 in the Session object to close the session and the underlying connection.

Sending messages to a PTP queue

The following general steps outline how an OpenEdge application connects to a SonicMQ Broker for a PTP session:

1. The application runs \texttt{jms/jmssession.p} or \texttt{jms/ptpsession.p} persistently to instantiate the appropriate \texttt{Session} object and calls the \texttt{beginSession} procedure to start the JMS session.
2. The application uses the handle of the Session object to create and send messages to a queue and to receive messages from a queue.
3. The application calls the \texttt{deleteSession} procedure in the Session object to close the session and the underlying connection.
4. The application obtains a handle to the PTP Session object.
5. The following general steps outline how an OpenEdge application sends a message to a queue. The application creates a message by calling one of the following procedures from the Session object: \texttt{createBytesMessage procedure} on page 184, \texttt{createDataSetMessage procedure} on page 186, \texttt{createHeaderMessage procedure} on page 186, \texttt{createMapMessage procedure} on page 186, \texttt{createMultipartMessage procedure} on page 188, \texttt{createStreamMessage procedure} on page 189, \texttt{createTempTableMessage procedure} on page 191, or \texttt{createXMLMessage procedure} on page 192.
6. The application populates the header fields, properties, and body of the message.
7. The application calls the \texttt{sendToQueue} procedure on page 266 in the Session object with the message handle and the name of a queue as input parameters.
8. The application can use the message one or more times and then deletes it.

Receiving messages from a PTP queue

The following general steps outline how an OpenEdge application receives a message from a queue:
1. The application obtains a handle to the PTP Session object.
2. The application creates a Message Consumer object by calling the createMessageConsumer procedure on page 187.
3. The application calls the receiveFromQueue procedure on page 262 in the Session object with the name of a queue and the Message Consumer handle as input parameters.
4. The application executes a \texttt{WAIT–FOR} statement (or calls a waitForMessages procedure on page 320) and processes incoming messages and other ABL (Advanced Business Language) events.
5. The application deletes the messages after it finishes using them.

Receiving a reply

The following general steps outline how an OpenEdge application receives a reply:

1. The application calls the requestReply procedure on page 264 in the Session object with the message handle, the name of a destination (a queue name for PTP), and the Message Consumer handle as input parameters.
2. The application executes a \texttt{WAIT–FOR} statement (or calls a waitForMessages procedure on page 320), which waits for the replies to arrive while processing other ABL events.
3. The Message Consumer object handles the replies.
4. The application deletes the replies after it finishes using them.

Temporary queues

A temporary queue allows an application to create and delete temporary queues on the current connection to the SonicMQ Broker. The SonicMQ Broker provides the name of the queue to the application. A temporary queue allows the SonicMQ Broker to hold JMS messages during the JMS session. Messages in a temporary queue are available to any application that knows the name of the temporary queue. A temporary queue is automatically deleted when the application that created it terminates the session. When the JMS session ends, any messages remaining in the temporary queue are deleted.

\textbf{Note:} A temporary queue can be used in OpenEdge client code or can be used by ABL code running in an AppServer.

Manage temporary queues by using the createTemporaryQueue procedure on page 190 and deleteTemporaryQueue procedure on page 194.

Deleting objects

An OpenEdge application must explicitly delete ABL objects after using them:

- The PTP Session object calls the deleteSession procedure on page 194.
- The Message Object calls the deleteMessage procedure on page 193.
- The Message Consumer object calls the deleteConsumer procedure on page 192.

In addition to deleting the objects, these calls delete the resources allocated by the OpenEdge Adapter for SonicMQ and the server-side resources.
Methods unique to Point-to-Point messaging

The following table lists the unique methods for Point-to-Point messaging.

Table 9: Unique PTP messaging methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sendToQueue procedure</td>
<td>Sends a message to a queue</td>
</tr>
<tr>
<td>receiveFromQueue procedure</td>
<td>Receives messages from a queue</td>
</tr>
<tr>
<td>browseQueue procedure</td>
<td>Allows applications to view messages in a queue without consuming them</td>
</tr>
</tbody>
</table>

For an example, see the PTP message example on page 109.

Using PUB/SUB messaging

Pub/Sub messaging allows you to produce a single message to be consumed by many receivers. A publisher sends messages to a topic. A receiver subscribes to those topics in which it is interested and receives all messages published to those topics.

The following general steps outline how to exchanged messages from an OpenEdge application to a SonicMQ Broker for a Pub/Sub session:

1. Create a session procedure and connect to a SonicMQ Broker.
2. Publish a message to a Pub/Sub topic.
3. Subscribe to a Pub/Sub topic and receive messages.
4. Send a message and receive a reply.
5. Delete objects.

Creating a session procedure and connecting to a SonicMQ Broker

The following general steps outline how an OpenEdge application connects to a SonicMQ Broker for a Pub/Sub session:

1. The application runs jms/jmssession.p or jms/pubsubsession.p persistently to instantiate the appropriate Session object.
2. The application sets connection parameters for SonicMQ.
3. The application calls the beginSession procedure on page 180 to connect to the OpenEdge Adapter for SonicMQ and the SonicMQ Broker and starts the JMS session.
4. The application uses the handle of the Session object to create and publish messages to topics and to subscribe to and receive messages from topics.

5. The application calls the `deleteSession` procedure on page 194 in the Session object to close the session and the underlying connection.

### Publishing a message to a Pub/Sub topic

The following general steps outline how an OpenEdge application publishes a message to a topic:

1. The application obtains a handle to the Pub/Sub Session object.
2. The application creates a message by calling one of the following in the Session object: `createBytesMessage` procedure on page 184, `createDataSetMessage` procedure on page 186, `createHeaderMessage` procedure on page 186, `createMapMessage` procedure on page 186, `createMultipartMessage` procedure on page 188, `createStreamMessage` procedure on page 189, `createTempTableMessage` procedure on page 191, or `createXMLMessage` procedure on page 192.
3. The application populates the header fields, properties, and body of the message.
4. The application calls the `publish` procedure on page 254 in the Session object with the message handle and the name of a topic as input parameters.
5. If the application is not going to use the message after publishing, it deletes the message.

### Subscribing to a Pub/Sub topic and receiving messages

The following general steps outline how an OpenEdge application subscribes to a topic and receives messages:

1. The application obtains a handle to the Pub/Sub Session object.
2. The application creates a Message Consumer object by calling the `createMessageConsumer` procedure on page 187.
3. The application calls the `startReceiveMessages` procedure on page 317 in the Session object with the message handle, the name of a destination (a topic name for Pub/Sub), and the Message Consumer handle as input parameters.
4. The application executes a `WAIT–FOR` statement (or calls a `waitForMessages` procedure on page 320) and processes incoming messages and other ABL events.
5. The application deletes the messages after the application finishes using them.

### Receiving a reply

The following general steps outline how an OpenEdge application receives a reply:

1. The application calls the `requestReply` procedure on page 264 in the Session object with the message handle, the name of a destination (a queue name for PTP), and the Message Consumer handle as input parameters.
2. The application executes a `WAIT–FOR` statement (or calls a `waitForMessages` procedure on page 320), which waits for the replies to arrive while processing other ABL events.
3. The Message Consumer object handles the replies.
4. The application deletes the replies after it finishes using them.
**Durable subscriptions**

Topics are destinations in Pub/Sub messaging. When messages are published, they are delivered to all active subscribers. Some subscribers register an interest in receiving messages sent while they were inactive. These are *durable subscriptions*. The broker notes the durable subscription and ensures that all messages from the topic’s publishers are retained until they either are acknowledged by the durable subscriber or have expired.

Durable subscriptions provide a mechanism to save messages for an unavailable client. Whenever a subscriber reconnects to the topic under the name it registered for its durable subscription, all undelivered messages to that topic that have not expired are delivered in order. The administrator can terminate durable subscriptions or a client can use the `cancelDurableSubscription` procedure on page 182 or the `subscribe` procedure on page 319 to close the durable subscription.

**Note:** A durable subscription is not allowed for a temporary topic.

**Temporary topic**

A temporary topic allows an application to create and delete temporary topic on the current connection to the SonicMQ Broker. The SonicMQ Broker provides the name of the temporary topic to the application. A temporary topic allows the SonicMQ Broker to hold JMS messages during the JMS session. Messages in a temporary topic are available to any application that knows the name of the temporary topic. A temporary topic is automatically deleted when the application that created it terminates the session. When the JMS session ends, any messages remaining in the temporary topic are deleted.

**Note:** A temporary topic can be used in OpenEdge client code or can be used by ABL code running in an AppServer.

Manage temporary topics by using the `createTemporaryTopic` procedure on page 190 and `deleteTemporaryTopic` procedure on page 195.

**Deleting objects**

An OpenEdge application must explicitly delete ABL objects after using them:

- The Pub/Sub Session object calls the `deleteSession` procedure on page 194.
- The Message object calls the `deleteMessage` procedure on page 193.
- The Message Consumer object calls the `deleteConsumer` procedure on page 192.

In addition to deleting the objects, these calls delete the resources allocated by the OpenEdge Adapter for SonicMQ and the server-side resources.

**Methods unique to Pub/Sub messaging**

The following table lists the unique methods for Pub/Sub messaging.
Table 10: Unique Pub/Sub messaging methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>publish procedure on page 254</td>
<td>Publishes a message to a topic</td>
</tr>
<tr>
<td>subscribe procedure on page 319</td>
<td>Subscribes to a topic</td>
</tr>
<tr>
<td>cancelDurableSubscription procedure on page 182</td>
<td>Cancels a durable subscription</td>
</tr>
</tbody>
</table>

For an example, see the Pub/Sub messaging example on page 113.
Implementing Messaging

In order to exchange messages using JMS messaging model, an OpenEdge client establishes a connection to a SonicMQ Broker, creates the message, and sends the message to a receiver.

Programming instructions are provided in Programming for the OpenEdge Adapter for SonicMQ with the ABL - JMS API on page 109.

For an alphabetical API reference, see ABL - JMS API Reference on page 161.

For details, see the following topics:

• Managing connections and sessions
• Externally managed connections
• Working with messages
• Consuming messages
• Transaction and recovery procedures
• Error and condition handling

Managing connections and sessions

In order to use the messaging capabilities of JMS and Sonic MQ, a connection must be created to the SonicMQ Broker. An active connection receives messages. A session is a single-threaded context for sending and receiving messages. Since ABL is single-threaded, there is no compelling reason for multiple sessions per connection, nor for exposing the distinction between sessions and connections. In the context of the ABL–JMS API, the term session refers to the combination of a session and a connection.
Note: In JMS, a Java client can create several sessions per connection.

When more than one session per connection is required (for example, to send and receive messages concurrently), a second session is used implicitly in the OpenEdge Adapter for SonicMQ, transparent to the ABL programmer.

The following sections describe:

• Creating a JMS session on page 66
• Deleting a JMS session on page 67
• Connection options on page 67
• Managing fail-over support on page 69
• Setting and getting JMS connection and session attributes on page 70
• Connecting to the OpenEdge Adapter for SonicMQ on page 70
• Load balancing on page 70
• Client persistence on page 70
• Fault tolerance on page 71
• Establishing session control on page 72
• Accessing message delivery parameters on page 72
• Request/Reply on page 73
• Message selectors on page 74

Creating a JMS session

These are the general steps to create a JMS session in ABL:

1. Run jms/jmssession.p, jms/pubsubsession.p or jms/ptpsession.p persistently with the OpenEdge Adapter for SonicMQ connection parameters as INPUT CHAR parameters.
2. (Optional) Set JMS attributes and parameters by calling internal procedures in the session procedure.
3. Start the actual JMS connection and session by calling the beginSession procedure on page 180.

Note: Session attributes cannot be modified after calling the beginSession procedure on page 180.

Creating multiple sessions

An OpenEdge application can create and connect to multiple Session objects concurrently. You must create separate sessions to connect to each domain (PTP or Pub/Sub) with a separate SonicMQ Broker or a single SonicMQ Broker’s unified domain.

Note: It is recommended that you use the JMS session domain and minimize the number of Session objects. Each session represents a separate SonicMQ client session and you want to minimize the number of SonicMQ client sessions.
Deleting a JMS session

An application calls the deleteSession procedure on page 194 in the Session object to close and delete the session. This call terminates the underlying JMS connection and sessions, disconnects the OpenEdge client from the OpenEdge Adapter for SonicMQ, deletes all the Message Consumer objects, and deletes the session's persistent procedure.

The deleteSession procedure on page 194 call does not delete the ABL Message objects associated with the session; those messages remain for possible use with other sessions.

Connection options

The OpenEdge Adapter for SonicMQ supports most of the same connection options as does the OpenEdge AppServer. You specify the desired options as the value of the adapterConnection parameter. The following table lists and explains the valid formats for expressing these options.

Table 11: OpenEdge Adapter for SonicMQ connection options

<table>
<thead>
<tr>
<th>Connection option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-H [host_name</td>
<td>IP-address]</td>
</tr>
<tr>
<td>-S [port-number</td>
<td>service-name ]</td>
</tr>
<tr>
<td>-DirectConnect</td>
<td>If included, causes the -H and -S parameters to be interpreted as the network address and TCP/IP port number of a specific OpenEdge Adapter for SonicMQ. If the -DirectConnect switch is omitted, the -H and -S parameters are interpreted as the network address and UDP port number of a NameServer.</td>
</tr>
<tr>
<td>-ssl</td>
<td>If included, specifies a secure connection to an SSL-enabled OpenEdge Adapter for SonicMQ. For more information, see OpenEdge Getting Started: Core Business Services.</td>
</tr>
<tr>
<td>-URL Web-or-AppServer-path</td>
<td>An HTTP- or HTTPS-based URL to an AppServer Internet Adapter (AIA), or an AppServer-based URL to a OpenEdge Adapter for SonicMQ to which you connect either directly or through a NameServer. This URL is identical in format to the URL used to connect Open Clients to an AppServer. The use of the -URL option</td>
</tr>
</tbody>
</table>
### Connection options

<table>
<thead>
<tr>
<th>Connection option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>precludes the use of the -H, -S, -DirectConnect, and -ssl options. The -URL parameter contains the necessary host and port information; it provides equivalent support for direct connections and secure SSL connections via AppServerDC, AppServerDCS, AppServerS, and HTTPS. For more information on the -URL connection option, see OpenEdge Application Server: Developing AppServer Applications.</td>
</tr>
<tr>
<td><strong>-nohostverify</strong></td>
<td>If included, turns off host verification for a secure (SSL) connection. In the case of an SSL connection, unless this switch is included, the client compares the host name (specified in the -H parameter or the -URL parameter) with the Common Name specified in the server certificate, and raises an error if they do not match. With -nohostverify in effect, the client never raises the error. This option works only in the context of a secure connection; that is, in combination with the -ssl switch or with an HTTPS, AppServerS, or AppServerDCS parameter to the -URL switch. For more information, see OpenEdge Getting Started: Core Business Services.</td>
</tr>
<tr>
<td><strong>-nosessionreuse</strong></td>
<td>If included, prevents the application from reusing the session ID when reconnecting to the same SSL-enabled OpenEdge Adapter for SonicMQ. This option works only in the context of a secure (SSL) connection; that is, in combination with the -ssl switch or with an HTTPS, AppServerS, or AppServerDCS parameter to the -URL switch. For more information, see OpenEdge Getting Started: Core Business Services.</td>
</tr>
<tr>
<td><strong>-pf pathname</strong></td>
<td>Specifies a text file containing any of the AppServer connection parameters described in this table. Any other OpenEdge startup parameters in the file are ignored.</td>
</tr>
<tr>
<td><strong>-SMQConnect</strong></td>
<td>Creates a direct connection to the SonicMQ Broker using the jmsession.p Session object.</td>
</tr>
</tbody>
</table>

The following table shows several examples of valid adapterConnection parameters.
### Table 12: Connection parameter examples

<table>
<thead>
<tr>
<th>Connection parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;&quot;</td>
<td>By default, connection to the NameServer running on UDP port 5162 on localhost</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;-H host1 -S 5163&quot;</td>
<td>Connection to the NameServer running on UDP port 5163 on the machine host1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;-H fortress -S 3621 -DirectConnect -ssl -nohostverify&quot;</td>
<td>Secure connection directly to the SSL-enabled OpenEdge Adapter for SonicMQ running on TCP/IP port 3621 on the machine fortress, with host verification disabled</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;-URL <a href="http://host1:3099/external/aia1?adapter.progress.jms">http://host1:3099/external/aia1?adapter.progress.jms</a>&quot;</td>
<td>Connection by a WebClient via HTTP protocol to an AIA running on the machine host1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;-URL AppServerDCS://fortress:3621/-nosessionreuse&quot;</td>
<td>Secure connection via AppServer protocol directly to an SSL-enabled OpenEdge Adapter for SonicMQ, with session reuse disabled</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>RUN jms/jmsession.p PERSISTENT SET sessionH (&quot;-SMQConnect&quot;).</td>
<td>Connects directly to the SonicMQ Broker allowing better connection and session management</td>
</tr>
</tbody>
</table>

See the following manuals for more information:

- *OpenEdge Application Server: Developing AppServer Applications* for details on URL-based connections. Although the discussion is in the context of AppServer connections, the same syntax rules apply to all OpenEdge Adapter for SonicMQ connections.

- *OpenEdge Application Server: Administration* (and online help for the Progress Explorer tool) for instructions on enabling the OpenEdge Adapter for SonicMQ for SSL connections.

- *OpenEdge Getting Started: Core Business Services* for a comprehensive discussion of SSL connections, management of keys and certificates, and other security considerations in OpenEdge.

### Managing fail-over support

Sonic allows a client to specify a list of Sonic brokers to connect. This makes it easier for the client to establish a connection when one or more brokers are not available. Sonic also allows the application to specify whether to try connecting to the brokers in the list sequentially or randomly. The following table lists the methods for managing broker connections.

### Table 13: Managing fail-over support

<table>
<thead>
<tr>
<th>Setting</th>
<th>Getting</th>
</tr>
</thead>
<tbody>
<tr>
<td>setConnectionURLs procedure on page 276</td>
<td>getConnectionURLs function on page 205 setSequential function on page 242</td>
</tr>
<tr>
<td>setSequential procedure on page 310</td>
<td></td>
</tr>
</tbody>
</table>
Setting and getting JMS connection and session attributes

After creating the session procedure, the application specifies connection and session attributes and retrieves values related to connection and session attributes. The following table lists the methods for handling connection and session attributes.

Table 14: Connection and session attributes

<table>
<thead>
<tr>
<th>Setting procedure/function</th>
<th>Getting function/procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>setAdapterService</code> on page 267</td>
<td><code>getAdapterService</code> function on page 196</td>
</tr>
<tr>
<td><code>setJMSServerName</code> procedure on page 293</td>
<td><code>getJMSServerName</code> function on page 223</td>
</tr>
<tr>
<td><code>setBrokerURL</code> procedure on page 270</td>
<td><code>getBrokerURL</code> function on page 197</td>
</tr>
<tr>
<td><code>setPingInterval</code> procedure on page 301</td>
<td><code>getConnectionMetaData</code> function on page 205</td>
</tr>
<tr>
<td><code>setUser</code> procedure on page 316</td>
<td><code>getUser</code> function on page 248</td>
</tr>
<tr>
<td><code>setPassword</code> procedure on page 301</td>
<td>For more information, see <code>MultipartMessage</code> on page 88 and the discussion of the <code>getPassword</code> function on page 234 in the <code>MultipartMessage</code> example on page 356.</td>
</tr>
<tr>
<td><code>setClientID</code> procedure on page 273</td>
<td><code>getClientID</code> function on page 202</td>
</tr>
<tr>
<td><code>setTransactedReceive</code> procedure on page 315</td>
<td><code>getTransactedReceive</code> function on page 247</td>
</tr>
<tr>
<td><code>setTransactedSend</code> procedure on page 316</td>
<td><code>getTransactedSend</code> function on page 247</td>
</tr>
</tbody>
</table>

Connecting to the OpenEdge Adapter for SonicMQ

After setting the previously described attributes as required, the application starts the JMS session and connection using the `beginSession` procedure on page 180.

Load balancing

Sonic supports the creation of load-balanced clusters. By default, connect-time load balancing is enabled for all SonicMQ Brokers within a cluster. When load balancing is in effect, connection requests can be redirected to other brokers in the cluster for more efficient processing.

To manage load balancing for the current request, use the `setLoadBalancing` procedure on page 294 and the `getLoadBalancing` function on page 224.

Client persistence

Client persistence provides a higher level of reliability than is defined in the JMS specification. *Client persistence* allows the JMS session to continue sending messages regardless of the SonicMQ Broker status. If the SonicMQ Broker is not available, the messages are stored locally and sent when the SonicMQ Broker becomes available.
**Storing undeliverable messages**

When the connection to the SonicMQ Broker fails, messages are persisted to disk, and replayed when the connection is re-established. Each connection must have a local directory specified where messages will be stored when a connection fails.

The following table lists the methods for managing client persistence.

**Table 15: Managing client persistence**

<table>
<thead>
<tr>
<th>Setting</th>
<th>Getting</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>setLocalStoreDirectory</code> procedure on page 294</td>
<td><code>getLocalStoreDirectory function</code> on page 225</td>
</tr>
<tr>
<td><code>setLocalStoreSize</code> procedure on page 295</td>
<td><code>getLocalStoreSize function</code> on page 225</td>
</tr>
<tr>
<td><code>setReconnectTimeout</code> procedure on page 304</td>
<td><code>getReconnectTimeout function</code> on page 237</td>
</tr>
<tr>
<td><code>setReconnectInterval</code> procedure on page 304</td>
<td><code>getReconnectInterval function</code> on page 236</td>
</tr>
<tr>
<td><code>setClientPersistence</code> procedure on page 273</td>
<td><code>getClientPersistence function</code> on page 202</td>
</tr>
<tr>
<td><code>createRejectedMessageConsumer</code> procedure on page 188</td>
<td></td>
</tr>
</tbody>
</table>

Additionally, client persistence requires using the `setClientID` procedure on page 273. The `clientID` must be unique for each client. The application may optionally call the `setPingInterval` procedure on page 301 to enable connection checking between the client and the SonicMQ Broker.

---

**Fault tolerance**

Fault tolerant connections allow a JMS client to reconnect to a SonicMQ Broker and enable reconnection to the same SonicMQ Broker, or to one of the SonicMQ Broker specified in a list, if this has been defined before the session is created. Fault tolerance is set on the client but must be supported by the SonicMQ Broker.

**Note:** Fault tolerance is only available to OpenEdge clients running in ClientConnect and ServerConnect mode.

An OpenEdge client specifies participation in a fault tolerant session when the client connects to licensed fault tolerant SonicMQ Brokers. In a fault tolerant session, when the SonicMQ Broker or the network experiences a fault, the session resumes when the SonicMQ Broker or its backup is available. The client maintains connection and session information waiting for the SonicMQ Broker to be available.
Replicated SonicMQ Brokers

Replicated SonicMQ Brokers provide additional broker availability. The active SonicMQ Broker and the replicated SonicMQ Broker synchronize all client information and data. If the active SonicMQ Broker goes down, the replicated SonicMQ Broker takes over as the lead broker. Clients running a fault tolerant connection seamlessly connect to the replicated SonicMQ Broker.

The following table lists the methods for managing fault tolerant connections.

Table 16: Managing fault tolerance

<table>
<thead>
<tr>
<th>Setting</th>
<th>Getting</th>
</tr>
</thead>
<tbody>
<tr>
<td>setFaultTolerant procedure on page 285</td>
<td>isFaultTolerant function on page 251</td>
</tr>
<tr>
<td>setClientTransactionBufferSize procedure on page 274</td>
<td>getFaultTolerant function on page 213</td>
</tr>
<tr>
<td>setInitialConnectionTimeout procedure on page 290</td>
<td>getClientTransactionBufferSize function on page 203</td>
</tr>
<tr>
<td>setFaultTolerantReconnectTimeout procedure on page 286</td>
<td>getInitialConnectionTimeout function on page 217</td>
</tr>
<tr>
<td>createChangeStateConsumer procedure on page 185</td>
<td>getFaultTolerantReconnectTimeout function on page 214</td>
</tr>
</tbody>
</table>

Establishing session control

The following table lists the methods an application uses to manage session control.

Table 17: Setting session methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>startReceiveMessages procedure on page 317</td>
<td>Starts receiving messages after creating a new session or after calling stopReceiveMessages procedure on page 318</td>
</tr>
<tr>
<td>stopReceiveMessages procedure on page 318</td>
<td>Causes the SonicMQ Broker to stop delivering messages to the OpenEdge client</td>
</tr>
<tr>
<td>deleteSession procedure on page 194</td>
<td>Closes a session and its underlying connection and deletes the session procedure</td>
</tr>
<tr>
<td>getConnectionID function on page 204</td>
<td>Returns the AppServer connection ID</td>
</tr>
</tbody>
</table>

Accessing message delivery parameters

Message delivery parameters set on the Session object are used as defaults for all messages sent in that session. The default can be changed by setting the parameters of the call to publish procedure on page 254, the call to sendToQueue procedure on page 266, or the call to requestReply procedure on page 264. These values cannot be changed after the beginSession procedure on page 180 is called.

The following table lists the methods for setting and getting delivery parameters.
### Table 18: Setting and getting delivery parameters

<table>
<thead>
<tr>
<th>Setting</th>
<th>Getting</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>setDefaultPriority</code> procedure on page 282</td>
<td><code>getDefaultPriority</code> function on page 212</td>
</tr>
<tr>
<td><code>setDefaultTimeToLive</code> procedure on page 283</td>
<td><code>getDefaultTimeToLive</code> function on page 212</td>
</tr>
<tr>
<td><code>setDefaultPersistency</code> procedure on page 281</td>
<td><code>getDefaultPersistency</code> function on page 211</td>
</tr>
</tbody>
</table>

### Setting the maximum number of messages

The default maximum number of active JMS messages in an OpenEdge session is 50. This is the maximum number of messages that have been created, but not deleted by the application. To change the default, the main procedure of the OpenEdge application must include the definition of `JMS-MAXIMUM-MESSAGES` global variable on page 252.

**Note:** If you exceed the limit on the `JMS-MAXIMUM-MESSAGES` global variable on page 252, an error is returned.

### Discardable messages

SonicMQ supports a `DISCARDABLE` message delivery mode. For non-transacted Pub/Sub sessions, `DISCARDABLE` delivers all messages to subscribers that are keeping up with the flow of messages, but drops the oldest messages waiting for lagging subscribers when new messages arrive, under any of the following conditions:

- When the message server's internal buffers for that subscriber session are full
- When a neighbor cluster member containing a Topic subscription is unavailable and a subscriber is located on the other cluster member
- When an intended durable subscriber is unavailable

An application controls message delivery mode using the `publish` procedure on page 254 and the `setDefaultPersistency` procedure on page 281.

### Request/Reply

*Request/Reply* is a mechanism for the `JMSReplyTo` message header field to specify the destination where replies to a message should be sent. To specify the message destination, use the `requestReply` procedure on page 264.

**Note:** The term *destination* refers to both topics and queues.

Java–JMS supports a manual approach through the `JMSReplyTo` field, whereas the ABL–JMS implementation automates the request/reply sequence by:

- Sending a reply by setting the `reply` output parameter of the message handler
- Requesting a reply by calling the `requestReply` procedure on page 264 with a handle to a Message Consumer for the reply
The ABL–JMS implementation uses a temporary destination for the reply. It is an error to set the JMSReplyTo field of the message explicitly if requestReply procedure on page 264 is used. The reply is received by the Message Consumer asynchronously, just like any other message. The temporary destination is deleted when the Message Consumer object is deleted.

Message selectors

SonicMQ messages can be filtered so that only messages meeting a specific criteria will be received. Message consumers in both domains can apply message selectors to filter messages. Message selectors filter messages so a client does not receive all the messages. Message selectors evaluate message header fields and properties. They do not access the body of a message.

The default behavior of message selector filtering is:

• For PTP sessions, the filtering is always performed by the SonicMQ Broker
• For Pub/Sub sessions, all messages for a subscribed topic are by default delivered to the subscriber, then the filter is applied by the SonicMQ client to decide which messages to consume

To have the SonicMQ Broker perform the filtering for a Pub/Sub session, use the setSelectorAtBroker procedure on page 309 and getSelectorAtBroker function on page 242. Choosing to perform message selection at the SonicMQ Broker reduces message traffic between the broker and the client but increases the workload of the SonicMQ Broker.

Note: Server-based message selectors are available with all adapters.

Externally managed connections

Client applications can dynamically adjust to redefinition of the broker connections and the destinations where messages are sent and received. This is achieved when client applications look up connection information in serialized connection objects or a store of JMS administered objects. The following sections describe:

• Using serialized connection objects on page 74
• Finding administered objects in JNDI or proprietary directories on page 82

Using serialized connection objects

A serialized connection object contains all the connection information required by a client to connect to a SonicMQ Broker, including userid and password. A SonicMQ administrator creates the serialized connection object as a file using the Sonic Management Console and provides the serialized connection object to the OpenEdge client. The OpenEdge client uses the setConnectionFile procedure on page 275 with the file containing the serialized connection object when creating the messaging session.

The serialized connection object file is used when connecting to a SonicMQ Broker. The following example shows how to use the serialized connection object file MyConnectionObject.sjo:

```plaintext
RUN jms/jmsession.p PERSISTENT SET hSession ("-SMQConnect").
RUN setConnectionFile IN hSession ("MyConnectionObject.sjo").
RUN beginSession IN hSession.
```
Connection file parameters

A connection file allows you to configure a set of connection parameters. The following table lists the connection parameters that can be set for each connection type.

Table 19: Connection file parameters (Continued)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Connection type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Broker-side selectors</td>
<td>All²</td>
</tr>
<tr>
<td>Client ID</td>
<td>All</td>
</tr>
<tr>
<td>Connection URL</td>
<td>All</td>
</tr>
<tr>
<td>Connect ID</td>
<td>All</td>
</tr>
<tr>
<td>Default password</td>
<td>All</td>
</tr>
<tr>
<td>Default user name</td>
<td>All</td>
</tr>
<tr>
<td>Durable message order</td>
<td>All</td>
</tr>
<tr>
<td>Flow to disk</td>
<td>All</td>
</tr>
<tr>
<td>Load Balancing</td>
<td>All</td>
</tr>
<tr>
<td>Max delivery count</td>
<td>All</td>
</tr>
<tr>
<td>Persistent delivery mode</td>
<td>All</td>
</tr>
<tr>
<td>Ping interval</td>
<td>All</td>
</tr>
<tr>
<td>Prefetch count</td>
<td>All</td>
</tr>
<tr>
<td>Prefetch threshold</td>
<td>All</td>
</tr>
<tr>
<td>Sequential</td>
<td>All</td>
</tr>
<tr>
<td>Client transaction buffer size</td>
<td>Client-connect, Server-connect</td>
</tr>
<tr>
<td>Enable fault tolerant</td>
<td>Client-connect, Server-connect</td>
</tr>
<tr>
<td>Enable local store</td>
<td>Client-connect, Server-connect</td>
</tr>
<tr>
<td>Fault tolerant reconnect timeout</td>
<td>Client-connect, Server-connect</td>
</tr>
<tr>
<td>Initial connect timeout</td>
<td>Client-connect, Server-connect</td>
</tr>
<tr>
<td>Local store directory</td>
<td>Client-connect, Server-connect</td>
</tr>
</tbody>
</table>

² All connections include Broker-connect, Client-connect, Server-connect
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Connection type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local store size</td>
<td>Client-connect, Server-connect</td>
</tr>
<tr>
<td>Reconnect interval</td>
<td>Client-connect, Server-connect</td>
</tr>
<tr>
<td>Reconnect timeout</td>
<td>Client-connect, Server-connect</td>
</tr>
</tbody>
</table>

Note: The combined length of the Client ID, Default password, and other connection file parameters listed in the table above must not exceed 30,000 non-Unicode characters when connecting to an AppServer.

Parameter values are fixed when the `beginSession procedure` on page 180 is called. When the `beginSession procedure` on page 180 is called, the following process takes place for each parameter to determine its value:

1. If the parameter value was set by a call to the corresponding `set<parameter>` routine, the values set in the call is used. If the same `set<parameter>` routine is called more than once, the last setting is used.

2. If the parameter value was not set by a `set<parameter>` routine, and a parameter file is specified, and the parameter is set in the connection file, then the value in the connection file is used.

3. If the parameter is not set by a `set<parameter>` routine, and it is not set in a specified connection file, then the default value is used.
The following figure depicts the process for determining the parameter value for each parameter during execution of the `beginSession` procedure on page 180.

**Figure 2: Connection parameter value decision flow**

All connection parameters must be set prior to calling `beginSession`.

### Creating serialized connection objects

The Sonic Management Console allows an administrator to create serialized connection objects and save them as a file.

To create a serialized connection object file:

1. Ensure the Domain Manager for SonicMQ is running.
2. Ensure the Sonic Management Console is started and connected to the Domain Manager.
3. On the Sonic Management Console menu bar, select **Tools** and then **JMS Administered objects**.
4. Select the **File System** and navigate to the **Directory** you want the serialized connection object file to reside, as shown:

![Image of Sonic Management Console with File System selected](image.png)

5. Select the **Connect** button.

6. In the left pane, select the Object Store directory that you created and then select the **Connection Factories** tab, as shown:
7. Select the **New** button and enter the required connection information for **Lookup Name** (name of serialized connection object file), **Connection URL(s)**, **Default User Name**, **Default Password**, and **Confirm Password**, as shown:
Note: All other information is optional for the connection object.

8. Select **Update**. The serialized connection object appears an entry, as shown:
The serialized connection object file `MyConnectionObject.sjo` exists in the Object Stores specified directory. An OpenEdge client uses the file `MyConnectionObject.sjo` to connect to the SonicMQ Broker.
Finding administered objects in JNDI or proprietary directories

A JMS-administered object is an object created by a JMS administrator and registered with a directory (typically a JNDI-compliant directory) under a name that is meaningful to the JMS clients. The object contains JMS configuration information that is created by a JMS administrator and later used by JMS clients. Java Naming and Directory Interface (JNDI) is an interface for JMS administrators to create and configure administered objects and store them in a namespace.

The SonicMQ-administered objects are:

- TopicConnectionFactory
- QueueConnectionFactory
- Topic
- Queue

For example, the administrator creates a TopicConnectionFactory object, which contains all the JMS server connection parameters (communication protocol host and port), assigns it a name, and stores it in a JNDI directory. The client does not have to know the connection parameters to connect to the JMS server. The client finds the object by name in the directory and uses it to create connection objects. The administrator can change the connection parameters later without affecting client applications.

The administrator can give the Topic and Queue objects meaningful aliases to shield the client from their internal names. For example, a topic with the internal JMS name of sports.USA.Northeast.golf could be stored in the directory under northern.golfers. For more information on administered objects, see the Java Message Service specification, SonicMQ Programming Guide, and SonicMQ Configuration and Management Guide.

Using the OpenEdge Adapter for the SonicMQ and the ABL - JMS API with administered objects

JMS does not impose any specific directory for storing administered objects (although it establishes the convention of using JNDI-compliant directories, such as LDAP). Also, the process of connecting to a JNDI server and obtaining an initial context is not standardized.

Therefore, to use directory-stored JMS objects, you must implement a Java class, compile it, and install the class file on the OpenEdge Adapter for the SonicMQ host under the OpenEdge Adapter for the SonicMQ’s CLASSPATH. (For more information on CLASSPATH, see Setting the CLASSPATH on page 83.) The OpenEdge Adapter for the SonicMQ looks for that class when it starts up. If it finds the class, it creates an instance object of it and uses it to locate administered objects. If it does not find the class, the OpenEdge Adapter for the SonicMQ creates objects as required.

jmsfrom4gl.AdminObjectFinder class

The following code is the skeleton of the jmsfrom4gl.AdminObjectFinder class. Use it as a template to create a class file and install it on the OpenEdge Adapter for the SonicMQ host; Unified Broker host for BrokerConnect, OpenEdge client host for ClientConnect, and AppServer or WebSpeed Transaction server host for the ServerConnect option.

The jmsfrom4gl.AdminObjectFinder name is mandatory. The class and the get...() methods must be declared public. The AdminObjectFinder class must be part of the jmsfrom4gl package and placed in a directory called jmsfrom4gl. The directory that contains jmsfrom4gl must be on the CLASSPATH of the OpenEdge Adapter for SonicMQ.
For example:

```java
package jmsfrom4gl;
import javax.jms.TopicConnectionFactory;
import javax.jms.QueueConnectionFactory;
import javax.jms.Topic;
import javax.jms.Queue;
public class AdminObjectFinder
{
    public TopicConnectionFactory getTopicConnectionFactory(String name)
        throws Exception
    {
        TopicConnectionFactory factory = null;
        // Write code to populate factory
        return factory;
    }
    public QueueConnectionFactory getQueueConnectionFactory(String name)
        throws Exception
    {
        QueueConnectionFactory factory = null;
        // Write code to populate factory
        return factory;
    }
    public Topic getTopic(String name)
        throws Exception
    {
        Topic topic = null;
        // Write code to populate topic
        return topic;
    }
    public Queue getQueue(String name)
        throws Exception
    {
        Queue queue = null;
        // Write code to populate queue
        return queue;
    }
}
```

**Note:** The `brokerURL` startup parameter is used as the input parameter for the `getTopicConnectionFactory` and `getQueueConnectionFactory` methods. For example, if the OpenEdge application calls the `setBrokerURL` procedure on page 270 passing in the input parameter `directory_factory_name`, the ABL–JMS implementation on the server side calls the `getTopicConnectionFactory` method with `directory_factory_name` as the parameter. If the `getTopicConnectionFactory` and `getQueueConnectionFactory` methods are implemented, the `jmsServerName` startup parameter is ignored (since the identity of the server’s vendor is encapsulated in the object). It is sufficient to implement methods for those objects that should be obtained from the directory. For example, it is legal to have an `AdminObjectFinder` class with only the `getTopicConnectionFactory` method. The ABL–JMS implementation looks for the methods dynamically and does not fail if the other methods are missing. If the object finder method returns null, the ABL–JMS implementation tries to create the object as if the method is not there. For more information on `CLASSPATH`, see Setting the `CLASSPATH` on page 83.

**Setting the CLASSPATH**

In Windows and on UNIX, you can set the `CLASSPATH` by using the `PluginPolicy.Progress.SonicMQ` section in the `AdminServerPlugins.properties` file. `BrokerConnect` uses the `pluginclasspath` property. `ClientConnect` and `ServerConnect` use the `classpath` property.
Working with messages

SonicMQ supports several types of messages with different formats. Each message type represents the message body in a different format.

The following sections describe:

- Message life cycle on page 84
- Creating, populating, and accessing messages on page 85
- Sending messages to a queue on page 93
- Publishing messages to a topic on page 94
- Clearing messages on page 94
- Deleting messages on page 94
- Accessing message header properties on page 94
- Accessing message properties on page 95

Message life cycle

An ABL–JMS message has a life cycle for sending and a life cycle for receiving.

Managing the sending life cycle of an ABL - JMS message

To manage the sending life cycle of an ABL–JMS message:

1. Create a message by running one of the following procedures in the Session object: `createBytesMessage procedure` on page 184, `createDataSetMessage procedure` on page 186, `createHeaderMessage procedure` on page 186, `createMapMessage procedure` on page 186, `createMultipartMessage procedure` on page 188, `createStreamMessage procedure` on page 189, `createTempTableMessage procedure` on page 191, or `createXMLMessage procedure` on page 192.

2. Populate a message by running `set...` and `write...` for header and data information.

3. Send the message to a destination.

4. Run the `deleteMessage procedure` on page 193 to delete the message.

Managing the receiving life cycle of an ABL - JMS message

To manage the receiving life cycle of an ABL–JMS message:

1. Receive a message in a Message Consumer object.

2. Run `get...` and `read...` to extract header information and body data.

3. Run the `deleteMessage procedure` on page 193 to delete the message.
Creating, populating, and accessing messages

The Session object provides a method for creating each type of message. Each message provides methods for setting the content of the message body. The following sections describe:

- **TextMessage** on page 85
- **HeaderMessage** on page 86
- **MapMessage** on page 86
- **StreamMessage** on page 87
- **BytesMessage** on page 88
- **MultipartMessage** on page 88
- **XMLMessage** on page 89
- **DataSetMessage** on page 90
- **TempTableMessage** on page 90
- **Java Object messages** on page 90
- **Message size limits** on page 91
- **Storing and extracting data** on page 91

**TextMessage**

A **TextMessage** is a message type whose body contains text data.

The following table lists the methods for handling text messages.

<table>
<thead>
<tr>
<th>Create method</th>
<th>Populate method</th>
<th>Access method</th>
</tr>
</thead>
<tbody>
<tr>
<td>createTextMessage procedure on page 191</td>
<td>setText procedure on page 315</td>
<td>eofStream function on page 195</td>
</tr>
<tr>
<td></td>
<td>setLongText procedure on page 298</td>
<td>getCharCount function on page 201</td>
</tr>
<tr>
<td></td>
<td>appendText procedure on page 180</td>
<td>getText function on page 245</td>
</tr>
<tr>
<td></td>
<td>reset procedure on page 265</td>
<td>getTextSegment function on page 246</td>
</tr>
<tr>
<td></td>
<td>clearBody procedure on page 182</td>
<td>getLongText function on page 229</td>
</tr>
<tr>
<td></td>
<td></td>
<td>getLongTextCP function on page 229</td>
</tr>
</tbody>
</table>

For any **TextMessage** smaller than 32K, text data can be extracted and stored in a message by the getText function and the setText procedure method. For a **TextMessage** longer than 32K, the setLongText procedure and the getLongText function are available. Use of these calls is recommended for new code developed to process large character strings.

**Note:** You can continue to use the appendText procedure and the getTextSegment function provided in earlier versions of the OpenEdge Adapter for SonicMQ, when ABL imposed a 32K character limit on text strings. However, programming with these older calls is significantly more complex than using the newer setLongText procedure and the getLongText function.
The appendText procedure and the getTextSegment function concatenate text segments. With multiple appendText procedure calls, an OpenEdge client can create a TextMessage up to the limit of the JMS server. The JMS non-OpenEdge client receives a single TextMessage resulting from the concatenation of all the text segments.

The OpenEdge Adapter for SonicMQ divides the received TextMessage into text segments of 8K (8192) or fewer characters. An application can then use multiple getTextSegment function calls to retrieve these segments. If getText function is called instead, the ABL–JMS API returns all of the text, and a run-time error occurs if the TextMessage is too large for the ABL interpreter to handle. An application can use the getCharCount function call to determine the total number of characters in a message.

For example, if the message value is UNKNOWN, or "", or a String of 5,000 characters, an application can use one getText function call (or one getTextSegment function call). If the message size is 16,400 characters, the first two getTextSegment function calls return 8192 characters each, and the last getTextSegment function call returns 16 characters.

The endOfStream function function returns true when all of the segments are retrieved (that is, when the number of getTextSegment function calls matches the number of segments). The setText procedure call implicitly calls clearBody procedure before setting the new text. The reset procedure and getText function calls transfer the message from write-only to read-only mode and position the message cursor before the first segment.

For more information, see Read-only and write-only modes on page 93.

**Note:** The 8K segment size is guaranteed. An OpenEdge application need not use the endOfStream function on page 195 for messages smaller than 8K, since there is only one segment. For information about code page conversions and text size limits, see XML code page encoding on page 346.

---

**HeaderMessage**

A HeaderMessage is a header-only message type that handles bodyless JMS messages. Use the createHeaderMessage procedure on page 186 to handle header messages. See Accessing message header properties on page 94 for information on methods that access message header information.

---

**MapMessage**

A MapMessage is a message type that contains a set of name/value pairs where values are Java primitives.

The following table lists the methods for handling map messages.
### Table 21: Methods for handling map messages

<table>
<thead>
<tr>
<th>Create method</th>
<th>Populate method</th>
<th>Access method</th>
</tr>
</thead>
<tbody>
<tr>
<td>createMapMessage procedure on page 186</td>
<td>setBoolean procedure on page 268</td>
<td>getMapNames function on page 230</td>
</tr>
<tr>
<td></td>
<td>setByte procedure on page 270</td>
<td>getItemType function on page 217</td>
</tr>
<tr>
<td></td>
<td>setBytesFromRaw procedure on page 271</td>
<td>getBytesToRaw function on page 200</td>
</tr>
<tr>
<td></td>
<td>setChar procedure on page 272</td>
<td>getChar function on page 200</td>
</tr>
<tr>
<td></td>
<td>setDateTime procedure on page 277</td>
<td>getDateTime function on page 207</td>
</tr>
<tr>
<td></td>
<td>setDateTime-TZ procedure on page 279</td>
<td>getDateTime-TZ function on page 208</td>
</tr>
<tr>
<td></td>
<td>setDouble procedure on page 280</td>
<td>getDecimal function on page 209</td>
</tr>
<tr>
<td></td>
<td>setFloat procedure on page 283</td>
<td>getInt function on page 210</td>
</tr>
<tr>
<td></td>
<td>setInt procedure on page 287</td>
<td>getInt64 function on page 215</td>
</tr>
<tr>
<td></td>
<td>setLong procedure on page 288</td>
<td>getLogical function on page 216</td>
</tr>
<tr>
<td></td>
<td>setLongString procedure on page 289</td>
<td>getLongString function on page 227</td>
</tr>
<tr>
<td></td>
<td>setLongStringCP procedure on page 298</td>
<td>getLongStringCP function on page 228</td>
</tr>
<tr>
<td></td>
<td>setShort procedure on page 310</td>
<td>clearBody procedure on page 182</td>
</tr>
<tr>
<td></td>
<td>setString procedure on page 313</td>
<td></td>
</tr>
</tbody>
</table>

### StreamMessage

A **StreamMessage** is a message type that allows applications to send and receive an unspecified number of items; each item is a Java data type. All basic Java data types are supported. When receiving any arbitrary Java data type, an application uses methods to read and specify an ABL data type. When writing a message from ABL, an application uses methods to send any of those Java data types and to specify the data.

The following table lists the methods for handling stream messages.

### Table 22: Methods for handling stream messages

<table>
<thead>
<tr>
<th>Create method</th>
<th>Populate method</th>
<th>Access method</th>
</tr>
</thead>
<tbody>
<tr>
<td>createStreamMessage procedure on page 189</td>
<td>writeBoolean procedure on page 321</td>
<td>eofStream function on page 195</td>
</tr>
<tr>
<td></td>
<td>writeByte procedure on page 321</td>
<td>moveToNext procedure on page 253</td>
</tr>
<tr>
<td></td>
<td>writeBytesFromRaw procedure on page 322</td>
<td>readBytesToRaw procedure on page 256</td>
</tr>
<tr>
<td></td>
<td>writeChar procedure on page 323</td>
<td>readChar function on page 256</td>
</tr>
<tr>
<td></td>
<td>writeDateTime procedure on page 323</td>
<td>readDateTime function on page 257</td>
</tr>
<tr>
<td></td>
<td>writeDateTime-TZ procedure on page 324</td>
<td>readDateTime-TZ function on page 258</td>
</tr>
<tr>
<td></td>
<td>writeDouble procedure on page 325</td>
<td>readDouble function on page 258</td>
</tr>
<tr>
<td></td>
<td>writeFloat procedure on page 326</td>
<td>readFloat function on page 259</td>
</tr>
<tr>
<td></td>
<td>writeInt procedure on page 326</td>
<td>readInt function on page 259</td>
</tr>
<tr>
<td></td>
<td>writeInt64 procedure on page 327</td>
<td>readInt64 function on page 259</td>
</tr>
<tr>
<td></td>
<td>writeLong procedure on page 328</td>
<td>readLogical function on page 260</td>
</tr>
<tr>
<td></td>
<td>writeLongString procedure on page 328</td>
<td>readLongString function on page 261</td>
</tr>
<tr>
<td></td>
<td>writeLongStringCP procedure on page 329</td>
<td>readLongStringCP function on page 261</td>
</tr>
<tr>
<td></td>
<td>writeShort procedure on page 330</td>
<td>clearBody procedure on page 182</td>
</tr>
<tr>
<td></td>
<td>writeString procedure on page 330</td>
<td></td>
</tr>
</tbody>
</table>
**BytesMessage**

A **BytesMessage** is a message type that contains an uninterpreted stream of bytes. This message type allows the passing of data "as is" without any interpretation by the ABL–JMS API or the JMS server.

The following table lists the methods for handling bytes messages.

<table>
<thead>
<tr>
<th>Create method</th>
<th>Populate method</th>
<th>Access method</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>createBytesMessage</code> procedure on page 184</td>
<td><code>setMemptr procedure</code> on page 299, <code>writeBytesFromRaw procedure</code> on page 322, <code>reset procedure</code> on page 265, <code>clearBody procedure</code> on page 182</td>
<td><code>readBytesToRaw procedure</code> on page 256, <code>getMemptr function</code> on page 230, <code>endOfStream function</code> on page 195, <code>getBytesCount function</code> on page 198</td>
</tr>
</tbody>
</table>

To write data to a **BytesMessage**, an application uses **RAW** or **MEMPTR** variables with `writeBytesFromRaw` procedure or `setMemptr` procedure. To read data, it uses `readBytesToRaw` procedure or `getMemptr` function.

**Note:** The **RAW** data type has a 32K size limit. To bypass this limit, an application uses the `writeBytesFromRaw` procedure and the `readBytesToRaw` procedure repeatedly. The **MEMPTR** data type does not have a 32K limit. To access **MEMPTR** bytes data, an application uses the `setMemptr` procedure and the `getMemptr` function.

For an example, see **Publishing, subscribing, and receiving an XML document in a BytesMessage** on page 344.

For example, a **BytesMessage** can pass an XML document encoded in a code page that does not match the OpenEdge client's code page.

For more information, see **XML code page encoding** on page 346. For an example, see **Publishing, receiving, and parsing an XMLMessage** on page 342.

**MultipartMessage**

A **MultipartMessage** is a message type that contains one or more discreet parts. A part can be a SonicMQ message, Character data, or Byte data. Parts are identified by a unique content ID character value and can be accessed by ID or index. Each part also contains a content-type value for identifying the data in the part. For message parts, the content-type is defined by Sonic and represents each message type supported by Sonic. A bytes part or text part has a user-defined content-type. There are no restrictions on what this content-type can be, but it is recommended that you use standard MIME types such as text/XML for XML data and text/plain for character data.

The following table lists the methods for handling multi-part messages.
Table 24: Methods for handling multi-part messages

<table>
<thead>
<tr>
<th>Create method</th>
<th>Populate method</th>
<th>Access method</th>
</tr>
</thead>
<tbody>
<tr>
<td>createMultipartMessage procedure</td>
<td>addMessagePart procedure on page 187</td>
<td>isMessagePart function on page 251</td>
</tr>
<tr>
<td>on page 188</td>
<td>addBytesPart procedure on page 178</td>
<td>getBytesPartByIndex function on page 199</td>
</tr>
<tr>
<td></td>
<td>addTextPart procedure on page 179</td>
<td>getBytesPartByID function on page 198</td>
</tr>
<tr>
<td></td>
<td>appendText procedure on page 180</td>
<td>getType function on page 205</td>
</tr>
<tr>
<td></td>
<td>clearBody procedure on page 182</td>
<td>getPartCount function on page 234</td>
</tr>
</tbody>
</table>

Note: The RAW data type has a 32K size limit. To bypass this limit, an application uses the writeBytesFromRaw procedure and the readBytesToRaw procedure repeatedly. The MEMPTR data type does not have a 32K limit. To access MEMPTR bytes data, an application uses the setMemptr procedure and the getMemptr function.

XMLMessage

An XMLMessage is a message type whose body contains a well-formed XML document (a SAX-WRITER, SAX-READER, or X-DOCUMENT). ABL has built in support to send and receive XML messages. The following table lists the methods for handling XML messages.

Table 25: Methods for handling XML messages

<table>
<thead>
<tr>
<th>Create method</th>
<th>Populate method</th>
<th>Access method</th>
</tr>
</thead>
<tbody>
<tr>
<td>createXMLMessage procedure on page 192</td>
<td>setText procedure on page 315</td>
<td>endOfStream function on page 195</td>
</tr>
<tr>
<td></td>
<td>setLongText procedure on page 298</td>
<td>getCharCount function on page 201</td>
</tr>
<tr>
<td></td>
<td>appendText procedure on page 180</td>
<td>getText function on page 245</td>
</tr>
<tr>
<td></td>
<td>reset procedure on page 265</td>
<td>getTextSegment function on page 246</td>
</tr>
<tr>
<td></td>
<td>setX-Document procedure on page 317</td>
<td>getLongText function on page 229</td>
</tr>
<tr>
<td></td>
<td>setSaxReader procedure on page 309</td>
<td>deleteSaxWriter procedure on page 193</td>
</tr>
<tr>
<td></td>
<td>deleteSaxWriter procedure on page 192</td>
<td>clearBody procedure on page 182</td>
</tr>
</tbody>
</table>

The XMLMessage is an extension of a JMS TextMessage. XMLMessage supports the same methods as TextMessage.

XML messages can be used in conjunction with the ABL XML parser:

- **Incoming messages** — Parse the XML text using the getX-Document function or the setSaxReader procedure
- **Outgoing messages** — Save the XML text using the setX-Document procedure or the getSaxWriter function

It is important to consider the code page of XML messages. (A code page is a table that maps each character on it to a unique numeric value.) Theoretically, any code page can be used to encode XML documents. However, each XML parser supports some or all code pages, and XML parsers differ with respect to the code page conversions that they can do.
With the ABL–JMS API, the conversion rules are straightforward. The text stored in an XML message by the OpenEdge application is expected to be encoded in the internal code page of the OpenEdge client (the \texttt{–cpinternal} startup parameter). For more information on the \texttt{–cpinternal} startup parameter, see \textit{OpenEdge Deployment: Startup Command and Parameter Reference}.

The ABL–JMS implementation automatically converts the text to \textit{Unicode} when a SonicMQ XML message is created. Unicode is an encoding format that provides a unique number for every character, regardless of platform, program, or language. The ABL–JMS implementation also converts the Unicode text received in XML messages to the internal code page of the OpenEdge client when the text is extracted.

For more information, see \textit{XML code page encoding} on page 346.

**DataSetMessage**

A \textit{DataSetMessage} is a message type whose body contains a ProDataSet. The information sent and received is based upon the existing \textit{XMLMessage}. ABL has built in support to transform a ProDataSet into XML. The following table lists the methods for handling DataSet messages.

<table>
<thead>
<tr>
<th>Create method</th>
<th>Populate method</th>
<th>Access method</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{createDataSetMessage procedure} on page 186</td>
<td>\texttt{setDataSet procedure} on page 277</td>
<td>\texttt{getDataSet function} on page 206</td>
</tr>
<tr>
<td></td>
<td>\texttt{reset procedure} on page 265</td>
<td></td>
</tr>
<tr>
<td></td>
<td>\texttt{clearBody procedure} on page 182</td>
<td></td>
</tr>
</tbody>
</table>

For an example of a \textit{DataSetMessage}, see \textit{DataSetMessage} on page 128.

**TempTableMessage**

A \textit{TempTableMessage} is a message type whose body contains a temp-table. The information sent and received is based upon the existing \textit{XMLMessage}. ABL has built in support to transform a temp-table into XML. The following table lists the methods for handling TempTable messages.

<table>
<thead>
<tr>
<th>Create method</th>
<th>Populate method</th>
<th>Access method</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{createTempTableMessage procedure} on page 191</td>
<td>\texttt{setTempTable procedure} on page 314</td>
<td>\texttt{getTempTable function} on page 244</td>
</tr>
<tr>
<td></td>
<td>\texttt{reset procedure} on page 265</td>
<td></td>
</tr>
<tr>
<td></td>
<td>\texttt{clearBody procedure} on page 182</td>
<td></td>
</tr>
</tbody>
</table>

For an example of a \textit{TempTableMessage}, see \textit{TempTableMessage} on page 125.

**Java Object messages**

The ABL–JMS API \textbf{does not} support Java Object messages. If a Java Object message is received on behalf of an OpenEdge client, the client's asynchronous error handler receives a \textit{TextMessage} with the header of the Java Object message and a text body with the string "ObjectMessage: Not Supported." (For more information, see \textit{Error and condition handling} on page 103.)
Message size limits

There is no limit to the ABL message size. However, ABL imposes a 32K limit on each item of a StreamMessage or MapMessage. For more information about text size limits, see XML code page encoding on page 346.

SonicMQ does not have a hard-coded maximum message size; the largest tested message is 1MB.

When using very large messages (exceeding 1MB), you might need to modify the JVM's memory limit values, specified in the jvmArgs property of the AdminServerPlugins.properties file. For example, if the OpenEdge Adapter for SonicMQ fails with an OutofMemory error in the log, you should modify the arguments for the sizes of the memory heap (-mx) and the stack (-ss). The following sample entry specifies 40MB for the memory heap and 8MB for the stack:

```
jvmArgs= -Xmx40m -Xss8m
```

Note: You can use the mergeprop utility installed with OpenEdge to manually edit the AdminServerPlugins.properties file. For information on using mergeprop, see OpenEdge Getting Started: Installation and Configuration.

Storing and extracting data

When writing data to a message, an application uses the name of the data type to specify the Java data type in the message; the ABL name is identical to the Java name. For example, Java uses the writeShort procedure on page 330 to write a number to a StreamMessage as short. The ABL counterpart is the internal procedure writeShort(INTEGER).

In the context of extracting data from a message, there is an important difference between the ABL model and the Java model with respect to the names of the methods:

- In Java, the name of the method determines the data type to which the extracted data is converted. For example, readLongString function on page 261 extracts a value (for example, an INTEGER value from a StreamMessage) and converts it to a String value.

- In ABL, the equivalent function is the readChar function on page 256 to convert a value (such as an INTEGER value) to an ABL CHARACTER value.

The following table maps the ABL data types to the JMS data types for data storage.

<table>
<thead>
<tr>
<th>ABL data type</th>
<th>JMS data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOGICAL</td>
<td>boolean</td>
</tr>
<tr>
<td>INTEGER</td>
<td>byte</td>
</tr>
<tr>
<td>INTEGER</td>
<td>short</td>
</tr>
<tr>
<td>INTEGER</td>
<td>int</td>
</tr>
<tr>
<td>INT64</td>
<td>long</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>long</td>
</tr>
</tbody>
</table>
The following table maps the available conversions from JMS data types to ABL data types for data extraction.

**Table 29: JMS and ABL data types for extracting data**

<table>
<thead>
<tr>
<th>JMS data type</th>
<th>ABL data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>LOGICAL or CHARACTER</td>
</tr>
<tr>
<td>byte</td>
<td>INTEGER, DECIMAL, or CHARACTER</td>
</tr>
<tr>
<td>short</td>
<td>INTEGER, DECIMAL, or CHARACTER</td>
</tr>
<tr>
<td>int</td>
<td>INTEGER, DECIMAL, or CHARACTER</td>
</tr>
<tr>
<td>long</td>
<td>INT64, DECIMAL or CHARACTER</td>
</tr>
<tr>
<td>float</td>
<td>DECIMAL or CHARACTER</td>
</tr>
<tr>
<td>double</td>
<td>DECIMAL or CHARACTER</td>
</tr>
<tr>
<td>String</td>
<td>CHARACTER or LONGCHAR</td>
</tr>
<tr>
<td>char</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>byte array</td>
<td>RAW or MEMPTR (MEMPTR is available only with BytesMessage)</td>
</tr>
<tr>
<td>Java date string</td>
<td>DATE, DATETIME, or DATE-TZ</td>
</tr>
</tbody>
</table>
Read-only and write-only modes

As in Java–JMS, the StreamMessage, TextMessage, XMLMessage, and BytesMessage are created in write-only mode. In write-only mode, an application can use only data-setting methods, not data-extraction methods.

The reset procedure on page 265 puts the cursor before the first item of the message and transfers it to read-only mode.

**Note:** The publish procedure on page 254, sendToQueue procedure on page 266, and requestReply procedure on page 264 call the reset procedure implicitly.

The message is received by the receiver in read-only mode. The clearBody procedure on page 182 clears the message body and transfers the message to write-only mode.

**Note:** Read-only and write-only refer to the body of the message, not its header. Read-only and write-only modes do not apply to Header messages, which lack a body.

Unlike in Java–JMS, a MapMessage in the ABL–JMS implementation is always in read/write mode; there is no read-only or write-only mode for a MapMessage.

**Note:** The reset procedure has no effect when called on Map and Header messages.

clearBody and clearProperties

The clearBody procedure and clearProperties procedure, supported by all message types, function as follows:

- The clearBody procedure on page 182 deletes all data from the message body.
- The clearProperties procedure on page 183 deletes all header properties (but not the JMS-predefined header fields).

Sending messages to a queue

In the PTP domain, applications send messages to a queue. To send a message to a queue with Java–JMS, an application obtains a handle to a queue object, creates a Queue Sender object, and uses the queue sender to send messages. Sending a message to a queue with the ABL–JMS API involves these general steps:

1. The application calls the sendToQueue procedure on page 266 in the ptpsession.p or jmssession.p object.
2. The application specifies the queue name as an **INPUT** parameter of type **CHARACTER**.

The application can set other sending parameters (such as **persistency**, **timeToLive**, and **priority**) in the Session object as a default for all the messages it sends, or it can set these parameters at each sendToQueue procedure call.
Publishing messages to a topic

In the Pub/Sub domain, applications publish messages to topics. To publish a message with Java–JMS, an application obtains a handle to a Topic object and creates a Publisher object. It then uses the Publisher Object to publish messages. Publishing a message to a topic with the ABL–JMS API involves these general steps:

1. The application publishes messages through the publish procedure on page 254 of the pubsubsession.p or jmssession.p object.

2. The application specifies the topic name as an INPUT parameter of type CHARACTER.

The application can set other sending parameters (such as persistency, timeToLive, and priority) in the Session object as a default for all the messages it sends, or it can set these parameters at each publish procedure call.

Clearing messages

An application clears the body of a message, leaving header and property values unchanged, using the clearBody procedure on page 182. The clearProperties procedure on page 183 deletes all header properties (but not the JMS-predefined header fields).

Deleting messages

An application explicitly deletes a message using the deleteMessage procedure on page 193. For example:

```
RUN deleteMessage IN messageH.
```

Accessing message header properties

The message header provides envelope information about a message. The message header interface is supported by all message types, and all message types have the same header information.

The message header is not created directly by the application. When any type of message is created, its header procedure is automatically created. The message procedure delegates header method calls to its header procedure.

The following table lists the methods to access message header information.
Table 30: Accessing message header information

<table>
<thead>
<tr>
<th>Set header methods</th>
<th>Get header methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>setJMSReplyTo procedure on page 292</td>
<td>getJMSReplyTo function on page 222</td>
</tr>
<tr>
<td>setReplyToDestinationType procedure on page 308</td>
<td>getReplyToDestinationType function on page 240</td>
</tr>
<tr>
<td>setJMSCorrelationID procedure on page 291</td>
<td>getJMSCorrelationID function on page 218</td>
</tr>
<tr>
<td>setJMSCorrelationIDAsBytes procedure on page 292</td>
<td>getJMSCorrelationIDAsBytes function on page 219</td>
</tr>
<tr>
<td>setJMSType procedure on page 293</td>
<td>getJMSType function on page 224 getJMSDestination function on page 220 getJMSRedelivered function on page 222 getMessageType function on page 233 getJMSMessageID function on page 221 getJMSDeliveryMode function on page 219 getJMSTimestamp function on page 223 getJMSExpiration function on page 220 getJMSPriority function on page 221 hasReplyTo function on page 249</td>
</tr>
</tbody>
</table>

Accessing message properties

Message properties can add more envelope information about a message. The number of header fields is fixed, but properties are flexible. An application can add any number of property name-and-value pairs. The following table lists the methods to access message properties.

Table 31: Accessing message properties

<table>
<thead>
<tr>
<th>Setting message properties</th>
<th>Getting message properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>setBooleanProperty procedure on page 269</td>
<td>getCharProperty function on page 201 getDateTimeProperty function on page 207 getDateTimeTzProperty function on page 209 getDecimalProperty function on page 210 getInt64Property function on page 211 getIntProperty function on page 215 getFloatProperty function on page 216 getLogicalProperty function on page 227 getBooleanProperty function on page 236 getPropertyNames function on page 235 setByteProperty procedure on page 271 setDateTimeProperty procedure on page 280 setDoubleProperty procedure on page 284 setFloatProperty procedure on page 287 setIntProperty procedure on page 289 setInt32Property procedure on page 290 setLongProperty procedure on page 297 setShortProperty procedure on page 311 setStringProperty procedure on page 313 clearProperties procedure on page 183</td>
</tr>
</tbody>
</table>

The requesting application clears the properties of a message (keeps header and body values unchanged) using clearProperties procedure.
Consuming messages

A Message Consumer consumes messages either asynchronously or synchronously. The message consumption type is set for a session. Messages are processed by the Message Consumer when the ABL Virtual Machine (AVM) is in a `WAIT–FOR` state or other IO-blocking state. While the application is in such a state, all other UI and non-UI events are handled normally. `WAIT–FOR` can be called explicitly by the ABL code. It can also be called through the `waitForMessages` procedure on page 320 on the Session object, which works the same for GUI, character, batch, AppServer, and WebSpeed applications. Once a message is consumed, the content of the message is inaccessible.

The following sections describe:

- Creating a Message Consumer object on page 96
- Creating a message handler process on page 97
- Setting reply properties on page 97
- Receiving messages from a queue on page 97
- Subscribing to a topic on page 98
- Terminating the Message Consumer object on page 98
- Processing messages on page 98
- Controlling flow of messages on page 98
- Reusing messages on page 98
- Message-reception issues on page 98
- Reply mechanisms on page 100

Creating a Message Consumer object

The OpenEdge application uses a Message Consumer object to receive messages from a destination or to receive asynchronous error messages. In a Session object, the application creates a Message Consumer object using the `createMessageConsumer` procedure on page 187.

The life cycle of a Message Consumer object includes these general steps:

1. An application implements a procedure to handle the messages.
2. The application creates the Message Consumer, specifying the message-handling procedure.
3. The application uses the Message Consumer object to do one of the following: subscribe to a topic (Pub/Sub) or receive messages from the queue (PTP); set an error handler and receive error messages asynchronously from SonicMQ through the OpenEdge Adapter for SonicMQ; or receive replies in a request/reply cycle.
4. After using the Message Consumer object, the application can activate it by getting into a `WAIT FOR` state (or any IO-blocking state where the application processes events).
5. When the Message Consumer finishes processing all messages of interest, the application calls the `deleteConsumer` procedure on page 192 to release the resources in the OpenEdge application, the OpenEdge Adapter for SonicMQ, and the SonicMQ Broker.
Creating a message handler process

A message handler processes the incoming message from the Message Consumer. When an incoming JMS or error message is received, the message handler is called automatically so that the application can process the message. The ABL programmer creates a message handler using the `messageHandler` procedure on page 252. The OpenEdge application passes context to the message handler using the `setApplicationContext` procedure on page 268.

Accessing message handler information

The following table lists procedures for getting message handler properties and type of message being handled.

Table 32: Methods for the message handler

<table>
<thead>
<tr>
<th>Getting message handler properties</th>
<th>Type of message handled</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getApplicationContext</code> function on page 197</td>
<td><code>inErrorHandling</code> function on page 249</td>
</tr>
<tr>
<td><code>getDestinationName</code> function on page 213</td>
<td><code>inMessageHandling</code> function on page 250</td>
</tr>
<tr>
<td><code>getProcHandle</code> function on page 235</td>
<td><code>inQueueBrowsing</code> function on page 250</td>
</tr>
<tr>
<td><code>getProcName</code> function on page 235</td>
<td><code>inReplyHandling</code> function on page 250</td>
</tr>
<tr>
<td><code>getReplyAutoDelete</code> function on page 238</td>
<td></td>
</tr>
<tr>
<td><code>getReplyPersistency</code> function on page 238</td>
<td></td>
</tr>
<tr>
<td><code>getReplyPriority</code> function on page 239</td>
<td></td>
</tr>
<tr>
<td><code>getReplyTimeToLive</code> function on page 239</td>
<td></td>
</tr>
<tr>
<td><code>getSession</code> function on page 241</td>
<td></td>
</tr>
</tbody>
</table>

Setting reply properties

The Message Consumer sets the reply properties by using the `setReplyPriority` procedure, `setReplyTimeToLive` procedure, `setReplyPersistency` procedure, and `setReplyAutoDelete` procedure.

Receiving messages from a queue

In the PTP domain, applications receive messages from a queue. The application calls the `receiveFromQueue` procedure on page 262 in `ptpsession.p` or `jmssession.p` with the queue name and a handle to the Message Consumer object. The application can pass a JMS properties selector expression to the `receiveFromQueue` procedure call to specify which messages the receiver wants to receive from the queue.

Queue browsing

The PTP model supports `queue browsing`, a mechanism that lets an application view the content of messages in a queue without actually consuming (receiving) the messages. The ABL–JMS API supports queue browsing through the `browseQueue` procedure on page 181 in the PTP Session object or JMS Session object.

The messages can be handled by the message handler in the same way as messages coming from a `receiveFromQueue` procedure on page 262 call, but they are not acknowledged and are not subject to the transactional context of the session. (See the Java Message Service specification and SonicMQ Programming Guide for details on queue browsing.)
Subscribing to a topic

In the Pub/Sub domain, applications subscribe to topics of interest. The application calls the subscribe procedure on page 319 in pubsubsession.p or jmsession.p with the topic name and a handle to the Message Consumer object. The application implements a message-handling routine for handling the incoming messages, as well as a Message Consumer object that contains the message handler and provides context to the application when it processes messages.

Durable subscriptions

A subscriber typically receives messages while it is active. Some applications might require that a subscriber receives all messages even if the subscriber is inactive when the messages are published. In order to meet this requirement, you can create a durable subscriber. A durable subscription guarantees message delivery. A durable subscription is registered with the SonicMQ Broker with a unique identity; the broker retains the subscription’s messages until they are received by the application or until they expire. The application can pass a JMS properties selector expression to the subscribe procedure on page 319 to specify which messages the subscriber wants to receive. The application can also specify whether it wants to receive its own published messages. Use the cancelDurableSubscription procedure on page 182 to cancel a durable subscription.

Terminating the Message Consumer object

In a Session object, the application deletes the Message Consumer object and releases resources in the OpenEdge application, the OpenEdge Adapter for SonicMQ, and the SonicMQ Broker using the deleteConsumer procedure on page 192.

Processing messages

To control message processing use the waitForMessages procedure on page 320.

Controlling flow of messages

The application controls the flow of messages to the SonicMQ client from a queue using the setPrefetchCount procedure on page 302 and the setPrefetchThreshold procedure on page 303.

**Note:** When the OpenEdge Adapter for SonicMQ sends a message to a queue that is full or to a topic that is full, an error is raised.

Reusing messages

The application sets message reuse using the setReuseMessage procedure on page 308 and getReuseMessage function on page 240.

Message-reception issues

The sections that follow discuss several message-reception issues in the PTP and Pub/Sub domains.
Stopping and starting message reception

To actually start receiving messages, the OpenEdge application must call the `startReceiveMessages` procedure on page 317 in the Session object. One call to the `startReceiveMessages` procedure is sufficient for the session. The application typically calls the `startReceiveMessages` procedure after subscribing to all topics of interest (in the Pub/Sub domain) or registering Message Consumer objects with the queues of interest (in the PTP domain).

The application can also call the `stopReceiveMessages` procedure on page 318 to temporarily stop the reception of messages. To resume message reception, it can call the `startReceiveMessages` procedure again.

In the Pub/Sub domain, calling the `stopReceiveMessages` procedure does not cancel existing subscriptions; however, for any nondurable subscription, messages published while reception is stopped are not delivered.

In the PTP domain, the messages are queued while the client is in the `stopReceiveMessages` procedure state and are delivered to the client after the `startReceiveMessages` procedure is called again.

Stopping the reception of messages is recommended when an application is not going to process messages for a while.

**Note:** After calling the `stopReceiveMessages` procedure, the OpenEdge client might receive one message sent from the server prior to execution of the call.

**Caution:** `StopReceiveMessages` procedure should not be invoked in a message handler.

Message Consumer scope

A Message Consumer object can be used to handle only one subscription (in the Pub/Sub domain) or receive messages from only one queue (in the PTP domain).

When the `deleteConsumer` procedure on page 192 is called, message reception is canceled and the Message Consumer object is deleted.

**Note:** To delete a durable subscription (in the Pub/Sub domain), the `cancelDurableSubscription` procedure on page 182 in `pubsubsession.p` or `jmssession.p` must be called as well, since `deleteConsumer` procedure only suspends the subscription in the current session. There is no equivalent to a durable subscription in the PTP domain. It is an error to call the `cancelDurableSubscription` procedure while there is an active Message Consumer for that subscription. First call the `deleteConsumer` procedure to delete the Message Consumer.

When a Message Consumer object is used for receiving replies through the `requestReply` procedure on page 264 call, it can be used many times; there is no need to create one for every call. The `deleteSession` procedure on page 194 call deletes all Message Consumer objects for that session.

OpenEdge run-time message-processing states

An OpenEdge application receives and processes messages when it is in an I/O-blocking state. The same rules that determine when asynchronous completion procedures are fired also determine when message handlers are called. The OpenEdge application should typically use the `WAIT–FOR` statement or the `waitForMessages` procedure API session call for processing messages as well as for other events.

The `waitForMessages` procedure on page 320 is a convenient way to write message-handling code that is independent of the environment in which the OpenEdge application is executed (GUI, CHUI, batch, AppServer, or WebSpeed). It processes all events that occur while the application is waiting, including user-interface events and asynchronous call events, and it allows the application to specify when to stop waiting.
The `waitForMessages` procedure takes three input parameters: a procedure handle, the name of a user-defined function in the procedure that returns a `LOGICAL` value, and a `timeOut` parameter of type `INTEGER` (specifying an interval in seconds). The `waitForMessages` procedure waits and processes events as long as: a) the user-defined function returns `TRUE`; and b) the interval specified by the `timeOut` value elapses without any messages being received.

The user-defined function is evaluated by the ABL–JMS API after the message handler is executed. Typically, the OpenEdge application should have logic for changing the return value of the function in the message handler.

### Synchronous message reception

ABL does not explicitly support receiving messages synchronously, but the same effect can be achieved by use of the `WAIT–FOR` statement or the `waitForMessages` procedure on page 320. These constructs wait for a user-defined event. When the desired message is received, the message handler can trigger the termination of the `WAIT–FOR` statement or the `waitForMessages` procedure—for example, by applying the specified user-defined event.

### Reply mechanisms

This section applies to both the Pub/Sub and the PTP domains.

Java–JMS provides no built-in mechanism for replies. It is the responsibility of the application to:

- Designate a Destination object (typically a temporary destination) for replies
- Send this Destination object to the receiver (typically through the `ReplyTo` field in the message header, a set of fields containing values to identify and route the message)

The receiver must extract the reply destination from the message and follow the normal publish (or send) steps to reply.

The ABL–JMS API simplifies this process, both for the sender needing a reply and for the receiver needing to reply:

- **Sender** — The ABL–JMS API `requestReply` procedure on page 264 can publish messages in the same way as the publish procedure, or send messages to a queue in the same way as the `sendToQueue` procedure. In addition, a Message Consumer object for replies is passed to the `requestReply` procedure as an input parameter. The ABL–JMS implementation automatically routes all the replies to that Message Consumer object. See Request/Reply on page 73 for additional information.

- **Receiver** — To reply, the message receiver returns a `reply` message handle as an output parameter in the message-handling routine. The application can call the `setReplyPersistency` procedure on page 306 in the Message Consumer object to automatically delete replies after sending them.

An application can also publish a reply message or send it to a queue by first calling the `getReplyToDestinationType` function on page 240 to extract the name of the reply destination, and then calling the publish procedure or `sendToQueue` procedure directly.

**Note:** If the `ReplyTo` destination is a temporary destination, an application must send a reply before deleting the original message. (See the Java Message Service specification and SonicMQ Programming Guide for information on temporary destinations.) Deleting the original message tells the ABL–JMS implementation that the `ReplyTo` temporary destination will no longer be used.

By default, the type of the `ReplyTo` destination matches the type of the origin of the message:
• If the message was created by a Pub/Sub Session object, the value of the `ReplyTo` field is considered a topic name.

• If the message was created by a PTP Session object, the value of the `ReplyTo` field is considered a queue name.

However, it is legal to designate a queue for replying to a published message, or a topic for replying to messages received from a queue. To accommodate this, the ABL–JMS API supports the `setReplyToDestinationType` procedure on page 308 and the `getReplyToDestinationType` function on page 240, both of which support the CHARACTER values `topic` and `queue`.

The `setReplyToDestinationType` procedure can be called if the OpenEdge application calls the `setJMSReplyTo` procedure and sets a destination from a domain other than that of the session. The `getReplyToDestinationType` function must be called when the OpenEdge application receives a message and wants to reply to it, but is not certain about the `ReplyTo` domain.

### Transaction and recovery procedures

**Transacted session**

A *transacted session* allows an application to send or receive groups of messages as one atomic operation:

• A session that is transacted for sending guarantees that either all messages in a group are sent, or none is sent.

• A session that is transacted for receiving guarantees that a group of received messages are acknowledged only after all messages in the group are successfully processed.

The following table lists the methods available for controlling the execution and recovery of transactions.

**Table 33: Managing transaction attributes**

<table>
<thead>
<tr>
<th>Setting</th>
<th>Rolling back</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>commitSend</code> procedure on page 184</td>
<td><code>rollbackSend</code> procedure on page 266</td>
</tr>
<tr>
<td><code>commitReceive</code> procedure on page 183</td>
<td><code>rollbackReceive</code> procedure on page 265</td>
</tr>
</tbody>
</table>

The typical Java–JMS transacted application uses two sessions, one for transacted sending and one for transacted receiving. The ABL–JMS implementation uses two JMS sessions behind the scenes, but at the ABL API level, there is only one Session object.

The application controls whether sending, receiving, or both are transacted. It makes the session transacted by calling the `setTransactedSend` procedure on page 316, the `setTransactedReceive` procedure on page 315, or both in the Session object.

A session that is transacted for sending, receiving, or both is constantly in a transaction mode. When a transaction is committed or rolled back, a new one is automatically started.

**Transacted sending**

When an application calls the `commitSend` procedure on page 184 in a Session object, all messages that have been published or sent to a queue with the current transaction are sent. When an application calls the `rollbackSend` procedure on page 266 in a Session object, all such messages are discarded.
Transacted receiving

When an application calls the `commitReceive` procedure on page 183 in a Session object, all messages that have been received with the current transaction are acknowledged. When an application calls the `rollbackReceive` procedure on page 265 in a Session object, all such messages are re-received (yielding the same effect as calling the `recover` procedure on page 263 in a nontransacted session).

Illegal calls: recover and setNoAcknowledge

Since message acknowledgement and recovery are handled automatically in a transacted session, it is an error to call the recover procedure and setNoAcknowledge procedure in a session that is transacted for receiving.

ABL transactions and JMS transacted sessions

ABL transactions and JMS transactions are not integrated. For example, a `DO TRANSACTION` block might be rolled back while the JMS calls inside the transaction block are committed. The OpenEdge application must synchronize between ABL transactions and JMS transactions.

---

Note: For information about the handling of errors and error conditions, see Error and condition handling on page 103.

---

Message acknowledgement, forwarding, and recovery

A client sends an acknowledgement to tell the SonicMQ Broker that the client received and processed a message and does not need to receive that message again. Acknowledgement of a message prevents the message and all previous messages from being delivered to that session again.

The following table lists the methods the application uses to set message acknowledgement.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Getting</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>setSingleMessageAcknowledgement procedure</code> on page 312</td>
<td><code>getSingleMessageAcknowledgement function</code> on page 243</td>
</tr>
<tr>
<td><code>setNoAcknowledge procedure</code> on page 300</td>
<td><code>getNoAcknowledge function</code> on page 233</td>
</tr>
<tr>
<td><code>acknowledgeAndForward procedure</code> on page 177</td>
<td></td>
</tr>
</tbody>
</table>

The sections that follow describe automatic message acknowledgement, preventing message acknowledgement, and message recovery.

Automatic message acknowledgement

With the ABL–JMS API, an incoming message is acknowledged automatically when the message handler finishes execution. Acknowledgement is sent on the request for the next message, improving performance.

A client or communication failure between the time the message handler finishes execution and the time the ABL–JMS implementation sends the acknowledgement can result in the messages being redelivered (according to the JMS message redelivery rules). An application can use a transacted session to avoid this message redelivery problem.

Unlike Java–JMS, the ABL–JMS API does not support the explicit acknowledgement of messages or the "lazy" acknowledgement of messages (the JMS `CLIENT_ACKNOWLEDGE` and `DUPS_OK_ACKNOWLEDGE` modes).
Preventing message acknowledgement

An OpenEdge application can explicitly prevent acknowledgement of a message by calling the setNoAcknowledge procedure of the Message Consumer object. (The Message Consumer object is passed as a parameter to the message-handling procedure.) The setNoAcknowledge procedure is typically used when the application wants to receive the same message again because of an error in processing it, or when receipt of a group of messages is to be signaled by explicit acknowledgement of only the last message in the group.

Single-message acknowledgement

Normally, an OpenEdge client application automatically acknowledges a message when the message handler procedure completes. In SINGLE_MESSAGE_ACKNOWLEDGE mode, however, each message requires its own acknowledgement; if you choose not to acknowledge a message, it is never acknowledged.

To turn on SINGLE_MESSAGE_ACKNOWLEDGE mode, an OpenEdge client application calls the setSingleMessageAcknowledgement procedure of the session handle with the input parameter set to TRUE. To turn off this mode, the application calls the same method with the input parameter set to FALSE.

Message recovery

If an application wants to receive all unacknowledged messages again, it can call the recover procedure on page 263 in the Session object. If the recover procedure is called on a session stopped by the stopReceiveMessages procedure on page 318, the session is recovered and message delivery is restarted.

Example

Consider the following scenario:

1. A client retrieves a message from a broker’s queue.
2. The broker wants to be notified when the message reaches its ultimate destination.
3. The ultimate destination is a remote queue.
4. The client sends the message on its way.

To acknowledge receipt of a message whose ultimate destination is a remote queue, you might enclose the message and acknowledgement in a single transaction; but this introduces the overhead and complexity of transaction processing. SonicMQ provides a cleaner solution, embodied in the following steps:

1. Run the setSingleMessageAcknowledgement procedure to set the session to SINGLE_MESSAGE_ACKNOWLEDGE.
2. Run the acknowledgeAndForward procedure within the message event handler, specifying a destination queue name, the original message handle, and optional message-delivery properties (priority, time to live, and persistency). If the method is successful, the message is acknowledged and forwarded in a single atomic operation.

Error and condition handling

This section provides information about handling of errors and conditions with the ABL–JMS API.

From the point of view of the ABL programmer, there are two types of errors and conditions, programming errors and run-time conditions:
A **programming error** is an erroneous sequence of calls to the ABL–JMS API, or the calling of the API with invalid parameters. Typically, programming errors should not occur in a deployed application. An example of a programming error is an attempt by the application to make a TextMessage call such as setText procedure, setLongText procedure, and appendText procedure in a StreamMessage. Programming errors should be tracked down and fixed at development time. The primary source of information for that phase is the ABL–JMS API (see ABL - JMS API Reference on page 161).

A **run-time condition** is an event that disturbs the normal flow of the application. Such events can occur in a deployed application, so the ABL programmer should try to handle them programmatically. Examples of run-time conditions include attempts to connect to a JMS server that is not currently running, and attempts to subscribe to a topic without the proper authorization. For information about programmatically handling run-time conditions, refer to SonicMQ API Reference, which is installed in sonicmq_docs\sonicmq_api under the SonicMQ installation directory (open index.html in this directory, or access SonicMQ API Reference from the SonicMQ Documentation Portal).

A second criterion for classifying errors and conditions is whether the problem is reported by the ABL–JMS implementation synchronously or asynchronously:

- A problem is reported **synchronously** if it occurs and is detected while the OpenEdge application is executing an ABL–JMS API call.
- A problem is reported **asynchronously** when it comes from the asynchronous error reporting system of the JMS server (OnException Events) or from the ABL–JMS mechanism that delivers messages asynchronously to the OpenEdge client.

Programming errors are usually reported synchronously. Run-time conditions are reported either synchronously or asynchronously.

**Note:** For a complete description of ABL error handling, see OpenEdge Development: Error Handling.

### Handling errors

To manage errors use the setErrorHandler procedure on page 284 and setNoErrorDisplay procedure on page 300.

### Synchronously reported errors and conditions

Errors are reported synchronously when something goes wrong at a method call. The problems can be either programming errors or run-time conditions. Examples include attempts to publish to an unauthorized topic or attempts to receive from a nonexistent queue.

An ABL API function reports problems synchronously by returning an **Unknown value (?)**.

Some programming errors are not detected by the ABL–JMS API but rather by the ABL interpreter. For example, an attempt to call the setText procedure in a StreamMessage causes error 6456:

```
Procedure message.p has no entry point for setText. (6456)
```
To report a problem synchronously, the ABL–JMS API internal procedure calls:

```
RETURN ERROR <error-message>
```

This call raises an error condition at the caller. The caller can use regular ABL techniques to handle the error: a NO-ERROR phrase or an ON ERROR block, coupled with checking the RETURN-VALUE value to obtain the error message. If an application uses the NO-ERROR phrase, it must check the STATUS-ERROR: error flag to determine whether a problem has occurred.

By default, every synchronously reported error or condition is displayed by the ABL–JMS API, which calls:

```
MESSAGE <error-message> VIEW-AS ALERT-BOX.
```

This mechanism allows a quick analysis and resolution of the problem at development time. At deployment time, however, the application developer might want to handle problems programmatically and prevent the message from appearing. Calling the setNoErrorDisplay procedure on page 300 in the Session object suppresses the message display.

**Note:** Message objects inherit the display/noDisplay property from the session that created them. However, after a message is created, it is independent of the session. The setNoErrorDisplay procedure on page 300 must be called in the Message object itself to change this property.

### Asynchronously reported conditions

Typically, problems reported asynchronously are run-time conditions, such as the failure of the SonicMQ Broker or the failure of communication between the OpenEdge Adapter for SonicMQ and the SonicMQ Broker. (See OpenEdge Application Server: Administration.) Another example is the failure to send an automatic reply (the message handler is set with a reply message, but the SonicMQ server fails to send the reply).

The error condition is reported in a TextMessage, with several possible CHAR message properties in the message header: exception, errorCode, linkedException-1, linkedException-2… linkedException-n (where n is a number of additional exceptions linked to the main exception). Use the getPropertyNames function on page 235 to get a list of properties in the error message header. See Messaging Examples on page 333 for an example.

The application should handle problems of this type programmatically by creating a Message Consumer object and passing it to the setErrorHandler procedure in the Session object. If an application does not set an error handler, a default error handler displays the error message and the properties in alert boxes.

**Note:** An application must call the beginSession procedure before creating the error-handling Message Consumer object and calling the setErrorHandler procedure on page 284.
Run-time conditions

Typically, run-time exceptions are generated by the Java–JMS code on the server. In such cases, the format of the error message obtained from the RETURN–VALUE is:

\[ <\text{java-exception}>:<\text{error-message}>. \]

The ABL programmer can look up the types of exceptions thrown by SonicMQ and handle some of them programmatically. The most typical run-time error conditions are connection and authorization failures.

Connection and communication failures

The most common run-time error condition is a connection failure. The \textit{beginSession procedure} on page 180, which creates the connection to the OpenEdge Adapter for SonicMQ and the JMS server, reports connection failures synchronously by calling:

\[ \text{RETURN ERROR } <\text{error-message}>. \]

The error can result from a failure to connect either to the OpenEdge Adapter for SonicMQ or to the JMS server. If the connection to the JMS server fails, the format of the error message is:

\[ <\text{java-exception}>:<\text{error-message}>. \]

A communication failure that occurs after a successful connection might be detected:

- Synchronously (for example, when the application is trying to publish a message)
- Asynchronously through the error handler

It might take several minutes for the timeout mechanism to trigger a communication failure event. To detect potential communication failures more quickly, use the \textit{setPingInterval procedure} on page 301 (a SonicMQ extension) to instruct the OpenEdge Adapter for SonicMQ to actively ping the SonicMQ Broker every \( n \) seconds.

Message handler errors and conditions

A message-handling procedure is an arbitrary ABL program, and the programmer is free to use any ABL technique to handle problems that occur during the processing of a message. However, the following issues and limitations exist:

- Message handlers should handle \texttt{ERROR}, \texttt{STOP}, and \texttt{QUIT} conditions and not propagate them. An unhandled condition is considered a programming error.

- Since the message handler returns control to the ABL–JMS implementation and the message handler cannot raise a condition, there must be a mechanism to allow the message handler to communicate problems to the rest of the OpenEdge application. You can use the \textit{setApplicationContext procedure} on page 268 call to pass an ABL procedure handle to the Message Consumer object. The message handler can obtain the procedure handle by calling the \textit{getApplicationContext function} on page 197 in the Message Consumer object and can then make the appropriate internal procedure calls.
• As mentioned in Message acknowledgement, forwarding, and recovery on page 102, the message handler can call the setNoAcknowledge procedure on page 300 of the Message Consumer to prevent the message from being acknowledged in a session that is not transacted for receiving.

• Calling WAIT–FOR is allowed inside a message handler, but no further messages from that Session object are received until the message handler returns.

• The following recursive calls from the message handler into the ABL–JMS API of the same Session object are considered programming errors: deleteSession procedure on page 194, deleteConsumer procedure on page 192, and recover procedure on page 263. There are no restrictions on calling these API entries of another Session object.

### Interrupts

An interrupt (CTRL+C on UNIX platforms or CTRL+BREAK on Microsoft platforms) while an ABL–JMS call is executing can cause the call to return either an ABL STOP condition or an ERROR condition, depending on the exact timing. The ABL–JMS implementation guarantees that partial messages will not be sent or received as the result of an interrupt.

### OpenEdge Adapter for SonicMQ failure

If communication with the OpenEdge Adapter for SonicMQ is lost, or if the OpenEdge Adapter for SonicMQ shuts down while the OpenEdge client is performing a WAIT–FOR or waitForMessages procedure on page 320 statement, an ABL STOP condition is raised.

If communication with the OpenEdge Adapter for SonicMQ is lost, or if the OpenEdge Adapter for SonicMQ shuts down while the ABL–JMS implementation is actively trying to communicate to it (for example, when the OpenEdge application calls the publish procedure or the subscribe procedure ), an ERROR or STOP condition is raised, depending on the exact point at which the failure is discovered.
Programming for the OpenEdge Adapter for SonicMQ with the ABL - JMS API

This section contains instructions for accessing SonicMQ messaging from the ABL (Advanced Business Language) through the ABL–JMS API provided by the OpenEdge® Adapter for SonicMQ.

For an alphabetical API reference, see ABL - JMS API Reference on page 161.

For details, see the following topics:

- PTP message example
- Pub/Sub messaging example
- Programming scenarios

PTP message example

A PTP messaging example consists of the basic steps described in the following sections:

- Creating a PTP session procedure on page 110
- Connecting to the broker on page 110
- Creating a Message Consumer on page 110
- Preparing to receive messages on page 110
- Sending messages to the queue on page 111
Creating a PTP session procedure

In the following session example, the application creates a session object by calling `ptpsession.p` persistently.

Creating a PTP session

```
DEFINE VARIABLE hPTPSession AS HANDLE NO-UNDO.
RUN jms/ptpsession.p PERSISTENT SET hPTPSession ("-H localhost -S 5162").
```

Creating the session object specifies the connection parameters to the SonicMQ Broker. This allows an application to set different session-level attributes before starting the JMS session. The connection to the SonicMQ Broker and the JMS session does not occur until the application calls the `beginSession` procedure on page 180.

Connecting to the broker

In the following connection example, the OpenEdge application connects to the SonicMQ Broker to begin exchanging messages.

Connecting to the broker

```
RUN setBrokerURL IN hPTPSession ("tcp://machinename:2506").
RUN beginSession IN hPTPSession.
```

Creating a Message Consumer

The OpenEdge client requires a queue for sending messages. You create a queue using the Sonic Management Console. Queues must be defined before starting your session. Then you create a Message Consumer to receive requests from queue, as follows:

Creating a Message Consumer

```
RUN createMessageConsumer IN hPTPSession
    (THIS-PROCEDURE, "myintproc", OUTPUT hConsumer).
```

Preparing to receive messages

The OpenEdge application begins listening on the queue and prepares to receive messages from the queue, as follows:
Preparing to receive messages

```plaintext
RUN receiveFromQueue IN hPTPSession ("myQueue", ?, hConsumer).
RUN startReceiveMessages IN hPTPSession.
/* Wait to receive the messages. */
WAIT-FOR u1 OF THIS-PROCEDURE.
```

The Message Consumer (hConsumer handle) listens on myqueue and handles messages using the myintproc internal procedure. The startReceiveMessages procedure on page 317 starts receipt of incoming messages.

Sending messages to the queue

The application sends a message to the queue using the sendToQueue procedure on page 266, as follows:

```
DEFINE VARIABLE hMessage AS HANDLE NO-UNDO.
/* Code to create message */
RUN sendToQueue IN hPTPSession ("myQueue", hMessage, ?, ?, ?)
```

Note: The queue must be created on the Sonic Management Console.

Receiving messages from the queue

The Message Consumer receives a message from the queue and executes the business logic.

Deleting a message

The application deletes the messages after it finishes using them, as shown in the following example.

```
RUN deleteMessage IN hMessage.
```
Complete code for sending a message using a PTP session

The following sample summarizes the steps for sending a message.

```ABL
/* Sending a message to myqueue */
DEFINE VARIABLE hMessage AS HANDLE NO-UNDO.
DEFINE VARIABLE hPTPSession AS HANDLE NO-UNDO.
/* Creates PTP session*/
RUN jms/ptpsession.p PERSISTENT SET hPTPSession ("-H localhost -S 5162")
/*Connects to the broker */
RUN setBrokerURL IN hPTPSession ("tcp://machinename:2506").
RUN beginSession IN hPTPSession.
/* Create a message */
RUN create...Message IN hPTPSession (OUTPUT hMessage).
RUN set... IN hMessage ("Message").
/*Send the message to "myqueue" */
RUN sendToQueue IN hPTPSession ("myQueue", hMessage, ?, ?, ?)
/* Delete message and session */
RUN deleteMessage IN hMessage.
RUN deleteSession IN hPTPSession.
```

Complete code for receiving a message using a PTP session

The following sample summarizes the steps for receiving a message.

```ABL
/* Receives a message from myqueue. */
DEFINE VARIABLE hConsumer AS HANDLE NO-UNDO.
DEFINE VARIABLE hPTPSession AS HANDLE NO-UNDO.
/* Creates PTP session*/
RUN jms/ptpsession.p PERSISTENT SET hPTPSession ("-H localhost -S 5162")
/*Connects to the broker */
RUN setBrokerURL IN hPTPSession ("tcp://machinename:2506").
RUN beginSession IN hPTPSession.
/* Messages received from myqueue are handled by the "myintproc" procedure. */
RUN createMessageConsumer IN hPTPSession
   (THIS-PROCEDURE, "myintproc", OUTPUT hConsumer).
RUN receiveFromQueue IN hPTPSession ("myQueue", ?, hConsumer).
RUN startReceiveMessages IN hPTPSession.
/* Wait to receive the messages. */
WAIT-FOR u1 OF THIS-PROCEDURE.
/* Delete session */
RUN deleteSession IN hPTPSession.
PROCEDURE myintproc:
   DEFINE INPUT PARAMETER hMessage AS HANDLE NO-UNDO.
   DEFINE INPUT PARAMETER hConsumer AS HANDLE NO-UNDO.
   DEFINE OUTPUT PARAMETER hReply AS HANDLE NO-UNDO.
   /* Business logic here */
   . . .
/* Delete message */
RUN deleteMessage IN hMessage.
APPLY "U1" TO THIS-PROCEDURE.
END.
```
Pub/Sub messaging example

A Pub/Sub messaging example consists of the basic steps described in the following sections:

- Creating a Pub/Sub session procedure on page 113
- Connecting to the broker on page 110
- Creating a Message Subscriber on page 113
- Subscribing to a topic on page 114
- Publishing to a topic on page 114
- Consuming messages from a topic on page 114
- Deleting a message on page 111
- Summary on page 112

Creating a Pub/Sub session procedure

In the following example, the application creates a session object by calling `pubsubsession.p` persistently.

```
DEFINE VARIABLE hPTPSession AS HANDLE NO-UNDO.
RUN jms/pubsubsession.p PERSISTENT
    SET hPubSubSession ("-H localhost -S 5162 ").
```

Creating the session object specifies the connection parameters to the SonicMQ Broker. This allows an application to set different session-level attributes before starting the JMS session. The connection to the SonicMQ Broker and the JMS session does not occur until the application calls the `beginSession procedure` on page 180.

Connecting to the broker

The OpenEdge application connects to the SonicMQ Broker to begin exchanging messages, as follows:

```
RUN setBrokerURL IN hPubSubSession ("tcp://machinename:2506").
RUN beginSession IN hPubSubSession.
```

Creating a Message Subscriber

You create a message subscriber to receive the message from the topic `newtopic`. The subscriber handles the message using the internal procedure `myintproc`, as shown in the following example.
Creating a Message Subscriber

```abl
/* Receives requests from the newTopic */
RUN createMessageConsumer IN hPubSubSession
   (THIS-PROCEDURE, "myintproc", OUTPUT hConsumer).
```

**Note:** Topics can be configured at run time.

Subscribing to a topic

Applications subscribe to topics of interest. To subscribe to a topic, the application subscribes to a topic and prepares to receive messages from the topic, as shown in the following example.

```
/* Subscribes to newtopic */
RUN SUBSCRIBE IN hPubSubSession ("newTopic", ?, ?, NO, hConsumer).
/* Start receiving requests */
RUN startReceiveMessages IN hPubSubSession.
/* Wait to receive the messages. */
WAIT-FOR u1 OF THIS-PROCEDURE.
```

Publishing to a topic

An application uses the publish procedure to publish messages to a topic, as shown in the following example.

```
DEFINE VARIABLE hMessage AS HANDLE NO-UNDO.
/* Code to create message */
RUN publish IN hPubSubSession ("newTopic", hMessage, ?, ?, ?).
```

Consuming messages from a topic

The Message Consumer receives a message from the topic and executes the business logic.

Deleting messages

The application deletes the messages after it finishes using them, as shown in the following example.

```
RUN deleteMessage IN hMessage.
```
Complete code for publish a message using a Pub/Sub session

The following sample summarizes the steps for sending a message.

```apl
/* Publishes a message to newtopic. */
DEFINE VARIABLE hMessage AS HANDLE NO-UNDO.
DEFINE VARIABLE hPubSubSession AS HANDLE NO-UNDO.
/* Creates the Pub/Sub session */
RUN jms/pubsubsession.p PERSISTENT
  SET hPubSubSession ("-H localhost -S 5162 ").
/*Connects to the broker */
RUN setBrokerURL IN hPubSubSession ("tcp://machinename:2506").
RUN beginSession IN hPubSubSession.
/* Create a message */
RUN create...Message IN hPTPSession (OUTPUT hMessage).
RUN set... IN hMessage ("Message").
/* Publish the message on the "newTopic" topic */
RUN publish IN hPubSubSession ("newTopic", hMessage, ?, ?, ?).
/* Delete message and session */
RUN deleteMessage IN hMessage.
RUN deleteSession IN hPubSubSession.
```

Complete code for receiving a message using a Pub/Sub session

The following sample summarizes the steps for receiving a message.

```apl
/* Subscribes and receives a message from myTopic. */
DEFINE VARIABLE hConsumer AS HANDLE NO-UNDO.
DEFINE VARIABLE hPubSubSession AS HANDLE NO-UNDO.
/* Creates the Pub/Sub session. */
RUN jms/pubsubsession.p PERSISTENT
  SET hPubSubSession ("-H localhost -S 5162 ").
/*Connects to the broker */
RUN setBrokerURL IN hPubSubSession ("tcp://machinename:2506").
RUN beginSession IN hPubSubSession.
/* Subscriptions to the newTopic topic. Received messages are handled by the
myintproc internal procedure. */
RUN createMessageConsumer IN hPubSubSession
  (THIS-PROCEDURE, "myintproc", OUTPUT hConsumer).
/* Subscriptions to newtopic */
RUN SUBSCRIBE IN hPubSubSession ("newTopic", ?, ?, NO, hConsumer).
/* Start receiving requests */
RUN startReceiveMessages IN hPubSubSession.
/* Wait to receive the messages. */
WAIT-FOR u1 OF THIS-PROCEDURE.
/* Delete session */
RUN deleteSession IN hPTPSession.
PROCEDURE myintproc:
  DEFINE INPUT PARAMETER hMessage AS HANDLE NO-UNDO.
  DEFINE INPUT PARAMETER hConsumer AS HANDLE NO-UNDO.
  DEFINE OUTPUT PARAMETER hReply AS HANDLE NO-UNDO.
/* Business logic here */
  ...
/* Delete message. */
RUN deleteMessage IN hMessage.
APPLY "U1" TO THIS-PROCEDURE.
END.
```
Programming scenarios

The following sections describe additional programming considerations and scenarios:

- **Using JMS 1.1 unified domain model** on page 116
- **Using ServerConnect and ClientConnect** on page 116
- **Client persistence** on page 117
- **Enhanced XML support** on page 118
- **TempTableMessage** on page 125
- **DataSetMessage** on page 128
- **Fault tolerance** on page 131

Using JMS 1.1 unified domain model

Prior to OpenEdge Release 10.1, OpenEdge clients were required to create a JMS session that was either for PTP or Pub/Sub messaging. If the client needed to use both queues and topics, it was necessary to create two JMS sessions.

Currently, OpenEdge supports JMS 1.1. JMS 1.1 unifies the two messaging domains into one domain. Therefore, OpenEdge clients may utilize both PTP or Pub/Sub messaging within the same JMS session. You can access both queues and topics using the same JMS session object. The following example shows the ABL code for using queues and topics in the same JMS session object:

```
RUN jms/jmsession.p PERSISTENT SET hSession (adapterConnection).
```

**Note:** The ModChat example demonstrates using the `jmsession` to perform both PTP and Pub/Sub messaging. The example uses a serialized connection object and server-based message selectors. For information on locating the examples, see the OpenEdge messages on page 24. For an alphabetical API reference, see ABL - JMS API Reference on page 161.

Using ServerConnect and ClientConnect

OpenEdge clients can connect directly to a SonicMQ Broker by using the OpenEdge Adapter for SonicMQ for a messaging session. By connecting directly to the SonicMQ Broker, the OpenEdge client has better control over connection management, and there is no need to manage and configure a OpenEdge Adapter for SonicMQ server process. Additional benefits include the availability of client persistence and fault tolerance. For more information on client persistence, see **Client persistence** on page 70. For more information on fault tolerance, see **Fault tolerance** on page 71.

**Caution:** This method creates a larger run-time footprint for your OpenEdge client or AppServer/WebSpeed process.
Using SMQConnect on a client

In the following example, the application creates a session procedure by calling `jmssession.p` persistently specifying the `-SMQConnect` connection parameter.

```
DEFINE VARIABLE hSession AS HANDLE NO-UNDO.
RUN jms/jmssession.p PERSISTENT SET hSession ("-SMQConnect").
RUN setBrokerURL IN hSession ("MQBrokerHost:2506").
RUN beginSession IN hSession.
```

**Note:** Each messaging session creates a connection to the SonicMQ Broker. You can minimize the number of connections to the SonicMQ Broker by using the AppServer or WebSpeed process to execute a shared ABL–JMS session.

Prior to using ServerConnect in an AppServer or WebSpeed server, the AppServer or WebSpeed server must be enabled for **SonicMQ ServerConnect enabled** using Progress Explorer.

To enable the AppServer or WebSpeed server for ServerConnect:

1. Select the Messaging properties for the AppServer or WebSpeed server.
2. Select the **SonicMQ ServerConnect enabled** check box, as shown:

   ![SonicMQ ServerConnect enabled properties](image)

3. Select unique broker and server log filenames.
4. Select the logging level.

   These settings start a SonicMQ ServerConnect process when the AppServer or WebSpeed server starts with specified logging options. After starting the AppServer or WebSpeed server, ensure the SonicMQ Broker is running.

**Client persistence**

*Client persistence* allows the JMS session to continue sending messages regardless of the SonicMQ Broker status. If the SonicMQ Broker is not available, the messages are stored locally and sent when the SonicMQ Broker becomes available.
For more information on client persistence, see Client persistence on page 70.

The following code sample shows how to set up client persistence.

**Client persistence example**

```abl
DEFINE VARIABLE hSession AS HANDLE NO-UNDO.
DEFINE VARIABLE rejectedMsgH AS HANDLE NO-UNDO.
/* Run adapter as symbiotic process */
RUN jms/jmsession.p PERSISTENT SET hSession ("-SMQConnect").
/* Set local store directory off of current working directory */
RUN setLocalStoreDirectory IN hSession ("mqstore").
RUN setLocalStoreSize IN hSession (5000).
/* Set timeouts - Retry every 5 minutes and give up if broker down 10 hours */
RUN setReconnectTimeout IN hSession (600).
RUN setReconnectInterval IN hSession (300).
RUN setClientPersistence IN hSession (TRUE).
RUN setBrokerURL IN hSession ("MQBrokerHost:2506").
RUN setClientID IN hSession ("SomeUniqueName").
RUN beginSession IN hSession.
/* Once session is established, create rejected Message Consumer */
RUN createRejectedMessageConsumer IN hSession (THIS-PROCEDURE, "RejectedMsgHandler", OUTPUT rejectedMsgH).

PROCEDURE RejectedMsgHandler:
  DEFINE INPUT PARAMETER hMessage AS HANDLE NO-UNDO.
  DEFINE INPUT PARAMETER hMessageConsumer AS HANDLE NO-UNDO.
  /* hAutoReply is not used in this example */
  DEFINE OUTPUT PARAMETER hAutoReply AS HANDLE NO-UNDO.
  DEFINE VARIABLE contentType AS CHARACTER NO-UNDO.
  DEFINE VARIABLE errorCode AS CHARACTER NO-UNDO.
  DEFINE VARIABLE errorText AS CHARACTER NO-UNDO.
  DEFINE VARIABLE exceptionCode AS CHARACTER NO-UNDO.
  DEFINE VARIABLE msgType AS CHARACTER NO-UNDO.
  MESSAGE "Reject message" VIEW-AS ALERT-BOX.
  errorCode = DYNAMIC-FUNCTION("getCharProperty" IN hMessage, "errorCode").
  errorText = DYNAMIC-FUNCTION("getCharProperty" IN hMessage, "errorText").
  exceptionCode = DYNAMIC-FUNCTION("getCharProperty" IN hMessage, "exception").
  MESSAGE errorText VIEW-AS ALERT-BOX.
  DISPLAY errorCode exceptionCode.
  iNumParts = DYNAMIC-FUNCTION("getPartCount" IN hMessage).
  IF DYNAMIC-FUNCTION("isMessagePart" IN hMessage, i) = TRUE THEN DO:
    contentType = DYNAMIC-FUNCTION("getMessageType":U IN hMessage, INPUT iNumParts, OUTPUT hMessagePart).
    msgType = DYNAMIC-FUNCTION("getMessageType":U IN hMessagePart).
    DISPLAY iNumParts msgType contentType.
  END.
RUN deleteMessage IN hMessage.
END PROCEDURE.
```

**Enhanced XML support**

Prior to OpenEdge Release 10.1, the OpenEdge Adapter for SonicMQ supported using the XMLMessage type if the client created the message as text in a well-formed XML document.

Currently, OpenEdge clients can send additional types of data, such as temp-tables and ProDatasets, as XMLMessage. The TempTableMessage and DataSetMessage transport data to the SonicMQ Broker using XML. ABL has built-in functionality to transform TEMP-TABLE or ProDataSet data into XML. Additionally, OpenEdge clients read, write, and parse XML using SAX-READER, SAX-WRITER, and X-DOCUMENT.

For more information on accessing the examples files, see OpenEdge messages on page 24.
The following example shows how to use the **SAX-WRITER** object.

**Saxwriter message**

```plaintext
DEFINE VARIABLE hSaxWriter AS HANDLE NO-UNDO.
DEFINE VARIABLE hSession AS HANDLE NO-UNDO.
DEFINE VARIABLE hMesg AS HANDLE NO-UNDO.
RUN jms/jmssession.p PERSISTENT SET hSession ("-SMQConnect").
RUN SetBrokerURL IN hSession ("-H localhost -S 5162 ").
RUN beginSession IN hSession.
RUN createXMLMessage IN hSession (OUTPUT hMesg).
/* Will write an envelope address */
hSaxWriter = DYNAMIC-FUNCTION('getSaxWriter':u IN hMesg, ?).
/* Want to format this so it is easy to read */
hSaxWriter:FORMATTED = TRUE.
hSaxWriter:START-DOCUMENT().
/* The ENCODING attribute defaults to UTF-8 */
/* The FRAGMENT attribute defaults to FALSE */
/* The STRICT attribute defaults to TRUE */
hSaxWriter:START-ELEMENT("mailingaddress").
RUN xmlData(INPUT "name", INPUT "Joe Perry").
hSaxWriter:START-ELEMENT("address").
/* This node contains an attribute */
hSaxWriter:INSERT-ATTRIBUTE("type", "personal").
RUN xmlData(INPUT "street", INPUT "11 Sugarland St.").
RUN xmlData(INPUT "city", INPUT "Somerville").
RUN xmlData(INPUT "state", INPUT "MA").
RUN xmlData(INPUT "zipcode", INPUT "02143").
hSaxWriter:END-ELEMENT("address").
hSaxWriter:START-ELEMENT("address").
hSaxWriter:INSERT-ATTRIBUTE("type", "business").
RUN xmlData(INPUT "name", INPUT "Progress Software").
RUN xmlData(INPUT "street", INPUT "14 Oak Park").
RUN xmlData(INPUT "city", INPUT "Bedford").
RUN xmlData(INPUT "state", INPUT "MA").
RUN xmlData(INPUT "zip", INPUT "01730").
hSaxWriter:WRITE-EMPTY-ELEMENT("default").
hSaxWriter:INSERT-ATTRIBUTE("type", "personal").
hSaxWriter:END-ELEMENT("mailingaddress").
hSaxWriter:END-DOCUMENT().
RUN deleteSaxWriter IN hMesg (saxWriterH).
RUN sendToQueue IN hSession ("SampleQ1", hMesg, ?, ?, ?).
RUN deleteMessage IN hMesg.
PROCEDURE xmlData:
  DEFINE INPUT PARAMETER xmlNode AS CHARACTER NO-UNDO.
  DEFINE INPUT PARAMETER charData AS CHARACTER NO-UNDO.
  hSaxWriter:START-ELEMENT(xmlNode).
  hSaxWriter:WRITE-CHARACTERS(charData).
  hSaxWriter:END-ELEMENT(xmlNode).
END PROCEDURE.
```

The following code sample shows how to use the **setSaxReader** procedure:

---

OpenEdge Development: Messaging and ESB

Programming scenarios
setSaxReader example

```
DEFINE VARIABLE hSax AS HANDLE NO-UNDO.
CREATE SAX-READER hSax.
/* SAX-READER setup as needed */
PROCEDURE messageHandler:
    DEFINE INPUT PARAMETER hMessage AS HANDLE NO-UNDO.
    DEFINE INPUT PARAMETER hMessageConsumer AS HANDLE NO-UNDO.
    DEFINE OUTPUT PARAMETER hAutoReply AS HANDLE NO-UNDO.
    DEFINE VARIABLE hResult AS HANDLE NO-UNDO.
    DEFINE VARIABLE mType AS CHARACTER NO-UNDO.
    mType = DYNAMIC-FUNCTION('getMessageType':u IN hMessage).
    CASE mType:
        WHEN "TempTableMessage" THEN DO:
            hResult = DYNAMIC-FUNCTION('GetTempTable':u IN hMessage, ?, ?, ?).
            /* TempTable actions as needed */
        END.
        WHEN "DatasetMessage" THEN DO:
            hResult = DYNAMIC-FUNCTION('GetDataSet':u IN hMessage, ?, ?, ?).
            /* DataSet actions as needed */
        END.
        WHEN "XMLMessage" THEN DO:
            RUN setSaxReader IN hMessage (hSax).
            hSax:SAX-PARSE().
        END.
    END CASE.
    RUN deleteMessage IN hMessage.
    END.
```

The SAX-WRITER object reads XML from a file using the SAX-READER object and send it to a queue using an XMLMessage. The following example shows how to use the SAX-WRITER object.
saxSender.p

DEFINE VARIABLE hd1 AS HANDLE NO-UNDO.
DEFINE VARIABLE hd2 AS HANDLE NO-UNDO.
DEFINE VARIABLE hMesg AS HANDLE NO-UNDO.
DEFINE VARIABLE hSession AS HANDLE NO-UNDO.
/* Start up the session to the SonicMQ broker */
RUN jms/ptpsession.p PERSISTENT SET hSession ("-H localhost -S 5162 ").
RUN setBrokerURL IN hSession ("localhost:2506").
RUN beginSession IN hSession.
/* Create the Sonic XML message */
RUN createXMLMessage IN hSession (OUTPUT hMesg).
/* The Adapter function GetSaxWriter will return a handle to a newly created 
SAX-WRITER and set its output to a destination local to the Adapter. The 
application may then make normal SAX-WRITER calls using this handle.*/
hd2 = DYNAMIC-FUNCTION('GetSaxWriter':u IN hMesg, ?).
hd2:START-DOCUMENT().
/* Create a SAX-READER to read in the xml file to be sent in a SonicMQ XML 
message. The SAX-READER callback procedures will use the handle to the 
SAX-WRITER just created to copy the XML. */
CREATE SAX-READER hd1.
hd1:SET-INPUT-SOURCE("FILE", "personal.xml").
hd1:SAX-PARSE().
DELETE OBJECT hd1.
hd2:END-DOCUMENT().
/* The Adapter procedure DeleteSaxWriter will copy the XML written with the 
SAX-WRITER calls into the SonicMQ XML message. It will then delete the 
SAX-WRITER specified by the handle. */
RUN DeleteSaxWriter IN hMesg (hd2).
/* Send the XML message. */
RUN sendToQueue IN hSession ("SampleQ1", hMesg, ?, ?, ?).
/* Disconnect the session from the SonicMQ broker */
RUN deleteMessage IN hMesg.
RUN deleteSession IN hSession.
/***/
/* callbacks for the SAX-READER function SAX-PARSE() */
/***/
/* Procedure is called when the parser finds a start tag for an element. */
PROCEDURE StartElement:
DEFINE INPUT PARAMETER namespaceURI AS CHARACTER NO-UNDO.
DEFINE INPUT PARAMETER cLocalName AS CHARACTER NO-UNDO.
DEFINE INPUT PARAMETER qname AS CHARACTER NO-UNDO.
DEFINE INPUT PARAMETER attributes AS HANDLE NO-UNDO.
hd2:START-ELEMENT(cLocalName, namespaceURI).
END PROCEDURE.
/* This callback gets passed the character data for an element. */
PROCEDURE Characters:
DEFINE INPUT PARAMETER mCharData AS MEMPTR NO-UNDO.
DEFINE INPUT PARAMETER iNumChars AS INTEGER NO-UNDO.
DEFINE VARIABLE cData AS CHARACTER NO-UNDO.
cData = GET-STRING(mCharData, 1, GET-SIZE(mCharData)).
hd2:WRITE-CHARACTERS(cData).
END PROCEDURE.
/* This callback is called when the parser finds the end tag for an Element. */
PROCEDURE EndElement:
DEFINE INPUT PARAMETER namespaceURI AS CHARACTER NO-UNDO.
DEFINE INPUT PARAMETER cLocalName AS CHARACTER NO-UNDO.
DEFINE INPUT PARAMETER qName AS CHARACTER NO-UNDO.
DEFINE INPUT PARAMETER attributes AS HANDLE NO-UNDO.
hd2:END-ELEMENT(cLocalName, namespaceURI). 
END.

The SAX-READER object reads an XMLMessage from a queue and writes it to a LONGCHAR. The following example shows how to use the SAX-READER object.
saxReceiver.p

DEFINE VARIABLE hdl2 AS HANDLE NO-UNDO.
DEFINE VARIABLE hSession AS HANDLE NO-UNDO.
DEFINE VARIABLE lch AS LONGCHAR NO-UNDO
    VIEW-AS EDITOR SIZE 70 BY 30 LARGE.
DEFINE VARIABLE hMsgConsumer AS HANDLE NO-UNDO.
DEFINE VARIABLE hMesg AS HANDLE NO-UNDO.
DEFINE VARIABLE stillWaiting AS LOGICAL NO-UNDO INITIAL TRUE.
/* Start up the session to the SonicMQ broker */
RUN jms/ptpsession.p PERSISTENT SET hSession ("-H localhost -S 5162 ").
RUN setBrokerURL IN hSession ("localhost:2506").
RUN beginSession IN hSession.
/* Create the message consumer and start receiving messages. */
RUN createMessageConsumer IN hSession
    (THIS-PROCEDURE, "messagehandler", OUTPUT hMsgConsumer).
RUN receiveFromQueue IN hSession ("SampleQ1", ?, hMsgConsumer).
RUN startReceiveMessages IN hSession.
/* Wait for all messages to be received. */
RUN waitForMessages IN hSession ("inWait", THIS-PROCEDURE, ?).
RUN deleteSession IN hSession.
/* Message handler procedure */
PROCEDURE messageHandler:
    DEFINE INPUT PARAMETER hMessage AS HANDLE NO-UNDO.
    DEFINE INPUT PARAMETER hMessageConsumer AS HANDLE NO-UNDO.
    DEFINE OUTPUT PARAMETER hAutoReply AS HANDLE NO-UNDO.
    DEFINE VARIABLE hdl1 AS HANDLE NO-UNDO.
    CREATE SAX-WRITER hdl2.
    hdl2:SET-OUTPUT-DESTINATION("LONGCHAR",lch).
    CREATE SAX-READER hdl1.
    /* The Adapter procedure SetSaxReader will set the input source for a 
    SAX-READER to the XML message that has been received. The application may 
    then use normal SAX-READER calls to access the XML from the message. */
    RUN SetSaxReader IN hMessage (hdl1).
    hdl1:SAX-PARSE().
    DELETE OBJECT hdl1.
    RUN deleteMessage IN hMessage.
    hdl2:END-DOCUMENT().
    DISPLAY lch.
    ASSIGN
        lch = ""
        stillWaiting = FALSE.
END PROCEDURE.
FUNCTION inWait RETURNS LOGICAL:
    RETURN stillWaiting.
END.
******************************************************************************
/* callbacks for the SAX-READER function SAX-PARSE() */
******************************************************************************
/* Procedure is called when the parser finds the start tag for an element. */
PROCEDURE StartElement:
    DEFINE INPUT PARAMETER namespaceURI AS CHARACTER NO-UNDO.
    DEFINE INPUT PARAMETER cLocalName AS CHARACTER NO-UNDO.
    DEFINE INPUT PARAMETER qname AS CHARACTER NO-UNDO.
    DEFINE INPUT PARAMETER attributes AS HANDLE NO-UNDO.
    hdl2:START-ELEMENT(cLocalName, namespaceURI).
END.
/* This callback gets passed the character data for an element.*/
PROCEDURE Characters:
    DEFINE INPUT PARAMETER charData AS MEMPTR NO-UNDO.
    DEFINE INPUT PARAMETER nNumChars AS INTEGER NO-UNDO.
    DEFINE VARIABLE data AS CHARACTER NO-UNDO.
    data = GET-STRING(charData, 1, GET-SIZE(charData)).
    hdl2:WRITE-CHARACTERS(data).
END PROCEDURE.
/* This callback is called when the parser finds the end tag for an Element. */
PROCEDURE EndElement:
The following code sample shows how to use the `setX-Document` procedure on page 317.

### setX-Document example

```plaintext
CREATE X-DOCUMENT hd1.
hd1:LOAD("file", "4k.xml", false).
RUN createXMLMessage IN hSession (OUTPUT hMesg).
RUN setX-Document IN hMesg(hdl1).
RUN sendToQueue IN hSession ("SampleQ1", hMesg, ?, ?, ?).
RUN deleteMessage IN hMesg.
```

The following sample code shows how to use `getX-Document` function on page 248.

### getX-Document example

```plaintext
PROCEDURE messageHandler:
DEFINE INPUT PARAMETER hMessage AS HANDLE NO-UNDO.
DEFINE INPUT PARAMETER hMessageConsumer AS HANDLE NO-UNDO.
DEFINE OUTPUT PARAMETER hAutoReply AS HANDLE NO-UNDO.
DEFINE VARIABLE mtype AS CHARACTER NO-UNDO.
DEFINE VARIABLE hResult AS HANDLE NO-UNDO.
mtype = DYNAMIC-FUNCTION('getMessageType':u IN hMessage).
CASE mtype:
  WHEN "TemptableMessage" THEN DO:
    hResult = DYNAMIC-FUNCTION('GetTempTable':u IN hMessage, ?, ?, ?).
    /* TempTable actions as needed */
  END.
  WHEN "DatasetMessage" THEN DO:
    hResult = DYNAMIC-FUNCTION('GetDataSet':u IN hMessage, ?, ?, ?).
    /* DataSet actions as needed */
  END.
  WHEN "XMLMessage" THEN DO:
    hResult = DYNAMIC-FUNCTION('getX-Document':u IN hMessage).
    /* X-DOCUMENT calls as needed */
  END.
END CASE.
RUN deleteMessage IN hMessage.
END.
```

The `X-DOCUMENT` object reads XML from a file and sends it to a queue using an `XMLMessage`. The following example shows how to use `X-DOCUMENT` object to send a message.
The X-DOCUMENT object reads an XMLMessage from a queue and writes it to a LONGCHAR. The following example shows how to use X-DOCUMENT object to receive a message.
domReceiver.p

```plaintext
DEFINE VARIABLE hSession AS HANDLE NO-UNDO.
DEFINE VARIABLE lch AS LONGCHAR NO-UNDO
    VIEW-AS EDITOR SIZE 70 BY 30 LARGE.
DEFINE VARIABLE hMsgConsumer AS HANDLE NO-UNDO.
DEFINE VARIABLE stillWaiting AS LOGICAL NO-UNDO INITIAL TRUE.
/* Start up the session to the SonicMQ broker */
RUN jms/ptpsession.p PERSISTENT SET hSession ('-H localhost -S 5162 ').
RUN setBrokerURL IN hSession ('localhost:2506').
RUN beginSession IN hSession.
/* Create the message consumer and start receiving messages. */
RUN createMessageConsumer IN hSession
    (THIS-PROCEDURE, "messageHandler", OUTPUT hMsgConsumer).
RUN receiveFromQueue IN hSession ("SampleQ1", ?, hMsgConsumer).
RUN startReceiveMessages IN hSession.
/* Wait for all messages to be received. */
RUN waitForMessages IN hSession ("inWait", THIS-PROCEDURE, ?).
RUN deleteSession IN hSession.
/* Message handler procedure */
PROCEDURE messageHandler:
    DEFINE INPUT PARAMETER hMessage AS HANDLE NO-UNDO.
    DEFINE INPUT PARAMETER hMessageConsumer AS HANDLE NO-UNDO.
    DEFINE OUTPUT PARAMETER hAutoReply AS HANDLE NO-UNDO.
    DEFINE VARIABLE hd1 AS HANDLE NO-UNDO.
    /* The Adapter function GetX-Document will return a handle to a newly created X-DOCUMENT and load it with the XML message that has been received. The application may then use normal X-DOCUMENT calls to access the XML from the message. */
    hd1 = DYNAMIC-FUNCTION('GetX-Document':u IN hMessage).
    hd1:SAVE("LONGCHAR",lch).
    DELETE OBJECT hd1.
    RUN deleteMessage IN hMessage.
    DISPLAY lch.
    ASSIGN
        lch = ""
        stillWaiting = FALSE.
END PROCEDURE.
FUNCTION inWait RETURNS LOGICAL:
    RETURN stillWaiting.
END.
```

TempTableMessage

OpenEdge applications use temp-tables for data. The new TempTableMessage supports using temp-tables for JMS messaging. The XMLMessage is the basis for the TempTableMessage. A non-OpenEdge application receives a TempTableMessage as an XMLMessage. The JMS header property signals an OpenEdge application that the incoming message is a TempTableMessage.

For more information on accessing the examples files, see OpenEdge messages on page 24.

For an alphabetical API reference, see ABL - JMS API Reference on page 161.

The following example shows sending a TempTableMessage.
DEFINE VARIABLE hMessage AS HANDLE NO-UNDO.
DEFINE VARIABLE hSession AS HANDLE NO-UNDO.
DEFINE VARIABLE numRecs AS INTEGER NO-UNDO.
DEFINE VARIABLE ptp AS LOGICAL NO-UNDO.
DEFINE VARIABLE ttH AS HANDLE NO-UNDO.
DEFINE TEMP-TABLE ttCustomer NO-UNDO LIKE customer.
/* Creates a session object. */
RUN jms/jmssession.p PERSISTENT SET hSession ("-SMQConnect").
RUN setBrokerURL IN hSession ("localhost:2506").
RUN beginSession IN hSession.
FOR EACH customer NO-LOCK:
    CREATE ttCustomer.
    ASSIGN
        ttCustomer.CustNum = customer.CustNum
        ttCustomer.Name = customer.Name
        ttCustomer.Address = customer.Address
        ttCustomer.Address2 = customer.Address2
        ttCustomer.City = customer.City
        ttCustomer.State = customer.State
        numRecs = numRecs + 1.
END.

ttH = TEMP-TABLE ttCustomer:HANDLE.
RUN createTempTableMessage IN hSession (OUTPUT hMessage).
RUN setTempTable IN hMessage (ttH, ?, TRUE).
IF ptp THEN
    RUN sendToQueue IN hSession ("SampleQ1", hMessage, ?, ?, ?).
ELSE
    RUN PUBLISH IN hSession ("TestTopic", hMessage, ?, ?, ?).
RUN deleteMessage IN hMessage.
DEDELETE ttCustomer.
MESSAGE "Number of records processed: " + STRING(numRecs).
RUN createTextMessage in hSession (OUTPUT hMessage).
RUN setText IN hMessage (STRING(numRecs)).
IF ptp THEN
    RUN sendToQueue IN hSession ("SampleQ1", hMessage, ?, ?, ?).
ELSE
    RUN PUBLISH IN hSession ("TestTopic", hMessage, ?, ?, ?).
RUN deleteMessage IN hMessage.
RUN deleteSession IN hSession.
**ttmsg_recv.p**

The following example shows receiving a TempTableMessage.

```plaintext
DEFINE VARIABLE hMsgConsumer AS HANDLE NO-UNDO.
DEFINE VARIABLE hMesg AS HANDLE NO-UNDO.
DEFINE VARIABLE hSession AS HANDLE NO-UNDO.
DEFINE VARIABLE numRecsRead AS INTEGER NO-UNDO.
DEFINE VARIABLE ptp AS LOGICAL NO-UNDO.
DEFINE VARIABLE stillWaiting AS LOGICAL NO-UNDO INITIAL TRUE.
RUN jms/jmssession.p PERSISTENT SET hSession ("-SMQConnect").
RUN setBrokerURL IN hSession ("localhost:2506").
RUN beginSession IN hSession.
RUN createMessageConsumer IN hSession
   (THIS-PROCEDURE, /* this procedure will handle it */
    "messageHandler", /* name of internal procedure */
    OUTPUT hMsgConsumer).
IF ptp THEN
   RUN receiveFromQueue IN hSession ("SampleQ1", ?, hMsgConsumer).
ELSE
   RUN subscribe IN hSession
      ("TestTopic", ?, /* durable subscription */
       ?, /* no message selector */
       TRUE, /* want to get my own publications */
       hMsgConsumer).
RUN startReceiveMessages IN hSession.
RUN waitForMessages IN hSession ("inWait", THIS-PROCEDURE, ?).
RUN deleteSession IN hSession.
MESSAGE "Number of records processed: " + STRING(numRecsRead).
PROCEDURE messageHandler:
   DEFINE INPUT PARAMETER hMessage AS HANDLE NO-UNDO.
   DEFINE INPUT PARAMETER hMessageConsumer AS HANDLE NO-UNDO.
   /* hAutoReply is not used in this example */
   DEFINE OUTPUT PARAMETER hAutoReply AS HANDLE NO-UNDO.
   DEFINE VARIABLE bh1 AS HANDLE NO-UNDO.
   DEFINE VARIABLE bh2 AS HANDLE NO-UNDO.
   DEFINE VARIABLE ttH1 AS HANDLE NO-UNDO.
   IF DYNAMIC-FUNCTION("getMessageType" in hMessage) =
      "TempTableMessage" THEN DO:
      ttH1 = DYNAMIC-FUNCTION("getTempTable" IN hMessage, ?, ?, ?).
      numRecsRead = numRecsRead + 1.
      bh1 = ttH1:DEFAULT-BUFFER-HANDLE.
      CREATE QUERY qh1.
      qh1:SET-BUFFERS(bh1).
      qh1:QUERY-PREPARE("for each tcust").
      qh1:QUERY-OPEN().
      qh1:GET-FIRST.
      REPEAT WHILE NOT qh1:QUERY-OFF-END:
         bh2 = bh1:BUFFER-FIELD("name").
         MESSAGE bh2:STRING-VALUE().
         qh1:GET-NEXT.
      DELETE OBJECT bh2.
   END.
   qh1:QUERY-CLOSE.
   DELETE OBJECT ttH1.
   DELETE OBJECT qh1.
END. /* getMessageType */
ELSE
   stillWaiting = FALSE.
RUN deleteMessage IN hMessage.
END PROCEDURE.
FUNCTION inWait RETURNS LOGICAL:
   RETURN stillWaiting.
END.
```
DataSetMessage

OpenEdge applications use ProDataSets for data. The new DataSetMessage supports using ProDataSets for JMS messaging. The XMLMessage is the basis for the DataSetMessage. The OpenEdge Adapter for SonicMQ converts the ProDataSet to/from XML and uses the XMLMessage to send/receive ProDataSets. A non-OpenEdge application receives a DataSetMessage as an XMLMessage. The JMS header property signals an OpenEdge application that the incoming message is a DataSetMessage.

For more information on accessing the examples files, see OpenEdge messages on page 24.

For an alphabetical API reference, see ABL - JMS API Reference on page 161.

The following example shows sending a DataSetMessage.
**dsmsg_send.p**

```plaintext
DEFINE VARIABLE hds1 AS HANDLE NO-UNDO.
DEFINE VARIABLE hMessage AS HANDLE NO-UNDO.
DEFINE VARIABLE hSession AS HANDLE NO-UNDO.
DEFINE VARIABLE ptp AS LOGICAL NO-UNDO INITIAL TRUE.
DEFINE VARIABLE ret AS LOGICAL NO-UNDO.

/* Definition for TEMP-TABLE ttCustomer */
DEFINE TEMP-TABLE ttCustomer NO-UNDO BEFORE-TABLE ttCustBef
  FIELD CustNum LIKE Customer.CustNum
  FIELD Name LIKE Customer.Name COLUMN-LABEL "custlab"
  XML-NODE-TYPE "Attribute"
  FIELD Country LIKE Customer.Country
  FIELD Comments LIKE Customer.Comments FORMAT "x(40)"
  INDEX CustNum IS PRIMARY UNIQUE CustNum
  INDEX Name IS WORD-INDEX Comments.

/* Definition for TEMP-TABLE ttOrder */
DEFINE TEMP-TABLE ttOrder NO-UNDO
  FIELD OrderNum LIKE Order.OrderNum
  FIELD CustNum LIKE Order.CustNum
  FIELD OrderDate LIKE Order.OrderDate
  INDEX OrderNum IS PRIMARY UNIQUE OrderNum
  INDEX CustOrder IS UNIQUE CustNum OrderNum
  INDEX OrderDate OrderDate.

DEFINE DATASET myds
  NAMESPACE-URI "urn:myds" NAMESPACE-PREFIX "ds"
  FOR ttCustomer, ttOrder
  DATA-RELATION custOrd FOR ttCustomer, ttOrder
  REPOSITION RELATION-FIELDS (CustNum, CustNum).

/* Creates a session object. */
IF ptp THEN
  RUN jms/ptpsession.p PERSISTENT SET hSession ("-SMQConnect").
ELSE
  RUN jms/pubsubsession.p PERSISTENT SET hSession ("-SMQConnect").
RUN setBrokerURL IN hSession ("localhost:2506").
RUN beginSession IN hSession.
FOR EACH Customer NO-LOCK WHERE Customer.CustNum < 4:
  CREATE ttCustomer.
  BUFFER-COPY Customer TO ttCustomer.
  FOR EACH Order OF Customer NO-LOCK:
    CREATE ttOrder.
    BUFFER-COPY Order TO ttOrder.
END. /* FOR EACH Order */
END. /* FOR EACH Customer */
hds1 = DATASET myds:HANDLE.
/* Uncomment to write XML to a file
*/
RUN createDatasetMessage in hSession (OUTPUT hMessage).
RUN dataSet IN hMessage( hds1, ?, TRUE).
IF ptp THEN
  RUN sendToQueue IN hSession ("SampleQ1", hMessage, ?, ?, ?).
ELSE
  RUN PUBLISH IN hSession ("TestTopic", hMessage, ?, ?, ?).
RUN deleteMessage IN hMessage.
RUN deleteSession in hSession.
```

The following example shows receiving a `DataSetMessage`.

**dsmsg_recv.p**

```plaintext
DEFINE VARIABLE hBuf AS HANDLE NO-UNDO.
DEFINE VARIABLE hds2 AS HANDLE NO-UNDO.
```
DEFINE VARIABLE hMesg  AS HANDLE NO-UNDO.
DEFINE VARIABLE hMsgConsumer AS HANDLE NO-UNDO.
DEFINE VARIABLE hq    AS HANDLE NO-UNDO.
DEFINE VARIABLE hrel  AS HANDLE NO-UNDO.
DEFINE VARIABLE ix    AS INTEGER NO-UNDO.
DEFINE VARIABLE jx    AS INTEGER NO-UNDO.
DEFINE VARIABLE numRecsRead AS INTEGER NO-UNDO.
DEFINE VARIABLE ptp   AS LOGICAL NO-UNDO INITIAL TRUE.
DEFINE VARIABLE ret   AS LOGICAL NO-UNDO.
DEFINE VARIABLE stillWaiting AS LOGICAL NO-UNDO INITIAL TRUE.
IF ptp THEN
  RUN jms/ptpsession.p PERSISTENT SET hSession ("-SMQConnect").
ELSE
  RUN jms/pubsubsession.p PERSISTENT SET hSession ("-SMQConnect").
RUN setBrokerURL IN hSession ("localhost:2506").
RUN beginSession IN hSession.
RUN createMessageConsumer IN hSession
  (THIS-PROCEDURE, /* this procedure will handle it */
   "messageHandler", /* name of internal procedure */
   OUTPUT hMsgConsumer).
IF ptp THEN
  RUN receiveFromQueue IN hSession ("SampleQ1", ?, hMsgConsumer).
ELSE
  RUN subscribe IN hSession
    ("TestTopic", ?, /* durable subscription */
     ?, /* no message selector */
     TRUE, /* want to get my own publications */
     hMsgConsumer).
RUN startReceiveMessages IN hSession.
RUN waitForMessages IN hSession ("inWait", THIS-PROCEDURE, ?).
RUN deleteSession IN hSession.
MESSAGE "Number of records processed: " + STRING(numRecsRead).
PROCEDURE messageHandler:
  DEFINE INPUT PARAMETER hMessage AS HANDLE NO-UNDO.
  DEFINE INPUT PARAMETER hMessageConsumer AS HANDLE NO-UNDO.
  DEFINE OUTPUT PARAMETER hAutoReply AS HANDLE NO-UNDO.
  DEFINE VARIABLE ttH1 AS HANDLE NO-UNDO.
  DEFINE VARIABLE bh1 AS HANDLE NO-UNDO.
  DEFINE VARIABLE bh2 AS HANDLE NO-UNDO.
  DEFINE VARIABLE qh1 AS HANDLE NO-UNDO.
  IF DYNAMIC-FUNCTION("getMessageType" in hMessage) =
    "DatasetMessage" THEN DO:
    ASSIGN
      hds2 = DYNAMIC-FUNCTION("getDataset" IN hMessage, ?, ?, ?)
      numRecsRead = numRecsRead + 1.
    MESSAGE
      "num-buffers: " hds2:NUM-BUFFERS "name: " hds2:NAME SKIP
      "namespace-info: " hds2:NAMESPACE-URI hds2:NAMESPACE-PREFIX SKIP
      "num-relations: " hds2:NUM-RELATIONS
      VIEW-AS ALERT-BOX.
    DO ix = 1 TO hds2:NUM-RELATIONS:
      hrel = hds2:GET-RELATION(ix).
      MESSAGE
        "rel name: " hrel:NAME SKIP
        "reposition: " hrel:REPOSITION SKIP
        "nested: " hrel:NESTED SKIP
        "where-str: " hrel:WHERE-STRING SKIP
        "parent: " hrel:PARENT-BUFFER:NAME SKIP
        "child: " hrel:CHILD-BUFFER:NAME SKIP
        "rel-fields: " hrel:RELATION-FIELDS
        VIEW-AS ALERT-BOX.
    END.
    DO jx = 1 TO hds2:NUM-BUFFERS:
      hBuf = hds2:GET-BUFFER-HANDLE(jx).
      MESSAGE "buf name: " hBuf:NAME VIEW-AS ALERT-BOX.
    DO ix = 1 TO hBuf:NUM-FIELDS:
Fault tolerance

Fault tolerant connections allow another SonicMQ Broker to take over if the original SonicMQ Broker fails. To ensure message delivery, use the fault-tolerant APIs to setup and enable fault tolerance. These APIs include the `setFaultTolerant` procedure on page 285, the `getFaultTolerant` function on page 213, the `isFaultTolerant` function on page 251, the `setConnectionURLs` procedure on page 276, the `setFaultTolerantReconnectTimeout` procedure on page 286, the `getFaultTolerantReconnectTimeout` function on page 214, the `setInitialConnectionTimeout` procedure on page 290, the `getInitialConnectionTimeout` function on page 217, the `setClientTransactionBufferSize` procedure on page 274, the `getClientTransactionBufferSize` function on page 203, and the `createChangeStateConsumer` procedure on page 185. Although you setup and enable fault tolerance from the SonicMQ client, the SonicMQ Broker must support it.

Note: Fault tolerance is only available to OpenEdge clients running in ClientConnect and ServerConnect.
After creating the session object, you must create the list of SonicMQ Brokers to use, set the fault tolerant property for the session, and then start the session.

Refer to the following sections for an:

- Example of setting up fault tolerance on page 132
- Example of a "ChangeState" handler (optional) on page 132

### Example of setting up fault tolerance

The following example shows how to set up a fault tolerant session.

#### Fault tolerance set up

```abl
DEFINE VARIABLE hSession AS HANDLE NO-UNDO.
RUN jms/jmssession.p PERSISTENT SET hSession ("-SMQConnect").
RUN setConnectionURLs IN hSession ("Primary:2508,BackupServer:9876").
RUN setFaultTolerant IN hSession(TRUE).
RUN beginSession IN hSession.
```

#### Example of a "ChangeState" handler (optional)

When the connection to the SonicMQ Broker is lost, SonicMQ has the ability to notify the application. A special asynchronous handler, "ChangeState" handler, notifies the client application whenever the state of the SonicMQ Broker changes. The character header property of the message passed to the "ChangeState" handler contains one of the following values: active, reconnecting, failed, or closed. You setup the handler by calling the `createChangeStateConsumer` procedure on page 185 after to calling the `beginSession` procedure on page 180.

#### Fault tolerant example

The following code sample shows how to use the `createChangeStateConsumer` procedure.

```abl
RUN createChangeStateConsumer IN hSession
    (THIS-PROCEDURE, "msgHandler", OUTPUT hMessage).
PROCEDURE msgHandler:
    DEFINE INPUT PARAMETER hMessage AS HANDLE NO-UNDO.
    DEFINE INPUT PARAMETER hMsgConsumer AS HANDLE NO-UNDO.
    DEFINE OUTPUT PARAMETER hReply AS HANDLE NO-UNDO.
    DEFINE VARIABLE cValue AS CHARACTER NO-UNDO.
    cValue = DYNAMIC-FUNCTION("getCharProperty" IN hMessage, "state").
    DISPLAY cValue.
    RUN deleteMessage IN hMessage.
END PROCEDURE.
```
Guidelines for Using and Programming for the OpenEdge Adapter for Sonic ESB

This section discusses certain general considerations related to OpenEdge Sonic ESB services and the operation of the OpenEdge Adapter for Sonic ESB. You can also find more detailed information in other documentation, as indicated in this section.

For details, see the following topics:

- Native Invocation methodology
- ESBOEGEN
- Configuring OpenEdge Architect for ESB annotations
- Web Services Invocation methodology
- Service definition considerations
- Sonic message handling run-time parameters

Native Invocation methodology

The OpenEdge Adapter for Sonic ESB supports a Native Invocation methodology for exposing ABL applications to the Sonic Enterprise Service Bus (ESB). When a Sonic ESB process is created using the Native Invocation methodology, ABL procedures are called directly via an OpenAPI call to an OpenEdge Application Server.

The Native Invocation methodology provides the following benefits:
• Simplified exposure of ABL code as a service
• Simplified process of mapping ABL parameters to Sonic messages
• Reduced overhead, improving run-time performance over Web Services Invocation methodology by eliminating conversions to and from SOAP messages

The Native Invocation methodology relies on invocation (.esboe) files that are imported directly into the Sonic Workbench.

At the highest level, the steps for exposing an ABL procedure are:

1. Develop your ABL code and create invocation files.
2. Import your invocation (.esboe) files into your Sonic project in Sonic Workbench.
3. Create a Sonic ESB Process, adding your invocation files to the process, one at a time, and mapping your input and output parameters to message definitions.
4. Save your ESB Process and upload it to your Sonic Domain for testing and deployment.

This process can be further simplified if you have at least Sonic V7.6 and have integrated OpenEdge Architect and Sonic Workbench into one Eclipse environment.

The sections that follow discuss these development steps in greater details. For a complete example of creating and testing an ESB process using a Native Invocation file, see Sample Native Invocation ESB process on page 361.

Creating an invocation file

OpenEdge developers have the choice of two approaches for creating an invocation file for ABL procedures and functions:

• **Declarative** — Capture information about publicly exposable procedures in the source code through the use of annotations

• **Non-declarative** — Capture information about publicly exposable procedures through the use of a tool such as ProxyGen

The declarative approach is recommended to OpenEdge developers as a best practice, entering relevant information for making a procedure, function or external procedure publicly exposable when the source code is written. This information is then stored with the source code, and during the build process captured as part of the r-code. In cases where the developer does not want to capture information about publicly exposable procedures with the source code, the non-declarative approach is supported.

Starting with OpenEdge Release 10.1C01 the ability, at development time, to drag and drop ABL procedure files directly into a Sonic ESB itinerary is supported. Dragging ABL procedure files directly into a Sonic ESB itinerary eliminates the need to create and import a .esboe file. If your source code is not already annotated with ESB annotations when you do the drag and drop, a wizard walks you through the steps required to add the necessary Native Invocation annotations.
The creation of an ESB itinerary based on .esboe files remains fully supported. The following figure graphically depicts the creation of an invocation .esboe file, following the declarative or non-declarative approach.

**Figure 3: Development flow: ABL source to ESB itinerary**

Declarative invocation files

Declarative invocation files rely on annotations. *Annotations* are a methodology to capture information in source code that extends the ABL language syntax. ABL source code annotations signal that external procedures, internal procedures, and user-defined functions are available to Sonic ESB processes. For external procedures, the annotation must precede all ABL statements in the source file. For internal procedures and user-defined functions, the annotation must precede the `PROCEDURE` or `FUNCTION` statement.
Annotation syntax

The Native Invocation annotation syntax is the constant "@openapi.openedge.export", followed by the scoping value "FILE" for external procedures, followed by a series of name-value pairs. The following table describes the annotation name-value pairs.

Table 35: Annotation name-value pairs (Continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>Default value</th>
<th>Mandatory/Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>ESB</td>
<td>Mandatory</td>
<td>Specifies the type of Open Client the procedure or function is exported to. The only type currently supported is ESB.</td>
</tr>
<tr>
<td>esboeFileName</td>
<td>For external procedures: %FILENAME% For internal procedures and functions: %FILENAME%_%PROCNAME%</td>
<td>Optional</td>
<td>Specifies the name of the .esboe file. The extension automatically appended. There is one file per annotation. If esboeFileName is not specified, the default value is used.</td>
</tr>
<tr>
<td>executionMode</td>
<td>external</td>
<td>Mandatory</td>
<td>Specifies the procedure will run persistently or externally. Specify executionMode=&quot;persistent&quot; for a file's external procedure before internal procedures and functions can be annotated. executionMode=&quot;persistent&quot; can only be specified for top-level external procedures.</td>
</tr>
<tr>
<td>useReturnValue</td>
<td>false</td>
<td>Optional</td>
<td>Specifies whether a return string is generated or not for a procedure. Specify useReturnValue=&quot;true&quot; to generate a return string. If useReturnValue is not specified, no return string is generated.</td>
</tr>
<tr>
<td>writeDataSetBeforeImage</td>
<td>false</td>
<td>Optional</td>
<td>Specifies whether before-image data is written out when ProDataSet parameters are serialized as XML. Specify writeDataSetBeforeImage=&quot;true&quot; to write the before-image data. Specify writeDataSetBeforeImage=&quot;false&quot; to not write the before-image data. If no ProDataSet parameters exist, writeDataSetBeforeImage is ignored.</td>
</tr>
</tbody>
</table>
Annotation examples

You can automatically add annotations in OpenEdge Architect, or you can add them manually. For details on adding annotations in OpenEdge Architect, see Annotating ABL in OpenEdge Architect on page 138. The following code samples illustrate annotations:

• External procedure

The following code sample illustrates annotation of an external procedure, foo.p. The only required name-value pairs are type and executionMode, as shown:

```plaintext
/* foo.p */
@openapi.openedge.export FILE (type="ESB", executionMode="external").
DEFINE INPUT PARAM bar AS INT.
DEFINE OUTPUT PARAM ney AS CHAR.
...
```

Processing this annotation creates the file foo.esboe that describes a non-persistent operation named foo. foo takes an input integer named bar and returns a character string named ney.

• Renamed .esboe file for an external procedure

The following code sample illustrates the annotation of an external procedure, foo.p, that renames the generated .esboe file:

```plaintext
/* foo.p */
@openapi.openedge.export FILE (type="ESB", executionMode="external").
esboeFileName="Renamed_foo".
DEFINE INPUT PARAM bar AS INT.
DEFINE OUTPUT PARAM ney AS CHAR.
...
```

This annotation creates the file Renamed_foo.esboe that describes a non-persistent operation on the OpenEdge Application Server.

• Internal procedure

The following code sample illustrates the annotation of an internal procedure, barr inside the file foo.p:

```plaintext
/* foo.p */
@openapi.openedge.export FILE (type="ESB", executionMode="persistent").
DEFINE INPUT PARAM bar AS INT.
DEFINE OUTPUT PARAM ney AS CHAR.
...@openapi.openedge.export(type="ESB")
PROCEDURE barr.
DEFINE INPUT PARAM abc AS INT.
DEFINE OUTPUT PARAM xyz AS CHAR.
...
```

The external procedure foo must be annotated as persistent before the internal procedure barr can be annotated. Three files are created based on these annotations:

• foo.esboe — foo.esboe runs the external procedure persistently

• foo_barr.esboe — foo_barr.esboe invokes the internal procedure
• foo_release.esboe — foo_release.esboe deletes the persistent procedure, unbinding the OpenEdge Application Server from the process

Annotating ABL in OpenEdge Architect

OpenEdge Architect's annotation capabilities include the creation of ESB Native Invocation annotations and the generation of .esboe files. See Configuring OpenEdge Architect for ESB annotations on page 154 to verify that OpenEdge Architect is configured to generate Native Invocation annotations. Once you have configured your project, you are ready to annotate your ABL source code. OpenEdge Architect provides several different methods:

• **ABL Editor** — You can add annotations by editing your ABL source and typing in the required information. You must match the syntax defined in Annotation syntax on page 136.

  **Note:** You can also start the Annotation wizard from the ABL Editor. Right mouse-click, and select Source > Add Annotation to invoke the wizard.

• **Annotation wizard** — You can annotate multiple source files at once with the Annotation wizard.

• **Adding annotations from Outline view** — You can add an annotation to a selected procedure or function from the Outline view.

**Annotating multiple source files at once**

To annotate multiple source files at once:

1. Start the wizard by selecting Source > Add Annotation. This displays the Add Annotations dialog:
2. Select **ESB Annotation - Main** from the **Select annotation or enter annotation text in editor** drop-down.

3. Check the files to annotate in **Available Resources**. Click **Finish** if you are only annotating external procedures, or click **Next** to also annotate internal procedures and functions.

4. ABL source files with internal procedures and functions are expandable in the **Selected Methods** tree view. Select the internal procedures and functions you want annotated and click **Finish**.

**Note:** The execution mode of the main annotation of an external procedure must be persistent if you are annotating internal procedures and functions. If not specified in this manner, you are prompted to change the execution mode of the external procedure.

### Adding annotations from the Outline view

**To add annotations from the Outline view:**

1. Highlight the procedure or function you want to annotate.

2. Right-click and select **New ESB Annotation** to bring up the following dialog box:
The **Detail ESB annotation** section is for your internal procedure or function. The **Main ESB annotation** section is for your external procedure.

3. You can change the default name of the generated .esboe file in the **ESBOE File name** field. If you want to use the return value, check **Use return value**. If you are using ProDataSets and want to write the dataset before-image data as serialized as XML, check **Write dataset before image**.

4. Click **OK** to add the annotation.

### Generating .esboe files

Once you have annotated your source, you generate your .esboe files. OpenEdge Architect provides two options for generating the files:

- If you configured your preferences to automatically build .esboe files, build your project. For instructions on configuring your project, see **Configuring OpenEdge Architect for ESB annotations** on page 154.

- Right mouse-click in either the **Resources** view or the ABL Editor, and select **OpenEdge > Generate Sonic ESB Invocation Files**.

Outside of OpenEdge Architect, use ESBOEGEN to generate your .esboe files from annotated ABL code. See the **ESBOEGEN** on page 153 for details.

**Note:** You can generate .esboe files from unannotated ABL code with ProxyGen. See **Non-declarative invocation files** on page 140 for more details.

Once you have generated your .esboe files, you are ready to create an ESB process in Sonic.

### Non-declarative invocation files

Non-declarative invocation files are generated from unannotated ABL code using Proxy Generator (ProxyGen). To generate .esboe files with Proxy Generator:
1. Ensure that your ABL procedures are compiled.

2. Start Proxy Generator and create a new AppObject. Fill in appropriate components and select procedures.

3. Select File > Generate to bring up the General tab of the Generate Proxies dialog box, as shown:

4. Select Sonic Native Invocation and specify the destination directory for your .esboe files in the Output Dir field.

5. Select the Sonic Native Invocation tab, as shown:

6. Select the output destination of your .esboe files, and click OK to generate them. If you check Save to Output Directory, your .esboe files are saved to the directory you specified on the General tab. If you check Deploy to Directory Service, you must specify the absolute path of your Sonic Directory Service in the Resource Dir field. If you check Create Deployment Archive, you must specify an archive (.xar) name in the Archive Name field.

**Import Native Invocation files into Sonic Workbench**

Once you have developed your ABL procedures and created your native invocation files, proceed to Sonic Workbench to import your invocation (.esboe) files and develop your ESB process.
To import your native invocation files into an ESB process:

1. Start Sonic Workbench and open an existing project or create a new one.
2. Choose **File > Import** to import your invocation files into the Sonic Directory Service.
3. On the process design canvas, select **OpenEdge Native Services** from the **Palette** and drop it into place on the canvas. Give the Service a meaningful name.
4. Fill in the Service information. You can drag an invocation file from the **Navigator** tab onto the Service, or right-click on the Service and choose **Open**.

If you dragged an invocation file onto the service, **Invocation File** displays the filename. If you did not drag a file, browse to the appropriate invocation file.

5. Select the **Request Mapping** tab:
The initial input message parts are mapped to the input parameters of the ABL procedure. Each parameter can be mapped to specific message parts or message headers, set to constants or ignored. The data can also be transformed using XPath or wrapped in an XML element.

6. Select the **Response Mapping** tab:
The output parameters can be placed in message parts or message headers, inserted as XML elements or attributes, or discarded.

7. Complete development of your process, adding additional invocation files, or other services.

**ABL parameter naming**

The Request and Response Mapping tabs show the list of input and output parameters, along with their XML schema types, required to invoke the operation. Each parameter can be mapped to the specific message parts or to messages headers, set to constants, or ignored. For request parameters, the data can also be transformed using XPath or wrapped in an XML element. For response parameters, the data can be placed in message parts or headers, inserted as XML elements or attributes, or discarded.

To help distinguish the ABL data type of parameters, a special naming convention is used. Each parameter is identified by its ABL parameter name, followed by its ABL type, and the mode of the parameter (IN or OUT). The construction is:

```xml
<ABLparameter>_<ABLdatatype>_<ABLparametertype>
```

The following table shows the resulting ESBOE parameters names, if you expose an ABL procedure that has an integer named `CustNum` as the input parameter, and returns a string named `CustName`.

**Table 36: Conversion of ABL parameter to ESBOE parameter**

<table>
<thead>
<tr>
<th>ABL parameter</th>
<th>ABL data type</th>
<th>ABL parameter type (IN/OUT)</th>
<th>ESBOE parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>CustNum</td>
<td>INTEGER</td>
<td>IN</td>
<td>CustNum_INTEGER_IN</td>
</tr>
<tr>
<td>CustName</td>
<td>CHARACTER</td>
<td>OUT</td>
<td>CustName_CHARACTER_OUT</td>
</tr>
</tbody>
</table>

**Internal procedure invocation**

Generating a native invocation file for an internal procedure or user-defined function, creates a total of three invocation files: one to run the external procedure persistently, one to invoke the internal procedure, and one to delete the persistent procedure and unbind the OpenEdge Application Server from the process. All three invocation files must be imported into your ESB process in order.
The following figure shows the three invocation files in a sample process.

**Figure 4: Internal procedure invocation**

![Image](image_url)

Calling the external procedure creates a proc-id parameter that is mapped throughout the process.

**ESB Process and session-managed OpenEdge Application Servers**

Your `.esboe` files do not contain information about your OpenEdge Application Server operating mode. OpenEdge installs process templates into your Sonic Directory Service for you to include in your process if you need a session-managed connection. The process template, `OESessionManaged`, adds two invocation files to your process: **Connect to Session-Managed AppServer** and **Disconnect from Session-Managed AppServer**. By default, the connection is made to the default `dev.OpenEdge` service in the `dev_OpenEdgeTest` container. You can change the service properties of your `dev.OpenEdge` service in your Sonic Management Console, or you can create a new service for your session-managed OpenEdge Application Server. If you create a new service, you must re-configure all your invocation files in your process to connect to your new service in your Sonic Workbench process editor.

Best practices for process design dictates that a session-managed session should not span more than a single process. Alter your process design to use sub-processes if you require multiple session-managed OpenEdge Application Servers.
The following figure shows the OESessionManaged template in Sonic Workbench.

**Figure 5: Session-managed invocation template**

![Diagram of OESessionManaged template in Sonic Workbench]

**ABL file drag and drop**

Dragging ABL procedure files directly into a Sonic ESB itinerary eliminates the need to create and import a `.esboe` file. Successfully dragging and dropping files requires that your environment be configured as follows:

- **Software**
  - OpenEdge Release 10.1C01 (Service pack 1) or higher for Windows. The OpenEdge Architect and the OpenEdge Adapter for Sonic ESB products must be installed.
  - Sonic Workbench 7.6 or higher.

- **Configuration** — Your OpenEdge Architect and Sonic Workbench Eclipse environments must be integrated. See PDSN for a matrix of supported versions and detailed instructions on integrating the two environments.

**Drag and drop execution**

Once your environment is configured, dragging and dropping your ABL procedure files into an ESB process itinerary is as simple as selecting the file and placing in the process editor.

To drag and drop an ABL file into an ESB process:
1. Open an ESB process editor window. Select an ABL file from an OpenEdge project in the Navigator window, and drag it into the ESB process, as shown:

2. After dropping your ABL procedure file, a wizard begins to guide you through the process of selecting the procedure you are including in your ESB process, as shown:
3. Complete the wizard as follows:

a) Select either **Main method** or **Internal procedure/function**.

b) If you select **Internal procedure/function**, you can check **Filter by ESB Annotations** to sort annotated procedures and functions higher in the selection box. Check **Generate handle and release steps** to run the external procedure persistently before calling the internal procedure, and to delete the persistent procedure after the internal procedure is run.

If you are adding steps that call more than one internal procedure from the same ABL file, you only need one set of handle and release steps.

a) Create a **Step Name**. By default, **Step Name** defaults to the ABL file name (without the `.p` suffix) for the main procedure, and the ABL file name and the internal procedure name separated by an underscore (_) for an internal procedure.

b) If your ABL source contains Native Invocation annotations, you can click **Finish**. If your ABL source is not annotated, or you need to modify the annotations, click **Next**. The following dialog opens:
4. If you have selected an internal procedure, both the **Detail ESB annotation** and **Main ESB annotation** sections appear. For an external procedure, only the Main section appears. For details on completing the annotation information, see [Annotating ABL in OpenEdge Architect](#) on page 138 for a complete definition of the Native Invocation annotation components. Click **Finish**.

5. Once you have completed the annotation wizard, your ABL procedure is added to the ESB process. An internal procedure added to the ESB process is shown:
If you modify the signature of your ABL procedure, such as the number of parameters, or parameter data types, you must delete and re-add the ESB process step for the procedure.

**ESB process details**

When viewing or developing your ESB process, double-clicking on your ABL procedure step provides you with additional detail information. The following figure shows the Service tab detail.
ESB process Service tab detail
When you have dragged an ABL procedure file into your ESB process, the Invocation type is **Inline OpenEdge Invocation**. Clicking **OpenEdge Invocation** displays the invocation details, as shown in the following figure.

**Figure 6: OpenEdge Invocation detail**

<table>
<thead>
<tr>
<th>File name:</th>
<th>src\custom.p</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method name:</td>
<td>N/A</td>
</tr>
<tr>
<td>Method Arguments:</td>
<td>Parameter:PRO_INTEGER INPUT in_param</td>
</tr>
<tr>
<td></td>
<td>Parameter:PRO_CHARACTER OUTPUT out_param</td>
</tr>
<tr>
<td>Method Returns:</td>
<td>N/A</td>
</tr>
</tbody>
</table>

## ABL file requirements

ABL files that can be dragged onto the ESB process editor must adhere to the following rules:

- The ABL file must be part of an OpenEdge project. Standalone files cannot be dragged into an ESB process.

- The ABL file must successfully compile. If the file is not compiled when dragged into the ESB process editor, it is compiled before the drop completes. If the compile fails, the file cannot be dropped.

- The file must not be in a modified state. If the file is not saved before it is dragged, it is saved before it is dropped.

In addition, only external procedures, internal procedures, and user-defined functions can be included in an ESB process itinerary, regardless of whether the ABL file is dragged, or a `.esboe` file is imported.
Testing an ESB Process containing OpenEdge Native Invocation Services

Once you have created your ESB process with native invocations, you can test it with the following general steps that describe the test process:

1. Save your ESB process and upload it to your Sonic Domain.
2. In Progress Explorer, configure and start an AppServer broker, typically `esbbroker1`, to run your ABL procedures.
3. In Sonic Management Console, confirm that the AppServer parameters for the default `OpenEdge Native Services` dev.OpenEdge service is configured to connect to the AppServer you started in the preceding step, by performing the following steps:
   a) Click the Configure tab and expand Services.
   b) Select OpenEdge Native Services.
   c) Select the dev.OpenEdge Service Name.
   d) Verify the AppServer parameters in the Init Parameters section. If necessary, modify to match the AppServer broker.
4. In Sonic Workbench, click the Container view. The default container for native invocations is `dev_OpenEdgeTest`. Select the dev_OpenEdgeTest container. Right-click and select Start to start the container.
5. Create and run Scenarios that verify the execution of your process, including a successful round-trip to and from your OpenEdge Application Server. For more information on creating and running Scenarios, and testing your ESB process, see your Sonic Workbench documentation.

ESBOEGEN

ESBOEGEN is a command line utility for processing annotated ABL source files to generate native invocation (.esboe) files. The command line syntax is as follows:

```
esboegen [-source directory]
[-esboe directory] [-archive filename]
[-rcode directory] [-recurse] [files]
```

Parameters:

- **-source directory**
  
  Specifies the directory containing annotated ABL files.

- **-esboe directory**
  
  Specifies the destination directory for generated .esboe files.
-archive filename

Specifies the name of an archive (.xar) file to hold all the generated .esboe files. filename must be a fully qualified filename.

rcode directory

Specifies the directory containing the compiled r-code that corresponds to either the directory specified with -source or the listed files. This parameter is required to process ABL code if it contains temp-table definitions containing the keyword LIKE.

-recurse

Directs ESBOEGEN to recursively search all subdirectories for ABL code. When specified, a corresponding directory tree is built in the output directory specified with -esboe or in the archive specified with -archive.

files

A comma separated list of ABL files. If a fully qualified file name is not specified, then ESBOEGEN looks for the file in the current working directory.

ESBOEGEN is only supported in Windows and can only execute when OpenEdge Architect is installed. ESBOEGEN provides the ability to write batch procedures to generate invocation files.

ESBOEGEN selects the output destination of the generated .esboe files according to the following order of precedence:

1. Destination specified by -esboe or -archive
2. Directory specified by -rcode
3. Directory specified -source
4. Directory specified for file name listed with files

You cannot specify both -esboe and -archive.

Configuring OpenEdge Architect for ESB annotations

By default, OpenEdge Architect is configured with a standard ESB annotation definition, and to build ESB invocation files. To verify or alter these settings, see the following sections:

- Default ESB Annotations on page 154
- Generating ESB invocation files on page 155

Default ESB Annotations

You can verify or modify default OpenEdge Architect annotations settings.

To verify OpenEdge Architect annotation settings:

1. Start OpenEdge Architect by selecting Programs > Start > OpenEdge > OpenEdge Architect .
2. Select a workspace if prompted.
3. Open an existing project or start a new project.
4. Select Window > Preferences.
5. In the tree view, expand OpenEdge Architect, then Editor, and select Annotations.
6. Select an ESB Annotation from Annotation names, as shown:

![Preferences dialog with Annotations selected](image)

7. Change the default annotation string by selecting Edit, if necessary. Click OK to exit the Preferences dialog box.

### Generating ESB invocation files

OpenEdge Architect can be configured to automatically generate Native Invocation files when you build your project. You can verify or change the default behavior of OpenEdge Architect for invocation file generation.

To configure Architect to generate Native Invocation files:

1. Start OpenEdge Architect by selecting Start > Programs > OpenEdge > OpenEdge Architect.
2. Select a workspace if prompted.
3. Open an existing project or start a new project.
4. Check your project properties.
   a) Select Project > Properties.
   b) In the tree view, expand OpenEdge and select Build.
c) Select **Generate Sonic invocation files on build** to have OpenEdge Architect automatically generate your `.esboe` files when your project is built.

d) Specify the output directory for your `.esboe` files in the **Invocation file destination** field as shown:

![Properties for ESB process example](image)

**Note:** If you do not specify a destination directory, OpenEdge Architect writes your `.esboe` files to your r-code destination directory.

5. Verify your property settings and click **OK**.

The next time your project containing annotated source is built, OpenEdge Architect will automatically generate your `.esboe` files.

### Web Services Invocation methodology

Functionally, there is no practical difference between the Web Services Invocation methodology in the OpenEdge Adapter for Sonic ESB and the OpenEdge Web Services Adapter (WSA). Both adapters perform the same conversions between the ABL (Advanced Business Language) and SOAP protocols. With a Web Services Invocation, an OpenEdge service of Sonic ESB is essentially identical to an OpenEdge Web service.

Any service developed for use with the OpenEdge Adapter for Sonic ESB can function as a standard Web service, either in the context of Sonic ESB or the WSA. Conversely, any existing Web service deployed to a WSA can be exported (by means of either the Progress Explorer or the WSAMAN Utility) and installed into Sonic ESB with its runtime properties intact.
Because of this close similarity between the two adapters, the same rules and guidelines apply in both cases to programming services and clients. *OpenEdge Development: Web Services* extensively discusses these topics in the context of the WSA. Please refer to that manual for programming information.

**Differences between the Web Services Invocation methodology in the OpenEdge Adapter for Sonic ESB and the WSA**

Although the OpenEdge Adapter for Sonic ESB and the WSA are very similar, there are some differences between them. The most significant difference is simply that Sonic ESB hosts a service in conjunction with the OpenEdge Adapter for Sonic ESB, whereas an OpenEdge Web service is hosted on a Web server or Java Servlet Engine (JSE) in conjunction with the WSA.

It is important to note that once installed, the OpenEdge Adapter for Sonic ESB does not appear as an entity of that name in any OpenEdge or Sonic UI component. In the Sonic Management Console, the primary tool for managing services and related functions, the presence of two OpenEdge services, **OpenEdge Native Services** and **OpenEdge Web Services**, nodes in the Services folder indicates that the OpenEdge Adapter for Sonic ESB is installed.

The following figure shows the OpenEdge services in the Sonic Management Console.

*Figure 7: Sonic Management Console showing OpenEdge Adapter for Sonic ESB installed*
There are a few additional differences, described in the sections that follow, that you should note if you are familiar with the WSA. The procedures to which these sections refer, as well as other important information about management of the adapters, is detailed in *OpenEdge Application Server: Administration*.

**Service deployment and management**

For a Sonic ESB service, once the AppServer application has been developed and its service definition has been generated, all deployment, configuration, and management activities associated with the service take place in the Sonic environment.

For a WSA-based service, you use the Progress Explorer or the WSAMAN Utility to perform deployment and administrative functions.

**WSM and WSD file usage**

When you install a Web Services Invocation OpenEdge service in Sonic ESB, you specify one of the following two files as the service definition:

- **The WSM file** — Created by Proxy Generator from the compiled r-code for an AppServer application
- **The WSD file** — Exported from an existing Web service

Sonic ESB stores a copy of the specified service definition file as a resource. The file retains its original name. The OpenEdge Adapter for Sonic ESB relies on the stored WSM or WSD file for the information needed to code and decode client SOAP messages and to access the appropriate service to execute requests.

When you deploy a Web service to a WSA instance, in contrast, the WSA stores the service definition as a copy of the original WSM or WSD file with the name `FriendlyName.wsad`. The WSA relies on this WSAD file for the necessary information about SOAP processing and service identification.

WSAD files do not exist in Sonic ESB.

**Storage of property information**

Default run-time properties of OpenEdge services for Sonic ESB are hard-coded in the OpenEdge Adapter for Sonic ESB. To change these properties for a given service, you edit its associated WSM or WSD service definition file, which is stored as a resource in Sonic ESB as mentioned in the preceding section. You use the custom OpenEdge Resource Editor for this purpose.

In the WSA environment, default runtime properties are stored in the `default.props` file associated with each WSA instance. You can modify these properties for a given service by means of the Progress Explorer or the WSAMAN Utility. Each service’s runtime properties are stored in a file named `FriendlyName.props`.

**WSDL file generation**

When your Sonic ESB service is ready for deployment, you use the custom OpenEdge Resource Editor available in the Sonic ESB Explorer to generate the associated WSDL file that defines the client interface to the service. This procedure differs from the corresponding procedure for WSA-based services in that the WSDL file for those services is generated through the Progress Explorer.
Service definition considerations

The sections that follow contain guidelines for using the Proxy Generator tool to create a service definition. For detailed information on Proxy Generator, refer to OpenEdge Development: Open Client Introduction and Programming and to the Proxy Generator online Help.

Session models

The OpenEdge Adapter for Sonic ESB, like the WSA, supports two session models:

- **Session-managed** — While a transaction is in progress between the service and a client, the connected AppServer is dedicated exclusively to that client. The AppServer maintains the context of the transaction until it has responded to all requests from the client and the transaction is completed. The session-managed model is available for the stateless, state-aware, and state-reset AppServer operating modes, and it can be used with all Open Client types (.NET and Java clients as well as Web service clients) and OpenEdge clients.

- **Session-free** — The AppServer does not maintain a transaction context. The AppServer returns a complete result in response to each single request, and it does so without regard to any previous responses to the client from itself or any other Application Server process. Any available Application Server processes launched by any qualified AppServer broker can process requests from the client in parallel, since each transaction is independent of all others. The session-free model is available on all clients, and it requires that the AppServer be running in the state-free mode.

In programming for services managed by the OpenEdge Adapter for Sonic ESB and their clients, it is generally preferable to write applications to take advantage of the session-free model. By doing so, you avoid the need to write code to manage connections. Similarly, it is desirable to avoid including SubAppObjects or ProcObjects (persistent procedures) when generating the client proxy for the AppServer application, because such objects are processed in a session-managed mode even if the session-free model is specified for the top-level AppObject. In general, the session-free model without the use of sub-objects requires less complex programming. It also allows for better performance and scalability, since multiple AppServer instances and processes can be made available to handle requests in parallel.
The following figure shows the Proxy Generator window and the Progress Explorer window where you set the applicable options for the session model and the AppServer operating mode.

Figure 8: Recommended options for OpenEdge Adapter for Sonic ESB services

Sonic message handling run-time parameters

All messages in Sonic ESB are multi-part messages. Currently, the OpenEdge Adapter for Sonic ESB looks for its input message in the first part, and creates a new message with its output. However, there are cases where you want to have flexibility in how messages are handled. You can have this flexibility by using service run-time parameters. The following run-time parameters let you specify an input message part and an output message part:

- **InputMessagePart** — An integer parameter that specifies which message part to use for the input to the OpenEdge service. The message part must be the proper XML that the service is expecting. There is no difference in functionality when using a value of 0 or 1. They will reflect the current functionality of using the first part for the input message. The default is 0.

- **OutputMessagePart** — An integer parameter that specifies which message part to use for the output of the service call. A value of 0 (the default) retains current functionality by returning a new message. A value of 1 retains the original message and replaces the first part with the output. This is useful if you want to retain the message properties of the original message.

You set the values of run-time parameters when you create a process. The values are evaluated when the service is invoked. If you specify an output message part, the original message is kept intact, with the output of the service call placed in the specified part. By default, there is no change in functionality if you do not specify these run-time parameters.

For more information on Sonic run-time parameters, see the Sonic documentation.
This appendix provides reference information on the procedures and functions (and a global variable) for the OpenEdge Adapter for SonicMQ ABL–JMS API. All procedures and functions are supported for all adapter personalities unless otherwise indicated in the reference entry.

For more information on using this information in context, see Programming for the OpenEdge Adapter for SonicMQ with the ABL - JMS API on page 109.

For details, see the following topics:

- Session objects
- jmssession.p
- ptpsession.p
- pubsubsession.p
- Methods in the Session objects
- Methods in the Message Consumer objects
- Methods in the Message objects
- acknowledgeAndForward procedure
- addBytesPart procedure
- addMessagePart procedure
- addTextPart procedure
- appendText procedure
• beginSession procedure
• browseQueue procedure
• cancelDurableSubscription procedure
• clearBody procedure
• clearProperties procedure
• commitReceive procedure
• commitSend procedure
• createBytesMessage procedure
• createChangeStateConsumer procedure
• createDataSetMessage procedure
• createHeaderMessage procedure
• createMapMessage procedure
• createMessageConsumer procedure
• createMultipartMessage procedure
• createRejectedMessageConsumer procedure
• createStreamMessage procedure
• createTemporaryQueue procedure
• createTemporaryTopic procedure
• createTempTableMessage procedure
• createTextMessage procedure
• createXMLMessage procedure
• deleteConsumer procedure
• deleteMessage procedure
• deleteSaxWriter procedure
• deleteSession procedure
• deleteTemporaryQueue procedure
• deleteTemporaryTopic procedure
• eofStream function
• getAdapterService function
• getApplicationContext function
• getBrokerURL function
• getBytesCount function
• getBytesPartByID function
- `getBytesPartByIndex` function
- `getBytesToRaw` function
- `getChar` function
- `getCharCount` function
- `getCharProperty` function
- `getClientID` function
- `getClientPersistence` function
- `getClientTransactionBufferSize` function
- `getConnectID` function
- `getConnectionID` function
- `getConnectionMetaData` function
- `getConnectionURLs` function
- `getContentType` function
- `getDataSet` function
- `getDate` function
- `getDateProperty` function
- `getDateTime` function
- `getDateTimeProperty` function
- `getDateTime-TZ` function
- `getDateTimeTzProperty` function
- `getDecimal` function
- `getDecimalProperty` function
- `getDefaultPersistency` function
- `getDefaultPriority` function
- `getDefaultTimeToLive` function
- `getDestinationName` function
- `getFaultTolerant` function
- `getFaultTolerantReconnectTimeout` function
- `getFlowToDisk` function
- `getInt` function
- `getIntProperty` function
- `getInt64` function
- `getInt64Property` function
• getInitialConnectionTimeout function
• getItemType function
• getJMSCorrelationID function
• getJMSCorrelationIDAsBytes function
• getJMSDeliveryMode function
• getJMSDestination function
• getJMSExpiration function
• getJMSMessageID function
• getJMSPriority function
• getJMSRedelivered function
• getJMSReplyTo function
• getJMSServerName function
• getJMSTimestamp function
• getJMSType function
• getLoadBalancing function
• getLocalStoreDirectory function
• getLocalStoreSize function
• getLocalStoreWaitTime function
• getLogical function
• getLogicalProperty function
• getLongString function
• getLongStringCP function
• getLongText function
• getLongTextCP function
• getMapNames function
• getMemptr function
• getMessagePartByID function
• getMessagePartByIndex function
• getMessageType function
• getNoAcknowledge function
• getPartCount function
• getPassword
• getProcHandle function
• getProcName function
• getPropertyNames function
• getPropertyType function
• getReconnectInterval function
• getReconnectTimeout function
• getReplyAutoDelete function
• getReplyPersistency function
• getReplyPriority function
• getReplyTimeToLive function
• getReplyToDestinationType function
• getReuseMessage function
• getSaxWriter function
• getSession function
• getSelectorAtBroker function
• getSequential function
• getShutdownWaitFor function
• getSingleMessageAcknowledgement function
• getTempTable function
• getText function
• getTextPartByID function
• getTextPartByIndex function
• getTextSegment function
• getTransactedReceive function
• getTransactedSend function
• getUser function
• getX-Document function
• hasReplyTo function
• inErrorHandling function
• inMessageHandling function
• inQueueBrowsing function
• inReplyHandling function
• isFaultTolerant function
• isMessagePart function
• JMS-MAXIMUM-MESSAGES global variable
• messageHandler procedure
• moveToNext procedure
• publish procedure
• readBytesToRaw procedure
• readChar function
• readDate function
• readDateTime function
• readDateTime-TZ function
• readDecimal function
• readInt function
• readInt64 function
• readLogical function
• readLongString function
• readLongStringCP function
• receiveFromQueue procedure
• recover procedure
• requestReply procedure
• reset procedure
• rollbackReceive procedure
• rollbackSend procedure
• sendToQueue procedure
• setAdapterService procedure
• setApplicationContext procedure
• setBoolean procedure
• setBooleanProperty procedure
• setBrokerURL procedure
• setByte procedure
• setByteProperty procedure
• setBytesFromRaw procedure
• setChar procedure
• setClientID procedure
• setClientPersistence procedure
• setClientTransactionBufferSize procedure
• setConnectID procedure
• setConnectionFile procedure
• setConnectionURLs procedure
• dataSet procedure
• setDate procedure
• setDateProperty procedure
• setDateTime procedure
• setDateTimeProperty procedure
• setDateTime-TZ procedure
• setDateTimeTzProperty procedure
• setDefaultPersistency procedure
• setDefaultPriority procedure
• setDefaultTimeToLive procedure
• setDouble procedure
• setDoubleProperty procedure
• setErrorHandler procedure
• setFaultTolerant procedure
• setFaultTolerantReconnectTimeout procedure
• setFloat procedure
• setFloatProperty procedure
• setFlowToDisk procedure
• setInt procedure
• setIntProperty procedure
• setInt64 procedure
• setInt64Property procedure
• setInitialConnectionTimeout procedure
• setJMSCorrelationID procedure
• setJMSCorrelationIDAsBytes procedure
• setJMSReplyTo procedure
• setJMSServerName procedure
• setJMSType procedure
• setLoadBalancing procedure
• setLocalStoreDirectory procedure
• setLocalStoreSize procedure
• setLocalStoreWaitTime procedure
• setLong procedure
• setLongProperty procedure
• setLongString procedure
• setLongText procedure
• setMemptr procedure
• setNoAcknowledge procedure
• setNoErrorDisplay procedure
• setPassword procedure
• setPingInterval procedure
• setPrefetchCount procedure
• setPrefetchThreshold procedure
• setReconnectInterval procedure
• setReconnectTimeout procedure
• setReplyAutoDelete procedure
• setReplyPersistency procedure
• setReplyPriority procedure
• setReplyTimeToLive procedure
• setReplyToDestinationType procedure
• setReuseMessage procedure
• setSaxReader procedure
• setSelectorAtBroker procedure
• setSequential procedure
• setShort procedure
• setShortProperty procedure
• setShutdownWaitFor procedure
• setSingleMessageAcknowledgement procedure
• setString procedure
• setStringProperty procedure
• setTempTable procedure
• setText procedure
• setTransactedReceive procedure
• setTransactedSend procedure
• setUser procedure
• setX-Document procedure
• startReceiveMessages procedure
• stopReceiveMessages procedure
• subscribe procedure
• waitForMessages procedure
• writeBoolean procedure
• writeByte procedure
• writeBytesFromRaw procedure
• writeChar procedure
• writeDate procedure
• writeDateTime procedure
• writeDateTime-TZ procedure
• writeDouble procedure
• writeFloat procedure
• writeInt procedure
• writeInt64 procedure
• writeLong procedure
• writeLongString procedure
• writeShort procedure
• writeString procedure

Session objects

The Session objects are:
• jmssession.p on page 170
• ptpsession.p on page 170
• pubsubsession.p on page 171

For information on the methods available for the Session objects, see Methods in the Session objects on page 171.
**jmssession.p**

Starts a JMS session using the unified domain to access both queues and topics in one JMS session object.

**Syntax**

```plaintext
RUN jms/jmssession.p PERSISTENT SET jmssession (adapterConnection).
```

**Parameter**

`adapterConnection`

Desired adapter connection options. For more information, see the Connection options.

**Applies to**

Session objects

**See also**

jmssession.p on page 170, ptpsession.p on page 170, pubsubsession.p on page 171

For more information, see Methods in the Session objects on page 171.

**ptpsession.p**

Starts a JMS session using queues.

**Syntax**

```plaintext
RUN jms/ptpsession.p PERSISTENT SET ptpsession (adapterConnection).
```

**Parameter**

`adapterConnection`

Desired adapter connection options. For more information, see the Connection options.

**Applies to**

Session objects

**See also**

jmssession.p on page 170, ptpsession.p on page 170, pubsubsession.p on page 171

For more information, see Methods in the Session objects on page 171.
pubsubsession.p

Starts a JMS session using topics.

Syntax

```plaintext
RUN jms/pubsubsession.p PERSISTENT SET pubsubsession (adapterConnection).
```

Parameter

adapterConnection

Desired adapter connection options. For more information, see the Connection options.

Applies to

Session objects

See also

jmssession.p on page 170, ptpsession.p on page 170, pubsubsession.p on page 171

For more information, see Methods in the Session objects on page 171.

Methods in the Session objects

The following table lists the methods in the Session objects.

Table 37: Methods in Session objects

<table>
<thead>
<tr>
<th>Method</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>beginSession procedure</code></td>
<td>180</td>
</tr>
<tr>
<td><code>cancelDurableSubscription procedure</code></td>
<td>182</td>
</tr>
<tr>
<td><code>commitSend procedure</code></td>
<td>184</td>
</tr>
<tr>
<td><code>createChangeStateConsumer procedure</code></td>
<td>185</td>
</tr>
<tr>
<td><code>createChangeStateConsumer procedure</code></td>
<td>185</td>
</tr>
<tr>
<td><code>createHeaderMessage procedure</code></td>
<td>186</td>
</tr>
<tr>
<td><code>createMessageConsumer procedure</code></td>
<td>187</td>
</tr>
<tr>
<td><code>createRejectedMessageConsumer procedure</code></td>
<td>188</td>
</tr>
<tr>
<td><code>createTemporaryQueue procedure</code></td>
<td>190</td>
</tr>
<tr>
<td><code>createTempTableMessage procedure</code></td>
<td>191</td>
</tr>
<tr>
<td><code>createTemporaryTopic procedure</code></td>
<td>190</td>
</tr>
<tr>
<td><code>createTextMessage procedure</code></td>
<td>191</td>
</tr>
</tbody>
</table>
Methods in the Message Consumer objects

The following table lists the methods in the Message Consumer objects.

Table 38: Methods in Message Consumer Objects

<table>
<thead>
<tr>
<th>Method rentals in Message Consumer objects</th>
<th>Page references</th>
</tr>
</thead>
<tbody>
<tr>
<td>setConnectionURLs procedure</td>
<td>276</td>
</tr>
<tr>
<td>setDefaultPriority procedure</td>
<td>282</td>
</tr>
<tr>
<td>setFaultTolerant procedure</td>
<td>285</td>
</tr>
<tr>
<td>setInitialConnectionTimeout procedure</td>
<td>290</td>
</tr>
<tr>
<td>setFlowToDisk procedure</td>
<td>288</td>
</tr>
<tr>
<td>setLoadBalancing procedure</td>
<td>294</td>
</tr>
<tr>
<td>setLocalStoreSize procedure</td>
<td>295</td>
</tr>
<tr>
<td>setNoErrorDisplay procedure</td>
<td>300</td>
</tr>
<tr>
<td>setReconnectTimeout procedure</td>
<td>304</td>
</tr>
<tr>
<td>setPingInterval procedure</td>
<td>301</td>
</tr>
<tr>
<td>setPrefetchThreshold procedure</td>
<td>303</td>
</tr>
<tr>
<td>setSequential procedure</td>
<td>310</td>
</tr>
<tr>
<td>setSingleMessageAcknowledgement procedure</td>
<td>312</td>
</tr>
<tr>
<td>setTransactedSend procedure</td>
<td>316</td>
</tr>
<tr>
<td>startReceiveMessages procedure</td>
<td>317</td>
</tr>
<tr>
<td>subscribe procedure</td>
<td>319</td>
</tr>
<tr>
<td>acknowledgeAndForward procedure</td>
<td>177</td>
</tr>
<tr>
<td>deleteConsumer procedure</td>
<td>192</td>
</tr>
<tr>
<td>getApplicationContext function</td>
<td>197</td>
</tr>
<tr>
<td>getDestinationName function</td>
<td>213</td>
</tr>
<tr>
<td>getNoAcknowledge function</td>
<td>233</td>
</tr>
<tr>
<td>getProcHandle function</td>
<td>235</td>
</tr>
<tr>
<td>getProcName function</td>
<td>235</td>
</tr>
<tr>
<td>getReplyAutoDelete function</td>
<td>238</td>
</tr>
<tr>
<td>getReplyPersistency function</td>
<td>238</td>
</tr>
<tr>
<td>getReplyPriority function</td>
<td>239</td>
</tr>
<tr>
<td>setFaultTolerantReconnectTimeout procedure</td>
<td>286</td>
</tr>
<tr>
<td>setErrorHandler procedure</td>
<td>284</td>
</tr>
<tr>
<td>setFaultTolerant procedure</td>
<td>285</td>
</tr>
<tr>
<td>setDefaultPersistency procedure</td>
<td>281</td>
</tr>
<tr>
<td>setDefaultTimeToLive procedure</td>
<td>283</td>
</tr>
<tr>
<td>setFaultTolerantReconnectTimeout procedure</td>
<td>286</td>
</tr>
<tr>
<td>setInitialConnectionTimeout procedure</td>
<td>290</td>
</tr>
<tr>
<td>setFlowToDisk procedure</td>
<td>288</td>
</tr>
<tr>
<td>setLoadBalancing procedure</td>
<td>294</td>
</tr>
<tr>
<td>setLocalStoreDirectory procedure</td>
<td>294</td>
</tr>
<tr>
<td>setLocalStoreWaitTime procedure</td>
<td>296</td>
</tr>
<tr>
<td>setNoErrorDisplay procedure</td>
<td>300</td>
</tr>
<tr>
<td>setPassword procedure</td>
<td>301</td>
</tr>
<tr>
<td>setReconnectInterval procedure</td>
<td>304</td>
</tr>
<tr>
<td>setPreferences procedure</td>
<td>302</td>
</tr>
<tr>
<td>setSelectorAtBroker procedure</td>
<td>309</td>
</tr>
<tr>
<td>setShUTDOWNWaitFor procedure</td>
<td>312</td>
</tr>
<tr>
<td>setTransactedReceive procedure</td>
<td>315</td>
</tr>
<tr>
<td>stopReceiveMessages procedure</td>
<td>318</td>
</tr>
<tr>
<td>waitForMessages procedure</td>
<td>320</td>
</tr>
<tr>
<td>subscribe procedure</td>
<td>319</td>
</tr>
</tbody>
</table>
For more information, see the Message Consumer objects.

### Methods in the Message objects

The following table lists the methods in the Message objects.

**Note:** Not all methods are in all message types.

**Table 39: Methods in Message Objects**

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>addBytesPart procedure</td>
<td>178</td>
</tr>
<tr>
<td>addMessagePart procedure</td>
<td>178</td>
</tr>
<tr>
<td>addTextPart procedure</td>
<td>179</td>
</tr>
<tr>
<td>appendText procedure</td>
<td>180</td>
</tr>
<tr>
<td>clearBody procedure</td>
<td>182</td>
</tr>
<tr>
<td>clearProperties procedure</td>
<td>183</td>
</tr>
<tr>
<td>deleteMessage procedure</td>
<td>193</td>
</tr>
<tr>
<td>deleteSaxWriter procedure</td>
<td>193</td>
</tr>
<tr>
<td>eofStream function</td>
<td>195</td>
</tr>
<tr>
<td>getBytesCount function</td>
<td>198</td>
</tr>
<tr>
<td>getBytesPartByID function</td>
<td>198</td>
</tr>
<tr>
<td>getBytesPartByIndex function</td>
<td>199</td>
</tr>
<tr>
<td>getBytesToRaw function</td>
<td>200</td>
</tr>
<tr>
<td>getChar function</td>
<td>200</td>
</tr>
<tr>
<td>getCharCount function</td>
<td>201</td>
</tr>
<tr>
<td>getCharProperty function</td>
<td>201</td>
</tr>
<tr>
<td>getConnectionURLs function</td>
<td>205</td>
</tr>
<tr>
<td>getContentType on page</td>
<td>205</td>
</tr>
<tr>
<td>getDataSet function</td>
<td>206</td>
</tr>
<tr>
<td>getDate function</td>
<td>207</td>
</tr>
<tr>
<td>getDateProperty function</td>
<td>207</td>
</tr>
<tr>
<td>getDateTime function</td>
<td>208</td>
</tr>
<tr>
<td>getDateTimeProperty function</td>
<td>208</td>
</tr>
<tr>
<td>getDateTime-TZ function</td>
<td>209</td>
</tr>
<tr>
<td>getDecimal function</td>
<td>210</td>
</tr>
<tr>
<td>getDecimal function</td>
<td>210</td>
</tr>
<tr>
<td>Method Name</td>
<td>Page</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td><code>getInt</code> function</td>
<td>215</td>
</tr>
<tr>
<td><code>getInt64</code> function</td>
<td>216</td>
</tr>
<tr>
<td><code>getIntProperty function</code></td>
<td>215</td>
</tr>
<tr>
<td><code>getInt64Property function</code></td>
<td>216</td>
</tr>
<tr>
<td><code>getItemType</code> function</td>
<td>217</td>
</tr>
<tr>
<td><code>getJMSCorrelationID function</code></td>
<td>218</td>
</tr>
<tr>
<td><code>getJMSCorrelationIDAsBytes function</code></td>
<td>219</td>
</tr>
<tr>
<td><code>getJMSDeliveryMode function</code></td>
<td>219</td>
</tr>
<tr>
<td><code>getJMSExpiration function</code></td>
<td>220</td>
</tr>
<tr>
<td><code>getJMSPriority function</code></td>
<td>221</td>
</tr>
<tr>
<td><code>getJMSReplyTo function</code></td>
<td>222</td>
</tr>
<tr>
<td><code>getJMSReplyToDestinationType function</code></td>
<td>240</td>
</tr>
<tr>
<td><code>getJMSReplyTo function</code></td>
<td>222</td>
</tr>
<tr>
<td><code>getJMSReplyToDestinationType function</code></td>
<td>240</td>
</tr>
<tr>
<td><code>getJMSRedelivered function</code></td>
<td>222</td>
</tr>
<tr>
<td><code>getJMSRedelivered function</code></td>
<td>222</td>
</tr>
<tr>
<td><code>getJMSDestination function</code></td>
<td>220</td>
</tr>
<tr>
<td><code>getJMSExpiration function</code></td>
<td>220</td>
</tr>
<tr>
<td><code>getJMSRedelivered function</code></td>
<td>222</td>
</tr>
<tr>
<td><code>getJMSReplyTo function</code></td>
<td>222</td>
</tr>
<tr>
<td><code>getJMSReplyToDestinationType function</code></td>
<td>240</td>
</tr>
<tr>
<td><code>getJMSReplyTo function</code></td>
<td>222</td>
</tr>
<tr>
<td><code>getLogical function</code></td>
<td>227</td>
</tr>
<tr>
<td><code>getLogicalProperty function</code></td>
<td>227</td>
</tr>
<tr>
<td><code>getLongString function</code></td>
<td>228</td>
</tr>
<tr>
<td><code>getLongStringCP function</code></td>
<td>228</td>
</tr>
<tr>
<td><code>getLongString function</code></td>
<td>228</td>
</tr>
<tr>
<td><code>getLongText function</code></td>
<td>229</td>
</tr>
<tr>
<td><code>getLoadBalancing function</code></td>
<td>224</td>
</tr>
<tr>
<td><code>getLoadBalancing function</code></td>
<td>224</td>
</tr>
<tr>
<td><code>getMapNames function</code></td>
<td>230</td>
</tr>
<tr>
<td><code>getMemptr function</code></td>
<td>230</td>
</tr>
<tr>
<td><code>getMessagePartByID function</code></td>
<td>231</td>
</tr>
<tr>
<td><code>getMessagePartByIndex function</code></td>
<td>232</td>
</tr>
<tr>
<td><code>getMessageType function</code></td>
<td>233</td>
</tr>
<tr>
<td><code>getMessagePartByID function</code></td>
<td>231</td>
</tr>
<tr>
<td><code>getMessagePartByIndex function</code></td>
<td>232</td>
</tr>
<tr>
<td><code>getMessageType function</code></td>
<td>233</td>
</tr>
<tr>
<td><code>getPropertyNames function</code></td>
<td>235</td>
</tr>
<tr>
<td><code>getPropertyType function</code></td>
<td>236</td>
</tr>
<tr>
<td><code>getReplyToDestinationType function</code></td>
<td>240</td>
</tr>
<tr>
<td><code>getReplyToDestinationType function</code></td>
<td>240</td>
</tr>
<tr>
<td><code>getReplyToDestinationType function</code></td>
<td>240</td>
</tr>
<tr>
<td><code>getText function</code></td>
<td>245</td>
</tr>
<tr>
<td><code>getTextPartByID function</code></td>
<td>245</td>
</tr>
<tr>
<td><code>getTextPartByIndex function</code></td>
<td>246</td>
</tr>
<tr>
<td><code>getTextSegment function</code></td>
<td>246</td>
</tr>
<tr>
<td><code>getTextPartByIndex function</code></td>
<td>246</td>
</tr>
<tr>
<td><code>getTextSegment function</code></td>
<td>246</td>
</tr>
<tr>
<td><code>getX-Document function</code></td>
<td>248</td>
</tr>
<tr>
<td><code>hasReplyTo function</code></td>
<td>249</td>
</tr>
<tr>
<td><code>isMessagePart function</code></td>
<td>251</td>
</tr>
<tr>
<td><code>isMessagePart function</code></td>
<td>251</td>
</tr>
<tr>
<td><code>messageHandler procedure</code></td>
<td>252</td>
</tr>
<tr>
<td><code>moveToNext procedure</code></td>
<td>253</td>
</tr>
<tr>
<td><code>moveToNext procedure</code></td>
<td>253</td>
</tr>
<tr>
<td><code>readBytesToRaw procedure</code></td>
<td>256</td>
</tr>
<tr>
<td><code>readBytesToRaw procedure</code></td>
<td>256</td>
</tr>
<tr>
<td><code>readChar function</code></td>
<td>256</td>
</tr>
<tr>
<td><code>readChar function</code></td>
<td>256</td>
</tr>
<tr>
<td><code>readDateTime function</code></td>
<td>258</td>
</tr>
<tr>
<td><code>readDateTime function</code></td>
<td>258</td>
</tr>
<tr>
<td><code>readDateTime-TZ function</code></td>
<td>258</td>
</tr>
<tr>
<td><code>readDateTime-TZ function</code></td>
<td>258</td>
</tr>
<tr>
<td><code>readDecimal function</code></td>
<td>259</td>
</tr>
<tr>
<td><code>readDecimal function</code></td>
<td>259</td>
</tr>
<tr>
<td><code>readInt function</code></td>
<td>259</td>
</tr>
<tr>
<td><code>readInt function</code></td>
<td>259</td>
</tr>
<tr>
<td><code>readInt64 function</code></td>
<td>260</td>
</tr>
<tr>
<td><code>readInt64 function</code></td>
<td>260</td>
</tr>
<tr>
<td><code>readLogical function</code></td>
<td>261</td>
</tr>
<tr>
<td><code>readLogical function</code></td>
<td>261</td>
</tr>
</tbody>
</table>

**OpenEdge Development: Messaging and ESB**
<table>
<thead>
<tr>
<th>Function/Procedure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>readLongString function</td>
<td>261</td>
</tr>
<tr>
<td>readLongStringCP function</td>
<td>262</td>
</tr>
<tr>
<td>reset procedure</td>
<td>265</td>
</tr>
<tr>
<td>setBoolean procedure</td>
<td>268</td>
</tr>
<tr>
<td>setBooleanProperty procedure</td>
<td>269</td>
</tr>
<tr>
<td>setByte procedure</td>
<td>270</td>
</tr>
<tr>
<td>setByteProperty procedure</td>
<td>271</td>
</tr>
<tr>
<td>setChar procedure</td>
<td>272</td>
</tr>
<tr>
<td>setDataSet procedure</td>
<td>277</td>
</tr>
<tr>
<td>setDate procedure</td>
<td>277</td>
</tr>
<tr>
<td>setDateProperty procedure</td>
<td>278</td>
</tr>
<tr>
<td>setDateTime procedure</td>
<td>279</td>
</tr>
<tr>
<td>setDateTimeProperty procedure</td>
<td>280</td>
</tr>
<tr>
<td>setDateTime-TZ procedure</td>
<td>280</td>
</tr>
<tr>
<td>setDateTimeTzProperty procedure</td>
<td>281</td>
</tr>
<tr>
<td>setDouble procedure</td>
<td>283</td>
</tr>
<tr>
<td>setDoubleProperty procedure</td>
<td>284</td>
</tr>
<tr>
<td>setFloat procedure</td>
<td>287</td>
</tr>
<tr>
<td>setFloatProperty procedure</td>
<td>287</td>
</tr>
<tr>
<td>setInt procedure</td>
<td>288</td>
</tr>
<tr>
<td>setInt64 procedure</td>
<td>289</td>
</tr>
<tr>
<td>setInt64Property procedure</td>
<td>290</td>
</tr>
<tr>
<td>setIntProperty procedure</td>
<td>289</td>
</tr>
<tr>
<td>setJMSCorrelationID procedure</td>
<td>291</td>
</tr>
<tr>
<td>setJMSCorrelationIDAsBytes procedure</td>
<td>292</td>
</tr>
<tr>
<td>setJMSReplyTo procedure</td>
<td>292</td>
</tr>
<tr>
<td>setJMSReplyToDestinationType procedure</td>
<td>292</td>
</tr>
<tr>
<td>set JMSType procedure</td>
<td>293</td>
</tr>
<tr>
<td>setLong procedure</td>
<td>296</td>
</tr>
<tr>
<td>setLongProperty procedure</td>
<td>297</td>
</tr>
<tr>
<td>setLongString procedure</td>
<td>298</td>
</tr>
<tr>
<td>setLongText procedure</td>
<td>298</td>
</tr>
<tr>
<td>setMemptr procedure</td>
<td>299</td>
</tr>
<tr>
<td>setErrorDisplay procedure</td>
<td>300</td>
</tr>
<tr>
<td>setReplyToDestinationType procedure</td>
<td>308</td>
</tr>
<tr>
<td>SAXReader procedure</td>
<td>309</td>
</tr>
<tr>
<td>setSequential procedure</td>
<td>310</td>
</tr>
<tr>
<td>setShort procedure</td>
<td>310</td>
</tr>
<tr>
<td>setShortProperty procedure</td>
<td>311</td>
</tr>
<tr>
<td>setSingleMessageAcknowledgement procedure</td>
<td>312</td>
</tr>
<tr>
<td>setString procedure</td>
<td>313</td>
</tr>
<tr>
<td>setStringProperty procedure</td>
<td>313</td>
</tr>
<tr>
<td>setTempTable procedure</td>
<td>314</td>
</tr>
<tr>
<td>setText procedure</td>
<td>315</td>
</tr>
<tr>
<td>setX-Document procedure</td>
<td>317</td>
</tr>
<tr>
<td>writeBoolean procedure</td>
<td>321</td>
</tr>
<tr>
<td>writeByte procedure</td>
<td>321</td>
</tr>
<tr>
<td>writeBytesFromRaw procedure</td>
<td>322</td>
</tr>
<tr>
<td>writeChar procedure</td>
<td>323</td>
</tr>
<tr>
<td>writeDate procedure</td>
<td>323</td>
</tr>
<tr>
<td>writeDateTime procedure</td>
<td>324</td>
</tr>
<tr>
<td>writeDateTime-TZ procedure</td>
<td>325</td>
</tr>
</tbody>
</table>
For more information, see the Message objects.

**acknowledgeAndForward procedure**

Forwards and acknowledges a message in a single operation.

**Note:** The JMS providers other than SonicMQ do not support this method.

**Syntax**

```
PROCEDURE acknowledgeAndForward.
DEFINE INPUT PARAMETER destinationName AS CHARACTER.
DEFINE INPUT PARAMETER messageH AS HANDLE.
DEFINE INPUT PARAMETER priority AS INTEGER.
DEFINE INPUT PARAMETER timeToLive AS DECIMAL.
DEFINE INPUT PARAMETER persistency AS CHARACTER.
```

**Applies to**

Message Consumer objects

**Notes**

- This procedure applies inside a message event handler.
- The session must be set to **SINGLE_MESSAGE_ACKNOWLEDGE**.
- The procedure expects a destination queue name, the original message handle, and optional message-delivery properties. If the message-delivery properties are set to the **Unknown value (?)**, the procedure uses the original values from the message.
- If the procedure is not successful—for example, if the destination does not exist—the message is not acknowledged and eventually returns to the queue.

**See also**

- acknowledgeAndForward procedure on page 177, setSingleMessageAcknowledgement procedure on page 312, 
- getSingleMessageAcknowledgement function on page 243, setNoAcknowledge procedure on page 300 , 
- getNoAcknowledge function on page 233

For more information, see the Message acknowledgement, forwarding, and recovery and the Single-message acknowledgement.
addBytesPart procedure

Adds any arbitrary part to a MultipartMessage.

Syntax

PROCEDURE addBytesPart.
DEFINE INPUT PARAMETER memptr AS MEMPTR.
DEFINE INPUT PARAMETER contentTypeString AS CHARACTER.
DEFINE INPUT PARAMETER contentIDString AS CHARACTER.

Applies to
Message objects

Notes
• The part can be text or binary. The Sonic message is created as usual.
• A content type and a content ID must be specified.
• To conserve resources, after calling this procedure, the application must delete the memory pointer (represented by memptr).

See also
createMultipartMessage procedure on page 188, addMessagePart procedure on page 178, addTextPart procedure on page 179, isMessagePart function on page 251, getPartCount function on page 234, getContentType on page 205, getMessagePartByID function on page 231, getMessagePartByIndex function on page 232, writeBytesFromRaw procedure on page 322, readBytesToRaw procedure on page 256, setMemptr procedure on page 299, getMemptr function on page 230

For more information, see the MultipartMessage and the MultiPartMessage example.

addMessagePart procedure

Adds a SonicMQ message to a MultipartMessage.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

PROCEDURE addMessagePart.
DEFINE INPUT PARAMETER messagePartH AS HANDLE.
DEFINE INPUT PARAMETER contentIDString AS CHARACTER.
Applies to
Message objects

Notes
• The Sonic message is created as usual.
• Its content type is defined by Sonic. The content-ID string (represented by `contentIDString`) sets the content ID of the part and is used to identify it.
• To conserve resources, after calling this procedure, the application must delete the message-part handle (represented by `messagePartH`).

See also
`createMultipartMessage procedure` on page 188, `addBytesPart procedure` on page 178, `addMessagePart procedure` on page 178, `addTextPart procedure` on page 179, `isMessagePart function` on page 251, `getContentType` on page 205, `getPartCount function` on page 234, `getMessagePartByID function` on page 231, `getMessagePartByIndex function` on page 232, `writeBytesFromRaw procedure` on page 322, `readBytesToRaw procedure` on page 256, `setMemptr procedure` on page 299, `getMemptr function` on page 230

For more information, see the MultipartMessage and the MultiPartMessage example.

---

addTextPart procedure

Adds a text part to a MultipartMessage.

Syntax

```plaintext
PROCEDURE addTextPart.
DEFINE INPUT PARAMETER charString AS CHARACTER.
DEFINE INPUT PARAMETER contentTypeString AS CHARACTER.
DEFINE INPUT PARAMETER contentIDString AS CHARACTER.
```

Applies to
Message objects

Notes
The method resembles `addBytesPart procedure` on page 178 except that it takes a CHARACTER string instead of a MEMPTR.

See also
`createMultipartMessage procedure` on page 188, `addBytesPart procedure` on page 178, `addMessagePart procedure` on page 178, `isMessagePart function` on page 251, `getContentType` on page 205, `getPartCount function` on page 234, `getMessagePartByID function` on page 231, `getMessagePartByIndex function` on page 232, `writeBytesFromRaw procedure` on page 322, `readBytesToRaw procedure` on page 256, `setMemptr procedure` on page 299, `getMemptr function` on page 230

For more information, see the MultipartMessage and the MultiPartMessage example.
**appendText procedure**

Appends text to the message in write-only mode using several calls to overcome the OpenEdge 32K limit on the number of characters.

**Syntax**

```abl
PROCEDURE appendText.
DEFINE INPUT PARAMETER textValue AS CHARACTER.
```

**Applies to**
Message objects

**Notes**

- This procedure is designed for use in conjunction with `setText procedure` on page 315.
- This procedure is not needed when using `setLongText procedure` on page 298, which writes `LONGCHAR` data.

**See also**

- `createTextMessage procedure` on page 191
- `setText procedure` on page 315
- `setLongText procedure` on page 298
- `appendText procedure` on page 180
- `endOfStream function` on page 195
- `getCharCount function` on page 201
- `getText function` on page 245
- `getTextSegment function` on page 246
- `getLongText function` on page 229
- `getLongTextCP function` on page 229

For more information, see the `TextMessage`.

**beginSession procedure**

Connects to the OpenEdge Adapter for SonicMQ and starts a JMS connection and session.

**Syntax**

```abl
PROCEDURE beginSession.
```

**Applies to**
Session objects

**Notes**

- If `beginSession procedure` on page 180 returns an error, the Session object is automatically deleted.
- This procedure executes remotely (sends a message to the OpenEdge Adapter for SonicMQ).
See also
beginSession procedure on page 180, getSession function on page 241, deleteSession procedure on page 194
For more information, see the Connecting to the OpenEdge Adapter for SonicMQ and the Establishing session control.

browseQueue procedure

Allows applications to view messages in a queue without consuming them.

Syntax

```plaintext
PROCEDURE browseQueue.
DEFINE INPUT PARAMETER queueName AS CHARACTER.
DEFINE INPUT PARAMETER messageSelector AS CHARACTER.
DEFINE INPUT PARAMETER messageConsumer AS HANDLE.
```

Parameters

*queueName*

The queue from which the messages are received.

*messageSelector*

A message selector.

*messageConsumer*

A Message Consumer object, which handles the messages asynchronously.

Applies to

Session objects

Notes

- This procedure receives (for browsing) all messages currently in the queue in the messageConsumer object.
- Browsed messages are not removed from the queue or acknowledged and are not subject to the transactional context of the session. (For more information, see the Java Message Service specification and the SonicMQ Programming Guide on queue browsing.)
- The session need not run startReceiveMessages procedure on page 317 to browse messages on a queue.
- This procedure executes remotely (sends a message to the OpenEdge Adapter for SonicMQ).

See also

browseQueue procedure on page 181, receiveFromQueue procedure on page 262, sendToQueue procedure on page 266
cancelDurableSubscription procedure

Cancels a durable subscription.

Syntax

```
PROCEDURE cancelDurableSubscription.
DEFINE INPUT PARAMETER subscriptionName AS CHARACTER.
```

Parameter

```
subscriptionName
```

Name of durable subscription.

Applies to

Session objects

Notes

• It is an error to call this procedure if there is an active Message Consumer for the subscription. Call deleteConsumer procedure on page 192 first to delete the Message Consumer.
• This procedure executes remotely (sends a message to the OpenEdge Adapter for SonicMQ).

See also

cancelDurableSubscription procedure on page 182 , publish procedure on page 254, subscribe procedure on page 319

For more information, see the Publishing messages to a topic, the Subscribing to a topic, the Durable subscriptions, and the Methods unique to Pub/Sub messaging.

For an example, see the Pub/Sub messaging example.

clearBody procedure

Clears the body of a message, keeping header and property values unchanged, and changes the mode of a message from read-only to write-only mode.

Syntax

```
PROCEDURE clearBody.
```
Applies to
Message objects

Notes
The clearBody procedure on page 182 transfers a StreamMessage, TextMessage, XMLMessage, BytesMessage, TempTableMessage, or DataSetMessage to write-only mode.

See also
clearProperties procedure on page 183
For more information, see the Accessing message properties.

clearProperties procedure
Clears the properties of the message, keeping the header and body values unchanged.

Syntax

```
PROCEDURE clearProperties.
```

Applies to
Message objects

See also

clearBody procedure on page 182
For more information, see the Accessing message properties.

commitReceive procedure
Acknowledges all messages received up to that point in the current transaction.

Syntax

```
PROCEDURE commitReceive.
```

Applies to
Session objects

Notes
• It is an error to call this method in a Session object that is not transacted for receiving.
• This procedure executes remotely (sends a message to the OpenEdge Adapter for SonicMQ).

See also
commitSend procedure on page 184, commitReceive procedure on page 183. rollbackSend procedure on page 266, rollbackReceive procedure on page 265, recover procedure on page 263
For more information, see the Transaction and recovery procedures.

commitSend procedure

Sends all messages published (or sent to a queue) up to that point in the current transaction.

Syntax

PROCEDURE commitSend.

Applies to
Session objects

Notes

• It is an error to call this method in a Session object that is not transacted for sending.
• This procedure executes remotely (sends a message to the OpenEdge Adapter for SonicMQ).

See also
commitSend procedure on page 184, commitReceive procedure on page 183. rollbackSend procedure on page 266, rollbackReceive procedure on page 265, recover procedure on page 263
For more information, see the Transaction and recovery procedures.

createBytesMessage procedure

Creates a new BytesMessage.

Syntax

PROCEDURE createBytesMessage.
DEFINE OUTPUT PARAMETER messageHandle AS HANDLE.

Applies to
Session objects
createChangeStateConsumer procedure

Creates a "Change State" handle to contain the state of the SonicMQ Broker connection changes.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

```plaintext
PROCEDURE createChangeStateConsumer
DEFINE INPUT PARAMETER procHandle AS HANDLE.
DEFINE INPUT PARAMETER procName AS CHARACTER.
DEFINE OUTPUT PARAMETER consumerHandle AS HANDLE.
```

Applies to

Session objects (ClientConnect and ServerConnect only).

Notes

- Only applicable for Fault Tolerant connections.
- Call createChangeStateConsumer procedure on page 185 after beginSession procedure on page 180.
- In the "Change State" handler, the character header property "state" will contain one of the following values: "active", "reconnecting", "failed", or "closed".
- Setting up a change-state handler is optional. If not done, the default is to not notify the application of state changes.

See also

setFaultTolerant procedure on page 285, getFaultTolerant function on page 213, isFaultTolerant function on page 251, createChangeStateConsumer procedure on page 185, setFaultTolerantReconnectTimeout procedure on page 286, getFaultTolerantReconnectTimeout function on page 214, setInitialConnectionTimeout procedure on page 290, getInitialConnectionTimeout function on page 217, setClientTransactionBufferSize procedure on page 274, getClientTransactionBufferSize function on page 203

For more information, see the Fault tolerance and the Fault tolerance.
createDataSetMessage procedure

Creates a new DataSetMessage.

Syntax

PROCEDURE createDataSetMessage.
DEFINE OUTPUT PARAMETER messageHandle AS HANDLE.

Applies to

Session objects

See also

createDataSetMessage procedure on page 186, dataSet procedure on page 277, getDataSet function on page 206

For an example, see the DataSetMessage.

createHeaderMessage procedure

Creates a new HeaderMessage.

Syntax

PROCEDURE createHeaderMessage.
DEFINE OUTPUT PARAMETER messageHandle AS HANDLE.

Applies to

Session objects

See also

For more information, see the HeaderMessage.

createMapMessage procedure

Creates a new MapMessage.
createMapMessage procedure

Syntax

```
PROCEDURE createMapMessage.
DEFINE OUTPUT PARAMETER messageHandle AS HANDLE.
```

Applies to

Session objects

See also

createMapMessage procedure on page 186, setBoolean procedure on page 268, setByte procedure on page 270, setBytesFromRaw procedure on page 271, setChar procedure on page 272, setDate procedure on page 277, setDateTime procedure on page 279, setDateTime-TZ procedure on page 280, setDouble procedure on page 283, setFloat procedure on page 287, setInt procedure on page 288, setLong procedure on page 296, setLongString procedure on page 298, setShort procedure on page 310, setString procedure on page 313, getMapNames function on page 230, getItemType function on page 217, getBytesToRaw function on page 200, getChar function on page 200, getDecimal function on page 210, getInt function on page 215, getLogical function on page 227, getLongString function on page 228, getLongStringCP function on page 228

For more information, see the MapMessage.

createMessageConsumer procedure

Creates a new Message Consumer object.

Syntax

```
PROCEDURE createMessageConsumer.
DEFINE INPUT PARAMETER procHandle AS HANDLE.
DEFINE INPUT PARAMETER procName AS CHARACTER.
DEFINE OUTPUT PARAMETER consumerHandle AS HANDLE.
```

Parameters

`procHandle`

The handle to a procedure `procName`.

`procName`

The name of an internal procedure for handling messages.

`consumerHandle`

The new Message Consumer object.

Applies to

Session objects
Notes
The application must pass the name of an internal procedure for handling messages and the handle to the internal procedure to createMessageConsumer procedure on page 187.

See also
createMessageConsumer procedure on page 187, deleteConsumer procedure on page 192, messageHandler procedure on page 252, waitForMessages procedure on page 320
For more information see the Message Consumer objects, the Consuming messages, the Terminating the Message Consumer object, and the Creating a message handler process.

createMultipartMessage procedure
Creates a MultipartMessage.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

| PROCEDURE createMultipartMessage.  
| DEFINE OUTPUT PARAMETER messageH AS HANDLE. |

Applies to
Session objects

See also
addBytesPart procedure on page 178, addMessagePart procedure on page 178, addTextPart procedure on page 179, isMessagePart function on page 251, getContentType on page 205, getPartCount function on page 234, getMessagePartByIndex function on page 231, getMessagePartByID function on page 232, writeBytesFromRaw procedure on page 322, readBytesToRaw procedure on page 256, setMemptr procedure on page 299, getMemptr function on page 230
For more information, see the MultipartMessage and the MultiPartMessage example.

createRejectedMessageConsumer procedure
Creates a Message Consumer to handle all rejected messages.

Note: The JMS providers other than SonicMQ do not support this method.
createRejectedMessageConsumer procedure

Syntax

```
PROCEDURE createRejectedMessageConsumer.
DEFINE INPUT PARAMETER procHandle AS HANDLE.
DEFINE INPUT PARAMETER procName AS CHARACTER.
DEFINE OUTPUT PARAMETER consumerHandle AS HANDLE.
```

Applies to

Session objects

Notes

- Only applicable for Client Persistence.
- Call createRejectedMessageConsumer procedure on page 188 after beginSession procedure on page 180.

See also


For more information, see the Client persistence and the Client persistence.

createStreamMessage procedure

Creates a new StreamMessage.

Syntax

```
PROCEDURE createStreamMessage.
DEFINE OUTPUT PARAMETER messageHandle AS HANDLE.
```

Applies to

Session objects
createTemporaryQueue procedure

Creates a temporary queue on the SonicMQ Broker and returns the queue name.

Syntax

```
PROCEDURE createTemporaryQueue.
DEFINE OUTPUT PARAMETER qname AS CHARACTER.
```

Applies to

Session objects

Notes

• An error is returned if beginSession procedure on page 180 has not yet been called.
• An error is returned if the temporary queue cannot be created.

See also

createTemporaryQueue procedure on page 190, deleteTemporaryQueue procedure on page 194,
createTemporaryTopic procedure on page 190, deleteTemporaryTopic procedure on page 195

For more information, see the Temporary queues and the Temporary topic.

createTemporaryTopic procedure

Creates a temporary topic on the SonicMQ Broker and returns the topic name.

Syntax

```
PROCEDURE createTemporaryTopic.
DEFINE OUTPUT PARAMETER tname AS CHARACTER.
```

For more information, see the Temporary queues and the Temporary topic.
createTempTableMessage procedure

Creates a new TempTableMessage.

Syntax

```
PROCEDURE createTempTableMessage.
DEFINE OUTPUT PARAMETER messageHandle AS HANDLE.
```

Applies to
Session objects

See also
createTemporaryQueue procedure on page 190, deleteTemporaryQueue procedure on page 194, createTemporaryTopic procedure on page 190, deleteTemporaryTopic procedure on page 195

For more information, see the Temporary queues and the Temporary topic.

createTextMessage procedure

Creates a new TextMessage object.

Syntax

```
PROCEDURE createTextMessage.
DEFINE OUTPUT PARAMETER messageHandle AS HANDLE.
```

Applies to
Session objects

See also
createTempTableMessage procedure on page 191, setTempTable procedure on page 314, getTempTable function on page 244

For more information, see the TempTableMessage.
See also
createTextMessage procedure on page 191, setText procedure on page 315, setLongText procedure on page 298, appendText procedure on page 180, endOfStream function on page 195, getCharCount function on page 201, getText function on page 245, getTextSegment function on page 246, getLongText function on page 229, getLongTextCP function on page 229
For more information, see the TextMessage and the StreamMessage.

createXMLMessage procedure

Creates a new XMLMessage.

Syntax

```ABL
PROCEDURE createXMLMessage.
DEFINE OUTPUT PARAMETER messageHandle AS HANDLE.
```

Applies to
Session objects

See also
For more information, see the XMLMessage.

deleteConsumer procedure

Deletes the Message Consumer object.

Syntax

```ABL
PROCEDURE deleteConsumer.
```

Applies to
Message Consumer objects

Notes

• In the Pub/Sub domain, deleteConsumer procedure on page 192 cancels the subscription. In the PTP domain, deleteConsumer procedure on page 192 removes the association with a queue.
• This procedure executes remotely (sends a message to the OpenEdge Adapter for SonicMQ).
See also
createMessageConsumer procedure on page 187, deleteConsumer procedure on page 192, messageHandler procedure on page 252, waitForMessages procedure on page 320

For more information see the Message Consumer objects, the Consuming messages, the Terminating the Message Consumer object, and the Creating a message handler process.

deleteMessage procedure

Deletes a message (including a TempTableMessage and DataSetMessage) and deallocates all of its memory and resources.

Syntax

PROCEDURE deleteMessage.

Applies to
Message objects

See also
For more information, see the Deleting messages.

deleteSaxWriter procedure

Terminates the SAX-WRITER and deletes the SAX-WRITER handle created by getSaxWriter function on page 241.

Syntax

PROCEDURE deleteSaxWriter.
INPUT PARAMETER hdl AS HANDLE.

Applies to
Message objects

Notes

• This procedure calls setLongText procedure on page 298 to place the XML created by the SAX-WRITER calls into the XMLMessage.

• The caller then sends the XMLMessage using the appropriate OpenEdge Adapter for SonicMQ calls.

• An error is returned if the specified handle is not a SAX-WRITER handle, by setLongText procedure on page 298, and by DELETE OBJECT.
deleteSession procedure

Closes a session and its underlying connection and deletes the session procedure.

Syntax

```plaintext
PROCEDURE deleteSession.
```

Applies to

Session objects

Notes

This procedure executes remotely (sends a message to the OpenEdge Adapter for SonicMQ).

See also

beginSession procedure on page 180, getSession function on page 241, deleteSession procedure on page 194

For more information, see the Establishing session control.

deleteTemporaryQueue procedure

Deletes the temporary queue on the SonicMQ Broker.

Syntax

```plaintext
PROCEDURE deleteTemporaryQueue.
DEFINE INPUT PARAMETER qname AS CHAR.
```

Applies to

Session objects

Notes

- Any Message Consumers for the temporary queue must be deleted with deleteConsumer procedure on page 192 before calling this procedure.
- An error is returned if beginSession procedure on page 180 has not yet been called.
An error is returned if the temporary queue cannot be deleted.

See also
createTemporaryQueue procedure on page 190, deleteTemporaryQueue procedure on page 194, createTemporaryTopic procedure on page 190, deleteTemporaryTopic procedure on page 195

For more information, see the Temporary queues and the Temporary topic.

deleteTemporaryTopic procedure
Deletes the temporary topic on the SonicMQ Broker.

Syntax

```
PROCEDURE deleteTemporaryTopic.
DEFINE INPUT PARAMETER tname AS CHARACTER.
```

Applies to
Session objects

Notes
• Any Message Consumers for the temporary topic must be deleted with deleteConsumer procedure on page 192 before calling this procedure.
• An error is returned if beginSession procedure on page 180 has not yet been called.
• An error is returned if the temporary topic cannot be deleted.

See also
createTemporaryQueue procedure on page 190, deleteTemporaryQueue procedure on page 194, createTemporaryTopic procedure on page 190, deleteTemporaryTopic procedure on page 195

For more information, see the Temporary queues and the Temporary topic.

endOfStream function
Returns TRUE if the application retrieved the last text segment, the last item of a stream, or the last byte segment.

Syntax

```
FUNCTION endOfStream RETURNS LOGICAL.
```

Applies to
Message objects
Notes

• An application should not call endOfStream function on page 195 if it used getMemptr function on page 230 for extracting the data.

• endOfStream function on page 195 is also used with the TextMessage, StreamMessage, and BytesMessage message types.

See also

createTextMessage procedure on page 191, setText procedure on page 315, setLongText procedure on page 298, appendText procedure on page 180, endOfStream function on page 195, getCharCount function on page 201, getText function on page 245, getTextSegment function on page 246, getLongText function on page 229, getLongTextCP function on page 229

For more information, see the TextMessage.

createBytesMessage procedure on page 184, setMemptr procedure on page 299, writeBoolean procedure on page 321, writeByte procedure on page 321, writeBytesFromRaw procedure on page 322, writeChar procedure on page 323, writeDate procedure on page 323, writeDateTime procedure on page 324, writeDateTime-TZ procedure on page 325, writeDouble procedure on page 326, writeFloat procedure on page 326, writeInt procedure on page 327, writeLong procedure on page 328, writeLongString procedure on page 329, writeShort procedure on page 330, writeString procedure on page 330, getBytesCount function on page 198, endOfStream function on page 195, moveToNext procedure on page 253, readBytesToRaw procedure on page 256, readChar function on page 257, readDateTime function on page 258, readDateTime-TZ function on page 258, readDecimal function on page 259, readInt function on page 259, readLogical function on page 261, readLongString function on page 261, readLongStringCP function on page 262, getMemptr function on page 230

For more information, see the BytesMessage.

For an example, see the Publishing, subscribing, and receiving an XML document in a BytesMessage.

createStreamMessage procedure on page 189, writeBoolean procedure on page 321, writeByte procedure on page 321, writeBytesFromRaw procedure on page 322, writeChar procedure on page 323, writeDate procedure on page 323, writeDateTime procedure on page 324, writeDateTime-TZ procedure on page 325, writeDouble procedure on page 326, writeFloat procedure on page 326, writeInt procedure on page 327, writeLong procedure on page 328, writeLongString procedure on page 329, writeShort procedure on page 330, writeString procedure on page 330, endOfStream function on page 195, moveToNext procedure on page 253, readBytesToRaw procedure on page 256, readChar function on page 257, readDateTime function on page 258, readDateTime-TZ function on page 258, readDecimal function on page 259, readInt function on page 259, readLogical function on page 261, readLongString function on page 261, readLongStringCP function on page 262

For more information, see the StreamMessage.

getAdapterService function

Returns the service name under which the OpenEdge Adapter for SonicMQ BrokerConnect is registered with the NameServer.

Syntax

FUNCTION getAdapterService RETURNS CHARACTER.
**getApplicationContext function**

Returns application context information.

**Syntax**

```FUNCTION getApplicationContext RETURNS HANDLE.```

**Applies to**

Message Consumer objects

**See also**

`setApplicationContext procedure` on page 268, `getApplicationContext function` on page 197

For more information, see the Accessing message handler information and the Creating a message handler process.

**getBrokerURL function**

Returns the value set by the preceding `setBrokerURL procedure` on page 270.

**Syntax**

```FUNCTION getBrokerURL RETURNS CHARACTER.```

**Applies to**

Session objects

**Notes**

If `setBrokerURL procedure` on page 270 was not called, Null is returned.
getBytesCount function

Returns the number of bytes in a BytesMessage.

Syntax

```
FUNCTION getBytesCount RETURNS INTEGER.
```

Applies to
Message objects

See also
createBytesMessage procedure on page 184, setMemptr procedure on page 299, writeBoolean procedure on page 321, writeBytes procedure on page 321, writeBytesFromRaw procedure on page 322, writeChar procedure on page 323, writeDate procedure on page 323, writeDateTime procedure on page 324, writeDateTime-TZ procedure on page 325, writeDouble procedure on page 326, writeFloat procedure on page 326, writeInt procedure on page 327, writeLong procedure on page 328, writeLongString procedure on page 329, writeShort procedure on page 330, writeString procedure on page 330, getBytesCount function on page 198, endOfStream function on page 195, moveToNext procedure on page 253, readBytesToRaw procedure on page 256, readChar function on page 256, readDate function on page 257, readDateTime function on page 258, readDateTime-TZ function on page 258, readDecimal function on page 259, readInt function on page 259, readLogical function on page 261, readLongString function on page 261, readLongStringCP function on page 262, getMemptr function on page 230

For more information, see the Publishing, subscribing, and receiving an XML document in a BytesMessage.

getBytesPartByID function

Retrieves a bytes part and returns the content type as a CHARACTER string.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

```
FUNCTION getBytesPartByID RETURNS CHARACTER 
    (INPUT contentID AS INTEGER, OUTPUT memPtr AS MEMPTR).
```
Applies to
Message objects

Notes
• Before calling this function, call SET-SIZE to free any memory allocated by the MEMPTR.
• The bytes part does not undergo any code-page conversion. If it consists of text data, it is encoded in UTF-8. To encode it differently, either convert the code page manually or use one of the getTextPartBy... functions.

See also
createMultipartMessage procedure on page 188, addBytesPart procedure on page 178, addMessagePart procedure on page 178, addTextPart procedure on page 179, isMessagePart function on page 251, getPartCount function on page 234, getMessagePartByID function on page 231, getMessagePartByIndex function on page 232, writeBytesFromRaw procedure on page 322, readBytesToRaw procedure on page 256, setMemptr procedure on page 299, getMemptr function on page 230

For more information, see the MultipartMessage and the MultiPartMessage example.

getBytesPartByIndex function

Retrieves a bytes part and returns the content type as a CHARACTER string.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

FUNCTION getMessagePartByIndex RETURNS CHARACTER
  (INPUT iIndex AS INTEGER, OUTPUT memPtr AS MEMPTR).

Applies to
Message objects

Notes
• Before calling this function, call SET-SIZE to free any memory allocated by the MEMPTR.
• The bytes part does not undergo any code-page conversion. If it consists of text data, it is encoded in UTF-8. To encode it differently, either convert the code page manually or use one of the getTextPartBy... functions.
See also
createBytesMessage procedure on page 184, setMemptr procedure on page 299, writeBoolean procedure on page 321, writeByte procedure on page 321, writeBytesFromRaw procedure on page 322, writeChar procedure on page 323, writeDate procedure on page 323, writeDateTime procedure on page 324, writeDateTime-TZ procedure on page 325, writeDouble procedure on page 326, writeFloat procedure on page 326, writeInt procedure on page 327, writeLong procedure on page 328, writeLongString procedure on page 329, writeShort procedure on page 330, writeString procedure on page 330, getBytesCount function on page 198, endOfStream function on page 195, moveToNext procedure on page 253, readBytesToRaw procedure on page 256, readChar function on page 256, readDate function on page 257, readDateTime function on page 258, readDateTime-TZ function on page 258, readDecimal function on page 259, readInt function on page 259, readLogical function on page 261, readLongString function on page 261, readLongStringCP function on page 262, getMemptr function on page 230

For more information, see the BytesMessage.
For an example, see the Publishing, subscribing, and receiving an XML document in a BytesMessage.
For more information, see the MultipartMessage and the MultiPartMessage example.

getBytesToRaw function

Gets a bytes item from a MapMessage.

Syntax

```
FUNCTION getBytesToRaw RETURNS RAW (itemName AS CHARACTER).
```

Applies to
Message objects

See also
createMapMessage procedure on page 186, setBoolean procedure on page 268, setByte procedure on page 270, setBytesFromRaw procedure on page 271, setChar procedure on page 272, setDate procedure on page 277, setDateTime procedure on page 279, setDateTime-TZ procedure on page 280, setDouble procedure on page 283, setFloat procedure on page 287, setInt procedure on page 288, setLong procedure on page 296, setLongString procedure on page 298, setShort procedure on page 310, setString procedure on page 313, getMapNames function on page 230, getItemType function on page 217, getBytesToRaw function on page 200, getChar function on page 200, getDate function on page 207, getDateTime function on page 208, getDateTime-TZ function on page 209, getDecimal function on page 210, getInt function on page 215, getLogical function on page 227, getLongString function on page 228, getLongStringCP function on page 228

For more information, see the MapMessage.

getChar function

Gets an item of any data type except byte from a MapMessage.
getChar function

Syntax

FUNCTION getChar RETURNS CHARACTER (itemName AS CHARACTER).

Applies to
Message objects

See also
createMapMessage procedure on page 186, setBoolean procedure on page 268, setByte procedure on page 270, setBytesFromRaw procedure on page 271, setChar procedure on page 272, setDate procedure on page 277, setDateTime procedure on page 279, setDateTime-TZ procedure on page 280, setDouble procedure on page 283, setFloat procedure on page 287, setText procedure on page 315, setLong procedure on page 296, setLongString procedure on page 298, setShort procedure on page 310, setString procedure on page 313, getMapNames function on page 230, getItemType function on page 217, getBytesToRaw function on page 200, getChar function on page 200, getDate function on page 207, setLogical function on page 227, getDecimal function on page 210, getLogical function on page 215, getCharCount function on page 201, getDateTime function on page 228, getLongString function on page 228, getLongStringCP function on page 228

For more information, see the MapMessage.

getCharCount function

Returns the total number of characters in a message.

Syntax

FUNCTION getCharCount RETURNS INTEGER.

Applies to
Message objects

See also
createTextMessage procedure on page 191, setText procedure on page 315, setLongText procedure on page 298, appendText procedure on page 180, endOfStream function on page 195, getCharCount function on page 201, getText function on page 245, getLongText function on page 229, getLongTextCP function on page 229

For more information, see the TextMessage.

getCharProperty function

Returns message properties of any data type.
Syntax

FUNCTION getCharProperty RETURNS CHARACTER (propertyName AS CHARACTER).

Applies to
Message objects

See also
clearProperties procedure on page 183
For more information, see the Accessing message properties.

getClientID function

Returns the client ID value for the SonicMQ Broker connection.

Syntax

FUNCTION getClientID RETURNS CHARACTER.

Applies to
Session objects

Notes
If setClientID procedure on page 273 was not called, Null is returned.

See also
setClientID procedure on page 273, getClientID function on page 202
For more information, see the Setting and getting JMS connection and session attributes.

ggetClientPersistence function

Returns the state of client persistence.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

FUNCTION getClientPersistence RETURNS LOGICAL.
Applies to
Session objects

Notes
• Only applicable for Client Persistence.
• The default is FALSE.

See also

For more information, see the Client persistence and the Client persistence.

ggetClientTransactionBufferSize function
Returns the client buffer size in bytes for Fault Tolerant transacted messages in memory.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

FUNCTION getClientTransactionBufferSize RETURNS INTEGER.

Applies to
Session objects (ClientConnect and ServerConnect only)

Notes
• Only applicable for Fault Tolerant connections.
• The default is 0. A value of 0 tells the SonicMQ client to use the default value as determined by the SonicMQ Broker. This value is the size of the buffer used by the SonicMQ Broker.

See also
setFaultTolerant procedure on page 285, getFaultTolerant function on page 213, isFaultTolerant function on page 251, createChangeStateConsumer procedure on page 185, setFaultTolerantReconnectTimeout procedure on page 286, getFaultTolerantReconnectTimeout function on page 214, setInitialConnectionTimeout procedure on page 290, getInitialConnectionTimeout function on page 217, setClientTransactionBufferSize procedure on page 274, getClientTransactionBufferSize function on page 203

For more information, see the Fault tolerance and the Fault tolerance.
getConnectID function

Returns the Sonic connection ID between the Sonic client and broker.

Syntax

FUNCTION getConnectID RETURNS CHARACTER.

Applies to
Session objects

Notes
• This value is set by calling setConnectID procedure on page 275.
• Returns UNKNOWN when called before setConnectID procedure on page 275.

See also
setConnectID procedure on page 275

getConnectionID function

Returns the AppServer connection ID.

Syntax

FUNCTION getConnectionID RETURNS CHARACTER.

Applies to
Session objects

Notes
• This value is typically used to correlate the session to log entries on the server side.
• Returns UNKNOWN when called before beginSession procedure on page 180.

See also
For more information, see the Establishing session control.
getConnectionMetaData function


Syntax

FUNCTION getConnectionMetaData RETURNS CHARACTER.

Applies to

Session objects

See also

For more information, see the Setting and getting JMS connection and session attributes.

getConnectionURLs function

Returns a comma-separated list of SonicMQ Broker URLs that the client will try to connect to.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

FUNCTION getConnectionURLs RETURNS CHARACTER.

Applies to

Session objects

See also

setConnectionURLs procedure on page 276, getConnectionURLs function on page 205, setSequential procedure on page 310, getSequential function on page 242

For more information, see the Managing fail-over support.

getContentType

Retrieves the content type of the message part corresponding to the index in a MultipartMessage.
Note: The JMS providers other than SonicMQ do not support this method.

Syntax

FUNCTION getContentType RETURNS CHARACTER (INPUT iIndex AS INTEGER).

Applies to
Message objects

See also

For more information, see the MultipartMessage and the MultiPartMessage example.

getDataSet function

Gets the handle to the newly created DataSetMessage.

Syntax

FUNCTION getDataSet (INPUT name, INPUT schemaLocation, INPUT fieldtypeMapping) RETURNS HANDLE.

Applies to
Message objects

Notes

• The handle parameter must be a declared handle. Any previous value of the handle parameter will be lost.

• The schema parameters specify the schema information and are passed directly to the READ-XML method. Specifying an Unknown value (?) for any of the schema parameters will result in the use of the default value for that parameter.

• The DataSet is created from reading the DataSetMessage and using the READ-XML method.

• The name parameter is the name of the widget-pool to be used when creating the DataSet. (For more information on widget pools, see the CREATE-DATASET entry in OpenEdge Development: ABL Reference.) A value of "?” will result in the use of the default pool.

See also
createDataSetMessage procedure on page 186, dataSet procedure on page 277, getDataSet function on page 206
getDateTime function

Returns a date value with no time or time zone information.

Syntax

FUNCTION getDate RETURNS DATE (itemName AS CHARACTER).

Applies to
Message objects

Notes
• Time information, if present, is removed.
• Time zone information, if present, is removed.
• If the application might receive messages originating in different time zones, the ABL programmer should ensure that date/time values are interpreted correctly.

See also
createMapMessage procedure on page 186, setBoolean procedure on page 268, setByte procedure on page 270, setBytesFromRaw procedure on page 271, setChar procedure on page 272, setDateTime procedure on page 277, setDateTimeTZ procedure on page 279, setDouble procedure on page 280, setFloat procedure on page 287, setInt procedure on page 288, setLong procedure on page 296, setLongString procedure on page 298, setShort procedure on page 310, setString procedure on page 313, getMapNames function on page 230, getItemType function on page 217, getBytesToRaw function on page 200, getChar function on page 200, getDate function on page 207, getDateTime function on page 208, getDateTimeTZ function on page 209, getDecimal function on page 210, getInt function on page 215, getLogical function on page 227, getLongString function on page 228, getLongStringCP function on page 228

For more information, see the MapMessage.

getDateProperty function

Returns a date value with no time or time zone information.

Syntax

FUNCTION getDateProperty RETURNS DATE (propertyName AS CHARACTER).

Applies to
Message objects
Notes

- Time information, if present, is removed.
- Time zone information, if present, is removed.
- If the application might receive messages originating in different time zones, the ABL programmer should ensure that date/time values are interpreted correctly.

See also
setDateProperty procedure on page 278, getDateProperty function on page 207

For more information, see the Accessing message properties.

getDateTime function

Returns a date-time value with no time zone information.

Syntax

```FUNCTION getDateTime RETURNS DATETIME (itemName AS CHARACTER).```

Applies to
Message objects

Notes

- Time zone information, if present, is removed.
- If time information is not present, the default time of 12:00AM (midnight) is added.
- If the application might receive messages originating in different time zones, the ABL programmer should ensure that date/time values are interpreted correctly.

See also
createMapMessage procedure on page 186, setBoolean procedure on page 268, setByte procedure on page 270, setBytesFromRaw procedure on page 271, setChar procedure on page 272, setDate procedure on page 277, setDateTime procedure on page 279, setDateTime-TZ procedure on page 280, setDouble procedure on page 283, setFloat procedure on page 287, setInt procedure on page 288, setLong procedure on page 296, setLongString procedure on page 298, setShort procedure on page 310, setString procedure on page 313, getMapNames function on page 230, getItemCount function on page 217, getBytesToRaw function on page 200, getChar function on page 200, getDate function on page 207, getDateTime function on page 208, getDateTime-TZ function on page 209, getDecimal function on page 210, getInt function on page 215, getLogical function on page 227, getLongString function on page 228, getLongStringCP function on page 228

For more information, see the MapMessage.
**getDateTimeProperty function**

Returns a date-time property with no time zone information.

**Syntax**

```plaintext
FUNCTION getDateTimeProperty RETURNS DATETIME (propertyName AS CHARACTER).
```

**Applies to**

Message objects

**Notes**

- Time zone information, if present, is removed.
- If time information is not present, the default time of 12:00AM (midnight) is added.
- If the application might receive messages originating in different time zones, the ABL programmer should ensure that date/time values are interpreted correctly.

**See also**

- `setDateTimeProperty procedure` on page 280
- `getDateTimeProperty function` on page 209

For more information, see the Accessing message properties.

**getDateTime-TZ function**

Returns a date-time value including time zone information.

**Syntax**

```plaintext
FUNCTION getDateTime-TZ RETURNS DATETIME-TZ (itemName AS CHARACTER).
```

**Applies to**

Message objects

**Notes**

- If time information is not present, the default time of 12:00AM (midnight) is added.
- If time zone information is not present, the default time zone of the client application is added.
See also
createMapMessage procedure on page 186, setBoolean procedure on page 268, setByte procedure on page 270, setBytesFromRaw procedure on page 271, setChar procedure on page 272, setDate procedure on page 277, setDateTime procedure on page 279, setDateTime-TZ procedure on page 280, setDouble procedure on page 283, setFloat procedure on page 287, setInt procedure on page 288, setLong procedure on page 296, setLongString procedure on page 298, setShort procedure on page 310, setString procedure on page 313, getMapNames function on page 230, getItemType function on page 217, getBytesToRaw function on page 200, getChar function on page 200, getDate function on page 207, getDateTime function on page 208, getDateTime-TZ function on page 209, getDecimal function on page 210, getInt function on page 215, getLogical function on page 227, getLongString function on page 228, getLongStringCP function on page 228
For more information, see the MapMessage.

getDateTimeTzProperty function

Returns a date-time property including time zone information.

Syntax

```
FUNCTION getDateTimeTzProperty RETURNS DATETIME-TZ (propertyName AS CHARACTER).
```

Applies to

Message objects

Notes

• If time information is not present, the default time of 12:00AM (midnight) is added.
• If time zone information is not present, the default time zone of the client application is added.

See also
setDateTimeTzProperty procedure on page 281 getDateTimeTzProperty function on page 210
For more information, see the Accessing message properties.

getDecimal function

Gets any numeric item from a MapMessage.

Syntax

```
FUNCTION getDecimal RETURNS DECIMAL (itemName AS CHARACTER).
```

Applies to

Message objects
getDecimalProperty function

Returns any numeric message property.

Syntax

```FUNCTION getDecimalProperty RETURNS DECIMAL (property AS CHARACTER).```  

Applies to

Message objects

See also

For more information, see the Accessing message properties.

defaultPersistency function

Returns the value specified by `setDefaultPersistency procedure` on page 281.

Syntax

```FUNCTION getDefaultPersistency RETURNS CHARACTER.```  

Applies to

Session objects

Notes

If `setDefaultPersistency procedure` on page 281 was not called, `PERSISTENT` is returned.
See also
setDefaultPersistency procedure on page 281, getDefaultPersistency function on page 211, setDefaultPriority procedure on page 282, getDefaultPriority function on page 212, setDefaultTimeToLive procedure on page 283, getDefaultTimeToLive function on page 212
For more information, see the Accessing message delivery parameters.

getDefaultPriority function
Returns the value specified by setDefaultPriority procedure on page 282.

Syntax

FUNCTION getDefaultPriority RETURNS INTEGER.

Applies to
Session objects

Notes
If setDefaultPriority procedure on page 282 was not called, 4 is returned.

See also
setDefaultPersistency procedure on page 281, getDefaultPersistency function on page 211, setDefaultPriority procedure on page 282, getDefaultPriority function on page 212, setDefaultTimeToLive procedure on page 283, getDefaultTimeToLive function on page 212
For more information, see the Accessing message delivery parameters.

getDefaultTimeToLive function
Returns the value specified by setDefaultTimeToLive procedure on page 283.

Syntax

FUNCTION getDefaultTimeToLive RETURNS DECIMAL.

Applies to
Session objects

Notes
If setDefaultTimeToLive procedure on page 283 was not called, UNKNOWN is returned.
See also
setDefaultPersistency procedure on page 281, getDefaultPersistency function on page 211, setDefaultPriority procedure on page 282getDefaultPriority function on page 212, setDefaultTimeToLive procedure on page 283, getDefaultTimeToLive function on page 212
For more information, see the Accessing message delivery parameters.

getDestinationName function
Returns the name of the destination that messages arrive from when the Message Consumer was passed to subscribe procedure on page 319 or receiveFromQueue procedure on page 262.

Syntax

```
FUNCTION getDestinationName RETURNS CHARACTER.
```

Applies to
Message Consumer objects

See also
For more information, see the Accessing message handler information.

getFaultTolerant function
Returns the current Fault Tolerant setting.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

```
FUNCTION getFaultTolerant RETURNS LOGICAL.
```

Applies to
Session objects (ClientConnect and ServerConnect only)

Notes
- Only applicable for Fault Tolerant connections.
- The default is FALSE.
See also
setFaultTolerant procedure on page 285, getFaultTolerant function on page 213, isFaultTolerant function on page 251, createChangeStateConsumer procedure on page 185, setFaultTolerantReconnectTimeout procedure on page 286, getFaultTolerantReconnectTimeout function on page 214, setInitialConnectionTimeout procedure on page 290, getInitialConnectionTimeout function on page 217, setClientTransactionBufferSize procedure on page 274, getClientTransactionBufferSize function on page 203

For more information, see the Fault tolerance and the Fault tolerance.

getFaultTolerantReconnectTimeout function

Returns the Fault Tolerant reconnection timeout.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

FUNCTION getFaultTolerantReconnectTimeout RETURNS INTEGER.

Applies to
Session objects (ClientConnect and ServerConnect only)

Notes
• Only applicable for Fault Tolerant connections.
• The default is 60 seconds.

See also
setFaultTolerant procedure on page 285, getFaultTolerant function on page 213, isFaultTolerant function on page 251, createChangeStateConsumer procedure on page 185, setFaultTolerantReconnectTimeout procedure on page 286, getFaultTolerantReconnectTimeout function on page 214, setInitialConnectionTimeout procedure on page 290, getInitialConnectionTimeout function on page 217, setClientTransactionBufferSize procedure on page 274, getClientTransactionBufferSize function on page 203

For more information, see the Fault tolerance and the Fault tolerance.

getFlowToDisk function

This function returns the current flow-to-disk setting.

Note: The JMS providers other than SonicMQ do not support this method.
**Syntax**

FUNCTION getFlowToDisk RETURNS INTEGER.

**Applies to**

Session objects

**See also**

setFlowToDisk procedure on page 288

For more information on this SonicMQ feature, see the "Flow to Disk" section in the "SonicMQ Client Sessions" chapter of the SonicMQ Application Programming Guide.

---

**getInt function**

Gets int, short, or byte items from a MapMessage.

**Syntax**

FUNCTION getInt RETURNS INTEGER (itemName AS CHARACTER).

**Applies to**

Message objects

**See also**

createMapMessage procedure on page 186, setBoolean procedure on page 268, setByte procedure on page 270, setBytesFromRaw procedure on page 271, setChar procedure on page 272, setDate procedure on page 277, setDateTime procedure on page 279, setDateTime-TZ procedure on page 280, setDouble procedure on page 283, setFloat procedure on page 287, setInt procedure on page 288, setLong procedure on page 296, setLongString procedure on page 298, setShort procedure on page 310, setString procedure on page 313, getMapNames function on page 230, getItemType function on page 217, getBytesToRaw function on page 200, getChar function on page 200, getDate function on page 207, getDateTime function on page 208, getDateTime-TZ function on page 209, getDecimal function on page 210, getInt64 function on page 216, getLogical function on page 227, getLongString function on page 228, getLongStringCP function on page 228

For more information, see the MapMessage.

---

**getIntProperty function**

Returns int, short, and byte message properties.
### getIntProperty Function

**Syntax**

```plaintext
FUNCTION getIntProperty RETURNS INTEGER (propertyName AS CHARACTER).
```

**Applies to**

Message objects

**See also**

`setIntProperty procedure` on page 289, `getIntProperty function` on page 215, `setInt64Property procedure` on page 290, `getIntProperty function` on page 215, `clearProperties procedure` on page 183

For more information, see the Accessing message properties.

---

### getInt64 Function

**Gets** INT64 **items from a MapMessage.**

**Syntax**

```plaintext
FUNCTION getInt64 RETURNS INT64 (itemName AS CHARACTER).
```

**Applies to**

Message objects

**See also**

`createMapMessage procedure` on page 186, `setBoolean procedure` on page 268, `setByte procedure` on page 270, `setBytesFromRaw procedure` on page 271, `setChar procedure` on page 272, `setDate procedure` on page 277, `setDateTime procedure` on page 279, `setDateTime-TZ procedure` on page 280, `setDouble procedure` on page 283, `setFloat procedure` on page 287, `setInt procedure` on page 288, `setLong procedure` on page 289, `setInt64 procedure` on page 289, `setLongString procedure` on page 310, `setShort procedure` on page 313, `setMapNames function` on page 230, `getItemType function` on page 217, `getBytesToRaw function` on page 200, `getChar function` on page 200, `getDate function` on page 207, `getDateTime function` on page 208, `getDateTime-TZ function` on page 209, `getDecimal function` on page 210, `getInt function` on page 215, `getLogical function` on page 227, `getLongString function` on page 228, `getLongStringCP function` on page 228

For more information, see the MapMessage.

---

### getInt64Property Function

**Returns** int64 **message properties.**
Syntax

FUNCTION getInt64Property RETURNS INT64 (propertyName AS CHARACTER).

Applies to
Message objects

See also
setInt64Property procedure on page 290, setIntProperty procedure on page 289, getIntProperty function on page 215, clearProperties procedure on page 183
For more information, see the Accessing message properties.

getInitialConnectionTimeout function

Returns the initial Fault Tolerant reconnection timeout in seconds.

**Note:** The JMS providers other than SonicMQ do not support this method.

Syntax

FUNCTION getInitialConnectionTimeout RETURNS INTEGER.

Applies to
Session objects (ClientConnect and ServerConnect only)

Notes
• Only applicable for Fault Tolerant connections.
• The default is 30 seconds.

See also
setFaultTolerant procedure on page 285, getFaultTolerant function on page 213, isFaultTolerant function on page 251, createChangeStateConsumer procedure on page 185, setFaultTolerantReconnectTimeout procedure on page 286, getFaultTolerantReconnectTimeout function on page 214, setInitialConnectionTimeout procedure on page 290, getInitialConnectionTimeout function on page 217, setClientTransactionBufferSize procedure on page 274, getClientTransactionBufferSize function on page 203
For more information, see the Fault tolerance and the Fault tolerance.

gGetItemType function

Returns the data type of an item in a MapMessage.
**Syntax**

```plaintext
FUNCTION getItemType RETURNS CHARACTER (itemName AS CHARACTER).
```

**Applies to**

Message objects

**Notes**

- Possible values include `UNKNOWN`, `boolean`, `byte`, `short`, `char`, `int`, `datetime-tz`, `long`, `float`, `double`, `string`, or `longchar`.
- It returns `UNKNOWN` if the item does not exist.
- The `getItemType` function cannot precisely determine certain data types. It is important to be aware of the following limitations:
  - **String values** — `getItemType` function on page 217 returns the `longchar` value for a data item consisting of a string longer than 32K. If the item is a string of 32K or less, the function returns the `string` value for both `CHARACTER` and `LONGCHAR` data. In the latter case, it is the responsibility of the ABL programmer to know the order of items in the `MapMessage` and to call the correct function to interpret the data appropriately.
  - **Date values** — `getItemType` function on page 217 returns the `datetime-tz` value for all date items.

**See also**

`createMapMessage procedure` on page 186, `setBoolean procedure` on page 268, `setByte procedure` on page 270, `setBytesFromRaw procedure` on page 271, `setChar procedure` on page 272, `setDate procedure` on page 277, `setDateTime procedure` on page 279, `setDateTime-TZ procedure` on page 280, `setDouble procedure` on page 283, `setFloat procedure` on page 287, `setInt procedure` on page 288, `setLong procedure` on page 296, `setLongString procedure` on page 298, `setShort procedure` on page 310, `setString procedure` on page 313, `getMapNames function` on page 230, `getItemType function` on page 217, `getBytesToRaw function` on page 200, `getChar function` on page 200, `getDate function` on page 207, `getDateTime function` on page 208, `getDateTime-TZ function` on page 209, `getDecimal function` on page 210, `getInt function` on page 217, `getLogical function` on page 227, `getLongString function` on page 228, `getLongStringCP function` on page 228

For more information, see the `MapMessage`.

**getJMSCorrelationID function**

Returns the correlation ID.

**Syntax**

```plaintext
FUNCTION getJMSCorrelationID RETURNS CHARACTER.
```

**Applies to**

Message objects
Notes
This value is an application-defined correlation ID, typically the ID of the message replied to.

See also
setJMSCorrelationID procedure on page 291, getJMSCorrelationID function on page 218
For more information, see the Accessing message header properties.

getJMSCorrelationIDAsBytes function
Returns a proprietary (JMS-provider-dependent) correlation ID.

Syntax

FUNCTION getJMSCorrelationIDAsBytes RETURNS RAW.

Applies to
Message objects

Notes
When accessing SonicMQ, the bytesCorrelationID field can be used for storing application-defined values.

See also
setJMSCorrelationIDAsBytes procedure on page 292 getJMSCorrelationIDAsBytes function on page 219
For more information, see the Accessing message header properties.

getJMSDeliveryMode function
Returns the delivery mode.

Syntax

FUNCTION getJMSDeliveryMode RETURNS CHARACTER.

Applies to
Message objects

Notes
• Possible values are PERSISTENT, NON_PERSISTENT, or DISCARDABLE.
• The message receiver never gets the NON_PERSISTENT_ASYNC value. A message sent using NON_PERSISTENT_ASYNC is received with the standard NON_PERSISTENT value.
See also
getJMSDestination function on page 220, getJMSRedelivered function on page 222, getMessageType function on page 233, getJMSMessageID function on page 221, getJMSDeliveryMode function on page 219, getJMSTimestamp function on page 223, getJMSCreationTimestamp function on page 223, getJMSExpiration function on page 220, getJMSPriority function on page 221
For more information, see the Accessing message header properties.

getJMSDestination function

Returns the name of the destination this message was sent to.

Syntax

```function
getJMSDestination
returns
character
```

Applies to

Message objects

Notes

The value is valid after the message was sent (at the sender side) and in the received message (at the receiver side).

See also

getJMSDestination function on page 220, getJMSRedelivered function on page 222, getMessageType function on page 233, getJMSMessageID function on page 221, getJMSDeliveryMode function on page 219, getJMSTimestamp function on page 223, getJMSCreationTimestamp function on page 223, getJMSExpiration function on page 220, getJMSPriority function on page 221
For more information, see the Accessing message header properties.

getJMSExpiration function

Returns the expiration time (GMT).

Syntax

```function
getJMSExpiration
returns
decimal
```

Applies to

Message objects
See also
getJMSDestination function on page 220, getJMSRedelivered function on page 222, getMessageType function on page 233, getJMSMessageID function on page 221, getJMSDeliveryMode function on page 219, getJMSTimestamp function on page 223, getJMSExpiration function on page 220, getJMSPriority function on page 221
For more information, see the Accessing message header properties.

getJMSMessageID function

Returns the message ID, a unique ID that the JMS server assigns to each message.

Syntax

FUNCTION getJMSMessageID RETURNS CHARACTER.

Applies to
Message objects

See also
getJMSDestination function on page 220, getJMSRedelivered function on page 222, getMessageType function on page 233, getJMSMessageID function on page 221, getJMSDeliveryMode function on page 219, getJMSTimestamp function on page 223, getJMSExpiration function on page 220, getJMSPriority function on page 221
For more information, see the Accessing message header properties.

getJMSPriority function

Returns priority values in the range of 0–9, where 4 is the default. Messages sent with higher priority can be expedited by the SonicMQ Broker. Priority values of 5 through 9 are expedited.

Syntax

FUNCTION getJMSPriority RETURNS INTEGER.

Applies to
Message objects

See also
getJMSDestination function on page 220, getJMSRedelivered function on page 222, getMessageType function on page 233, getJMSMessageID function on page 221, getJMSDeliveryMode function on page 219, getJMSTimestamp function on page 223, getJMSExpiration function on page 220, getJMSPriority function on page 221
For more information, see the Accessing message header properties.
getJMSRedelivered function

Returns **TRUE** (at the receiver side) if this is not the first delivery of this message.

**Syntax**

```plaintext
FUNCTION getJMSRedelivered RETURNS LOGICAL.
```

**Applies to**

Message objects

**Notes**

A second delivery can take place if the first delivery is not acknowledged by the receiver or, in a transacted session, if the transaction was rolled back.

**See also**

- [getJMSDestination function](#) on page 220
- [getJMSRedelivered function](#) on page 222
- [getMessageType function](#) on page 233
- [getJMSMessageID function](#) on page 221
- [getJMSDeliveryMode function](#) on page 219
- [getJMSTimestamp function](#) on page 223
- [getJMSExpiration function](#) on page 220
- [getJMSPriority function](#) on page 221

For more information, see the Accessing message header properties.

getJMSReplyTo function

Returns the reply destination.

**Syntax**

```plaintext
FUNCTION getJMSReplyTo RETURNS CHARACTER.
```

**Applies to**

Message objects

**Notes**

- The destination can be the name of a queue, even if the message is received from a Pub/Sub session, and the destination can be the name of a topic even if the message is received from a PTP session.
- An application must call [getReplyToDestinationType function](#) on page 240 if both a queue destination and a topic destination might be stored in the received message.
See also

setJMSReplyTo procedure on page 292, getJMSReplyTo function on page 222, hasReplyTo function on page 249, setReplyToDestinationType procedure on page 308, getReplyToDestinationType function on page 240

For more information, see the Accessing message header properties.

getJMSServerName function

Returns the value set by the preceding setJMSServerName procedure on page 293.

Syntax

FUNCTION getJMSServerName RETURNS CHARACTER.

Applies to

Session objects

Notes

If setJMSServerName procedure on page 293 is not called, the Unknown value (?) is returned.

See also

setJMSServerName procedure on page 293, getJMSServerName function on page 223

For more information, see the Setting and getting JMS connection and session attributes.

getJMSTimestamp function

Returns the message sending time, which is the difference, in milliseconds, between the message creation time and midnight, January 1, 1970 UTC.

Syntax

FUNCTION getJMSTimestamp RETURNS DECIMAL.

Applies to

Message objects

See also

getJMSDestination function on page 220, getJMSRedelivered function on page 222, getMessageType function on page 233, getJMSMessageID function on page 221, getJMSDeliveryMode function on page 219, getJMSTimestamp function on page 223, getJMSExpiration function on page 220, getJMSPriority function on page 221
For more information, see the Accessing message header properties.

getJMSType function

Returns a proprietary (JMS-provider-dependent) type name.

Syntax

```FUNCTION getJMSType RETURNS CHARACTER.```

Applies to

Message objects

Notes

When accessing SonicMQ, the JMSType field can be used for storing application-defined values.

See also

setJMSType procedure on page 293 getJMSType function on page 224

For more information, see the Accessing message header properties.

getLoadBalancing function

Returns a LOGICAL value indicating whether load balancing is enabled.

Syntax

```FUNCTION getLoadBalancing RETURNS LOGICAL.```

Applies to

Session objects

Notes

- **TRUE** indicates load balancing is enabled. **FALSE** indicates it is not enabled.
- With load balancing, the client is willing to have a connect request redirected to another SonicMQ Broker within a SonicMQ cluster.

See also

setLoadBalancing procedure on page 294, getLoadBalancing function on page 224

For more information, see the Load balancing.
getLocalStoreDirectory function

Returns the directory that will be used by the adapter to persist messages.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

FUNCTION getLocalStoreDirectory RETURNS CHARACTER.

Applies to

Session objects (ClientConnect and ServerConnect only).

Notes

• Only applicable for Client Persistence.
• This value is set by a call to setLocalStoreDirectory procedure on page 294.
• If setLocalStoreDirectory procedure on page 294 was never called, the value is the Unknown value (?).

See also

createRejectedMessageConsumer procedure on page 188, setClientPersistence procedure on page 273, getClientPersistence function on page 222, setLocalStoreDirectory procedure on page 294, getLocalStoreDirectory function on page 225, setLocalStoreSize procedure on page 295, getLocalStoreSize function on page 225, setLocalStoreWaitTime procedure on page 296, getLocalStoreWaitTime function on page 226, setReconnectTimeout procedure on page 304, getReconnectTimeout function on page 237, setReconnectInterval procedure on page 304, getReconnectInterval function on page 236

For more information, see the Client persistence and the Client persistence.

getLocalStoreSize function

Returns the maximum size of the local store in kilobytes.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

FUNCTION getLocalStoreSize RETURNS INTEGER.

Applies to

Session objects (ClientConnect and ServerConnect only).
Notes

• Only applicable for Client Persistence.
• The default size is 10000 (10MB).

See also


For more information, see the Client persistence and the Client persistence.

getLocalStoreWaitTime function

Returns the wait interval before Client Persistence begins.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

FUNCTION getLocalStoreWait RETURNS INTEGER.

Applies to

Session objects (ClientConnect and ServerConnect only).

Notes

• Only applicable for Client Persistence.
• The default is 5 seconds.

See also


For more information, see the Client persistence and the Client persistence.
getLogical function

Returns a boolean item by name from a MapMessage.

Syntax

```
FUNCTION getLogical RETURNS LOGICAL (itemName AS CHARACTER).
```

Applies to
Message objects

See also
createMapMessage procedure on page 186, setBoolean procedure on page 268, setByte procedure on page 270, setBytesFromRaw procedure on page 271, setChar procedure on page 272, setDateTime procedure on page 277, setDateTime-TZ procedure on page 280, setDouble procedure on page 283, setFloat procedure on page 287, setInt procedure on page 288, setLong procedure on page 296, setLongString procedure on page 298, setShort procedure on page 310, setString procedure on page 313, getMapNames function on page 230, getItemType function on page 217, getBytesToRaw function on page 200, getChar function on page 200, getDateTime function on page 207, getDateTime-TZ function on page 209, getDecimal function on page 210, getInt function on page 215, getLogical function on page 227, getLongString function on page 228, getLongStringCP function on page 228

For more information, see the MapMessage.

getLogicalProperty function

Returns a boolean message property.

Syntax

```
FUNCTION getLogicalProperty RETURNS LOGICAL (propertyName AS CHARACTER).
```

Applies to
Message objects

See also
clearProperties procedure on page 183

For more information, see the Accessing message properties.
getLongString function

Returns a LONGCHAR item by name from a MapMessage.

Syntax

```
FUNCTION getLongString RETURNS LONGCHAR (itemName AS CHARACTER).
```

Applies to

Message objects

Notes

The text is converted to the current default code page of the OpenEdge client application. To return text based on a different code page, use `getLongTextCP function` on page 229.

See also

createMapMessage procedure on page 186, setBoolean procedure on page 268, setByte procedure on page 270, setBytesFromRaw procedure on page 271, setChar procedure on page 272, setDate procedure on page 277, setDateTime procedure on page 279, setDateTime-TZ procedure on page 280, setDouble procedure on page 283, setFloat procedure on page 287, setInt procedure on page 288, setLong procedure on page 296, setLongString procedure on page 298, setShort procedure on page 310, setString procedure on page 313, getMapNames function on page 230, getItemType function on page 217, getBytesToRaw function on page 200, getChar function on page 200, getDate function on page 207, getDateTime function on page 208, getDateTime-TZ function on page 209, getDecimal function on page 210, getInt function on page 215, getLogical function on page 227, getLongString function on page 228, getLongStringCP function on page 228

For more information, see the MapMessage.

getLongStringCP function

Returns a String item by name from a MapMessage and converts the text to the specified code page.

Syntax

```
FUNCTION getLongString RETURNS LONGCHAR (code_page AS CHARACTER, itemName AS CHARACTER).
```

Applies to

Message objects
See also
createMapMessage procedure on page 186, setBoolean procedure on page 268, setByte procedure on page 270, setBytesFromRaw procedure on page 271, setChar procedure on page 272, setDateTime procedure on page 277, setDateTime-TZ procedure on page 280, setDouble procedure on page 283, setFloat procedure on page 287, setInt procedure on page 288, setLong procedure on page 296, setLongString procedure on page 298, setShort procedure on page 310, setString procedure on page 313, getMapNames function on page 230, getItemType function on page 217, getBytesToRaw function on page 200, getChar function on page 200, getDate function on page 207, getDateTime function on page 208, getDateTime-TZ function on page 209, getDecimal function on page 210, getInt function on page 215, getLogical function on page 227, getLongString function on page 228, getLongText function on page 229.

For more information, see the MapMessage.

getLongText function

Returns all text in a TextMessage.

Syntax

```
FUNCTION getLongText RETURNS LONGCHAR.
```

Applies to

Message objects

Notes

• Implicitly calls reset procedure on page 265.
• The text is converted to the current default code page of the OpenEdge client application. To return text based on a different code page, use getLongTextCP function on page 229.

See also
cREATEMapMessage procedure on page 191, setText procedure on page 315, setLongText procedure on page 298, appendText procedure on page 180, endOfStream function on page 195, getCharCount function on page 201, getText function on page 245, getTextSegment function on page 246, getLongText function on page 229, getLongTextCP function on page 229.

For more information, see the TextMessage.

getLongTextCP function

Returns all text in a TextMessage and converts it to the specified code page.

Syntax

```
FUNCTION getLongTextCP RETURNS LONGCHAR (code_page as CHARACTER).
```

OpenEdge Development: Messaging and ESB
Applies to
Message objects

Notes
• Implicitly calls reset procedure on page 265.
• The LONGCHAR data returned is converted to the code page specified by the code_page parameter.

See also
createTextMessage procedure on page 191, setText procedure on page 315, setLongText procedure on page 298, appendText procedure on page 180, eofStream function on page 195, getCharCount function on page 201, getText function on page 245, getTextSegment function on page 246, getLongText function on page 229, getLongTextCP function on page 229
For more information, see the TextMessage.

getMapNames function
Returns a comma-separated list of the item names in a MapMessage.

Syntax

FUNCTION getMapNames RETURNS CHARACTER.

Applies to
Message objects

See also
createMapMessage procedure on page 186, setBoolean procedure on page 268, setByte procedure on page 270, setBytesFromRaw procedure on page 271, setChar procedure on page 272, setDate procedure on page 277, setDateTime procedure on page 279, setDateTime-TZ procedure on page 280, setDouble procedure on page 283, setFloat procedure on page 287, setInt procedure on page 288, setLong procedure on page 296, setLongString procedure on page 298, setShort procedure on page 310, setString procedure on page 313, getMapNames function on page 230, getItemType function on page 217, getBytesToRaw function on page 200, getChar function on page 200, getDate function on page 207, getTime function on page 208, getTime-TZ function on page 209, getDecimal function on page 210, getInt function on page 215, getLogical function on page 227, getLongString function on page 228, getLongStringCP function on page 228
For more information, see the MapMessage.

getMemptr function
Returns a reference to a MEMPTR variable that contains exactly all the bytes of a BytesMessage.
**getMemptr function**

Retrieves a message part corresponding to the given `contentID`.

### Syntax

```plaintext
FUNCTION getMemptr RETURNS MEMPTR.
```

### Applies to

Message objects

### Notes

- This function implicitly calls `reset procedure` on page 265.
- If the message was in a write-only mode, it will be in a read-only/reset mode after the call.
- `getMemptr function` on page 230 does not create a copy of the `MEMPTR` variable; it returns a reference to the data maintained by the Message object.
- `deleteMessage procedure` on page 193 call releases the variable’s memory, and the caller must copy any data it needs or needs to modify before deleting the message.

### See also

- `createBytesMessage procedure` on page 184, `setMemptr procedure` on page 299, `writeBoolean procedure` on page 321, `writeByte procedure` on page 321, `writeBytesFromRaw procedure` on page 322, `writeChar procedure` on page 323, `writeDate procedure` on page 323, `writeDateTime procedure` on page 324, `writeDateTime-TZ procedure` on page 325, `writeDouble procedure` on page 326, `writeFloat procedure` on page 326, `writeInt procedure` on page 327, `writeLong procedure` on page 328, `writeLongString procedure` on page 329, `writeShort procedure` on page 330, `writeString procedure` on page 330, `getContentType` on page 205, `getBytesCount function` on page 198, `endOfStream function` on page 195, `moveToNext procedure` on page 253, `readBytesToRaw procedure` on page 256, `readChar function` on page 256, `readDate function` on page 257, `readDateTime function` on page 258, `readDateTime-TZ function` on page 258, `readDecimal function` on page 259, `readInt function` on page 259, `readLogical function` on page 261, `readLongString function` on page 261, `readLongStringCP function` on page 262, `getMemptr function` on page 230

For more information, see the see the StreamMessage and the BytesMessage.

For an example, see the Publishing, subscribing, and receiving an XML document in a BytesMessage.

**getMessagePartByID function**

Retrieves a message part corresponding to the given `contentID`.

### Syntax

```plaintext
FUNCTION getMessagePartByID RETURNS CHARACTER
    (INPUT contentID AS INTEGER, OUTPUT messagePartH AS HANDLE).
```

### Applies to

Message objects
Notes

• When you use the same handle variable to retrieve multiple message parts, after each retrieval, call deleteMessage procedure on page 193 on the handle variable to free the message part.

• The getMessagePartByID function returns the content-type of the message. You can use it to identify the message type of the message part.

See also

createMultipartMessage procedure on page 188, addBytesPart procedure on page 178, addMessagePart procedure on page 178, addTextPart procedure on page 179, isMessagePart function on page 251, getContentType on page 205, getPartCount function on page 234, getMessagePartById function on page 232, getMessagePartByIndex function on page 231, getMessagePartByIndex function on page 232, getMessagePartByIndex function on page 232, writeBytesFromRaw procedure on page 322, readBytesToRaw procedure on page 256, setMemptr procedure on page 299, getMemptr function on page 230

For more information, see the MultipartMessage and the MultiPartMessage example.

getMessagePartByIndex function

Retrieves a handle to the message part corresponding to the index.

Syntax

FUNCTION getMessagePartByIndex RETURNS CHARACTER
   (INPUT index AS INTEGER, OUTPUT messagePartH AS HANDLE).

Applies to

Message objects

Notes

• When you use the same handle variable to retrieve multiple message parts, between retrievals, call deleteMessage procedure on page 193 on the handle variable to free the message part.

• The getMessagePartByIndex function returns the content-type of the message. You can use it to identify the message type of the message part.

See also

createMultipartMessage procedure on page 188, addBytesPart procedure on page 178, addMessagePart procedure on page 178, addTextPart procedure on page 179, isMessagePart function on page 251, getPartCount function on page 234, getMessagePartById function on page 232, writeBytesFromRaw procedure on page 322, readBytesToRaw procedure on page 256, setMemptr procedure on page 299, getMemptr function on page 230

For more information, see the MultipartMessage and the MultiPartMessage example.
getMessageType function

Returns one of the following OpenEdge Adapter for SonicMQ message types: TextMessage, MapMessage, StreamMessage, BytesMessage, HeaderMessage, XMLMessage, MultipartMessage, TempTableMessage, or DataSetMessage.

Syntax

FUNCTION getMessageType RETURNS CHARACTER.

Applies to
Message objects

See also
getJMSDestination function on page 220, getJMSRedelivered function on page 222, getMessageType function on page 233, getJMSMessageID function on page 221, getJMSDeliveryMode function on page 219, getJMSTimestamp function on page 223, getJMSExpiration function on page 220, getJMSPriority function on page 221

For more information, see the Accessing message header properties.

getNoAcknowledge function

Returns TRUE if setNoAcknowledge procedure on page 300 was called.

Syntax

FUNCTION getNoAcknowledge RETURNS LOGICAL.

Applies to
Message Consumer objects

See also
acknowledgeAndForward procedure on page 177, setSingleMessageAcknowledgement procedure on page 312, getSingleMessageAcknowledgement function on page 243, setNoAcknowledge procedure on page 300, getNoAcknowledge function on page 233

For more information, see the Message acknowledgement, forwarding, and recovery and the Single-message acknowledgement.
**getPartCount function**

Returns the number of parts in a MultipartMessage.

**Syntax**

```ABL
FUNCTION getPartCount RETURNS INTEGER.
```

**Applies to**

Message objects

**See also**

createMultipartMessage procedure on page 188, addBytesPart procedure on page 178, addMessagePart procedure on page 178, addTextPart procedure on page 179, isMessagePart function on page 251, getContentType on page 205, getMessagePartByID function on page 231, getMessagePartByIndex function on page 232, writeBytesFromRaw procedure on page 322, readBytesToRaw procedure on page 256, setMemptr procedure on page 299, getMemptr function on page 230

**getPassword**

Returns the value set by the preceding setPassword procedure on page 301.

For more information, see the MultipartMessage and the MultiPartMessage example.getPassword function

**Syntax**

```ABL
FUNCTION getPassword RETURNS CHARACTER.
```

**Applies to**

Session objects

**Note**

If setPassword procedure on page 301 was not called, The Unknown value (?) is returned.

**See also**

setPassword procedure on page 301

For more information, see the Setting and getting JMS connection and session attributes.
getProcHandle function

Returns the handle to a procedure that contains the name of an internal procedure for handling messages.

Syntax

```cobol
FUNCTION getProcHandle RETURNS HANDLE.
```

Applies to
Message Consumer objects

See also
getProcName function on page 235, getProcHandle function on page 235

For more information, see the Accessing message handler information.

getProcName function

Returns the name of the internal procedure for handling messages.

Syntax

```cobol
FUNCTION getProcName RETURNS CHARACTER.
```

Applies to
Message Consumer objects

See also
getProcName function on page 235, getProcHandle function on page 235

For more information, see the Accessing message handler information.

getPropertyNames function

Returns a comma-separated list of the properties of a message.

Syntax

```cobol
FUNCTION getPropertyNames RETURNS CHARACTER.
```
getPropertyType function

Returns the message property's data type.

Syntax

```
FUNCTION getPropertyType RETURNS CHARACTER (propertyName AS CHARACTER).
```

Parameter

`propertyName`

The message property's data type. Possible values are: `UNKNOWN`, `boolean`, `byte`, `short`, `char`, `int`, `long`, `float`, `double`, or `string`.

Applies to

Message objects

Notes

• If the property was not set in the message, the `UNKNOWN` is returned.

• Since date-time values are transmitted as `String` data, the function cannot distinguish them from other strings. The ABL programmer must know the order of properties in the header and call the correct function to interpret date-time values appropriately.

See also

`clearProperties procedure` on page 183

For more information, see the Accessing message properties.

getReconnectInterval function

Returns the interval for reconnection attempts in seconds.

**Note:** The JMS providers other than SonicMQ do not support this method.
**getReconnectInterval**

**FUNCTION getReconnectInterval** **RETURNS** **INTEGER**

**Applies to**
Session objects (ClientConnect and ServerConnect only).

**Notes**
- Only applicable for Client Persistence.
- The default is 30 seconds.

**See also**

For more information, see the Client persistence and the Client persistence.

**getReconnectTimeout function**

Returns the timeout for reconnection attempts in minutes.

**Note:** The JMS providers other than SonicMQ do not support this method.

**Syntax**

```
FUNCTION getReconnectTimeout RETURNS INTEGER
```

**Applies to**
Session objects (ClientConnect and ServerConnect only).

**Notes**
- Only applicable for Client Persistence.
- The default is 0 indicating no timeout.
getReplyAutoDelete function

Returns whether all reply messages are to be automatically deleted or not.

Syntax

FUNCTION getReplyAutoDelete RETURNS LOGICAL.

Applies to

Message Consumer objects

See also


For more information, see the Accessing message handler information and the Setting reply properties.

getReplyPersistency function

Returns the value for message persistency.

Syntax

FUNCTION getReplyPersistency RETURNS CHARACTER.

Applies to

Message Consumer objects

Notes

If setReplyPersistency procedure on page 306 was not called, PERSISTENT is returned.
**See also**


For more information, see the Accessing message handler information and the Setting reply properties.

---

### getReplyPriority function

Returns the priority of the reply messages.

**Syntax**

```plaintext
FUNCTION getReplyPriority RETURNS INTEGER.
```

**Applies to**

Message Consumer objects

**Notes**

If setReplyPriority procedure on page 306 was not called, the returned value is 4.

**See also**


For more information, see the Accessing message handler information and the Setting reply properties.

---

### getReplyTimeToLive function

Returns the time to live value (in milliseconds) of the reply messages.

**Syntax**

```plaintext
FUNCTION getReplyTimeToLive RETURNS DECIMAL.
```

**Applies to**

Message Consumer objects

**Notes**

If setReplyTimeToLive procedure on page 307 was not called, UNKNOWN is returned.

---
getReplyToDestinationType function

Returns queue, topic, or UNKNOWN.

Syntax

FUNCTION getReplyToDestinationType RETURNS CHARACTER.

Applies to
Message objects

Notes
• Applications use this function when the domain of the ReplyTo field is not known.
• The type value can be queue or topic.
• If getReplyToDestinationType function on page 240 is not called, a default type is automatically set when the message is sent, according to the type of the session: queue for PTP or topic for Pub/Sub.
• For a jmsSession, the default is queue.

See also
setJMSReplyTo procedure on page 292, getJMSReplyTo function on page 222, hasReplyTo function on page 249, setReplyToDestinationType procedure on page 308, getReplyToDestinationType function on page 240

For more information, see the Accessing message header properties.

getReuseMessage function

Returns the value set by setReuseMessage procedure on page 308.

Syntax

FUNCTION getReuseMessage RETURNS LOGICAL.

Applies to
Message Consumer objects
Notes
TRUE if setReuseMessage procedure on page 308 was called; if not, it returns FALSE.

See also
setReuseMessage procedure on page 308, getReuseMessage function on page 240
For more information, see the Reusing messages.

getSaxWriter function
Creates a SAX-WRITER and sets the output destination to an internal longchar for intermediate storage.

Syntax

FUNCTION getSaxWriter (INPUT name) RETURNS HANDLE.

Applies to
Message objects

Notes
• The caller uses SAX-WRITER methods on the returned handle to create the XML document. (which will be written to the internal longchar).
• The name parameter is the name of the widget-pool to be used when creating the SAX-WRITER. The Unknown value (?) results in the use of the default pool.
• When XML creation is completed, call deleteSaxWriter procedure on page 193.
• Possible errors are returned by CREATE-SAX-WRITER or SET-OUTPUT-DESTINATION.

See also
For more information, see the XMLMessage.

getSession function
Returns a handle to the session.

Syntax

FUNCTION getSession RETURNS HANDLE.
Applies to
Message Consumer objects

See also
beginSession procedure on page 180, getSession function on page 241, deleteSession procedure on page 194
For more information, see the Accessing message handler information.

getcSelectorAtBroker function

Gets the broker selector setting.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

FUNCTION getSelectorAtBroker RETURNS LOGICAL.

Applies to
Session objects

Notes
The default is FALSE.

See also
setSelectorAtBroker procedure on page 309, getSelectorAtBroker function on page 242
For more information, see the Message selectors.

getcSequential function

Returns a LOGICAL value indicating how a fail-over list is used.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

FUNCTION getSequential RETURNS LOGICAL.

Applies to
Session objects
Notes

• When using a fail-over list, clients try to connect to brokers in the list either sequentially or randomly.

• \texttt{getSequential function} on page 242 returns \texttt{TRUE} if connection attempts are sequential and \texttt{FALSE} if connection attempts are random.

See also

\texttt{setConnectionURLs procedure} on page 276, \texttt{getConnectionURLs function} on page 205, \texttt{setSequential procedure} on page 310, \texttt{getSequential function} on page 242

For more information, see the Managing fail-over support.

\textbf{getShutdownWaitFor function}

Returns a \texttt{LOGICAL} value indicating the current value of the shutdown \texttt{WAIT-FOR} flag.

Syntax

\begin{verbatim}
FUNCTION getShutdownWaitFor RETURNS LOGICAL.
\end{verbatim}

Applies to

Session objects

Notes

• \texttt{getShutdownWaitFor function} on page 243 returns \texttt{YES} if \texttt{setShutdownWaitFor procedure} on page 312 was never called.

See also

\texttt{setShutdownWaitFor procedure} on page 312

\textbf{getSingleMessageAcknowledgement function}

Returns a logical value indicating whether a client session is configured to use single-message acknowledgement.

\textbf{Note:} The JMS providers other than SonicMQ do not support this method.

Syntax

\begin{verbatim}
FUNCTION getSingleMessageAcknowledgement RETURNS LOGICAL.
\end{verbatim}

Applies to

Session objects
Notes

- `getSingleMessageAcknowledgement function` on page 243 returns `TRUE` if the client session is configured to use single-message acknowledgement.
- `getSingleMessageAcknowledgement function` on page 243 returns `FALSE` if the client session is not so configured.

See also

`acknowledgeAndForward procedure` on page 177, `setSingleMessageAcknowledgement procedure` on page 312, `getSingleMessageAcknowledgement function` on page 243, `setNoAcknowledge procedure` on page 300, `getNoAcknowledgement function` on page 233

For more information, see the Message acknowledgement, forwarding, and recovery and the Single-message acknowledgement.

getTempTable function

Gets the handle to the newly created TempTable.

Syntax

```
FUNCTION getTempTable (INPUT name, INPUT schemaLocation,
                        INPUT fieldTypeMapping) RETURNS HANDLE.
```

Applies to

Message objects

Notes

- The handle parameter must be a declared handle. Any previous value of the handle parameter will be lost.
- The schema parameters specify the schema information and are passed directly to the `READ-XML` method. Specifying an `Unknown` value (?) for any of the schema parameters will result in the use of the default value for that parameter.
- The TempTable is created from reading the TempTableMessage and using the `READ-XML` method.
- The `name` parameter is the name of the widget-pool to be used when creating the TempTable. (For more information on widget pools, see the `CREATE-TEMP-TABLE` entry in *OpenEdge Development: ABL Reference.*) A value of "?" will result in the use of the default pool.

See also

`createTempTableMessage procedure` on page 191, `setTempTable procedure` on page 314, `getTempTable function` on page 244

For more information, see the TempTableMessage.
getText function

Returns all text in a TextMessage or XMLMessage.

Syntax

```FUNCTION getText RETURNS CHARACTER.
```

Applies to

Message objects

Notes

• A run-time error occurs if the message is too large to be handled by the ABL interpreter.
• Implicitly calls reset procedure on page 265.

See also

createTextMessage procedure on page 191, setText procedure on page 315, setLongText procedure on page 298, appendText procedure on page 180, endOfStream function on page 195, getCharCount function on page 201, getText function on page 245, getTextSegment function on page 246, getLongText function on page 229, getLongTextCP function on page 229

For more information, see the TextMessage.

getTextPartByID function

Retrieves a text part and returns the content type as a CHARACTER string.

Syntax

```FUNCTION getTextPartByID RETURNS CHARACTER
   (INPUT contentID AS INTEGER, OUTPUT partBody AS CHARACTER).
```

Applies to

Message objects

Notes

• This function converts the text part from UTF-8 to the SESSION:CPINTERNAL code page.
• If the message body exceeds 32K, this function raises an error. To avoid this, use getBytesPartByID function on page 198.
See also
getBytesPartByID function on page 198, getMessagePartByID function on page 231, getTextPartByID function on page 245
For more information, see MultipartMessage

**getTextPartByIndex function**
Retrieves a text part and returns the content type as a CHARACTER string.

**Syntax**

```plaintext
FUNCTION getTextPartByIndex RETURNS CHARACTER
    (INPUT iIndex AS INTEGER, OUTPUT partBody AS CHARACTER).
```

**Applies to**
Message objects

**Notes**
- This function converts the text part from UTF-8 to the SESSION:CPINTERNAL code page.
- If the message body exceeds 32K, this function raises an error. To avoid this, use getBytesPartByIndex function on page 199.

See also
getBytesPartByIndex function on page 199, getMessagePartByIndex function on page 232, getTextPartByIndex function on page 246
For more information, see the MultipartMessage.

**getTextSegment function**
Returns the next text segment when handling large messages in read-only mode.

**Syntax**

```plaintext
FUNCTION getTextSegment RETURNS CHARACTER.
```

**Applies to**
Message objects

**Notes**
- As an alternative to retrieving multiple text segments with getTextSegment function on page 246, you can use getLongText function on page 229 to retrieve LONGCHAR data in a single operation.
getTransactedReceive function

Returns the value set by the preceding setTransactedReceive procedure on page 315.

Syntax

```plaintext
FUNCTION getTransactedReceive RETURNS LOGICAL.
```

Applies to

Session objects

Notes

If setTransactedReceive procedure on page 315 was not called, FALSE is returned.

See also

setTransactedReceive procedure on page 315, getTransactedReceive function on page 247, setTransactedSend procedure on page 316, getTransactedSend function on page 247

For more information, see the Setting and getting JMS connection and session attributes.

getTransactedSend function

Returns the value set by the preceding setTransactedSend procedure on page 316.

Syntax

```plaintext
FUNCTION getTransactedSend RETURNS LOGICAL.
```

Applies to

Session objects

Notes

If setTransactedSend procedure on page 316 was not called, FALSE is returned.
See also
For more information, see the Setting and getting JMS connection and session attributes.

getUser function
Returns the value set by the preceding setUser procedure on page 316.

Syntax

```ABL
FUNCTION getUser RETURNS CHARACTER.
```

Applies to
Session objects

Notes
If setUser procedure on page 316 was not called, the Unknown value (?) is returned.

See also
setUser procedure on page 316, getUser function on page 248
For more information, see the Setting and getting JMS connection and session attributes.

getX-Document function
Parses an XML document from a SonicMQ XMLMessage into an X-DOCUMENT.

Syntax

```ABL
FUNCTION getX-Document RETURNS HANDLE.
```

Applies to
Message objects

Notes
• The handle to the created X-DOCUMENT is returned as the value of this function. CREATE X-DOCUMENT can return an error.
• The caller is responsible for deleting the handle when the application is done processing the X-DOCUMENT.
• Errors from getLongText function on page 229 and xdoc:LOAD will be returned.
hasReplyTo function

Returns TRUE if the JMSreplyTo header was set.

Syntax

FUNCTION hasReplyTo RETURNS LOGICAL.

Applies to
Message objects

See also
setJMSReplyTo procedure on page 292, getJMSReplyTo function on page 222, hasReplyTo function on page 249, setReplyToDestinationType procedure on page 308, getReplyToDestinationType function on page 240

For more information, see the Accessing message header properties.

inErrorHandling function

Returns TRUE when called from a message handler if the application is handling an error message.

Syntax

FUNCTION inErrorHandling RETURNS LOGICAL.

Applies to
Message Consumer objects

See also
inErrorHandling function on page 249, inMessageHandling function on page 250, inQueueBrowsing function on page 250, inReplyHandling function on page 250

For more information, see the Accessing message handler information.
inMessageHandling function

Returns TRUE when called from a message handler if the application is handling the data in a subscription (or queue) message.

Syntax

FUNCTION inMessageHandling RETURNS LOGICAL.

Applies to
Message Consumer objects

See also
inErrorHandling function on page 249, inMessageHandling function on page 250, inQueueBrowsing function on page 250, inReplyHandling function on page 250

For more information, see the Accessing message handler information.

inQueueBrowsing function

Returns TRUE when called from a message handler if an application is handling a queue browsing message.

Syntax

FUNCTION inQueueBrowsing RETURNS LOGICAL.

Applies to
Message Consumer objects

See also
inErrorHandling function on page 249, inMessageHandling function on page 250, inQueueBrowsing function on page 250, inReplyHandling function on page 250

For more information, see the Accessing message handler information.

inReplyHandling function

Returns TRUE when called from a message handler if an application is handling a reply message.
**isFaultTolerant function**

Determines if the SonicMQ Broker connected supports Fault Tolerance.

**Syntax**

```plaintext
FUNCTION isFaultTolerant RETURNS LOGICAL.
```

**Applies to**

Session objects

**Notes**

- Only applicable for Fault Tolerant connections.
- This function must be called after `beginSession` procedure on page 180 is called.

**See also**

- `setFaultTolerant` procedure on page 285, `getFaultTolerant` function on page 213, `isFaultTolerant` function on page 251, `createChangeStateConsumer` procedure on page 185, `setFaultTolerantReconnectTimeout` procedure on page 286, `getFaultTolerantReconnectTimeout` function on page 214, `setInitialConnectionTimeout` procedure on page 290, `getInitialConnectionTimeout` function on page 217, `setClientTransactionBufferSize` procedure on page 274, `getClientTransactionBufferSize` function on page 203

For more information, see the Fault tolerance and the Fault tolerance.

**Note:** The JMS providers other than SonicMQ do not support this method.

---

**isMessagePart function**

Returns **TRUE** if the part specified by `index` is a SonicMQ message.
Syntax

FUNCTION isMessagePart RETURNS LOGICAL (INPUT index AS INTEGER).

Applies to
Message objects

Notes
If isMessagePart function on page 251 returns TRUE, use message-part access methods.

See also
createMultipartMessage procedure on page 188, addBytesPart procedure on page 178, addMessagePart procedure on page 178, addTextPart procedure on page 179, isMessagePart function on page 251, getContentType on page 205, getPartCount function on page 234, getMessagePartByID function on page 231, getMessagePartByIndex function on page 232, writeBytesFromRaw procedure on page 322, readBytesToRaw procedure on page 256, setMemptr procedure on page 299, getMemptr function on page 230

For more information, see the MultipartMessage and the MultiPartMessage example.

JMS-MAXIMUM-MESSAGES global variable

Changes the maximum number of JMS messages in an OpenEdge session.

Syntax

DEFINE NEW GLOBAL SHARED VAR JMS-MAXIMUM-MESSAGES AS INTEGER INIT new-val.

Notes
• The total number of messages includes messages created by the application and messages received from JMS.
• The default is 50.
• If you exceed the message limit, an error is returned.
• To change the default to new-val, the variable definition must be included in the main procedure of the OpenEdge application.

See also
For information on this global variable in context, see the Setting the maximum number of messages.

messageHandler procedure

Handles incoming JMS and error messages.
Syntax

```
PROCEDURE messageHandler.
DEFINE INPUT PARAMETER message AS HANDLE.
DEFINE INPUT PARAMETER messageConsumer AS HANDLE.
DEFINE OUTPUT PARAMETER reply AS HANDLE.
```

Parameters

**message**

The message.

**messageConsumer**

The Message Consumer object that contains this message handler. The application can use the Message Consumer object to get context information about the message (for example, the session handle to the session that received that message) and the context (for example, the session handler).

**reply**

A handle to the reply message, if any. The application can reply to the message automatically without having to extract the reply to fields. The application can set the reply parameter with a reply message, which is automatically sent to the `JMSReplyTo` destination of the message. If the `setReplyAutoDelete` procedure on page 305 (true) Message Consumer procedure is called, the reply message is automatically deleted after being sent.

Applies to

Message objects

Notes

- The message handler is written by an application and must be registered with a Message Consumer object.
- When a message is received, the message handler is called automatically so the application can process the message.

See also

- `createMessageConsumer` procedure on page 187, `deleteConsumer` procedure on page 192, `messageHandler` procedure on page 252, `waitForMessages` procedure on page 320

For more information see the Message Consumer objects, the Consuming messages, the Terminating the Message Consumer object, and the Creating a message handler process.

**moveToNext procedure**

Moves the cursor to the next data item in a `StreamMessage` and returns its data type.
Syntax

```FUNCTION moveToNext RETURNS CHARACTER.
```

Applies to
Message objects

Notes

- Possible return values include the Unknown value (\(?\)), boolean, byte, short, char, int, long, float, double, string, or byte.
- The Unknown value (\(?\)) is returned when the value of the item is NULL.
- When the message is received or after reset procedure on page 265 is called, the cursor is set before the first data item.
- It is an error to try to move the cursor beyond the last item.
- moveToNext procedure on page 253 function cannot precisely determine certain data types. It is important to be aware of the following limitations:
  - String values — moveToNext procedure on page 253 returns the longchar value for a data item consisting of a string longer than 32K. If the item is a string of 32K or less, the function returns the string value for both CHARACTER and LONGCHAR data. In the latter case, it is the responsibility of the ABL programmer to know the order of items in the StreamMessage and to call the correct function to interpret the data appropriately.
  - Date values — moveToNext procedure on page 253 returns the string value for all date items. The ABL programmer must know the order of items in the StreamMessage and call the correct function to interpret the data appropriately.
  - INT64 values — moveToNext procedure on page 253 returns the long value for INT64 message data. It is the responsibility of the ABL programmer to know the order of items in the StreamMessage and to call the correct function to interpret the data appropriately.

See also
createStreamMessage procedure on page 189, writeBoolean procedure on page 321, writeByte procedure on page 321, writeBytesFromRaw procedure on page 322, writeChar procedure on page 323, writeDate procedure on page 323, writeDateTime procedure on page 324, writeDateTime-TZ procedure on page 325, writeDouble procedure on page 326, writeFloat procedure on page 326, writeInt procedure on page 327, writeLong procedure on page 328, writeLongString procedure on page 329, writeShort procedure on page 330, writeString procedure on page 330, endOfStream function on page 195, moveToNext procedure on page 253, readBytesToRaw procedure on page 256, readChar function on page 256, readDate function on page 257, readDateTime function on page 258, readDateTime-TZ function on page 258, readDecimal function on page 259, readInt function on page 259, readInt64 function on page 260, readLogical function on page 261, readLongString function on page 261, readLongStringCP function on page 262

For more information, see the StreamMessage.

publish procedure
Publishes a message to a topic.
Syntax

```lang
PROCEDURE publish.
DEFINE INPUT PARAMETER topicName AS CHARACTER.
DEFINE INPUT PARAMETER message AS HANDLE.
DEFINE INPUT PARAMETER priority AS INTEGER.
DEFINE INPUT PARAMETER timeToLive AS DECIMAL.
DEFINE INPUT PARAMETER deliveryMode AS CHARACTER.
```

Parameters

- **topicName**
  
  The topic to which the message is published.

- **message**
  
  The Message object.

- **priority**
  
  The message priority (optional): 0–9. Session default is used if UNKNOWN.

- **timeToLive**
  
  Time to live, in milliseconds (optional). Session default is used if UNKNOWN.

- **deliveryMode**
  
  The delivery mode (optional): PERSISTENT, NON_PERSISTENT, NON_PERSISTENT_ASYNC, DISCARDABLE, or UNKNOWN (?). Session default is used if UNKNOWN.

Applies to

- Session objects

Notes

- If the publication is in reply to a received message, `topicName` can be the ReplyTo field obtained from the original message.
- This procedure executes remotely (sends a message to the OpenEdge Adapter for SonicMQ).

See also

- `cancelDurableSubscription procedure` on page 182, `publish procedure` on page 254, `subscribe procedure` on page 319
- For more information, see the Publishing messages to a topic, the Subscribing to a topic, the Durable subscriptions, and the Methods unique to Pub/Sub messaging.
- For an example, see the Pub/Sub messaging example.
**readBytesToRaw procedure**

Returns byte array data from the body of a StreamMessage or a BytesMessage.

**Syntax**

```
FUNCTION readBytesToRaw RETURNS RAW.
```

**Applies to**

Message objects

**Notes**

- It can be called in read-only mode to return the next byte segment in a BytesMessage.
- The size of all the byte segments other than the last one is 8192; the size of the last one is 8192 or less.

**See also**

createStreamMessage procedure on page 189, getContentType on page 205, writeBoolean procedure on page 321, writeByte procedure on page 321, writeBytesFromRaw procedure on page 322, writeChar procedure on page 323, writeDate procedure on page 323, writeDateTime procedure on page 324, writeDateTime-TZ procedure on page 325, writeDouble procedure on page 326, writeFloat procedure on page 326, writeInt procedure on page 327, writeLong procedure on page 328, writeLongString procedure on page 329, writeShort procedure on page 330, writeString procedure on page 330, endOfStream function on page 195, moveToNext procedure on page 253, readBytesToRaw procedure on page 256, readChar function on page 256, readDate function on page 257, readDateTime function on page 258, readDateTime-TZ function on page 258, readDecimal function on page 259, readInt function on page 259, readLogical function on page 261, readLongString function on page 261, readLongStringCP function on page 262

For more information, see the StreamMessage and the BytesMessage.

**readChar function**

Returns any message data segment except bytes data from the body of a StreamMessage.

**Syntax**

```
FUNCTION readChar RETURNS CHARACTER.
```

**Applies to**

Message objects

**Notes**

The size of all the character segments other than the last one is 8192; the size of the last one is 8192 or less.
See also
createStreamMessage procedure on page 189, writeBoolean procedure on page 321, writeByte procedure on page 321, writeBytesFromRaw procedure on page 322, writeChar procedure on page 323, writeDate procedure on page 323, writeDateTime procedure on page 324, writeDateTime-TZ procedure on page 325, writeDouble procedure on page 326, writeFloat procedure on page 326, writeInt procedure on page 327, writeLong procedure on page 328, writeLongString procedure on page 329, writeShort procedure on page 330, writeString procedure on page 330, endOfStream function on page 195, moveToNext procedure on page 253, readBytesToRaw procedure on page 256, readChar function on page 256, readDate function on page 257, readDateTime function on page 258, readDateTime-TZ function on page 258, readDecimal function on page 259, readInt function on page 259, readLogical function on page 261, readLongString function on page 261, readLongStringCP function on page 262
For more information, see the StreamMessage.

readDate function

Returns a DATE value with no time zone information from the body of a StreamMessage.

Syntax

```
FUNCTION readDate RETURNS DATE.
```

Applies to
Message objects

Notes

• Time information, if present, is removed.
• Time zone information, if present, is removed.
• If the application might receive messages originating in different time zones, the ABL programmer should ensure that date values are interpreted correctly.

See also
createStreamMessage procedure on page 189, writeBoolean procedure on page 321, writeByte procedure on page 321, writeBytesFromRaw procedure on page 322, writeChar procedure on page 323, writeDate procedure on page 323, writeDateTime procedure on page 324, writeDateTime-TZ procedure on page 325, writeDouble procedure on page 326, writeFloat procedure on page 326, writeInt procedure on page 327, writeLong procedure on page 328, writeLongString procedure on page 329, writeShort procedure on page 330, writeString procedure on page 330, endOfStream function on page 195, moveToNext procedure on page 253, readBytesToRaw procedure on page 256, readChar function on page 256, readDate function on page 257, readDateTime function on page 258, readDateTime-TZ function on page 258, readDecimal function on page 259, readInt function on page 259, readLogical function on page 261, readLongString function on page 261, readLongStringCP function on page 262
For more information, see the StreamMessage.
**readDateTime function**

Returns a DATETIME value with no time zone information from the body of a StreamMessage.

**Syntax**

```
FUNCTION readDateTime RETURNS DATETIME.
```

**Applies to**

Message objects

**Notes**

- Time zone information, if present, is removed.
- If time information is not present, the default time of 12:00AM (midnight) is added.
- If the application may receive messages originating in different time zones, the ABL programmer should ensure that date/time values are interpreted correctly.

**See also**

createStreamMessage procedure on page 189, writeBoolean procedure on page 321, writeByte procedure on page 321, writeBytesFromRaw procedure on page 322, writeChar procedure on page 323, writeDateTime procedure on page 324, writeDateTime-TZ procedure on page 325, writeDouble procedure on page 326, writeFloat procedure on page 326, writeInt procedure on page 327, writeLong procedure on page 328, writeLongString procedure on page 329, writeShort procedure on page 330, writeString procedure on page 330, endOfStream function on page 195, moveToNext procedure on page 253, readBytesToRaw procedure on page 256, readChar function on page 256, readDate function on page 257, readDateTime function on page 258, readDateTime-TZ function on page 258, readDecimal function on page 259, readInt function on page 259, readLogical function on page 261, readLongString function on page 261, readLongStringCP function on page 262

For more information, see the StreamMessage.

**readDateTime-TZ function**

Returns a DATETIME-TZ value from the body of a StreamMessage.

**Syntax**

```
FUNCTION readDateTimeTz RETURNS DATETIME-TZ.
```

**Applies to**

Message objects
Notes

• If time information is not present, the default time of 12:00AM (midnight) is added.
• If time zone information is not present, the default time zone of the client application is added.

See also
createStreamMessage procedure on page 189, writeBoolean procedure on page 321, writeByte procedure on page 321, writeBytesFromRaw procedure on page 322, writeChar procedure on page 323, writeDate procedure on page 323, writeDateTime procedure on page 324, writeDateTime-TZ procedure on page 325, writeDouble procedure on page 326, writeFloat procedure on page 326, writeInt procedure on page 327, writeLong procedure on page 328, writeLongString procedure on page 329, writeShort procedure on page 330, writeString procedure on page 330, endOfStream function on page 195, moveToNext procedure on page 253, readBytesToRaw procedure on page 256, readChar function on page 256, readDate function on page 257, readDateTime function on page 258, readDateTime-TZ function on page 258, readDecimal function on page 259, readInt function on page 259, readLogical function on page 261, readLongString function on page 261, readLongStringCP function on page 262

For more information, see the StreamMessage.

readDecimal function

Returns any numeric data from the body of a StreamMessage.

Syntax

```
FUNCTION readDecimal RETURNS DECIMAL.
```

Applies to
Message objects

See also
createStreamMessage procedure on page 189, writeBoolean procedure on page 321, writeByte procedure on page 321, writeBytesFromRaw procedure on page 322, writeChar procedure on page 323, writeDate procedure on page 323, writeDateTime procedure on page 324, writeDateTime-TZ procedure on page 325, writeDouble procedure on page 326, writeFloat procedure on page 326, writeInt procedure on page 327, writeLong procedure on page 328, writeLongString procedure on page 329, writeShort procedure on page 330, writeString procedure on page 330, endOfStream function on page 195, moveToNext procedure on page 253, readBytesToRaw procedure on page 256, readChar function on page 256, readDate function on page 257, readDateTime function on page 258, readDateTime-TZ function on page 258, readDecimal function on page 259, readInt function on page 259, readLogical function on page 261, readLongString function on page 261, readLongStringCP function on page 262

For more information, see the StreamMessage.

readInt function

Returns int, short, or byte data from the body of a StreamMessage.
Syntax

FUNCTION readInt RETURNS INTEGER.

Applies to
Message objects

See also
createStreamMessage procedure on page 189, writeBoolean procedure on page 321, writeByte procedure on page 321, writeBytesFromRaw procedure on page 322, writeChar procedure on page 323, writeDate procedure on page 323, writeDateTime procedure on page 324, writeDateTime-TZ procedure on page 325, writeDouble procedure on page 326, writeFloat procedure on page 326, writeInt procedure on page 327, writeInt64 procedure on page 328, writeLong procedure on page 328, writeLongString procedure on page 329, writeShort procedure on page 330, writeString procedure on page 330, eofStream function on page 195, moveToNext procedure on page 253, readBytesToRaw procedure on page 256, readChar function on page 256, readDate function on page 257, readDateTime function on page 258, readDateTime-TZ function on page 258, readDecimal function on page 259, readInt64 function on page 260, readLogical function on page 261, readLongString function on page 261, readLongStringCP function on page 262

For more information, see the StreamMessage.

readInt64 function

Returns INT64 data from the body of a StreamMessage.

Syntax

FUNCTION readInt64 RETURNS INT64.

Applies to
Message objects

See also
createStreamMessage procedure on page 189, writeBoolean procedure on page 321, writeByte procedure on page 321, writeBytesFromRaw procedure on page 322, writeChar procedure on page 323, writeDate procedure on page 323, writeDateTime procedure on page 324, writeDateTime-TZ procedure on page 325, writeDouble procedure on page 326, writeFloat procedure on page 326, writeInt procedure on page 327, writeInt64 procedure on page 328, writeLong procedure on page 328, writeLongString procedure on page 329, writeShort procedure on page 330, writeString procedure on page 330, eofStream function on page 195, moveToNext procedure on page 253, readBytesToRaw procedure on page 256, readChar function on page 256, readDate function on page 257, readDateTime function on page 258, readDateTime-TZ function on page 258, readDecimal function on page 259, readInt function on page 259, readLogical function on page 261, readLongString function on page 261, readLongStringCP function on page 262

For more information, see the StreamMessage.
readLogical function

Returns boolean data from the body of a StreamMessage.

Syntax

```plaintext
FUNCTION readLogical RETURNS LOGICAL.
```

Applies to

Message objects

See also

createStreamMessage procedure on page 189, writeBoolean procedure on page 321, writeByte procedure on page 321, writeBytesFromRaw procedure on page 322, writeChar procedure on page 323, writeDate procedure on page 323, writeDateTime procedure on page 324, writeDateTime-TZ procedure on page 325, writeDouble procedure on page 326, writeFloat procedure on page 326, writeInt procedure on page 327, writeLong procedure on page 328, writeLongString procedure on page 329, writeShort procedure on page 330, writeString procedure on page 330, endOfStream function on page 195, moveToNext procedure on page 253, readBytesToRaw procedure on page 256, readChar function on page 256, readDate function on page 257, readDateTime function on page 258, readDateTime-TZ function on page 258, readDecimal function on page 259, readInt function on page 259, readLogical function on page 261, readLongString function on page 261, readLongStringCP function on page 262

For more information, see the StreamMessage.

readLongString function

Returns String data from the body of a StreamMessage.

Syntax

```plaintext
FUNCTION readLongString RETURNS LONGCHAR.
```

Applies to

Message objects

Notes

- The text is converted to the current default code page of the OpenEdge client application.
- To return text based on a different code page, use readLongStringCP function on page 262.
readLongStringCP function

Returns LONGCHAR data from the body of a StreamMessage and converts it to the specified code page.

Syntax

```plaintext
FUNCTION readLongStringCP RETURNS LONGCHAR (code_page as CHARACTER).
```

Applies to

Message objects

Notes

The LONGCHAR data returned is converted to the code page specified by the code_page parameter.

See also

createStreamMessage procedure on page 189, writeBoolean procedure on page 321, writeByte procedure on page 321, writeBytesFromRaw procedure on page 322, writeChar procedure on page 323, writeDate procedure on page 323, writeDateTime procedure on page 324, writeDateTime-TZ procedure on page 325, writeDouble procedure on page 326, writeFloat procedure on page 326, writeInt procedure on page 327, writeLong procedure on page 328, writeLongString procedure on page 329, writeShort procedure on page 330, writeString procedure on page 330, endOfStream function on page 195, moveToNext procedure on page 253, readBytesToRaw procedure on page 256, readChar function on page 256, readDate function on page 257, readDateTime function on page 258, readDateTime-TZ function on page 258, readDecimal function on page 259, readInt function on page 259, readLogical function on page 261, readLongString function on page 261, readLongStringCP function on page 262

For more information, see the StreamMessage.

receiveFromQueue procedure

 Receives messages from a queue.
Syntax

```
PROCEDURE receiveFromQueue.
DEFINE INPUT PARAMETER queueName AS CHARACTER.
DEFINE INPUT PARAMETER messageSelector AS CHARACTER.
DEFINE INPUT PARAMETER messageConsumer AS HANDLE.
```

Parameters

`queueName`

The queue from which the messages are received.

`messageSelector`

A message selector. If UNKNOWN, receives all messages.

`messageConsumer`

A Message Consumer object, which handles the messages asynchronously.

Applies to

Session objects

Notes

- The messages are handled asynchronously by the `messageConsumer` procedure.
- This procedure executes remotely (sends a message to the OpenEdge Adapter for SonicMQ).

See also

- `browseQueue procedure` on page 181, `receiveFromQueue procedure` on page 262, `sendToQueue procedure` on page 266
- For more information, see the Sending messages to a queue, the Receiving messages from a queue, and the Methods unique to Point-to-Point messaging.
- For an example, see the PTP message example.

---

**recover procedure**

Redelivers all unacknowledged messages received up to that point in the current session.

Syntax

```
PROCEDURE recover.
```

Applies to

Session objects
Notes

• It is an error to call this method if the session is transacted for receiving. Call the rollbackReceive procedure on page 265 instead.

• This procedure executes remotely (sends a message to the OpenEdge Adapter for SonicMQ).

See also commitSend procedure on page 184, commitReceive procedure on page 183. rollbackSend procedure on page 266, rollbackReceive procedure on page 265, recover procedure on page 263
For more information, see the Transaction and recovery procedures.

requestReply procedure

Creates a temporary queue or topic and sets the JMSReplyTo message header field. Then requestReply procedure sends the message to the destination specified and designates the messageConsumer parameter for processing replies.

Syntax

```
PROCEDURE requestReply.
DEFINE INPUT PARAMETER destination AS CHARACTER.
DEFINE INPUT PARAMETER message AS HANDLE.
DEFINE INPUT PARAMETER replySelector AS CHARACTER. /*UNKNOWN means receiving all replies*/
DEFINE INPUT PARAMETER messageConsumer AS HANDLE. /*UNKNOWN is illegal*/
DEFINE INPUT PARAMETER priority AS INTEGER. /*Session default is used if UNKNOWN.*/
DEFINE INPUT PARAMETER timeToLive AS DECIMAL. /*Session default is used if UNKNOWN.*/
DEFINE INPUT PARAMETER deliveryMode AS CHARACTER. /*Session default is used if UNKNOWN.*/
```

Applies to

Session objects

Notes

• The term destination is used for both topics and queues.

• The ABL–JMS implementation automates the request/reply sequence:
  • Sending a reply by setting the reply OUTPUT parameter of the message handler
  • Requesting a reply by calling requestReply procedure on page 264 with a reply Message Consumer

• The ABL–JMS implementation uses a temporary destination for the reply. It is an error to set the JMSReplyTo field of the message explicitly if requestReply is used. The reply is received by messageConsumer asynchronously, just like any other message reception. The temporary destination is deleted when the Message Consumer object is deleted.

• This procedure executes remotely (sends a message to the OpenEdge Adapter for SonicMQ).
See also
For more information, see the Request/Reply.

reset procedure
Changes the mode of a message from write-only to read-only mode and positions the cursor before the first data item.

Syntax

```plaintext
PROCEDURE reset.
```

Applies to
Message objects

Notes
• Sending the message causes an implicit call to reset procedure on page 265.
• The message becomes read-only and arrives at the receiver in a reset state.

See also
For more information, see the TextMessage, the StreamMessage, and the BytesMessage.

rollbackReceive procedure
Starts redelivering the messages received up to that point in the current transaction.

Syntax

```plaintext
PROCEDURE rollbackReceive.
```

Applies to
Session objects

Notes
• Redelivers messages that have been received, but not acknowledged.
• It is an error to call this procedure in a Session object that is not transacted for receiving.
• This procedure executes remotely (sends a message to the OpenEdge Adapter for SonicMQ).
See also
commitSend procedure on page 184, commitReceive procedure on page 183, rollbackSend procedure on page 266, rollbackReceive procedure on page 265, recover procedure on page 263
For more information, see the Transaction and recovery procedures.

rollbackSend procedure
Discards all messages sent up to that point in the current transaction.

Syntax

PROCEDURE rollbackSend.

Applies to
Session objects

Notes
• It is an error to call this method in a Session object that is not transacted for sending.
• This procedure executes remotely (sends a message to the OpenEdge Adapter for SonicMQ).

See also
commitSend procedure on page 184, commitReceive procedure on page 183, rollbackSend procedure on page 266, rollbackReceive procedure on page 265, recover procedure on page 263
For more information, see the Message acknowledgement, forwarding, and recovery.

sendToQueue procedure
Sends a message to a queue.

Syntax

PROCEDURE sendToQueue.
DEFINE INPUT PARAMETER queueName AS CHARACTER.
DEFINE INPUT PARAMETER message AS HANDLE.
DEFINE INPUT PARAMETER priority AS INTEGER.
DEFINE INPUT PARAMETER timeToLive AS DECIMAL.
DEFINE INPUT PARAMETER deliveryMode AS CHARACTER.
Parameters

queueName

The queue to which the message is sent.

message

The Message object.

priority

The message priority (optional): 0–9. If Unknown value (?), the session default is used.

timeToLive

Time to live, in milliseconds (optional). If Unknown value (?), the session default is used.

deliveryMode

The delivery mode (optional): PERSISTENT, NON_PERSISTENT, NON_PERSISTENT_ASYNC, or UNKNOWN (?). If Unknown value (?), the session default is used.

Applies to

Session objects

Notes

• If the sending is in reply to a received message, queueName can be the ReplyTo field obtained from the original message.

• This procedure executes remotely (sends a message to the OpenEdge Adapter for SonicMQ).

See also

browseQueue procedure on page 181, receiveFromQueue procedure on page 262, sendToQueue procedure on page 266

For more information, see the Sending messages to a queue, the Receiving messages from a queue, and the Methods unique to Point-to-Point messaging.

For an example, see the PTP message example.

setAdapterService procedure

Specifies the service name under which the OpenEdge Adapter for SonicMQ BrokerConnect is registered with the NameServer.

Syntax

PROCEDURE setAdapterService.
DEFINE INPUT PARAMETER serviceName AS CHARACTER,
Applies to
Session objects (for BrokerConnect only)

Notes
• The default is adapter.progress.jms.
• If the OpenEdge Adapter for SonicMQ uses adapter.progress.jms, calling setAdapterService procedure on page 267 is unnecessary.
• If the application uses a -URL parameter to connect to the OpenEdge Adapter for SonicMQ, that parameter includes the service name; any subsequent calls to setAdapterService are ignored.

See also
setAdapterService procedure on page 267, getAdapterService function on page 196
For more information, see the Setting and getting JMS connection and session attributes.

setApplicationContext procedure
Passes context to the message handler.

Syntax

```
PROCEDURE setApplicationContext.
DEFINE INPUT PARAMETER handler AS HANDLE.
```

Applies to
Message Consumer objects

Notes
• The handler parameter is typically a handle to a persistent procedure implemented by the application.
• When the message handler is called, it gets that handler and uses it, for example, to deposit error information in the application's context by calling a specific handler's internal procedure.

See also
setApplicationContext procedure on page 268, getApplicationContext function on page 197
For more information, see the Accessing message handler information and the Creating a message handler process.

setBoolean procedure
Converts data to the JMS boolean data type in a MapMessage.
**Syntax**

```plaintext
PROCEDURE setBoolean.
DEFINE INPUT PARAMETER itemName AS CHARACTER.
DEFINE INPUT PARAMETER value AS LOGICAL.
```

**Applies to**
Message objects

**Notes**

An Unknown value (?) is considered FALSE.

**See also**

createMapMessage procedure on page 186, setBoolean procedure on page 268, setByte procedure on page 270, setBytesFromRaw procedure on page 271, setChar procedure on page 272, setDate procedure on page 277, setDateTime procedure on page 279, setDateTime-TZ procedure on page 280, setDouble procedure on page 283, setFloat procedure on page 287, setInt procedure on page 288, setLong procedure on page 296, setLongString procedure on page 298, setShort procedure on page 310, setString procedure on page 313, getMapNames function on page 230, getItemType function on page 217, getBytesToRaw function on page 200, getChar function on page 200, getDate function on page 207, getDateTime function on page 208, getDateTime-TZ function on page 209, getDecimal function on page 210, getInt function on page 215, getLogical function on page 227, getLongString function on page 228, getLongStringCP function on page 228

For more information, see the MapMessage.

---

**setBooleanProperty procedure**

Sets a boolean message property.

**Syntax**

```plaintext
PROCEDURE setBooleanProperty.
DEFINE INPUT PARAMETER propertyName AS CHARACTER.
DEFINE INPUT PARAMETER propertyValue AS LOGICAL.
```

**Applies to**
Message objects

**Notes**

An Unknown value (?) is considered a FALSE value.

**See also**

clearProperties procedure on page 183

For more information, see the Accessing message properties.
setBrokerURL procedure

Sets the value of the SonicMQ Broker URL.

Syntax

```
PROCEDURE setBrokerURL.
DEFINE INPUT PARAMETER brokerURL AS CHARACTER.
```

Parameter

brokerURL

The URL for the SonicMQ Broker.

Applies to

Session objects

Notes

• If set on the client, it overwrites the default broker URL property set on the OpenEdge Adapter for SonicMQ side.
• The creation of a session fails if no value is set on the client or at the OpenEdge Adapter for SonicMQ.

See also

setBrokerURL procedure on page 270, getBrokerURL function on page 197

For more information, see the Setting and getting JMS connection and session attributes.

setByte procedure

Converts data in a MapMessage to the JMS byte data type.

Syntax

```
PROCEDURE setByte.
DEFINE INPUT PARAMETER itemName AS CHARACTER.
DEFINE INPUT PARAMETER value AS INTEGER.
```

Applies to

Message objects

Notes

• Byte values range from –128 to 127.
The server returns a `NumberFormatException` message for a value overflow. For example, calling `setByte("item1", 1000)` results in a value overflow.

See also
- `createMapMessage` procedure on page 186
- `setBoolean` procedure on page 268
- `setByte` procedure on page 270
- `setBytesFromRaw` procedure on page 271
- `setChar` procedure on page 272
- `setDate` procedure on page 277
- `setDateTime` procedure on page 279
- `setDateTime-TZ` procedure on page 280
- `setDouble` procedure on page 283
- `setFloat` procedure on page 287
- `setInt` procedure on page 288
- `setLong` procedure on page 296
- `setLongString` procedure on page 298
- `setShort` procedure on page 310
- `setString` procedure on page 313
- `getMapNames` function on page 230
- `getItemType` function on page 217
- `getBytesToRaw` function on page 200
- `getChar` function on page 207
- `getDate` function on page 207
- `getDateTime` function on page 208
- `getDateTime-TZ` function on page 209
- `getDecimal` function on page 210
- `getInt` function on page 215
- `getLogical` function on page 227
- `getLongString` function on page 228
- `getLongStringCP` function on page 228

For more information, see the `MapMessage`.

### setByteProperty procedure

Sets a byte property in a message; the values range from –128 to 127.

**Syntax**

```plaintext
PROCEDURE setByteProperty.
  DEFINE INPUT PARAMETER propertyName AS CHARACTER.
  DEFINE INPUT PARAMETER propertyValue AS INTEGER.
```

**Applies to**

Message objects

**Notes**

The server returns a `NumberFormatException` message for a value overflow. For example, calling `setByteProperty("prop1", 1000)` results in a value overflow.

See also
- `clearProperties` procedure on page 183

For more information, see the Accessing message properties.

### setBytesFromRaw procedure

Converts data in a `MapMessage` to the JMS byte data type.
setBytesFromRaw procedure

Syntax

PROCEDURE setBytesFromRaw.
DEFINE INPUT PARAMETER itemName AS CHARACTER.
DEFINE INPUT PARAMETER values AS RAW.

Applies to
Message objects

See also
createMapMessage procedure on page 186, setBoolean procedure on page 268, setByte procedure on page 270, setBytesFromRaw procedure on page 271, setChar procedure on page 272, setDate procedure on page 277, setDateTime procedure on page 279, setDateTime-TZ procedure on page 280, setDouble procedure on page 283, setFloat procedure on page 287, setInt procedure on page 288, setLong procedure on page 296, setLongString procedure on page 298, setShort procedure on page 310, setString procedure on page 313, getMapNames function on page 230, getItemType function on page 217, getBytesToRaw function on page 200, getChar function on page 200, getDate function on page 207, getDateTime function on page 208, getDateTime-TZ function on page 209, getDecimal function on page 210, getInt function on page 215, getLogical function on page 227, getLongString function on page 228, getLongStringCP function on page 228

For more information, see the MapMessage.

setChar procedure

Converts data in a MapMessage to the JMS char data type.

Syntax

PROCEDURE setChar.
DEFINE INPUT PARAMETER itemName AS CHARACTER.
DEFINE INPUT PARAMETER value AS CHARACTER.

Applies to
Message objects

Notes
The number of characters in the char value must be one.
See also
createMapMessage procedure on page 186, setBoolean procedure on page 268, setByte procedure on page 270, setBytesFromRaw procedure on page 271, setChar procedure on page 272, setDateTime procedure on page 277, setDateTime-TZ procedure on page 280, setDouble procedure on page 283, setFloat procedure on page 287, setInt procedure on page 288, setLong procedure on page 296, setLongString procedure on page 298, setShort procedure on page 310, setString procedure on page 313, getMapNames function on page 230, getItemType function on page 217, getBytesToRaw function on page 200, getChar function on page 200, getDate function on page 207, getDateTime function on page 208, getDateTime-TZ function on page 209, getDecimal function on page 210, getInt function on page 215, getLogical function on page 227, getLongString function on page 228, getLongStringCP function on page 228

For more information, see the MapMessage.

setClientID procedure

Sets the client ID value for the SonicMQ Broker connection and overwrites the default client ID set on the server side.

Syntax

```plaintext
PROCEDURE setClientID.
DEFINE INPUT PARAMETER clientID AS CHARACTER.
```

Parameter

clientID

Client ID value for the SonicMQ Broker connection.

Applies to

Session objects

Notes

• A client ID is required for durable subscriptions and for client persistence.
• If called, setClientID procedure on page 273 overwrites the default client ID set on the server side.

See also

setClientID procedure on page 273, getClientID function on page 202

For more information, see the Setting and getting JMS connection and session attributes, the Methods unique to Pub/Sub messaging, and the Subscribing to a topic.

setClientPersistence procedure

Sets client persistence.
Note: The JMS providers other than SonicMQ do not support this method.

Syntax

```abl
PROCEDURE setClientPersistence.
DEFINE INPUT PARAMETER enabled AS LOGICAL.
```

Applies to
Session objects (ClientConnect and ServerConnect only).

Notes

- Only applicable for Client Persistence.
- You must call `setClientID` procedure on page 273 to use client persistence.
- If the default is `FALSE`, client persistence is not enabled.
- You cannot change client persistence for an active session.
- Call `setClientPersistence` procedure on page 273 prior to calling `beginSession` procedure on page 180.
- To change the value of client persistence, you must stop and restart the session for any changes to take effect.

See also

For more information, see the Client persistence and the Client persistence.

**setClientTransactionBufferSize procedure**

Sets the SonicMQ client buffer size in bytes for Fault Tolerant transacted messages in memory.

Syntax

```abl
PROCEDURE setClientTransactionBufferSize.
DEFINE INPUT PARAMETER sz AS INTEGER.
```

Applies to
Session objects (ClientConnect and ServerConnect only).
Notes

• Only applicable for Fault Tolerant connections.

• Call setClientTransactionBufferSize procedure on page 274 before beginSession procedure on page 180 is called.

• A value of 0 tells the SonicMQ client to use the default value as determined by the SonicMQ Broker. This value is the size of the buffer used by the SonicMQ Broker.

See also

setFaultTolerant procedure on page 285, getFaultTolerant function on page 213, isFaultTolerant function on page 251, createChangeStateConsumer procedure on page 185, setFaultTolerantReconnectTimeout procedure on page 286, getFaultTolerantReconnectTimeout function on page 214, setInitialConnectionTimeout procedure on page 290, getInitialConnectionTimeout function on page 217, setClientTransactionBufferSize procedure on page 274, getClientTransactionBufferSize function on page 203

For more information, see the Fault tolerance and the Fault tolerance.

setConnectID procedure

Sets the connection ID between the Sonic client and Sonic broker.

Syntax

PROCEDURE setConnectID.
DEFINE INPUT PARAMETER connectID AS CHARACTER.

Applies to

Session objects.

Notes

Sets the connection ID between the Sonic client and Sonic broker. If not explicitly set, Sonic uses an internal value.

See also

getConnectID function on page 204

setConnectionFile procedure

Sets the connection file.

Note: The JMS providers other than SonicMQ do not support this method.
Syntax

```
PROCEDURE setConnectionFile.
DEFINE INPUT PARAMETER filename AS CHARACTER.
```

Applies to
Session objects.

Notes
• This file contains all the serialized connection object information used to connect to a Sonic MQ Broker.
• Parameter values in the connection file are superseded by any parameter set through a call to the corresponding `set<parameter>` procedure.
• Connection parameter values are fixed when `beginSession procedure` on page 180 is called.

See also
For more information, see the Using serialized connection objects.

**setConnectionURLs procedure**

Specifies a list of broker URLs for the client to try to connect to.

**Note:** The JMS providers other than SonicMQ do not support this method.

Syntax

```
PROCEDURE setConnectionURLs.
DEFINE INPUT PARAMETER brokerList AS CHARACTER.
```

Parameter

brokerList

A comma-separated list of Sonic Broker URLs for the client to use for connecting.

Applies to
Session objects

Notes
• If `brokerList` is not set to the `Unknown` value (?), it overrides the URL specified by `setBrokerURL procedure` on page 270.
• Call this procedure instead of `setBrokerURL procedure` on page 270 when there is a list of broker URLs.
See also
setConnectionURLs procedure on page 276, getConnectionURLs function on page 205, setSequential procedure on page 310, getSequential function on page 242
For more information, see the Managing fail-over support.

**setDataSet procedure**

Sets the **DataSetMessage**.

**Syntax**

```plaintext
PROCEDURE setDataSet.
DEFINE INPUT PARAMETER dsHdl AS HANDLE.
DEFINE INPUT PARAMETER schemaLocation AS CHARACTER.
DEFINE INPUT PARAMETER writeSchema AS LOGICAL.
```

**Applies to**

Message objects

**Notes**

- The handle parameter must be a handle to a valid DataSet.
- The schema parameters specify the schema information and are passed directly to the WRITE-XML method. Specifying an Unknown value (?) for any of the schema parameters will result in the use of the default value for that parameter.
- The DataSetMessage is converted to an XMLMessage using the WRITE-XML method and setLongText procedure on page 298.

See also
createDataSetMessage procedure on page 186, setDataSet procedure on page 277, getDataSet function on page 206
For an example, see the DataSetMessage.

**setDate procedure**

Sets a date value as **String** data in a **MapMessage**.

**Syntax**

```plaintext
PROCEDURE setDate.
DEFINE INPUT PARAMETER itemname as CHARACTER.
DEFINE INPUT PARAMETER value as DATE.
```
Applies to
Message objects

Notes
• The procedure writes value as a DATETIME-TZ value, adding default time and time zone information:
  • A time of 12:00AM (midnight)
  • The default time zone of the client application

• If the message might be consumed by an application in a different time zone from that of the sending application, the ABL programmer must ensure that the receiving application can interpret the value correctly.

• The format of the DATETIME-TZ value is a subset of the ISO8601 format.

• An error is returned if the Unknown value (?) is specified.

See also
createMapMessage procedure on page 186, setBoolean procedure on page 268, setByte procedure on page 270, setBytesFromRaw procedure on page 271, setChar procedure on page 272, setDate procedure on page 277, setDateTime procedure on page 279, setDateTime-TZ procedure on page 280, setDouble procedure on page 283, setFloat procedure on page 287, setInt procedure on page 288, setLong procedure on page 296, setLongString procedure on page 298, setShort procedure on page 310, setString procedure on page 313, getMapNames function on page 230, getItemType function on page 217, getBytesToRaw function on page 200, getChar function on page 200, getDate function on page 207, getDateTime function on page 208, getDateTime-TZ function on page 209, getDecimal function on page 210, getInt function on page 215, getLogical function on page 227, getLongString function on page 228, getLongStringCP function on page 228
For more information, see the MapMessage.

setDateProperty procedure
Sets a date property in a message header.

Syntax

```
PROCEDURE setDateProperty.
DEFINE INPUT PARAMETER propertyName AS CHARACTER.
DEFINE INPUT PARAMETER propertyValue AS DATE.
```

Applies to
Message objects

Notes
• The date value is set as a String message property.
• The procedure writes propertyValue as a DATETIME-TZ value, adding default time and time zone information:
  • A time of 12:00AM (midnight)
• The default time zone of the client application

• The format of the DATETIME-TZ value is a subset of the ISO8601 format.

• An error is returned if the Unknown value (?) is specified.

• If the message might be consumed by a non-OpenEdge application in a different time zone from that of the sending application, the ABL programmer must ensure that the receiving application can interpret the value correctly. OpenEdge applications that receive the message correctly interpret the DATETIME-TZ value.

See also
setDateProperty procedure on page 278, getDateProperty function on page 207

For more information, see the Accessing message properties.

setDateTime procedure

Sets a date-time value as String data in a MapMessage.

Syntax

PROCEDURE setDateTime.
DEFINE INPUT PARAMETER itemName AS CHARACTER.
DEFINE INPUT PARAMETER value AS DATETIME.

Applies to

Message objects

Notes

• The date-time value is set as a String message property.

• The procedure writes value as a DATETIME-TZ value, adding default time zone information. The format of the DATETIME-TZ value is a subset of the ISO8601 format.

• If the message might be consumed by a non-OpenEdge application in a different time zone from that of the sending application, the ABL programmer must ensure that the receiving application can interpret the value correctly. OpenEdge applications that receive the message correctly interpret the DATETIME-TZ value.

• An error is returned if the Unknown value (?) is specified.

See also

creatMapMessage procedure on page 186, setBoolean procedure on page 268, setByte procedure on page 270, setBytesFromRaw procedure on page 271, setChar procedure on page 272, setDate procedure on page 277, setDateTime procedure on page 279, setDateTime-TZ procedure on page 280, setDouble procedure on page 283, setFloat procedure on page 287, setInt procedure on page 288, setLong procedure on page 296, setLongString procedure on page 298, setShort procedure on page 310, setString procedure on page 313, getMapNames function on page 230, getItemType function on page 217, getBytesToRaw function on page 200, getChar function on page 200, getDate function on page 207, getDateProperty function on page 208, getDateTime function on page 209, getDateTime-TZ function on page 209, getDecimal function on page 210, getInt function on page 215, getLogical function on page 227, getLongString function on page 228, getLongStringCP function on page 228
For more information, see the MapMessage.

**setDateTimeProperty procedure**

Sets a date-time value.

**Syntax**

PROCEDURE setDateTimeProperty.
DEFINE INPUT PARAMETER propertyName AS CHARACTER.
DEFINE INPUT PARAMETER propertyValue AS DATETIME.

**Applies to**

Message objects

**Notes**

- The date-time value is set as a String property in a message header.
- The procedure writes `value` as a DATETIME-TZ value, adding default time zone information.

**See also**

setDateTimeProperty procedure on page 280, getDateTimeProperty function on page 209

For more information, see the Accessing message properties.

**setDateTime-TZ procedure**

Sets a date-time value, including time zone information in a MapMessage.

**Syntax**

PROCEDURE setDateTime-TZ.
DEFINE INPUT PARAMETER itemName as CHARACTER.
DEFINE INPUT PARAMETER value as DATETIME-TZ

**Applies to**

Message objects

**Notes**

- The date-time value, including time zone information, is set as a String message property.
- An error is returned if the Unknown value (?) is specified.
If the message might be consumed by a non-OpenEdge application in a different time zone from that of the
sending application, the ABL programmer must ensure that the receiving application can interpret the value
correctly. OpenEdge applications that receive the message correctly interpret the \texttt{DATETIME-TZ} value.

\textbf{See also}
\item createMapMessage procedure on page 186, setBoolean procedure on page 268, setByte procedure on page 270, setBytesFromRaw procedure on page 271, setChar procedure on page 272, setDateTime procedure on page 277, setDateTimeTz procedure on page 279, setDateTime-TZ procedure on page 280, setDouble procedure on page 283, setFloat procedure on page 287, setInt procedure on page 288, setLong procedure on page 296, setLongString procedure on page 298, setShort procedure on page 310, setString procedure on page 313, getMapNames function on page 230, getItemType function on page 217, getBytesToRaw function on page 200, getChar function on page 200, getDateTime function on page 207, getDateTimeTz function on page 208, getDateTime-TZ function on page 209, getDecimal function on page 210, getInt function on page 215, getLogical function on page 227, getLongString function on page 228, getLongStringCP function on page 228

For more information, see the MapMessage.

\textbf{setDateTimeTzProperty procedure}

Sets a date-time value, including time zone information.

\textbf{Syntax}

\begin{verbatim}
PROCEDURE setDateTimeTzProperty.
DEFINE INPUT PARAMETER propertyName AS CHARACTER.
DEFINE INPUT PARAMETER propertyValue AS DATETIME-TZ.
\end{verbatim}

\textbf{Applies to}

Message objects

\textbf{Notes}

- The date-time value, including time zone information, is set as a \texttt{String} message property in the message header.
- The procedure writes value as a \texttt{DATETIME-TZ} value, adding default time and time zone information.

\textbf{See also}

setDateTimeTzProperty procedure on page 281, getDateTimeTzProperty function on page 210

For more information, see the Accessing message properties.

\textbf{setDefaultPersistency procedure}

Sets the default message persistency value for all messages sent in that session.
Syntax

```
PROCEDURE setDefaultPersistency.
DEFINE INPUT PARAMETER deliveryMode AS CHARACTER.
```

Applies to
Session objects

Notes

• Possible values include: PERSISTENT, NON_PERSISTENT, NON_PERSISTENT_ASYNC, DISCARDABLE, and UNKNOWN (?).
• The default value is PERSISTENT.
• A call with an Unknown value (?) has no effect.
• Use DISCARDABLE only when publishing to a topic. If DISCARDABLE is used when publishing other than to a topic, an error is raised.
• The evaluation is case-insensitive.
• NON_PERSISTENT_ASYNC is a SonicMQ extension of the JMS specification.

See also
setDefaultPersistency procedure on page 281, getDefaultPersistency function on page 211, setDefaultPriority procedure on page 282, getDefaultPriority function on page 212, setDefaultTimeToLive procedure on page 283, getDefaultTimeToLive function on page 212

For more information, see the Accessing message delivery parameters.

**setDefaultPriority procedure**

Sets the default message priority for all messages sent in that session.

Syntax

```
PROCEDURE setDefaultPriority.
DEFINE INPUT PARAMETER priority AS INTEGER.
```

Applies to
Session objects

Notes

• The range of priority values is 0–9. The default is 4.
• Setting an Unknown value (?) has no effect.
setDefaultTimeToLive procedure

Sets the default time to live, the number of milliseconds from the time a message is sent to the time the SonicMQ Broker can delete the message from the system.

Syntax

```
PROCEDURE setDefaultTimeToLive.
DEFINE INPUT PARAMETER millis AS DECIMAL.
```

Applies to

Session objects

Notes

• A setting of 0 specifies that the message never expires.
• The default is JMS-broker-dependent; the SonicMQ default value is 0.
• Any fractional part of the decimal value is truncated.
• If the value does not fit in a Java long value, Java rules for decimal-to-long conversions are used.
• Setting an Unknown value (?) has no effect.

See also

setDefaultPersistency procedure on page 281, getDefaultPersistency function on page 211, setDefaultPriority procedure on page 282, getDefaultPriority function on page 212, setDefaultTimeToLive procedure on page 283, getDefaultTimeToLive function on page 212

For more information, see the Accessing message delivery parameters.

setDouble procedure

Converts data in a MapMessage to the JMS double data type.

Syntax

```
PROCEDURE setDouble.
DEFINE INPUT PARAMETER itemName AS CHARACTER.
DEFINE INPUT PARAMETER value AS DECIMAL.
```
setDoubleProperty procedure

Sets a double message property.

Syntax

```abl
PROCEDURE setDoubleProperty.
DEFINE INPUT PARAMETER propertyName AS CHARACTER.
DEFINE INPUT PARAMETER propertyValue AS DECIMAL.
```

Applies to
Message objects

See also
createMapMessage procedure on page 186, setBoolean procedure on page 268, setByte procedure on page 270, setBytesFromRaw procedure on page 271, setChar procedure on page 272, setDateTime procedure on page 277, setDateTime-TZ procedure on page 280, setDouble procedure on page 283, setFloat procedure on page 287, setInt procedure on page 288, setLong procedure on page 296, setLongString procedure on page 298, setShort procedure on page 310, setString procedure on page 313, getMapNames function on page 230, getItemType function on page 217, getBytesToRaw function on page 200, getChar function on page 200, getDate function on page 207, getDateTime function on page 208, getDateTime-TZ function on page 209, getDecimal function on page 210, getInt function on page 215, getLogical function on page 227, getLongString function on page 228, getLongStringCP function on page 228

For more information, see the MapMessage.

setErrorHandler procedure

Sets the error-handling procedure.

Syntax

```abl
PROCEDURE setErrorHandler.
DEFINE INPUT PARAMETER messageConsumer AS HANDLE.
```
Applies to
Session objects

Notes

• The application must create the error-handling messageConsumer object and call setErrorHandler procedure on page 284 after calling beginSession procedure on page 180. If the application does not call setErrorHandler procedure on page 284, a default error handler displays the error message and the properties in alert boxes.

• The application should handle asynchronously reported error conditions programmatically by creating an error-handling Message Consumer object and passing it to setErrorHandler procedure on page 284 in the Session object. setErrorHandler procedure on page 284 creates an error-handling Message Consumer object.

• Asynchronous conditions are always reported as a TextMessage with several possible CHARACTER message properties. The CHARACTER properties that might be included in the message header are: exception, errorCode, linkedException-1, linkedException-2 ... linkedException-n (where n is a number of additional exceptions linked to the main exception).

• getPropertyNames function on page 235 can be used to get the list of properties in the error message header.

See also
setErrorHandler procedure on page 284, setNoErrorDisplay procedure on page 300

For more information, see the Error and condition handling. For an example, see Messaging Examples on page 333.

setFaultTolerant procedure

Enables or disables Fault Tolerance for the session.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

PROCEDURE setFaultTolerant.
DEFINE INPUT PARAMETER enable AS LOGICAL.

Applies to
Session objects (ClientConnect and ServerConnect only)

Notes

• Only applicable for Fault Tolerant connections.

• A value of TRUE will enable fault-tolerance and a value of FALSE will disable it. If default is FALSE, fault-tolerance is not enabled.

• This procedure must be called before beginSession procedure on page 180 is called.

• You cannot change fault-tolerance for an active session. You must stop and restart the session for any changes to have an effect.
• The application must also call setConnectionURLs procedure on page 276 to provide a list of broker URLs to be connected to when the current connection fails. SonicMQ will connect to the URLs in the order they are listed, starting at the beginning of the list.

• setSequential procedure on page 310 may be called to connect to the urls in the order they are listed starting at a random place in the list. See the SonicMQ documentation for details.

See also
setFaultTolerant procedure on page 285, getFaultTolerant function on page 213, isFaultTolerant function on page 251, createChangeStateConsumer procedure on page 185, setFaultTolerantReconnectTimeout procedure on page 286, getFaultTolerantReconnectTimeout function on page 214, setInitialConnectionTimeout procedure on page 290, getInitialConnectionTimeout function on page 217, setClientTransactionBufferSize procedure on page 274, get_clientTransactionBufferSize function on page 203

For more information, see the Fault tolerance and the Fault tolerance.

setFaultTolerantReconnectTimeout procedure

Sets a time limit in seconds on reconnection attempts.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

| PROCEDURE setFaultTolerantReconnectTimeout.  
| DEFINE INPUT PARAMETER seconds AS INTEGER. |

Applies to

Session objects (ClientConnect and ServerConnect only).

Notes

• Only applicable for Fault Tolerant connections.

• The default is 60 seconds. A timeout value of 0 indicates no timeout (that is, the reconnect will be attempted indefinitely).

• This procedure must be called before beginSession procedure on page 180 is called.

• The URLs specified in setConnectionURLs procedure on page 276 are used to attempt reconnection to a SonicMQ Broker.

See also
setFaultTolerant procedure on page 285, getFaultTolerant function on page 213, isFaultTolerant function on page 251, createChangeStateConsumer procedure on page 185, setFaultTolerantReconnectTimeout procedure on page 286, getFaultTolerantReconnectTimeout function on page 214, setInitialConnectionTimeout procedure on page 290, getInitialConnectionTimeout function on page 217, setClientTransactionBufferSize procedure on page 274, get_clientTransactionBufferSize function on page 203

For more information, see the Fault tolerance and the Fault tolerance.
**setFloat procedure**

Converts data in a MapMessage to the JMS float data type.

**Syntax**

```plaintext
PROCEDURE setFloat.
DEFINE INPUT PARAMETER itemName AS CHARACTER.
DEFINE INPUT PARAMETER value AS DECIMAL.
```

**Applies to**

Message objects

**See also**

createMapMessage procedure on page 186, setBoolean procedure on page 268, setByte procedure on page 270, setBytesFromRaw procedure on page 271, setChar procedure on page 272, setDate procedure on page 277, setDateTime procedure on page 279, setDateTime-TZ procedure on page 280, setDouble procedure on page 283, setFloat procedure on page 287, setInt procedure on page 288, setLong procedure on page 296, setLongString procedure on page 298, setShort procedure on page 310, setString procedure on page 313, getMapNames function on page 230, getItemType function on page 217, getBytesToRaw function on page 200, getChar function on page 200, getDate function on page 207, getDateTime function on page 208, getDateTime-TZ function on page 209, getDecimal function on page 210, getBytesToRaw function on page 200, getLogical function on page 227, getLongString function on page 228, getLongStringCP function on page 228

For more information, see the MapMessage.

**setFloatProperty procedure**

Sets a float message property.

**Syntax**

```plaintext
PROCEDURE setFloatProperty.
DEFINE INPUT PARAMETER propertyName AS CHARACTER.
DEFINE INPUT PARAMETER propertyValue AS DECIMAL.
```

**Applies to**

Message objects

**See also**

clearProperties procedure on page 183

For more information, see the Accessing message properties.
setFlowToDisk procedure

Enables or disables the SonicMQ flow-to-disk functionality.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

```abl
PROCEDURE setFlowToDisk.
   DEFINE INPUT PARAMETER val AS INTEGER NO-UNDO.
```

Applies to
Session objects

Notes
• When enabled, the SonicMQ broker saves messages to disk if the client is blocked and cannot receive the messages.
• You can set up the SonicMQ broker to have this feature on or off by default.
• The input parameter can have the following possible values:
  • 0 specifies to use the broker setting.
  • 1 turns on flow-to-disk.
  • 2 turns off flow-to-disk.
• This procedure can only be called before the session is started with beginSession procedure on page 180.
• The SonicMQ broker only uses the flow-to-disk feature for pub/sub messages.

See also
getFlowToDisk function on page 214

For more information on this SonicMQ feature, see the "Flow to Disk" section in the "SonicMQ Client Sessions" chapter of the SonicMQ Application Programming Guide.

setInt procedure

Converts data in a MapMessage to the JMS int data type.

Syntax

```abl
PROCEDURE setInt.
   DEFINE INPUT PARAMETER itemName AS CHARACTER.
   DEFINE INPUT PARAMETER value AS INTEGER.
```
**setIntProperty procedure**

Converts an int message property.

**Syntax**

```plaintext
PROCEDURE setIntProperty.
DEFINE INPUT PARAMETER propertyName AS CHARACTER.
DEFINE INPUT PARAMETER propertyValue AS INTEGER.
```

**Applies to**
Message objects

**See also**
getIntProperty function on page 215, clearProperties procedure on page 183, setInt64Property procedure on page 290

For more information, see the Accessing message properties.

**setInt64 procedure**

Converts data in a MapMessage to the JMS long data type.

**Syntax**

```plaintext
PROCEDURE setInt64.
DEFINE INPUT PARAMETER itemName AS CHARACTER.
DEFINE INPUT PARAMETER value AS INT64.
```
Applies to
Message objects

Notes
The Unknown value (?) is allowed.

See also
createMapMessage procedure on page 186, setBoolean procedure on page 268, setByte procedure on page 270, setBytesFromRaw procedure on page 271, setChar procedure on page 272, setDate procedure on page 277, setDateTime procedure on page 279, setDateTime-TZ procedure on page 280, setDouble procedure on page 283, setFloat procedure on page 287, setInt procedure on page 288, setLong procedure on page 296, setLongString procedure on page 298, setShort procedure on page 310, setString procedure on page 313, getMapNames function on page 230, getItemType function on page 217, getBytesToRaw function on page 200, getChar function on page 200, getDate function on page 207, getDateTime function on page 208, getDateTime-TZ function on page 209, getDecimal function on page 210, getInt function on page 215, getInt64 function on page 216, getLogical function on page 227, getLongString function on page 228, getLongStringCP function on page 228
For more information, see the MapMessage.

**setInt64Property procedure**

Sets the value of a JMS message property to INT64.

**Syntax**

```abl
PROCEDURE setInt64Property.
DEFINE INPUT PARAMETER propertyName AS CHARACTER.
DEFINE INPUT PARAMETER propertyValue AS INT64.
```

Applies to
Message objects

Notes
The Unknown value (?) is allowed.

See also
getInt64Property function on page 216, setIntProperty procedure on page 289, getIntProperty function on page 215, clearProperties procedure on page 183
For more information, see the Accessing message properties.

**setInitialConnectionTimeout procedure**

Sets a time limit in seconds on the initial Fault tolerant connection attempt to the SonicMQ Broker.
Note: The JMS providers other than SonicMQ do not support this method.

Syntax

```
PROCEDURE setInitialConnectionTimeout.
DEFINE INPUT PARAMETER seconds AS INTEGER.
```

Applies to

Message objects

Notes

- Only for Fault Tolerant connections.
- The default is 30 seconds. The timeout value specifies the timeout in seconds. A timeout value of 0 indicates no timeout (that is, the connect will be attempted indefinitely).
- This procedure must be called before beginSession procedure on page 180 is called.
- The URLs specified in setConnectionURLs procedure on page 276 are used to attempt to connect to a SonicMQ Broker.
- If the time limit is reached without being able to connect, the application calls the Adapter error handler.

See also

setFaultTolerant procedure on page 285, getFaultTolerant function on page 213, isFaultTolerant function on page 251, createChangeStateConsumer procedure on page 185, setFaultTolerantReconnectTimeout procedure on page 286, getFaultTolerantReconnectTimeout function on page 214, setInitialConnectionTimeout procedure on page 290, getInitialConnectionTimeout function on page 217, setClientTransactionBufferSize procedure on page 274, getClientTransactionBufferSize function on page 203

For more information, see the Fault tolerance and the Fault tolerance.

setJMSCorrelationID procedure

Sets the correlation ID.

Syntax

```
PROCEDURE setJMSCorrelationID
DEFINE INPUT PARAMETER correlationID AS CHARACTER.
```

Applies to

Message objects

Notes

This value is application-defined; typically it is set to the ID of the message replied to.
See also
setJMSCorrelationIDAsBytes procedure on page 292, getJMSCorrelationIDAsBytes function on page 219
For more information, see the Accessing message header properties.

setJMSCorrelationIDAsBytes procedure
Sets the bytes correlation ID, a proprietary (JMS-provider-dependent) value.

Syntax

```abl
PROCEDURE setJMSCorrelationIDAsBytes
DEFINE INPUT PARAMETER bytesCorrelationID AS RAW.
```

Applies to
Message objects

Notes
• The bytes correlation ID usage is proprietary (JMS-provider-dependent).
• When accessing SonicMQ, the `bytesCorrelationID` field can be used for storing application-defined values.

See also
setJMSCorrelationIDAsBytes procedure on page 292, getJMSCorrelationIDAsBytes function on page 219
For more information, see the Accessing message header properties.

setJMSReplyTo procedure
Sets a destination for replies.

Syntax

```abl
PROCEDURE setJMSReplyTo
DEFINE INPUT PARAMETER destination AS CHARACTER.
```

Applies to
Message objects

Notes
• The destination can be a name of a queue if the message is sent by a Pub/Sub session.
• The destination can be the name of the topic if the message is sent by a PTP session.
setReplyToDestinationType procedure on page 308 must be called to set the correct destination type.

See also
setJMSReplyTo procedure on page 292, getJMSReplyTo function on page 222, hasReplyTo function on page 249, setReplyToDestinationType procedure on page 308, getReplyToDestinationType function on page 240

For more information, see the Accessing message header properties.

setJMSServerName procedure

Specifies the JMS broker implementation, SonicMQ.

Syntax

```plaintext
PROCEDURE setJmsServerName.
DEFINE INPUT PARAMETER jmsServerName AS CHARACTER.
```

Parameter

`jmsServerName`

Specifies the JMS broker implementation, SonicMQ.

Applies to

Session objects

Notes

If set on the client, it overwrites the `jmsServerName` property set on the OpenEdge Adapter for SonicMQ side.

See also

setJMSServerName procedure on page 293, getJMSServerName function on page 223

For more information, see the Setting and getting JMS connection and session attributes.

setJMSType procedure

Sets the type name, which is proprietary (JMS-provider-dependent).

Syntax

```plaintext
PROCEDURE setJMSType
DEFINE INPUT PARAMETER typeName AS CHARACTER.
```
Applies to
Message objects

Notes
When accessing SonicMQ, the JMSType field can be used for storing application-defined values.

See also
setJMSType procedure on page 293, getJMSType function on page 224
For more information, see the Accessing message header properties.

setLoadBalancing procedure

Turns client-side load balancing on or off.

Syntax

```
PROCEDURE setLoadBalancing
DEFINE INPUT PARAMETER loadBalancing AS LOGICAL.
```

Applies to
Session objects

Notes
• If client-side load balancing is turned on, the client allows redirection to another SonicMQ Broker in the cluster.
• If client-side load balancing is turned off, the client does not allow redirection.
• If beginSession procedure on page 180 has already been called, an error is raised.

See also
setLoadBalancing procedure on page 294, getLoadBalancing function on page 224
For more information, see the Load balancing.

setLocalStoreDirectory procedure

Sets the directory that will be used by the adapter to persist messages.

Note: The JMS providers other than SonicMQ do not support this method.
Syntax

PROCEDURE setLocalStoreDirectory
DEFINE INPUT PARAMETER localStoreDir AS CHARACTER.

Applies to
Session objects (ClientConnect and ServerConnect only).

Notes
• Only applicable for Client Persistence.
• The default is the current working directory.
• The directory used will be appended with the client ID that is set to ensure uniqueness when operating in MQ-ServerConnect mode.
• Errors for this call will occur after the call to beginSession procedure on page 180.

See also

For more information, see the Client persistence and the Client persistence.

setLocalStoreSize procedure

Sets the maximum size of the local store in kilobytes.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

PROCEDURE setLocalStoreSize
DEFINE INPUT PARAMETER storesize AS INTEGER.

Applies to
Session objects (ClientConnect and ServerConnect only).

Notes
• Only applicable for Client Persistence.
• The default size is 1000 (10MB).
Errors for this call will occur after the call to `beginSession procedure` on page 180.

See also

For more information, see the Client persistence and the Client persistence.

### setLocalStoreWaitTime procedure

Sets the interval in seconds before Client Persistence begins.

**Note:** The JMS providers other than SonicMQ do not support this method.

**Syntax**

```abl
PROCEDURE setLocalStoreWait.
DEFINE INPUT PARAMETER interval AS INTEGER.
```

**Applies to**

Session objects (ClientConnect and ServerConnect only).

**Notes**

- Only applicable for Client Persistence.
- The default is 5, which means the adapter will wait 5 seconds then persist messages to disk.
- Errors for this call will occur after the call to `beginSession procedure` on page 180.

See also
`createRejectedMessageConsumer procedure` on page 188, `setClientPersistence procedure` on page 273, `getClientPersistence function` on page 202, `setLocalStoreDirectory procedure` on page 294, `getLocalStoreDirectory function` on page 225, `setLocalStoreSize procedure` on page 295, `getLocalStoreSize function` on page 225, `setLocalStoreWaitTime function` on page 226, `setReconnectTimeout procedure` on page 304, `getReconnectTimeout function` on page 237, `setReconnectInterval procedure` on page 304, `getReconnectInterval function` on page 236

For more information, see the Client persistence and the Client persistence.

### setLong procedure

Converts `long` data in a `MapMessage` in Text and XML messages.
**setLongProperty procedure**

Sets a long message property.

### Syntax

```
PROCEDURE setLongProperty.
DEFINE INPUT PARAMETER propertyName AS CHARACTER.
DEFINE INPUT PARAMETER propertyValue AS DECIMAL.
```

### Applies to
Message objects

### Notes
Any fractional part of the DECIMAL value is truncated.

### See also
- `clearProperties procedure` on page 183
- For more information, see the Accessing message properties.
setLongString procedure

Sets String data in a MapMessage.

Syntax

```abl
PROCEDURE setLongString.
DEFINE INPUT PARAMETER itemName AS CHARACTER.
DEFINE INPUT PARAMETER value AS LONGCHAR.
```

Applies to
Message objects

See also
createMapMessage procedure on page 186, setBoolean procedure on page 268, setByte procedure on page 270, setBytesFromRaw procedure on page 271, setChar procedure on page 272, setDate procedure on page 277, setDateTime procedure on page 279, setDateTime-TZ procedure on page 280, setDouble procedure on page 283, setFloat procedure on page 287, setInt procedure on page 288, setLong procedure on page 296, setLongString procedure on page 298, setShort procedure on page 310, setString procedure on page 313, getMapNames function on page 230, getItemType function on page 217, getBytesToRaw function on page 200, getChar function on page 200, getDate function on page 207, getDateTime function on page 208, getDateTime-TZ function on page 209, getDecimal function on page 210, getInt function on page 215, getLogical function on page 227, getLongString function on page 228, getLongStringCP function on page 228

For more information, see the MapMessage.

setLongText procedure

Clears the message body and sets a new text or XML value of any length.

Syntax

```abl
PROCEDURE setLongText.
DEFINE INPUT PARAMETER textValue AS LONGCHAR.
```

Applies to
Message objects

Notes
An error is returned if the Unknown value (?) is specified.
setMemptr procedure

Sets the specified number of bytes from the MEMPTR variable starting at startIndex in a BytesMessage.

Syntax

```
PROCEDURE setMemptr.
DEFINE INPUT PARAMETER memptrVar AS MEMPTR.
DEFINE INPUT PARAMETER startIndex AS INTEGER.
DEFINE INPUT PARAMETER numBytes AS INTEGER.
```

Applies to

Message objects

Notes

• The first byte is 1.

• setMemptr procedure on page 299 implicitly calls clearBody procedure on page 182 before setting the data and resets after setting the data. Therefore, it can be used whether the message is in a read-only mode or a write-only mode prior to the call.

• The call makes a copy of the data. Thus, the memptrVar variable is not modified by the ABL–JMS implementation and can be modified by the OpenEdge application after the call without corrupting the message.

See also

createBytesMessage procedure on page 184, setMemptr procedure on page 299, writeBoolean procedure on page 321, writeByte procedure on page 321, writeBytesFromRaw procedure on page 322, writeChar procedure on page 323, writeDate procedure on page 323, writeDateTime procedure on page 324, writeDateTime-TZ procedure on page 325, writeDouble procedure on page 326, writeFloat procedure on page 326, writeInt procedure on page 327, writeLong procedure on page 328, writeLongString procedure on page 329, writeShort procedure on page 330, writeString procedure on page 330, getContentType on page 205, getBytesCount function on page 198, endOfStream function on page 195, moveToNext procedure on page 253, readBytesToRaw procedure on page 256, readChar function on page 256, readDate function on page 257, readDateTime function on page 258, readDateTime-TZ function on page 258, readDecimal function on page 259, readInt function on page 259, readLogical function on page 261, readLongString function on page 261, readLongStringCP function on page 262, getMemptr function on page 230

For more information, see the BytesMessage.

For an example, see the Publishing, subscribing, and receiving an XML document in a BytesMessage.
setNoAcknowledge procedure

Instructs the ABL–JMS implementation not to acknowledge this message.

Syntax

```abl
PROCEDURE setNoAcknowledge.
```

Applies to
Message Consumer objects

Notes

• This call should be made if the OpenEdge application fails to use the data in a message and must receive the message again.
• This call is an error if the session is transacted for receiving.
• If the Message Consumer object is used to handle error messages or for queue browsing, this call has no effect.

See also
acknowledgeAndForward procedure on page 177, setSingleMessageAcknowledgement procedure on page 312, getSingleMessageAcknowledgement function on page 243, setNoAcknowledge procedure on page 300, getNoAcknowledge function on page 233

For more information, see the Message acknowledgement, forwarding, and recovery and the Single-message acknowledgement.

setNoErrorDisplay procedure

Turns the automatic display of synchronous errors and conditions on and off.

Syntax

```abl
PROCEDURE setNoErrorDisplay.
DEFINE INPUT PARAMETER noDisplay AS LOGICAL.
```

Applies to
Session objects and Message objects

Notes

• The default value is **FALSE**. The ABL–JMS implementation automatically displays synchronously reported errors and conditions in alert boxes.
• If set to TRUE, synchronous errors and conditions are not automatically displayed by the ABL–JMS implementation.

• Messages inherit the noDisplay property from the session that created them.

• After the message is created, setNoErrorDisplay procedure on page 300 must be called in the message itself to change the noDisplay property.

• Errors caused by method calls are automatically displayed.

See also
setErrorHandler procedure on page 284, setNoErrorDisplay procedure on page 300
For more information, see the Error and condition handling.

setPassword procedure

Sets the password value for the SonicMQ Broker login and overwrites the default password property set on the OpenEdge Adapter for SonicMQ side.

Syntax

PROCEDURE setPassword.
DEFINE INPUT PARAMETER password AS CHARACTER.

Parameter

password

Password value for the SonicMQ Broker login.

Applies to

Session objects

Notes

If called, setPassword procedure on page 301 overwrites the default password property set on the OpenEdge Adapter for SonicMQ side.

See also

getPassword on page 234
For more information, see the Setting and getting JMS connection and session attributes.

setPingInterval procedure

Specifies the interval in seconds for the JMS Adapter to actively ping the SonicMQ Broker so communication failure can be detected promptly.
Note: The JMS providers other than SonicMQ do not support this method.

Syntax

```abl
PROCEDURE setPingInterval.
DEFINE INPUT PARAMETER pingInterval AS INTEGER.
```

Parameter

`pingInterval`

The interval (in seconds).

Applies to

Session objects

Notes

- No pinging is performed by default.
- The `setPingInterval` functionality is a SonicMQ extension (see *SonicMQ Programming Guide*).
- A `pingInterval` value can also be specified in the `ubroker.properties` file for all clients by using the `srvrStartupParam` property of the OpenEdge Adapter for SonicMQ, as shown:

  ```
srvrStartupParam=pingInterval=3
  ```

- The `setPingInterval` procedure must be called before `beginSession` procedure on page 180 is called.

See also

For more information, see the Setting and getting JMS connection and session attributes, and also see *SonicMQ Programming Guide*.

**setPrefetchCount procedure**

Sets the number of messages a SonicMQ client can retrieve in a single operation from a queue containing multiple messages.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

```abl
PROCEDURE setPrefetchCount.
DEFINE INPUT PARAMETER count AS INTEGER.
```
Applies to
Session objects

Notes
• The default is 3. For example, a count of 3 means that the Sonic client retrieves up to three messages from a queue.
• If this procedure is called after beginSession procedure on page 180 is called, an error is raised.

See also
setPrefetchCount procedure on page 302, setPrefetchThreshold procedure on page 303
For more information, see the Controlling flow of messages.

setPrefetchThreshold procedure
Determines when the SonicMQ client goes back to the broker for more messages.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

PROCEDURE setPrefetchThreshold.
DEFINE INPUT PARAMETER threshold AS INTEGER.

Parameter

threshold

Prefetch threshold.

Applies to
Session objects for ptpsession

Notes
• The default is 1. For example, a threshold value of 1 means that Sonic does not go back to the broker for more messages until the last message has been delivered.
• If this procedure is called after beginSession procedure on page 180 is called, an error is raised.

See also
setPrefetchCount procedure on page 302, setPrefetchThreshold procedure on page 303
For more information, see the Controlling flow of messages.
setReconnectInterval procedure

Sets the interval in seconds between reconnect attempts.

Syntax

PROCEDURE setReconnectInterval.
DEFINE INPUT PARAMETER interval AS INTEGER.

Applies to
Session objects (ClientConnect and ServerConnect only)

Note: The JMS providers other than SonicMQ do not support this method.

Notes
• Only applicable for Client Persistence.
• The default is 30, which means the adapter will attempt to reconnect to an MQ Broker every 30 seconds.
• Errors for this call will occur after the call to beginSession procedure on page 180.

See also
createRejectedMessageConsumer procedure on page 188, setClientPersistence procedure on page 273, getLocalStoreDirectory function on page 225, setLocalStoreWaitTime procedure on page 296, getLocalStoreWaitTime function on page 226, setReconnectInterval procedure on page 304, getReconnectInterval function on page 236
For more information, see the Client persistence and the Client persistence.

setReconnectTimeout procedure

Sets the maximum amount of time in minutes that the client will attempt to reconnect to a broker.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

PROCEDURE setReconnectTimeout.
DEFINE INPUT PARAMETER timeout AS INTEGER.
Applies to
Session objects (ClientConnect and ServerConnect only)

Notes
• Only applicable for Client Persistence.
• The default is 0 meaning there is no timeout.
• If the value is set, an asynchronous error will be sent to the OpenEdge application after the timeout has expired and no further reconnects will be attempted.
• Errors for this call will occur after the call to beginSession procedure on page 180.

See also

For more information, see the Client persistence and the Client persistence.

setReplyAutoDelete procedure
Specifies whether all reply messages are to be automatically deleted.

Syntax

```
PROCEDURE setReplyAutoDelete.
DEFINE INPUT PARAMETER val AS LOGICAL.
```

Applies to
Message Consumer objects

Notes
• The default value is FALSE.
• If the reply property val is set to TRUE, all reply messages returned through the message handler's OUTPUT parameter are automatically deleted after being sent.

See also

For more information, see the Accessing message handler information and the Setting reply properties.
setReplyPersistency procedure

Sets the value for message persistency when the Message Consumer is passed to requestReply procedure on page 264.

Syntax

```abl
PROCEDURE setReplyPersistency.
DEFINE INPUT PARAMETER deliveryMode AS CHARACTER.
```

Applies to

Message Consumer objects

Notes

- The values are: PERSISTENT, NON_PERSISTENT, NON_PERSISTENT_ASYNC, and UNKNOWN. The default value is PERSISTENT.
- The evaluation is case-insensitive. A call with an Unknown value (?) has no effect.
- The replyPersistency value can be set only once.
- NON_PERSISTENT_ASYNC is a SonicMQ extension.

See also


For more information, see the Accessing message handler information and the Setting reply properties.

setReplyPriority procedure

Sets the priority of the reply messages when the Message Consumer is passed to requestReply procedure on page 264.

Syntax

```abl
PROCEDURE setReplyPriority.
DEFINE INPUT PARAMETER priority AS INTEGER.
```

Applies to

Message Consumer objects
setReplyTimeToLive procedure

Sets the time to live value (in milliseconds) of the reply messages when the Message Consumer is passed to requestReply procedure on page 264.

Syntax

```
PROCEDURE setReplyTimeToLive.
DEFINE INPUT PARAMETER millis AS DECIMAL.
```

Applies to

Message Consumer objects

Notes

- Time to live is the number of milliseconds from the time the message is sent to the time the SonicMQ Broker can delete the message from the system.
- A value of 0 means the message never expires.
- The default is JMS-system-dependent; the SonicMQ default value is 0.
- The replyTimeToLive values can be set only once. The fractional part of the decimal value is truncated. If the value does not fit in a Java long value, Java rules for decimal-to-long conversion apply.

See also


For more information, see the Accessing message handler information and the Setting reply properties.
setReplyToDestinationType procedure

Sets the type of the destination specified by setJMSReplyTo procedure on page 292.

Syntax

```abl
PROCEDURE setReplyToDestinationType
DEFINE INPUT PARAMETER type AS CHARACTER.
```

Applies to

Message objects

Notes

• The `type` value can be `queue` or `topic`.
• If setReplyToDestinationType procedure on page 308 is not called, a default type is automatically set when the message is sent, according to the type of the session: `queue` for PTP or `topic` for Pub/Sub.
• For a `jmsSession`, the default is `queue`. To use topics, call setReplyToDestinationType procedure on page 308 to set `topic`.

See also

setJMSReplyTo procedure on page 292, getJMSReplyTo function on page 222, hasReplyTo function on page 249, setReplyToDestinationType procedure on page 308, getReplyToDestinationType function on page 240

For more information, see the Accessing message header properties.

setReuseMessage procedure

Instructs the Message Consumer object not to create a new message for each received message.

Syntax

```abl
PROCEDURE setReuseMessage.
```

Applies to

Message Consumer objects

Notes

• Calling setReuseMessage procedure on page 308 improves performance. If the procedure is not called, the Message Consumer object creates a new message for each received message.
• A message that is being reused should not be deleted before the session is deleted.
See also
setReuseMessage procedure on page 308, getReuseMessage function on page 240
For more information, see the Reusing messages.

**setSaxReader procedure**

Sets the input destination to an internal longchar that will be used as intermediate storage of the XML read from an XMLMessage.

**Syntax**

```
PROCEDURE setSaxReader.
INPUT PARAMETER hd1 AS HANDLE.
```

**Applies to**
Message objects

**Notes**

- The specified handle must already be initialized as a SAX-READER handle. If the specified handle is not a SAX-READER handle, an error is returned.
- The caller uses SAX-READER methods on the handle to read the XML document.

See also
For more information, see the XMLMessage.

**setSelectorAtBroker procedure**

Sets message filtering at the SonicMQ Broker (instead of the SonicMQ client).

**Syntax**

```
PROCEDURE setSelectorAtBroker.
DEFINE INPUT PARAMETER seq AS LOGICAL.
```

**Applies to**
Session objects
Notes

- Since the SonicMQ Broker typically runs on a machine with more resources than the machine running the SonicMQ client, it is desirable to have the SonicMQ Broker provide this filtering instead of the SonicMQ client.

- SonicMQ messages can be filtered so that only those meeting a specific criteria will be received. For point-to-point sessions, this filtering is always done by the SonicMQ Broker. For publish/subscribe sessions, this filtering is done by the SonicMQ client by default.

- This procedure must be called before `beginSession procedure` on page 180.

See also

`setSelectorAtBroker procedure` on page 309, `getSelectorAtBroker function` on page 242

For more information, see the Message selectors.

setSequential procedure

Sets the method the client application will use to connect to the broker.

Syntax

```
PROCEDURE setSequential.
DEFINE INPUT PARAMETER seq AS LOGICAL.
```

Applies to

Session objects

Notes

- Sonic lets clients try to connect to brokers in a connection list in two ways:
  - **Sequentially** — Starting with the first broker in the list and working sequentially
  - **Randomly** — Repeatedly picking a broker randomly

- The default is `TRUE`, which tells clients to try to connect sequentially.

- To attempt load balancing, set `seq` to `FALSE`, which tells clients to try to connect randomly.

See also

`setConnectionURLs procedure` on page 276, `getConnectionURLs function` on page 205, `setSequential procedure` on page 310, `getSequential function` on page 242

For more information, see the Managing fail-over support.

setShort procedure

Converts data to the JMS `short` data type in a `MapMessage`.
Syntax

PROCEDURE setShort.
DEFINE INPUT PARAMETER itemName AS CHARACTER.
DEFINE INPUT PARAMETER value AS INTEGER.

Applies to
Message objects

Notes
The server returns a NumberFormatException message for a value overflow.

See also
createMapMessage procedure on page 186, setBoolean procedure on page 268, setByte procedure on page 270, setBytesFromRaw procedure on page 271, setChar procedure on page 272, setDate procedure on page 277, setDateTime procedure on page 279, setDateTime-TZ procedure on page 280, setDouble procedure on page 283, setFloat procedure on page 287, setInt procedure on page 288, setLong procedure on page 296, setLongString procedure on page 298, setShort procedure on page 310, setString procedure on page 313, getMapNames function on page 230, getItemType function on page 217, getBytesToRaw function on page 200, getChar function on page 200, getDateTime function on page 207, getDateTime function on page 208, getDateTime-TZ function on page 209, getDecimal function on page 210, getInt function on page 215, getLogical function on page 227, getLongString function on page 228, getLongStringCP function on page 228

For more information, see the MapMessage.

setShortProperty procedure

Sets a short message property.

Syntax

PROCEDURE setShortProperty.
DEFINE INPUT PARAMETER propertyName AS CHARACTER.
DEFINE INPUT PARAMETER propertyValue AS INTEGER.

Applies to
Message objects

Notes
The server returns a NumberFormatException message for a value overflow.

See also
clearProperties procedure on page 183

For more information, see the Accessing message properties.
**setShutdownWaitFor procedure**

Sets a **LOGICAL** value indicating the setting of the shutdown **WAIT-FOR** flag.

**Syntax**

```
PROCEDURE setShutdownWaitFor.
DEFINE INPUT PARAMETER val AS LOGICAL.
```

**Applies to**

Session objects

**Notes**

- The default behavior of **deleteSession procedure** on page 194 is to execute the ABL statement **WAIT-FOR PROCEDURE COMPLETE**. The **WAIT-FOR** statement in **deleteSession procedure** on page 194 can cause an error if your application has an outstanding **WAIT-FOR** call. You can direct **deleteSession procedure** on page 194 to skip the **WAIT-FOR** call, by calling **setShutdownWaitFor procedure** on page 312 with a logical "NO" input parameter.

- If you direct **deleteSession procedure** on page 194 to skip the **WAIT-FOR** call, your application must issue a blocking-I/O statement to allow the adapter to process the shutdown message.

- The default value for the shutdown **WAIT-FOR** flag is **YES**, which tells **deleteSession procedure** on page 194 to execute the ABL statement **WAIT-FOR PROCEDURE COMPLETE**.

**See also**

**deleteSession procedure** on page 194, **getShutdownWaitFor function** on page 243.

**setSingleMessageAcknowledgement procedure**

Turns on single-message acknowledgement for a client session.

**Syntax**

```
PROCEDURE setSingleMessageAcknowledgement.
DEFINE INPUT PARAMETER ackMethod as LOGICAL.
```

**Applies to**

Session objects

**Notes**

- If a session is configured to use single-message acknowledgement, the following rules apply:
  - Groups of messages cannot be acknowledged in one operation.
• Acknowledge-and-forward can be used.

• This procedure must be called before `beginSession procedure` on page 180. Otherwise, an error is raised.

**See also**

`acknowledgeAndForward procedure` on page 177, `setSingleMessageAcknowledgement procedure` on page 312, `getSingleMessageAcknowledgement function` on page 243, `setNoAcknowledge procedure` on page 300, `getNoAcknowledge function` on page 233

For more information, see the Message acknowledgement, forwarding, and recovery and the Single-message acknowledgement.

### setString procedure

Converts data in a MapMessage to the JMS String data type.

**Syntax**

```
PROCEDURE setString.
DEFINE INPUT PARAMETER itemName AS CHARACTER.
DEFINE INPUT PARAMETER value AS CHARACTER.
```

**Applies to**

Message objects

**See also**

`createMapMessage procedure` on page 186, `setBoolean procedure` on page 268, `setByte procedure` on page 270, `setBytesFromRaw procedure` on page 271, `setChar procedure` on page 272, `setDate procedure` on page 277, `setDateTime procedure` on page 279, `setDateTime-TZ procedure` on page 280, `setDouble procedure` on page 283, `setFloat procedure` on page 287, `setInt procedure` on page 288, `setLong procedure` on page 296, `setLongString procedure` on page 298, `setShort procedure` on page 310, `setString procedure` on page 313, `getMapNames function` on page 230, `getItemType function` on page 217, `getBytesToRaw function` on page 200, `getChar function` on page 200, `getDate function` on page 207, `getDateTime function` on page 208, `getDateTime-TZ function` on page 209, `getDecimal function` on page 210, `getInt function` on page 215, `getLogical function` on page 227, `getLongString function` on page 228, `getLongStringCP function` on page 228

For more information, see the MapMessage.

### setStringProperty procedure

Sets a String message property.
Syntax

```
PROCEDURE setStringProperty.
DEFINE INPUT PARAMETER propertyName AS CHARACTER.
DEFINE INPUT PARAMETER propertyValue AS CHARACTER.
```

Applies to
Message objects

See also
`clearProperties procedure` on page 183
For more information, see the Accessing message properties.

**setTempTable procedure**

Sets the TempTableMessage.

Syntax

```
PROCEDURE setTempTable.
DEFINE INPUT PARAMETER tableHdl AS HANDLE.
DEFINE INPUT PARAMETER schemaLocation AS CHARACTER.
DEFINE INPUT PARAMETER writeSchema AS LOGICAL.
```

Applies to
Message objects

Notes

- The handle parameter must be a handle to a valid TempTable.
- The schema parameters specify the schema information and are passed directly to the WRITE-XML method. Specifying an Unknown value (?) for any of the schema parameters will result in the use of the default value for that parameter.
- The TempTableMessage is converted to an XMLMessage using the WRITE-XML method and `setLongText` procedure on page 298.

See also
`createTempTableMessage procedure` on page 191, `setTempTable procedure` on page 314, `getTempTable function` on page 244
For more information, see the TempTableMessage.
setText procedure

Clears the message body and sets a new text value.

Syntax

```
PROCEDURE setText.
DEFINE INPUT PARAMETER textValue AS CHARACTER.
```

Applies to
Message objects

Notes
- The call can be made when the message is in write-only or read-only mode.
- After the call, the message is in write-only mode. You can use appendText procedure on page 180 calls to append more text.
- As an alternative to concatenating multiple CHARACTER segments with the appendText procedure on page 180, use setLongText procedure on page 298.

See also
createTextMessage procedure on page 191, setText procedure on page 315, setLongText procedure on page 298, appendText procedure on page 180, endOfStream function on page 195, getCharCount function on page 201, getText function on page 245, getTextSegment function on page 246, getLongText function on page 229, getLongTextCP function on page 229
For more information, see the TextMessage.

setTransactedReceive procedure

Makes the session transacted for receiving.

Syntax

```
PROCEDURE setTransactedReceive.
```

Applies to
Session objects

Notes
A session is not transacted by default.
See also

setTransactedReceive procedure on page 315, getTransactedReceive function on page 247, setTransactedSend procedure on page 316, getTransactedSend function on page 247

For more information, see the Setting and getting JMS connection and session attributes.

setTransactedSend procedure

Makes the session transacted for sending.

Syntax

PROCEDURE setTransactedSend.

Applies to

Session objects

Notes

A session is not transacted by default.

See also

setTransactedReceive procedure on page 315, getTransactedReceive function on page 247, setTransactedSend procedure on page 316, getTransactedSend function on page 247

For more information, see the Setting and getting JMS connection and session attributes.

setUser procedure

Sets the user value for the SonicMQ Broker login and overwrites the default user property set on the OpenEdge Adapter for SonicMQ side.

Syntax

PROCEDURE setUser.
DEFINE INPUT PARAMETER user AS CHARACTER.

Parameter

user

SonicMQ Broker login.

Applies to

Session objects
Notes
If called, setUser procedure on page 316 overwrites the default user property set on the OpenEdge Adapter for SonicMQ side.

See also
setUser procedure on page 316, getUser function on page 248
For more information, see the Setting and getting JMS connection and session attributes.

setX-Document procedure
Copies an XML document specified by the handle parameter into the XMLMessage.

Syntax

```
PROCEDURE setX-Document.
DEFINE INPUT PARAMETER hdl AS HANDLE.
```

Applies to
Message objects

Notes
• The caller must have properly defined the handle and correctly loaded an XML document using X-DOCUMENT calls. If the handle is not an X-DOCUMENT or X-NODOREF handle, an error is returned.
• This procedure loads the XML into an internal longchar. Use setLongText procedure on page 298 to copy the longchar into the XMLMessage.
• Other errors include any error generated by xdoc:SAVE, any error generated by the XMLMessage, or by setLongText procedure on page 298.

See also
createXMLMessage procedure on page 192, setX-Document procedure on page 317, getX-Document function on page 248,.saxReader procedure on page 309, getSaxWriter function on page 241, deleteSaxWriter procedure on page 193
For more information, see the XMLMessage.

startReceiveMessages procedure
Starts receiving messages after creating a new session or after calling stopReceiveMessages procedure on page 318.
Syntax

PROCEDURE startReceiveMessages.

Applies to
Session objects

Notes
• Messages can be sent without calling startReceiveMessages procedure on page 317.
• This procedure executes remotely (sends a message to the OpenEdge Adapter for SonicMQ).

See also
startReceiveMessages procedure on page 317, stopReceiveMessages procedure on page 318
For more information, see the Establishing session control.

stopReceiveMessages procedure

Causes the OpenEdge Adapter for SonicMQ Broker to stop receiving messages on behalf of the OpenEdge client.

Syntax

PROCEDURE stopReceiveMessages.

Applies to
Session objects

Notes
• A subsequent call to startReceiveMessages procedure on page 317 resumes message reception and delivery.
• If this procedure is called in a pubsubsession object and the subscription is not durable, messages published while reception is stopped are not delivered.
• A single message that was already sent to the client before stopReceiveMessages procedure on page 318 was called might be received by the client after stopReceiveMessages procedure on page 318 call.
• This procedure executes remotely (sends a message to the OpenEdge Adapter for SonicMQ).

See also
startReceiveMessages procedure on page 317, stopReceiveMessages procedure on page 318
For more information, see the Establishing session control.
subscribe procedure

Subscribes to a topic.

Syntax

```
PROCEDURE subscribe.
DEFINE INPUT PARAMETER topicName AS CHARACTER.
DEFINE INPUT PARAMETER subscriptionName AS CHARACTER.
DEFINE INPUT PARAMETER messageSelector AS CHARACTER.
DEFINE INPUT PARAMETER noLocalPublications AS LOGICAL.
DEFINE INPUT PARAMETER messageConsumer AS HANDLE.
```

Parameters

topicName

The topic to which the procedure subscribes.

subscriptionName

A `subscriptionName` parameter with a value other than `UNKNOWN` specifies a durable subscription. Durable subscriptions require that the JMS client have a client ID identifier; the client must call `setClientID` in the `pubsubsession` object (or set the default client ID on the server side). If the `subscriptionName` value is `UNKNOWN`, the subscription is not durable.

messageSelector

A message selector (optional).

noLocalPublications

A Boolean flag controlling whether the application receives its own messages (optional). The default is `FALSE` (the session receives its own publications).

messageConsumer

The Message Consumer object.

Applies to

Session objects

Notes

- The messages are handled asynchronously by the `messageConsumer` object.
- If the `subscriptionName` value is `UNKNOWN` or an empty string, the subscription is not durable.
- Durable subscriptions require the JMS client to have a client ID identifier.
- The client must call `setClientID` procedure on page 273 in the `pubsubsession` object (or set the default client ID on the server side) if a durable subscription is desired. The default of `noLocalPublications` is `FALSE`. The session, by default, get its own publications.
• This procedure executes remotely (sends a message to the OpenEdge Adapter for SonicMQ).

See also
cancelDurableSubscription procedure on page 182, publish procedure on page 254, subscribe procedure on page 319

For more information, see the Publishing messages to a topic, the Subscribing to a topic, the Durable subscriptions, and the Methods unique to Pub/Sub messaging.

For an example, see the Pub/Sub messaging example.

waitForMessages procedure

Waits and processes events as long as the user-defined function is TRUE.

Syntax

PROCEDURE waitForMessages:
   DEFINE INPUT PARAMETER UDFName AS CHARACTER NO-UNDO.
   DEFINE INPUT PARAMETER procH AS HANDLE NO-UNDO.
   DEFINE INPUT PARAMETER timeOut AS INTEGER NO-UNDO.

Parameters

UDFName
   User-defined function.

procH
   Handle to user-defined function.

timeOut
   Time limit (in seconds) in which no messages are received.

Applies to

Session objects

Notes

• UDFName (in procH) returns TRUE and there is no period of more than timeOut seconds in which no messages are received.

• The user-defined function, UDFName, is evaluated each time after a message is handled.

See also
createMessageConsumer procedure on page 187, deleteConsumer procedure on page 192, messageHandler procedure on page 252, waitForMessages procedure on page 320
writeBoolean procedure

Writes boolean data to the body of a StreamMessage.

Syntax

PROCEDURE writeBoolean.
DEFINE INPUT PARAMETER value AS LOGICAL.

Applies to

Message objects

Notes

An Unknown value (?) is considered FALSE.

See also

createStreamMessage procedure on page 189, writeBoolean procedure on page 321, writeByte procedure on page 321, writeBytesFromRaw procedure on page 322, writeChar procedure on page 323, writeDate procedure on page 323, writeDateTime procedure on page 324, writeDateTime-TZ procedure on page 325, writeDouble procedure on page 326, writeFloat procedure on page 326, writeInt procedure on page 327, writeLong procedure on page 328, writeLongString procedure on page 329, writeShort procedure on page 330, writeString procedure on page 330, endOfStream function on page 195, moveToNext procedure on page 253, readBytesToRaw procedure on page 256, readChar function on page 256, readDate function on page 257, readDateTime function on page 258, readDateTime-TZ function on page 258, readDecimal function on page 259, readInt function on page 259, readLogical function on page 261, readLongString function on page 261, readLongStringCP function on page 262

For more information, see the StreamMessage.

writeByte procedure

Writes byte data to the body of a StreamMessage.

Syntax

PROCEDURE writeByte.
DEFINE INPUT PARAMETER value AS INTEGER.

Applies to

Message objects
Notes

• Byte values range from –128 to 127.
• The server returns a NumberFormatException message for a value overflow. For example, calling `writeByte(1000)` results in a value overflow.
• Used in write-only mode to write an additional bytes segment to a BytesMessage.

See also

createStreamMessage procedure on page 189, writeBoolean procedure on page 321, writeByte procedure on page 321, writeBytesFromRaw procedure on page 322, writeChar procedure on page 323, writeDate procedure on page 323, writeDateTime procedure on page 324, writeDateTime-TZ procedure on page 325, writeDouble procedure on page 326, writeFloat procedure on page 326, writeInt procedure on page 327, writeLong procedure on page 328, writeLongString procedure on page 329, writeShort procedure on page 330, writeString procedure on page 330, endOfStream function on page 195, moveToNext procedure on page 253, readBytesToRaw procedure on page 256, readChar function on page 256, readDate function on page 257, readDateTime function on page 258, readDateTime-TZ function on page 258, readDecimal function on page 259, readInt function on page 259, readLogical function on page 261, readLongString function on page 261, readLongStringCP function on page 262

For more information, see the StreamMessage.

writeBytesFromRaw procedure

Writes byte array data to the body of a StreamMessage or ByteMessage.

Syntax

PROCEDURE writeBytesFromRaw.
DEFINE INPUT PARAMETER bytesValue AS RAW.

Applies to

Message objects

Notes

• This procedure applies to both StreamMessage and ByteMessage message types.
• This procedure can be called in write-only mode to write additional bytes segments to a BytesMessage and work around the RAW data type limit of 32K.
writeChar procedure

Writes char data to the body of a StreamMessage.

Syntax

```plaintext
PROCEDURE writeChar.
DEFINE INPUT PARAMETER value AS CHARACTER.
```

Applies to

Message objects

Notes

The number of characters in the char value must be one.

See also

createStreamMessage procedure on page 189, writeBoolean procedure on page 321, writeByte procedure on page 321, writeBytesFromRaw procedure on page 322, writeChar procedure on page 323, writeDateTime procedure on page 323, writeDateTime-TZ procedure on page 324, writeDouble procedure on page 325, writeFloat procedure on page 326, writeInt procedure on page 327, writeLong procedure on page 328, writeLongString procedure on page 329, writeShort procedure on page 330, writeString procedure on page 330, endOfStream function on page 195, moveToNext procedure on page 253, readBytesToRaw procedure on page 256, readChar function on page 256, readDate function on page 257, readDateTime function on page 258, readDateTime-TZ function on page 258, readDecimal function on page 259, readInt function on page 259, readLogical function on page 261, readLongString function on page 261, readLongStringCP function on page 262

For more information, see the StreamMessage.

writeDate procedure

Writes a date value as String data to the body of a StreamMessage.
Syntax

PROCEDURE writeDate.
DEFINE INPUT PARAMETER value AS DATE.

Applies to
Message objects

Notes

• The procedure actually writes `value` as a `DATETIME-TZ` value, adding default time and time zone information:
  • A time of 12:00AM (midnight)
  • The default time zone of the client application

• If the message might be consumed by an application in a different time zone from that of the sending application, the ABL programmer must ensure that the receiving application can interpret the value correctly.

• The format of the `DATETIME-TZ` value is a subset of the ISO8601 format.

• An error is returned if the `Unknown value (?)` is specified.

See also
createStreamMessage procedure on page 189, writeBoolean procedure on page 321, writeByte procedure on page 321, writeBytesFromRaw procedure on page 322, writeChar procedure on page 323, writeDate procedure on page 323, writeDateTime procedure on page 324, writeDateTime-TZ procedure on page 325, writeDouble procedure on page 326, writeFloat procedure on page 326, writeInt procedure on page 327, writeLong procedure on page 328, writeLongString procedure on page 329, writeShort procedure on page 330, writeString procedure on page 330, endOfStream function on page 195, moveToNext procedure on page 253, readBytesToRaw procedure on page 256, readChar function on page 256, readDate function on page 257, readDateTime function on page 258, readDateTime-TZ function on page 258, readDecimal function on page 259, readInt function on page 259, readLogical function on page 261, readLongString function on page 261, readLongStringCP function on page 262

For more information, see the StreamMessage.

writeDateTime procedure

Writes a date-time value as String data to the body of a StreamMessage.

Syntax

PROCEDURE writeDateTime.
DEFINE INPUT PARAMETER value AS DATETIME.

Applies to
Message objects
Notes

• The procedure actually writes value as a DATETIME-TZ value, adding default time and time zone information.

• The format of the DATETIME-TZ value is a subset of the ISO8601 format.

• An error is returned if the Unknown value (?) is specified.

• If the message may be consumed by an application in a different time zone from that of the sending application, the ABL programmer must ensure that the receiving application can interpret the value correctly.

See also

createStreamMessage procedure on page 189, writeBoolean procedure on page 321, writeByte procedure on page 321, writeBytesFromRaw procedure on page 322, writeChar procedure on page 323, writeDate procedure on page 323, writeDateTime procedure on page 324, writeDateTime-TZ procedure on page 325, writeDouble procedure on page 326, writeFloat procedure on page 326, writeInt procedure on page 327, writeLong procedure on page 328, writeLongString procedure on page 329, writeShort procedure on page 330, writeString procedure on page 330, endOfStream function on page 195, moveToNext procedure on page 253, readBytesToRaw procedure on page 256, readChar function on page 256, readDate function on page 257, readDateTime function on page 258, readDateTime-TZ function on page 258, readDecimal function on page 259, readInt function on page 259, readLogical function on page 261, readLongString function on page 261, readLongStringCP function on page 262

For more information, see the StreamMessage.

writeDateTime-TZ procedure

Writes a date-time value, including time zone information, as String data to the body of a StreamMessage.

Syntax

```plaintext
PROCEDURE writeDateTime-TZ.
DEFINE INPUT PARAMETER value AS DATETIME-TZ.
```

Applies to

Message objects

Notes

An error is returned if the Unknown value (?) is specified.
writeDouble procedure

Writes double data to the body of a StreamMessage.

Syntax

```
PROCEDURE writeDouble.
DEFINE INPUT PARAMETER value AS DECIMAL.
```

Applies to
Message objects

See also
createStreamMessage procedure on page 189, writeBoolean procedure on page 321, writeByte procedure on page 321, writeBytesFromRaw procedure on page 322, writeChar procedure on page 323, writeDate procedure on page 323, writeDateTime procedure on page 324, writeDateTime-TZ procedure on page 325, writeDouble procedure on page 326, writeFloat procedure on page 326, writeInt procedure on page 327, writeLong procedure on page 328, writeLongString procedure on page 329, writeShort procedure on page 330, writeString procedure on page 330, endOfStream function on page 195, moveToNext procedure on page 253, readBytesToRaw procedure on page 256, readChar function on page 256, readDate function on page 257, readDateTime function on page 258, readDateTime-TZ function on page 258, readDecimal function on page 259, readInt function on page 259, readLogical function on page 261, readLongString function on page 261, readLongStringCP function on page 262

For more information, see the StreamMessage.

writeFloat procedure

Writes float data to the body of a StreamMessage.
**Syntax**

```plaintext
PROCEDURE writeFloat.
DEFINE INPUT PARAMETER value AS DECIMAL.
```

**Applies to**
Message objects

**Notes**
The fractional part of the DECIMAL value is truncated.

**See also**
- `createStreamMessage` on page 189
- `writeBoolean` on page 321
- `writeByte` on page 321
- `writeBytesFromRaw` on page 322
- `writeChar` on page 323
- `writeDateTime` on page 323
- `writeDateTime-TZ` on page 325
- `writeDouble` on page 326
- `writeFloat` on page 326
- `writeInt` on page 327
- `writeLong` on page 328
- `writeLongString` on page 329
- `writeShort` on page 330
- `writeString` on page 330
- `endOfStream` on page 195
- `moveToNext` on page 253
- `readBytesToRaw` on page 256
- `readChar` on page 256
- `readDate` on page 257
- `readDateTime` on page 258
- `readDateTime-TZ` on page 258
- `readDecimal` on page 259
- `readInt` on page 259
- `readLogical` on page 261
- `readLongString` on page 261
- `readLongStringCP` on page 262

For more information, see the StreamMessage.

**writeInt procedure**

Writes int data to the body of a StreamMessage.

**Syntax**

```plaintext
PROCEDURE writeInt.
DEFINE INPUT PARAMETER value AS INTEGER.
```

**Applies to**
Message objects
writeInt64 procedure

Writes INT64 data to the body of a StreamMessage.

Syntax

```
PROCEDURE writeInt64.
DEFINE INPUT PARAMETER value AS INT64.
```

Applies to

Message objects

Notes

The Unknown value (?) is allowed.

See also

createStreamMessage procedure on page 189, writeBoolean procedure on page 321, writeByte procedure on page 321, writeBytesFromRaw procedure on page 322, writeChar procedure on page 323, writeDate procedure on page 323, writeDateTime procedure on page 324, writeDateTime-TZ procedure on page 325, writeDouble procedure on page 326, writeFloat procedure on page 326, writeInt64 procedure on page 328, writeLong procedure on page 328, writeLongString procedure on page 329, writeShort procedure on page 330, writeString procedure on page 330, endOfStream function on page 195, moveToNext procedure on page 253, readBytesToRaw procedure on page 256, readChar function on page 256, readDate function on page 257, readDateTime function on page 258, readDateTime-TZ function on page 258, readDecimal function on page 259, readInt function on page 259, readInt64 function on page 260, readLogical function on page 261, readLongString function on page 261, readLongStringCP function on page 262

For more information, see the StreamMessage.

writeLong procedure

Writes long data to the body of a StreamMessage.
Syntax

```
PROCEDURE writeLong.
DEFINE INPUT PARAMETER value AS DECIMAL.
```

Applies to
Message objects

Notes
The fractional part of the DECIMAL value is truncated.

See also
createStreamMessage procedure on page 189, writeBoolean procedure on page 321, writeByte procedure on page 321, writeBytesFromRaw procedure on page 322, writeChar procedure on page 323, writeDateTime procedure on page 323, writeDateTime-TZ procedure on page 325, writeDouble procedure on page 326, writeFloat procedure on page 326, writeInt procedure on page 327, writeLong procedure on page 328, writeLongString procedure on page 329, writeShort procedure on page 330, writeString procedure on page 330, endOfStream function on page 195, moveToNext procedure on page 253, readBytesToRaw procedure on page 256, readChar function on page 256, readDate function on page 257, readDateTime function on page 258, readDateTime-TZ function on page 258, readDecimal function on page 259, readInt function on page 259, readLogical function on page 261, readLongString function on page 261, readLongStringCP function on page 262

For more information, see the StreamMessage.

**writeLongString procedure**

**Write String data of any length to the body of a StreamMessage.**

Syntax

```
PROCEDURE writeLongString.
DEFINE INPUT PARAMETER value AS LONGCHAR.
```

Applies to
Message objects
writeShort procedure

 Writes short data to the body of a StreamMessage.

 Syntax

 PROCEDURE writeShort.
 DEFINE INPUT PARAMETER value AS INTEGER.

 Applies to
 Message objects

 Notes
 The server returns a NumberFormatException message for a value overflow.

 See also
 createStreamMessage procedure on page 189, writeBoolean procedure on page 321, writeByte procedure on page 321, writeBytesFromRaw procedure on page 322, writeChar procedure on page 323, writeDate procedure on page 323, writeDateTime procedure on page 324, writeDateTime-TZ procedure on page 325, writeDouble procedure on page 326, writeFloat procedure on page 326, writeInt procedure on page 327, writeLong procedure on page 328, writeLongString procedure on page 329, writeShort procedure on page 330, writeString procedure on page 330, endOfStream function on page 195, moveToNext procedure on page 253, readBytesToRaw procedure on page 256, readChar function on page 256, readDate function on page 257, readDateTime function on page 258, readDateTime-TZ function on page 258, readDecimal function on page 259, readInt function on page 259, readLogical function on page 261, readLongString function on page 261, readLongStringCP function on page 262

 For more information, see the StreamMessage.

writeString procedure

 Writes String data to the body of a StreamMessage.
PROCEDURE writeString.
DEFINE INPUT PARAMETER value AS CHARACTER.

Applies to
Message objects

See also
createStreamMessage procedure on page 189, writeBoolean procedure on page 321, writeByte procedure on page 321, writeBytesFromRaw procedure on page 322, writeChar procedure on page 323, writeDate procedure on page 323, writeDateTime procedure on page 324, writeDateTime-TZ procedure on page 325, writeDouble procedure on page 326, writeFloat procedure on page 326, writeInt procedure on page 327, writeLong procedure on page 328, writeLongString procedure on page 329, writeShort procedure on page 330, writeString procedure on page 330, endOfStream function on page 195, moveToNext procedure on page 253, readBytesToRaw procedure on page 256, readChar function on page 256, readDate function on page 257, readDateTime function on page 258, readDateTime-TZ function on page 258, readDecimal function on page 259, readInt function on page 259, readLogical function on page 261, readLongString function on page 261, readLongStringCP function on page 262

For more information, see the StreamMessage.
Messaging Examples

This appendix provides ABL (Advanced Business Language) messaging example procedures of Pub/Sub and PTP messaging, as well as a sample application illustrating the gateway approach to integration with the native ABL publish and subscribe mechanism. Examples are written using the BrokerConnect option; however, the examples may be run using either the ClientConnect or ServerConnect options.

For information on locating the examples, see OpenEdge messages on page 24.

For an alphabetical API reference, see ABL - JMS API Reference on page 161.

For details, see the following topics:

• Pub/Sub messaging examples
• PTP messaging examples
• MultiPartMessage example
• Gateway sample application

Pub/Sub messaging examples

The Pub/Sub examples consist of sets of subscribers and publishers. You should run each messaging example interactively from its own window. Launch the subscriber first, because the message is discarded if the publisher publishes the message before there are any subscribers to the topic (or any durable subscriptions).

The examples include:

• Publishing and subscribing with a TextMessage on page 334
Publishing and subscribing with a TextMessage

The procedures example1.p and example2.p demonstrate basic Pub/Sub messaging. The procedure example1.p publishes a TextMessage to a topic, and the procedure example2.p subscribes to a topic and receives a TextMessage.

To run example1.p and example2.p:

1. Run example2.p so the subscriber is running before you publish, as shown:
example2.p

```cpp
/* Subscribes and receives a Text message. */
DEFINE VARIABLE hPubSubSession AS HANDLE NO-UNDO.
DEFINE VARIABLE hConsumer AS HANDLE NO-UNDO.
/* Creates a session object. */
RUN jms/pubsubsession.p PERSISTENT SET hPubSubSession
("-H localhost -S 5162 ").
RUN setBrokerURL IN hPubSubSession ("localhost:2506").
RUN beginSession IN hPubSubSession.
/* Subscribe to the GolfTopic topic. Messages are handled by the
 "golfHandler" internal procedure. */
RUN createMessageConsumer IN hPubSubSession
 (THIS-PROCEDURE, /* this procedure will handle it */
 "golfHandler", /* name of internal procedure */
 OUTPUT hConsumer).
RUN subscribe IN hPubSubSession
 ("GolfTopic", /* name of topic */
 ?, /* subscription is not durable */
 ?, /* no message selector */
 FALSE, /* want my own messages too */
hConsumer). /* handles the incoming messages*/
/* Start receiving messages */
RUN startReceiveMessages IN hPubSubSession.
/* Wait to receive the messages. Any other I/O-blocked statements can be
 used for receiving messages. */
WAIT-FOR u1 OF THIS-PROCEDURE.
PROCEDURE golfHandler:
DEFINE INPUT PARAMETER hMessage AS HANDLE NO-UNDO.
DEFINE INPUT PARAMETER msghConsumer AS HANDLE NO-UNDO.
DEFINE OUTPUT PARAMETER hReply AS HANDLE NO-UNDO.
/* Display the message - we assume that reply is not required. */
DISPLAY "Message text: "
DYNAMIC-FUNCTION('getText':U IN hMessage) FORMAT "x(70)".
RUN deleteMessage IN hMessage.
APPLY "U1" TO THIS-PROCEDURE.
END PROCEDURE.
```

2. Run example1.p to publish the TextMessage to a topic, as shown:

example1.p

```cpp
/* Publishes a text message. */
DEFINE VARIABLE hMessage AS HANDLE NO-UNDO.
DEFINE VARIABLE hPubSubSession AS HANDLE NO-UNDO.
/* Creates a session object. */
RUN jms/pubsubsession.p PERSISTENT SET hPubSubSession
("-H localhost -S 5162 ").
RUN setBrokerURL IN hPubSubSession ("localhost:2506").
RUN beginSession IN hPubSubSession.
/* Create a text message */
RUN createTextMessage IN hPubSubSession (OUTPUT hMessage).
RUN setText IN hMessage ("Golf shoes on sale today").
/* Publish the message on the "GolfTopic" topic */
RUN publish IN hPubSubSession ("GolfTopic", hMessage, ?, ?, ?).
RUN deleteMessage IN hMessage.
RUN deleteSession IN hPubSubSession.
```
Publishing with message properties and subscribing selectively

The procedure example3.p publishes a TextMessage from Super Golf Center to Sub Par Golf using the setStringProperty procedure on page 313. The procedure example4.p subscribes to a topic and only receives messages addressed to Sub Par Golf (by passing a selector to the subscribe procedure on page 319 call).

To run example3.p and example4.p:

1. Run example4.p first so the subscriber is running before you publish, as shown:

```
example4.p

/* Receives a text message with "TO" property equal to "Sub Par Golf" */
DEFINE VARIABLE hConsumer AS HANDLE NO-UNDO.
DEFINE VARIABLE hPubSubSession AS HANDLE NO-UNDO.
/* Creates a session object. */
RUN jms/pubsubsession.p PERSISTENT SET hPubSubSession
("-H localhost -S 5162 ").
RUN setBrokerURL IN hPubSubSession ("localhost:2506").
RUN beginSession IN hPubSubSession.
/* Subscribes to the GolfTopic topic. Messages are handled by the
"golfHandler" internal procedure. */
RUN createMessageConsumer IN hPubSubSession
_THIS-PROCEDURE, /* this procedure will handle it */
golfHandler", /* name of internal procedure */
OUTPUT hConsumer).
RUN subscribe IN hPubSubSession
"GolfTopic", /* name of topic */
?, /* subscription is not durable */
"TO = 'Sub Par Golf'", /* only messages from Sub Par Golf */
FALSE, /* want my own messages too */
hConsumer). /* handles the incoming messages */
/* Start receiving messages */
RUN startReceiveMessages IN hPubSubSession.
/* Wait to receive the messages. Any other I/O-blocked statements can be
used for receiving messages. */
WAIT-FOR u1 OF THIS-PROCEDURE.
PROCEDURE golfHandler:
DEFINE INPUT PARAMETER hMessage AS HANDLE NO-UNDO.
DEFINE INPUT PARAMETER hMsgConsumer AS HANDLE NO-UNDO.
DEFINE OUTPUT PARAMETER hReply AS HANDLE NO-UNDO.
/* Display the message - we assume that reply is not required. */
DISPLAY
"Message text: " DYNAMIC-FUNCTION
('getText':U IN hMessage) FORMAT "X(30)"
"Message from: " DYNAMIC-FUNCTION
('getCharProperty':U IN hMessage, "FROM").
RUN deleteMessage IN hMessage.
APPLY "U1" TO THIS-PROCEDURE.
END PROCEDURE.
```

2. Run example3.p, as shown:
Publishing with a reply handle, subscribing, and receiving an automatic reply

The procedures example5.p, example6.p, and example7.p illustrate publishing with a reply handle and receiving an automatic reply. The procedure example5.p subscribes with an automatic reply mechanism. It can only reply to messages that have the JMSReplyTo header field. The procedure example6.p subscribes with explicit reply by calling the publish procedure on page 254e directly. It can only reply to messages that have the JMSReplyTo header field. The procedure example7.p publishes using the requestReply procedure on page 264 for receiving reply messages from subscribers. It populates the JMSReplyTo header field automatically.

To run example5.p, example6.p, and example7.p:

1. Run example5.p so the subscriber is running before you publish, as shown:
example5.p

/* Using the automatic reply mechanism. Note that the received message
must have a JMSReplyTo header field for this to work. Example7 can be
used to receive the reply. */
DEFINE VARIABLE hConsumer AS HANDLE NO-UNDO.
DEFINE VARIABLE hPubSubSession AS HANDLE NO-UNDO.
/* Creates a session object. */
RUN jms/pubsubsession.p PERSISTENT SET hPubSubSession
("-H localhost -S 5162 ").
RUN setBrokerURL IN hPubSubSession ("localhost:2506").
RUN beginSession IN hPubSubSession.
/* Subscribe to the GolfTopic topic. Messages are handled by the
"golfHandler" internal procedure. */
RUN createMessageConsumer IN hPubSubSession
(TTHIS-PROCEDURE, /* this procedure will handle it */
"golfHandler", /* name of internal procedure */
OUTPUT hConsumer).
RUN subscribe IN hPubSubSession
("GolfTopic", /* name of topic */
?, /* subscription is not durable */
?, /* no message selector */
FALSE, /* want my own messages too */
hConsumer). /* handles the messages */
/* Start receiving messages */
RUN startReceiveMessages IN hPubSubSession.
/* Wait forever to receive messages since "u1" is never applied. */
WAIT-FOR u1 OF THIS-PROCEDURE.
PROCEDURE golfHandler:
DEFINE INPUT PARAMETER hMessage AS HANDLE NO-UNDO.
DEFINE INPUT PARAMETER hMsgConsumer AS HANDLE NO-UNDO.
DEFINE OUTPUT PARAMETER hReply AS HANDLE NO-UNDO.
/* Creates a reply message. The reply is published automatically when
control returns to the ABL-JMS implementation. */
DISPLAY DYNAMIC-FUNCTION('getText':U IN hMessage) FORMAT "x(60)".
IF DYNAMIC-FUNCTION('hasReplyTo':U IN hMessage) THEN DO:
  RUN createTextMessage IN hPubSubSession (OUTPUT hReply).
  RUN setText IN hReply ("Will bid. Send data in sportsXML format.").
END.
RUN deleteMessage IN hMessage.
END PROCEDURE.

2. Run example6.p to subscribe with explicit reply by calling the publish procedure directly, as shown:
/* Replying explicitly. Note that the received message must have a
JMSReplyTo header field for this to work. Example7 can be used to
receive the reply. */
DEFINE VARIABLE msgConsumer AS HANDLE NO-UNDO.
DEFINE VARIABLE hPubSubSession AS HANDLE NO-UNDO.
/* Creates a session object. */
RUN jms/pubsubsession.p PERSISTENT SET hPubSubSession
("-H localhost -S 5162 ").
RUN setBrokerURL IN hPubSubSession ("localhost:2506").
RUN beginSession IN hPubSubSession.
RUN createMessageConsumer IN hPubSubSession
(This-PROCEDURE, /* this procedure will handle it */
"messageHandler", /* name of internal procedure */
OUTPUT msgConsumer).
RUN subscribe IN hPubSubSession
("GolfTopic", ?, /* no durable subscription */
?, /* no message selector */
FALSE, /* want to get my own publications */
msgConsumer).
RUN startReceiveMessages IN hPubSubSession.
/* Wait forever to receive messages since "u1" is never applied. */
WAIT-FOR u1 OF THIS-PROCEDURE.
RUN deleteSession IN hPubSubSession.
PROCEDURE messageHandler:
DEFINE INPUT PARAMETER hMessage AS HANDLE NO-UNDO.
DEFINE INPUT PARAMETER hMessageConsumer AS HANDLE NO-UNDO.
/* hAutoReply is not used in this example */
DEFINE OUTPUT PARAMETER hAutoReply AS HANDLE NO-UNDO.
DEFINE VARIABLE hReply AS HANDLE NO-UNDO.
DISPLAY DYNAMIC-FUNCTION('getText':U IN hMessage) FORMAT "x(60)".
IF NOT DYNAMIC-FUNCTION('hasReplyTo':U IN hMessage) THEN RETURN.
/* Publishes a reply explicitly - using the publish call. */
RUN createTextMessage IN hPubSubSession (OUTPUT hReply).
RUN setText IN hReply("Will bid. Send data in sportsXML format.").
RUN publish IN hPubSubSession (DYNAMIC-FUNCTION
('getJMSReplyTo':U IN hMessage), hReply, ?, ?, ?).
RUN deleteMessage IN hMessage.
/* After we have sent the message, delete it. */
RUN deleteMessage IN hReply.
END PROCEDURE.

3. Run example7.p to publish using the requestReply procedure for receiving reply messages from subscribers.
It populates the JMSReplyTo header field automatically, as shown:
Publishing, receiving, and processing a StreamMessage

The procedures example8.p and example9.p publish, receive, and process a StreamMessage.

To run example8.p and example9.p:

1. Connect to the Sports database.
2. Run example9.p to receive the StreamMessage containing the customer names and numbers, as shown:
/* Receives a Stream message. */
DEFINE VARIABLE hConsumer AS HANDLE NO-UNDO.
DEFINE VARIABLE hPubSubSession AS HANDLE NO-UNDO.
/* Creates a session object. */
RUN jms/pubsubsession.p PERSISTENT SET hPubSubSession
("-H localhost -S 5162 ").
RUN setBrokerURL IN hPubSubSession ("localhost:2506").
RUN beginSession IN hPubSubSession.
/* Subscribe to the newCustomers topic. The newCustHandler internal
procedure handles the list of new customers. */
RUN createMessageConsumer IN hPubSubSession
(This-PROCEDURE, /* this procedure will handle it */
"newCustHandler", /* name of internal procedure */
OUTPUT hConsumer).
RUN subscribe IN hPubSubSession
("NewCustomers", /* name of topic */
?, /* subscription is not durable*/
?, /* no message selector. */
FALSE, /* want my own messages too */
hConsumer). /* handles the messages */
/* Start receiving messages */
RUN startReceiveMessages IN hPubSubSession.
/* Wait to receive the messages. Any other I/O-blocked statements can be
used for receiving messages. */
WAIT-FOR u1 OF THIS-PROCEDURE.
PROCEDURE newCustHandler:
DEFINE INPUT PARAMETER hMessage AS HANDLE NO-UNDO.
DEFINE INPUT PARAMETER hMsgConsumer AS HANDLE NO-UNDO.
DEFINE OUTPUT PARAMETER hReply AS HANDLE NO-UNDO.
/* Display the stream of customer names and customer numbers. The
moveToNext function moves the cursor to the next item in the stream
and returns the data type of that item. We assume the reply is not
required. */
IF NOT DYNAMIC-FUNCTION('getMessageType':U IN hMessage) =
"StreamMessage" THEN RETURN.
/* Note that the 'moveToNext' functions returns the item's data type. */
DO WHILE NOT DYNAMIC-FUNCTION('endOfStream':U IN hMessage) WITH DOWN:
DISPLAY DYNAMIC-FUNCTION('moveToNext':U IN hMessage)
DYNAMIC-FUNCTION('readChar':U IN hMessage)
DYNAMIC-FUNCTION('moveToNext':U IN hMessage)
DYNAMIC-FUNCTION('readInt':U IN hMessage).
DOWN.
END.
RUN deleteMessage IN hMessage.
APPLY "U1" TO THIS-PROCEDURE.
END PROCEDURE.

3. Run example8.p to publish a StreamMessage containing customer names and numbers, as shown:
### Publishing, receiving, and parsing an XMLMessage

The procedures example10.p and example11.p create, publish, receive, and parse an XMLMessage.

To run example10.p and example11.p:

1. Run example11.p so the subscriber is running before you publish. The following example subscribes, receives, and parses an XMLMessage:

```plaintext
/* Publishing a Stream message. */
DEFINE VARIABLE hMessage AS HANDLE NO-UNDO.
DEFINE VARIABLE hPubSubSession AS HANDLE NO-UNDO.
/* Creates a session object. */
RUN jms/pubsubsession.p PERSISTENT SET hPubSubSession
("-H localhost -S 5162 ").
RUN setBrokerURL IN hPubSubSession ("localhost:2506").
RUN beginSession IN hPubSubSession.
/* Create a stream message */
RUN createStreamMessage IN hPubSubSession (OUTPUT hMessage).
/* Load the message with a list of customer names and custnums. */
FOR EACH customer NO-LOCK:
    RUN writeString IN hMessage (customer.name).
    RUN writeInt IN hMessage (customer.custnum).
END.
/* Publish the message on the NewCustomers topic. */
RUN publish IN hPubSubSession ("NewCustomers", hMessage, ?, ?, ?).
```
2. Run example10.p to create and publish an XMLMessage with the data of 100 people, as shown:

```plaintext
/* Receives and parse an XML message. */
DEFINE VARIABLE hPubSubSession AS HANDLE NO-UNDO.
DEFINE VARIABLE msgConsumer1 AS HANDLE NO-UNDO.
DEFINE VARIABLE msgNum AS INTEGER NO-UNDO.
DEFINE VARIABLE stillWaiting AS LOGICAL NO-UNDO INITIAL TRUE.
RUN jms/pubsubsession.p PERSISTENT SET hPubSubSession
("-H localhost -S 5162 ").
RUN setBrokerURL IN hPubSubSession ("localhost:2506").
RUN beginSession IN hPubSubSession.
RUN createMessageConsumer IN hPubSubSession
    (THIS-PROCEDURE, "messageHandler", OUTPUT msgConsumer1).
RUN subscribe IN hPubSubSession
    ("people", ?, ?, FALSE, msgConsumer1) NO-ERROR.
RUN startReceiveMessages IN hPubSubSession.
RUN waitForMessages IN hPubSubSession ("inWait", THIS-PROCEDURE, ?).
RUN deleteSession IN hPubSubSession.
PROCEDURE messageHandler:
    DEFINE INPUT PARAMETER hMessage AS HANDLE NO-UNDO.
    DEFINE INPUT PARAMETER hMessageConsumer AS HANDLE NO-UNDO.
    DEFINE OUTPUT PARAMETER hReply AS HANDLE NO-UNDO.
    DEFINE VARIABLE hDoc AS HANDLE NO-UNDO.
    DEFINE VARIABLE hRoot AS HANDLE NO-UNDO.
    DEFINE VARIABLE ix AS INTEGER NO-UNDO.
    DEFINE VARIABLE mDoc AS MEMPTR NO-UNDO.
    DEFINE VARIABLE xmlText AS CHARACTER NO-UNDO.
    CREATE X-DOCUMENT hDoc.
    CREATE X-NODEREF hRoot.
    SET-SIZE(mDoc) = 400000. /* The size is an estimate. */
    ix = 1.
    DO WHILE NOT DYNAMIC-FUNCTION('endOfStream' IN hMessage):
        xmlText = DYNAMIC-FUNCTION('getTextSegment':U IN hMessage).
        PUT-STRING(mDoc, ix) = xmlText.
        ix = ix + LENGTH(xmlText).
    END.
    hDoc:LOAD("memptr", mDoc, FALSE).
    hDoc:GET-DOCUMENT-ELEMENT(hRoot).
    RUN getPeople(hRoot, 1).
    RUN deleteMessage IN hMessage.
    SET-SIZE(mDoc) = 0.
    stillWaiting = FALSE.
END PROCEDURE.

/* Displays the XML node names and XML text. */
PROCEDURE getPeople:
    DEFINE INPUT PARAMETER hParent AS HANDLE NO-UNDO.
    DEFINE INPUT PARAMETER iLevel AS INTEGER NO-UNDO.
    DEFINE VARIABLE hNodeRef AS HANDLE NO-UNDO.
    DEFINE VARIABLE ix AS INTEGER NO-UNDO.
    CREATE X-NODEREF hNodeRef.
    REPEAT ix = 1 TO hParent:NUM-CHILDREN.
        hParent:GET-CHILD(hNodeRef, ix).
        IF hNoderef:NAME = "#text" THEN
            MESSAGE "Text: " hNodeRef:NODE-VALUE.
        ELSE
            MESSAGE "Node name: " hNodeRef:NAME.
        RUN getPeople(hNodeRef, (iLevel + 1)).
    END.
    DELETE OBJECT hNoderef.
END PROCEDURE.
FUNCTION inWait RETURNS LOGICAL:
    RETURN stillWaiting.
END.
```
example10.p

/* Publishes an XML message. */
DEFINE VARIABLE hPubSubSession AS HANDLE NO-UNDO.
DEFINE VARIABLE hMesg AS HANDLE NO-UNDO.
DEFINE VARIABLE cPerson AS CHARACTER NO-UNDO.
DEFINE VARIABLE ix AS INTEGER NO-UNDO.
RUN jms/pubsubsession.p PERSISTENT SET hPubSubSession
  ("-H localhost -S 5162 ").
RUN setBrokerURL IN hPubSubSession ("localhost:2506").
RUN beginSession IN hPubSubSession.
RUN createXMLMessage IN hPubSubSession (OUTPUT hMesg).
/* Creates an XML message with 100 people. */
RUN appendText IN hMesg('<?xml version="1.0" ').
RUN appendText IN hMesg("encoding='ISO8859-1' ?>")..
/* Create personnel node */
RUN appendText IN hMesg("<personnel>").
REPEAT ix = 1 TO 100:
  ASSIGN
cPerson = "<person>"
cPerson = cPerson + "<name>"
cPerson = cPerson + "<family>SecondName</family>"
cPerson = cPerson + "<given>FirstName" + STRING(ix) + "</given>"
cPerson = cPerson + "</name>"
cPerson = cPerson + "<email>myEmail@subpargolf.com</email>"
cPerson = cPerson + "</person>".
  RUN appendText IN hMesg (cPerson).
END.
RUN appendText IN hMesg ("</personnel>").
RUN publish IN hPubSubSession ("people", hMesg, ?, ?, ?).
RUN deleteMessage IN hMesg.
RUN deleteSession IN hPubSubSession.

Publishing, subscribing, and receiving an XML document in a BytesMessage

The procedures example12.p and example13.p use a MEMPTR variable to publish and receive an XML document in a BytesMessage to prevent code-page conversions. The code pages of the document and the OpenEdge client do not have to match.

To run example12.p and example13.p:

1. Run example13.p to subscribe and receive a BytesMessage containing an XML document, as shown:
example13.p

/* Receives an XML document in a Bytes message. */
DEFINE VARIABLE hPubSubSession AS HANDLE NO-UNDO.
DEFINE VARIABLE msgConsumer1 AS HANDLE NO-UNDO.
DEFINE VARIABLE stillWaiting AS LOGICAL NO-UNDO INITIAL TRUE.
RUN jms/pubsubsession.p PERSISTENT SET hPubSubSession
("-H localhost -S 5162 ").
RUN setBrokerURL IN hPubSubSession ("localhost:2506").
RUN beginSession IN hPubSubSession.
RUN createMessageConsumer IN hPubSubSession
(THIS-PROCEDURE, "messageHandler", OUTPUT msgConsumer1).
RUN subscribe IN hPubSubSession
("xmlTopic", ?, /* not a durable subscription */
?, /* no message selector. */
FALSE, /* no local events */
msgConsumer1) NO-ERROR.
RUN startReceiveMessages IN hPubSubSession.
RUN waitForMessages IN hPubSubSession ("inWait", THIS-PROCEDURE, ?).
RUN deleteSession IN hPubSubSession.

PROCEDURE messageHandler:
DEFINE INPUT PARAMETER hMessage AS HANDLE NO-UNDO.
DEFINE INPUT PARAMETER hMessageConsumer AS HANDLE NO-UNDO.
DEFINE OUTPUT PARAMETER hReply AS HANDLE NO-UNDO.
DEFINE VARIABLE mDoc AS MEMPTR NO-UNDO.
DEFINE VARIABLE hDoc AS HANDLE NO-UNDO.
DEFINE VARIABLE hRoot AS HANDLE NO-UNDO.
mDoc = DYNAMIC-FUNCTION('getMemptr':U IN hMessage).
CREATE X-DOCUMENT hDoc.
CREATE X-NODEREF hRoot.
hDoc:LOAD("memptr", mDoc, FALSE).
hDoc:Get-DOCUMENT-ELEMENT(hRoot).
RUN GetChildren(hRoot, 1).
RUN deleteMessage IN hMessage.
stillWaiting = FALSE.
END PROCEDURE.

PROCEDURE GetChildren:
DEFINE INPUT PARAMETER hParent AS HANDLE NO-UNDO.
DEFINE INPUT PARAMETER iLevel AS INTEGER NO-UNDO.
DEFINE VARIABLE hNodeRef AS HANDLE NO-UNDO.
DEFINE VARIABLE ix AS INTEGER NO-UNDO.
CREATE X-NODEREF hNodeRef.
REPEAT ix = 1 TO hParent:NUM-CHILDREN.
hParent:GET-CHILD(hNodeRef, ix).
IF hNodeRef:NAME = "#text" THEN
  MESSAGE "Node text: " hNodeRef:NODE-VALUE.
ELSE
  MESSAGE "Node name: " hNodeRef:NAME.
RUN GetChildren(hNodeRef, (iLevel + 1)).
END.
DELETE OBJECT hNodeRef.
END PROCEDURE.
FUNCTION inWait RETURNS LOGICAL:
RETURN stillWaiting.
END.

2. Run example12.p to publish the BytesMessage containing an XML document, as shown:
XML code page encoding

OpenEdge applications work with the built-in XML parser. It is important to consider the code page encoding of XML messages. In principle, XML documents can be encoded with any code page. However, XML parsers support some or all code pages, and XML parsers also differ with respect to the code page conversions that they support.

OpenEdge clients set and get XML text using the ABL CHARACTER data type. CHARACTER data is encoded by the ABL interpreter according to the internal code page (the -cpinternal startup parameter). The ABL–JMS implementation automatically converts the text to Unicode when it is sent to the JMS server, and from Unicode to the internal client's code page when the text is sent from the server to the client.

In general, when the characters used by the XML document are from the 7-byte ASCII subset, there are no issues the ABL programmer has to consider. Otherwise, observe the following examples and guidelines in the following examples.

Code page example 1

In this example, two OpenEdge clients use the ISO8859-1 code page:

- Client1 sets XML text in an XMLMessage and sends it.
- Client2 receives the message, extracts the text, stores it in a MEMPTR variable, and creates an XML document. (See Publishing, receiving, and parsing an XMLMessage on page 342.)
The following code-page conversions take place:

1. ISO8859-1 (client1) to Unicode (SonicMQ XMLMessage)
2. Unicode (SonicMQ XMLMessage) to ISO8859-1 (client2)

In this example, the XML parser parses the XML document correctly if:

1. The header of the document specifies that the encoding is ISO8859-1.
2. The parser can handle ISO8859-1.

### Code page example 2

In this example, two OpenEdge clients use ISO8859–1 for their internal code page. Client1 saves a UTF–8 encoded XML document in a MEMPTR variable (calling the X–DOC:SAVE() ABL method) and then uses the ABL GET–STRING statement to extract the text from the MEMPTR and pass it into the XMLMessage. (This is a deliberate error.) UTF–8 (Unicode Transformation Format) is an 8-bit encoding form that serializes a Unicode scalar value as a sequence of one to four bytes.

An OpenEdge client cannot mix code pages. The text it sets in the XMLMessage must be encoded in the same code page as the client’s internal code page. In general, a MEMPTR variable must be used carefully, since it can have any data in it. The ABL programmer must be sure that it contains only NULL free text (no embedded NULL bytes), encoded with the same code page as the internal code page, before loading it into an XMLMessage.

In this example, if the OpenEdge client cannot be started up with –cpinternal UTF–8, but still wants to use ABL–JMS to pass that UTF–8 document, it can use a BytesMessage or bytes elements in a StreamMessage. When sent as bytes, the XML data will get to the receiver uninterpreted and unconverted. The ABL receiver can then set the data in a MEMPTR variable and load the parser.

A second option is to convert the text (and the document's header) to ISO8859–1 using the CODEPAGE–CONVERT ABL function. However, if –cpinternal represents all character, the conversion is automatic if you use LONGCHAR or CHAR. If –cpinternal represents all characters, the conversion is also automatic when you use the new built-in XML routines (SAX–WRITER or setX–Document). When you use the new built-in XML routines, you can create, send, and receive UTF-8 XML documents.

If the ABL receiver of an XMLMessage is unsure about the XML header encoding declaration, it must check it and perhaps modify it to match its internal code page before loading the parser.

### Publishing, subscribing, and receiving the customer table in a StreamMessage

The procedures example14.p and example1.5p use RAW transfer to publish, subscribe, and receive the customer table in a StreamMessage. The procedure example14.p publishes the customer table in a StreamMessage; each customer record is a bytes item. The procedure example15.p subscribes and receives the customer table in a StreamMessage; each customer record is a bytes item.

To run Examples 14 and 15:

2. Run example15.p so the subscriber is running before you publish, as shown:
example15.p

```
/* Receives the customer table in a Stream message. */
DEFINE VARIABLE hPubSubSession AS HANDLE NO-UNDO.
DEFINE VARIABLE msgConsumer1 AS HANDLE NO-UNDO.
DEFINE TEMP-TABLE ttCustomer NO-UNDO LIKE customer.
RUN jms/pubsubsession.p PERSISTENT SET hPubSubSession
   ("-H localhost -S 5162 ").
RUN setBrokerURL IN hPubSubSession ("localhost:2506").
RUN beginSession IN hPubSubSession.
RUN createMessageConsumer IN hPubSubSession
   (THIS-PROCEDURE, "messageHandler", OUTPUT msgConsumer1).
RUN subscribe IN hPubSubSession
   ("topic1",
    ?, /* not a durable subscription */
    ?, /* no message selector */
    FALSE, /* no local events */
    msgConsumer1).
RUN startReceiveMessages IN hPubSubSession.
WAIT-FOR u1 OF THIS-PROCEDURE.
FOR EACH ttCustomer:
   DISPLAY ttCustomer WITH 2 COLUMN.
END.
RUN deleteSession IN hPubSubSession.
PROCEDURE messageHandler:
   DEFINE INPUT PARAMETER hMessage AS HANDLE NO-UNDO.
   DEFINE INPUT PARAMETER hMessageConsumer AS HANDLE NO-UNDO.
   DEFINE OUTPUT PARAMETER hReply AS HANDLE NO-UNDO.
   DEFINE VARIABLE rawCust AS RAW NO-UNDO.
   DO WHILE NOT DYNAMIC-FUNCTION('endOfStream' IN hMessage):
      DYNAMIC-FUNCTION('moveToNext':U IN hMessage).
      rawCust = DYNAMIC-FUNCTION('readBytesToRaw':U IN hMessage).
      RAW-TRANSFER rawCust TO ttCustomer.
      RELEASE ttCustomer.
   END.
RUN deleteMessage IN hMessage.
APPLY "U1" TO THIS-PROCEDURE.
END PROCEDURE.
```

3. Run example14.p, as shown:

example14.p

```
/* Publishes the customer table in a Stream message. */
DEFINE VARIABLE hMesg AS HANDLE NO-UNDO.
DEFINE VARIABLE hPubSubSession AS HANDLE NO-UNDO.
DEFINE VARIABLE rawCust AS RAW NO-UNDO.
RUN jms/pubsubsession.p PERSISTENT SET hPubSubSession
   ("-H localhost -S 5162 ").
RUN setBrokerURL IN hPubSubSession ("localhost:2506").
RUN beginSession IN hPubSubSession.
RUN createStreamMessage IN hPubSubSession (OUTPUT hMesg).
FOR EACH customer NO-LOCK:
   RAW-TRANSFER customer TO rawCust.
   RUN writeBytesFromRaw IN hMesg(rawCust).
END.
RUN publish IN hPubSubSession ("topic1", hMesg, ?, ?, ?,).
RUN deleteMessage IN hMesg.
RUN deleteSession IN hPubSubSession.
```
Publishing and receiving a group of messages in a transaction

The procedures example22.p and example23.p publish and receive a group of messages in a single transaction. The procedure example22.p creates a session that is transacted for sending, and The procedure example23.p creates a session that is transacted for receiving.

To publish and receive a group of messages in a transaction:

1. Run example23.p so the subscriber is running before you publish, as shown:

```plaintext
example23.p
/* Subscribes and receives three messages in a single transaction. */
DEFINE VARIABLE hConsumer AS HANDLE NO-UNDO.
DEFINE VARIABLE hPubSubSession AS HANDLE NO-UNDO.
DEFINE VARIABLE iMsgNum AS INTEGER NO-UNDO.
/* Creates a transaction for receiving session. */
RUN jms/pubsubsession.p PERSISTENT SET hPubSubSession
  ("-H localhost -S 5162 ").
RUN setBrokerURL IN hPubSubSession ("localhost:2506").
RUN setTransactedReceive IN hPubSubSession.
RUN beginSession IN hPubSubSession.
/* Subscribe to the TestTopic topic. Messages are handled by the
"msgHandler" internal procedure. */
RUN createMessageConsumer IN hPubSubSession
  (THIS-PROCEDURE, /* this procedure will handle it */
   "msgHandler", /* name of internal procedure */
   OUTPUT hConsumer).
RUN subscribe IN hPubSubSession
  ("TestTopic", /* name of topic */
  ?, /* subscription is not durable */
  ?, /* no message selector */
  FALSE, /* want my own messages too */
  hConsumer). /* handles the incoming messages*/
/* Start receiving messages */
RUN startReceiveMessages IN hPubSubSession.
/* Wait to receive the three messages. */
WAIT-FOR u1 OF THIS-PROCEDURE.
PROCEDURE msgHandler:
  DEFINE INPUT PARAMETER hMessage AS HANDLE NO-UNDO.
  DEFINE INPUT PARAMETER hMsgConsumer AS HANDLE NO-UNDO.
  DEFINE OUTPUT PARAMETER hReply AS HANDLE NO-UNDO.
  /* Display the message - we assume that reply is not required. */
  DISPLAY "Message text: ",
    DYNAMIC-FUNCTION('getText':U IN hMessage) FORMAT "x(70)".
RUN deleteMessage IN hMessage.
iMsgNum = iMsgNum + 1.
/* Commit the reception of the three messages. */
IF iMsgNum = 3 THEN DO:
  RUN commitReceive IN hPubSubSession.
  MESSAGE "committed!".
  APPLY "U1" TO THIS-PROCEDURE.
END.
END PROCEDURE.
```

2. Run example22.p to subscribe and receive messages from example22.p in a single transaction, as shown:
/* Publishes A group of Text messages in a single transaction. */
DEFINE VARIABLE hMessage AS HANDLE NO-UNDO.
DEFINE VARIABLE hPubSubSession AS HANDLE NO-UNDO.
/* Creates a transcated for sending session. */
RUN jms/pubsubsession.p PERSISTENT SET hPubSubSession
  ("-H localhost -S 5162 ").
RUN setBrokerURL IN hPubSubSession ("localhost:2506").
RUN setTransactedSend IN hPubSubSession.
RUN beginSession IN hPubSubSession.
/* Create a text message */
RUN createTextMessage IN hPubSubSession (OUTPUT hMessage).
/* Publish three messages */
RUN setText IN hMessage ("message1").
RUN publish IN hPubSubSession ("TestTopic", hMessage, ?, ?, ?).
RUN setText IN hMessage ("message2").
RUN publish IN hPubSubSession ("TestTopic", hMessage, ?, ?, ?).
RUN setText IN hMessage ("message3").
RUN publish IN hPubSubSession ("TestTopic", hMessage, ?, ?, ?).
/* Commit the publication of the messages. */
RUN commitSend IN hPubSubSession.
RUN deleteMessage IN hMessage.
RUN deleteSession IN hPubSubSession.

Installing an error handler to handle an asynchronous error

The procedure example16.p installs an error handler to detect a JMS server communication loss.

To install an error handler to handle an asynchronous error:

1. Run example16.p, as shown:
example16.p

```pascal
/* Installs an error handler to deal with a JMS server communication loss. */
DEFINE VARIABLE errorConsumer AS HANDLE NO-UNDO.
DEFINE VARIABLE hPubSubSession AS HANDLE NO-UNDO.
DEFINE VARIABLE jmsIsOk AS LOGICAL NO-UNDO INITIAL TRUE.
RUN jms/pubsubsession.p PERSISTENT SET hPubSubSession
    ("-H localhost -S 5162 ").
RUN setBrokerURL IN hPubSubSession ("localhost:2506").
RUN beginSession IN hPubSubSession.
RUN createMessageConsumer IN hPubSubSession (THIS-PROCEDURE, "errorHandler", OUTPUT errorConsumer).
RUN setErrorHandler IN hPubSubSession (errorConsumer).
RUN startReceiveMessages IN hPubSubSession.
/* Wait forever for messages until the connection with the JMS server is lost with error code "-5" (shutdown the SonicMQ Broker to simulate that). */
RUN waitForMessages IN hPubSubSession ("inWait", THIS-PROCEDURE, ?).
IF NOT jmsIsOk THEN DO:
    MESSAGE "Disconnected from JMS Server..." VIEW-AS ALERT-BOX.
    RUN deleteSession IN hPubSubSession.
END.
FUNCTION inWait RETURNS LOGICAL:
    RETURN jmsIsOk.
END.
PROCEDURE errorHandler:
    DEFINE INPUT PARAMETER hMessage AS HANDLE NO-UNDO.
    DEFINE INPUT PARAMETER hMessageConsumer AS HANDLE NO-UNDO.
    DEFINE OUTPUT PARAMETER hReply AS HANDLE NO-UNDO.
    DEFINE VARIABLE errorCode AS CHARACTER NO-UNDO.
    DEFINE VARIABLE errorText AS CHARACTER NO-UNDO.
    ASSIGN
        errorCode = DYNAMIC-FUNCTION
            ("getCharProperty":U IN hMessage, "errorCode")
        errorText = DYNAMIC-FUNCTION("getText":U IN hMessage).
    RUN deleteMessage IN hMessage.
    MESSAGE errorText errorCode VIEW-AS ALERT-BOX.
    IF errorCode = ",-5" THEN
        jmsIsOk = FALSE.
    END.
END.
```

2. Shut down the SonicMQ Broker to simulate the communication loss.

**Installing an error handler for synchronous errors**

The procedure example17.p publishes a `TextMessage` to a nonexistent topic and handles the error conditions, as shown:
example17.p

```plaintext
/* Publishes A Text message to an illegal topic name and handles the
error conditions. */
DEFINE VARIABLE hPubSubSession AS HANDLE NO-UNDO.
DEFINE VARIABLE hMessage AS HANDLE NO-UNDO.
DEFINE VARIABLE lSuccess AS LOGICAL NO-UNDO.
/* Creates a session object. */
RUN jms/pubsubsession.p PERSISTENT SET hPubSubSession
   ("-H localhost -S 5162 ").
RUN setNoErrorDisplay IN hPubSubSession (true).
RUN setBrokerURL IN hPubSubSession ("localhost:2506").
RUN beginSession IN hPubSubSession.
/* Create a text message */
RUN createTextMessage IN hPubSubSession (OUTPUT hMessage).
RUN setText IN hMessage ("Golf shoes on sale today.").
/* Publish the message on the illegal '*' topic */
DO ON ERROR UNDO, LEAVE:
   RUN publish IN hPubSubSession ("*", hMessage, ?, ?, ?).
   lSuccess = TRUE.
END.
If NOT lSuccess THEN
   MESSAGE "Failed to publish to topic ": " RETURN-VALUE VIEW-AS ALERT-BOX.
RUN deleteMessage IN hMessage.
RUN deleteSession IN hPubSubSession.
```

PTP messaging examples

The PTP examples consist of a sender and a receiver, and each set should run together. Note that queues
cannot be generated on the fly by the clients; queues must be created using the administration tool of the
SonicMQ Broker.

These examples include:

- Sending a message to a queue and receiving a message from a queue on page 352
- Achieving scalable server architecture with PTP queuing on page 354

Sending a message to a queue and receiving a message from a queue

The procedures example18.p and example19.p send and receive a message from a queue.

To send a message to a queue and receive a message from a queue:

1. Create the GolfQueue queue using the SonicMQ Explorer. (See SonicMQ Programming Guide for
   information about creating queues.)
2. Run example18.p to send a TextMessage to the GolfQueue, as shown:
/* Sends A Text message to a queue. */
DEFINE VARIABLE hMessage AS HANDLE NO-UNDO.
DEFINE VARIABLE hPTPSession AS HANDLE NO-UNDO.
/* Creates a session object. */
RUN jms/ptpsession.p PERSISTENT SET hPTPSession
("-H localhost -S 5162 ").
RUN setBrokerURL IN hPTPSession ("localhost:2506").
RUN beginSession IN hPTPSession.
/* Create a text message */
RUN createTextMessage IN hPTPSession (OUTPUT hMessage).
RUN setText IN hMessage ("Golf shoes on sale today.").
/* Sends the message to the "GolfQueue" queue */
RUN sendToQueue IN hPTPSession ("GolfQueue", hMessage, ?, ?, ?).
RUN deleteMessage IN hMessage.
RUN deleteSession IN hPTPSession.

3. Run example19.p to receive a message from the GolfQueue, as shown:

/* Receives a Text message from a queue. */
DEFINE VARIABLE hConsumer AS HANDLE NO-UNDO.
DEFINE VARIABLE hPTPSession AS HANDLE NO-UNDO.
/* Creates a session object. */
RUN jms/ptpsession.p PERSISTENT SET hPTPSession
("-H localhost -S 5162 ").
RUN setBrokerURL IN hPTPSession ("localhost:2506").
RUN beginSession IN hPTPSession.
/* GolfQueue Messages are handled by the "golfHandler" procedure. */
RUN createMessageConsumer IN hPTPSession
(THIS-PROCEDURE, /* this procedure will handle it */
"golfHandler", /* name of internal procedure */
OUTPUT hConsumer).
RUN receiveFromQueue IN hPTPSession
("GolfQueue", /* name of queue */
?, /* no message selector */
hConsumer). /* handles incoming messages*/
/* Start receiving messages */
RUN startReceiveMessages IN hPTPSession.
/* Wait to receive the messages. Any other I/O-blocked statements can be
used for receiving messages. */
WAIT-FOR u1 OF THIS-PROCEDURE.
PROCEDURE golfHandler:
    DEFINE INPUT PARAMETER hMessage AS HANDLE NO-UNDO.
    DEFINE INPUT PARAMETER hMsgConsumer AS HANDLE NO-UNDO.
    DEFINE OUTPUT PARAMETER hReply AS HANDLE NO-UNDO.
    /* Display the message - we assume that reply is not required. */
    DISPLAY "Message text: ",
        DYNAMIC-FUNCTION('getText':U IN hMessage) FORMAT "x(70)".
    RUN deleteMessage IN hMessage.
    APPLY "U1" TO THIS-PROCEDURE.
END PROCEDURE.
Achieving scalable server architecture with PTP queuing

The procedures example20p and example21.p use PTP queuing to achieve scalable server architecture. Several instances of example20.p send requests to a single JMS queue and receive replies from servers that run example21.p. You can add more instances to handle an increasing volume of requests.

To run example20.p and example21.p:

1. Create the requestQueue queue using the SonicMQ Explorer.
2. Run example20.p to send requests to the requestQueue queue, as shown:

```asp
/* Sends a request to a queue and receives a reply from the server. */
DEFINE VARIABLE hConsumer AS HANDLE NO-UNDO.
DEFINE VARIABLE hPTPSession AS HANDLE NO-UNDO.
DEFINE VARIABLE hRequest AS HANDLE NO-UNDO.
DEFINE VARIABLE request AS CHARACTER NO-UNDO.

/* Creates a session object. */
RUN jms/ptpsession.p PERSISTENT SET hPTPSession
("-H localhost -S 5162 ").
RUN setBrokerURL IN hPTPSession ("localhost:2506").
RUN beginSession IN hPTPSession.
/* Create a text message */
RUN createTextMessage IN hPTPSession (OUTPUT hRequest).
/* Creates a consumer for the reply */
RUN createMessageConsumer IN hPTPSession
(This-PROCEDURE, /* this procedure will handle it */
"replyHandler", /* name of internal procedure */
OUTPUT hConsumer).
/* Start the reply receiving */
RUN startReceiveMessages IN hPTPSession.
/* Loop forever. */
REPEAT:
UPDATE request WITH FRAME f1 CENTERED.
RUN setText IN hRequest (request).
/* Sends a request to the requestQueue and handles the reply in the
replyHandler internal procedure. */
RUN requestReply IN hPTPSession
("requestQueue", hRequest,
?, /* No reply selector. */
hConsumer, ?, ?, ?).
/* Wait for the reply. */
WAIT-FOR u1 OF THIS-PROCEDURE.
END.
PROCEDURE replyHandler:
DEFINE INPUT PARAMETER hReply AS HANDLE NO-UNDO.
DEFINE INPUT PARAMETER hMsgConsumer AS HANDLE NO-UNDO.
DEFINE OUTPUT PARAMETER responseH AS HANDLE NO-UNDO.
/* Display the reply from the server. */
DISPLAY "reply text: "
DYNAMIC-FUNCTION('getText':U IN hReply) FORMAT "X(30)".
RUN deleteMessage IN hReply.
APPLY "U1" TO THIS-PROCEDURE.
END PROCEDURE.
```

3. Run example21.p to receive requests from the requestQueue queue, execute them, and reply to the requester, as shown:
This example implements a server who gets requests from a JMS queue, executes the request, and replies to the requester. Run several instances of this server and several instances of a client (example20) to observe the scalability of this configuration. */

DEFINE INPUT PARAMETER serverName AS CHARACTER NO-UNDO.
DEFINE VARIABLE hConsumer AS HANDLE NO-UNDO.
DEFINE VARIABLE hPTPSession AS HANDLE NO-UNDO.
DEFINE VARIABLE replyMessage AS HANDLE NO-UNDO.
/* Creates a session object. */
RUN jms/ptpsession.p PERSISTENT SET hPTPSession
("-H localhost -S 5162 ").
RUN setBrokerURL IN hPTPSession ("localhost:2506").
RUN beginSession IN hPTPSession.
/* Uses one message for all the replies. */
RUN createTextMessage IN hPTPSession (OUTPUT replyMessage).
/* receives requests from the requestQueue */
RUN createMessageConsumer IN hPTPSession
(This-PROCEDURE, /* this procedure will handle it */
"requestHandler", /* name of internal procedure */
OUTPUT hConsumer).
RUN receiveFromQueue IN hPTPSession
("requestQueue", /* request queue */
?, /* no message selector */
hConsumer). /* handles the messages */
/* Start receiving requests */
RUN startReceiveMessages IN hPTPSession.
/* Process requests forever. */
RUN waitForMessages IN hPTPSession ("inWait", THIS-PROCEDURE, ?).
PROCEDURE requestHandler:
DEFINE INPUT PARAMETER hRequest AS HANDLE NO-UNDO.
DEFINE INPUT PARAMETER hMsgConsumer AS HANDLE NO-UNDO.
DEFINE OUTPUT PARAMETER hReply AS HANDLE NO-UNDO.
/* Creates a reply message. The reply is sent automatically when 
control returns to the ABL-To-JMS implementation. */
replyText = serverName + " executed " +
DYNAMIC-FUNCTION ('getText':U IN hRequest).
RUN deleteMessage IN hRequest.
hReply = replyMessage.
RUN setText IN hReply (replyText).
END PROCEDURE.
FUNCTION inWait RETURNS LOGICAL:
RETURN true.
END.
MultiPartMessage example

The following fragment creates a MultiPartMessage:

```plaintext
/* Create a multipart message */
RUN createMultipartMessage IN hPTPSession (OUTPUT hMessage).
/* Create a Sonic text message */
RUN createTextMessage IN hPTPSession (OUTPUT messagePartH).
RUN setText IN messagePartH (cTextString).
/* Add part to multipart message */
RUN addMessagePart IN hMessage (INPUT messagePartH, INPUT contentIDString).
/* Add a memptr part */
RUN addBytesPart IN hMessage
  (memptr, contentTypeString, contentIDString).
/* Add a text part */
RUN addTextPart IN hMessage
  (memptr, msgTextString, contentTypeString, contentIDString).
```

First, the fragment creates a MultiPartMessage just as it would create any other supported message type. The createMultipartMessage procedure on page 188 returns a message handle, which supports methods for adding parts.

Next, the fragment creates a text message and adds it to the MultiPartMessage. Each message part has two main identifiers: content type and content ID. Content type identifies the type of part, while content ID identifies a particular part. Since a Sonic text message already has a content type, when the text message is added, only the content ID must be specified.

Finally, the fragment adds a bytes part, comprising an arbitrary set of bytes represented as a MEMPTR. Adding the bytes part resembles adding the text message except that the content type must also be specified.

Gateway sample application

The gateway sample application demonstrates a framework for integrating the native ABL publish and subscribe mechanism (named events) with the ABL–JMS API. (See ABL - JMS API Reference on page 161.)

The following sections describe:

- **Application files** on page 356
- **Running the sample application** on page 359

Application files

The sample application manages a set of customer records loaded from the sports.customer table. For each country, there is one instance of the application that manages the subset of customers from that country. The country is specified as an application startup parameter.

The gateway sample application consists of three files:

- **appDriver.p** — Drives the publish and subscribe gateway example
- **JMSgateway.p** — Establishes a gateway between local and remote publish and subscribe events
• customers.p — Updates customer records from a specified country while keeping the other records identical to the master copy

The main loop of the application is in appDriver.p:

1. The user specifies the Customer.Cust-num value.
2. The application finds the customer and allows the user to update the record if the Customer.Country field matches the startup country.
3. If the Customer.Country field does not match the startup country, the user can only view the customer record.
4. Several applications, each managing one country, run concurrently. Each application is connected to a JMS server through a local JMS gateway object. The goal is to keep the records identical across the different locations.
5. When an application modifies a customer record, it publishes the new record through an ABL PUBLISH CustUpdate call.
6. The local JMS gateway object subscribes to the CustUpdate event. It packs the published parameters in a JMS MapMessage and publishes it to the JMS CustUpdate topic.
7. The other JMS gateway objects subscribe to the JMS CustUpdate topic. They receive the JMS MapMessage, unpack the parameters, and publish the updated record locally through an ABL PUBLISH CustUpdate call.
8. The application picks up the updated record and updates the local copy.

The procedure appDriver.p drives the publish and subscribe gateway example, as shown:

```abla
/* appDriver.p: Drives the Pub/Sub gateway example. */
DEFINE INPUT PARAMETER cCountry AS CHARACTER NO-UNDO.
DEFINE VARIABLE hCustomers AS HANDLE NO-UNDO.
DEFINE VARIABLE hGateway AS HANDLE NO-UNDO.
DEFINE VARIABLE iCustNum AS INTEGER NO-UNDO.
/* Initialization */
RUN customers.p PERSISTENT SET hCustomers.
RUN loadCustomers IN hCustomers.
RUN JMSgateway.p PERSISTENT SET hGateway ("-H localhost -S 5162 ").
REPEAT:
    iCustNum = ?.
    UPDATE iCustNum LABEL "cust-num"
       WITH FRAME ff CENTERED TITLE "Find Customer".
    RUN updateCustInteractive IN hCustomers (iCustNum, cCountry).
END.
RUN deleteGateway IN hGateway.
```

The procedure JMSgateway.p establishes a gateway between local and remote publish and subscribe events, as shown:
/* JMSgateway.p: A gateway between local and remote Pub/Sub events. */
DEFINE INPUT PARAMETER connectionParams AS CHARACTER NO-UNDO.
/* JMS objects */
DEFINE VARIABLE hPubSubSession AS HANDLE NO-UNDO.
DEFINE VARIABLE outMessage AS HANDLE NO-UNDO.
DEFINE VARIABLE hConsumer AS HANDLE NO-UNDO.
FUNCTION bufferToRaw RETURNS RAW (bufferH AS HANDLE) FORWARD.
/* Raw Transfer Declarations */
DEFINE TEMP-TABLE ttRaw NO-UNDO
FIELD rValue AS RAW.
CREATE ttRaw.
/* Initializes the JMS server and subscribes to the CustUpdate topic */
RUN jms/pubsubsession.p PERSISTENT SET hPubSubSession (connectionParams).
RUN setBrokerURL IN hPubSubSession ("localhost:2506").
RUN beginSession IN hPubSubSession.
RUN createMapMessage IN hPubSubSession (OUTPUT outMessage).
RUN createMessageConsumer IN hPubSubSession
  (THIS-PROCEDURE, "handleRemoteEvent", OUTPUT hConsumer).
RUN subscribe IN hPubSubSession
  ("CustUpdate", /* topic name */
   ?, /* not durable */
   ?, /* no message selector */
   TRUE, /* local events */
   hConsumer).
RUN startReceiveMessages IN hPubSubSession.
/* Subscribes to local CustUpdate events */
SUBSCRIBE TO "CustUpdate" ANYWHERE RUN-PROCEDURE "handleLocalEvent".
/* Publish locally a remote message from the CustUpdate topic. */
PROCEDURE handleRemoteEvent:
  DEFINE INPUT PARAMETER hMessage AS HANDLE NO-UNDO.
  DEFINE INPUT PARAMETER hMessageConsumer AS HANDLE NO-UNDO.
  DEFINE OUTPUT PARAMETER hReply AS HANDLE NO-UNDO.
  PUBLISH "CustUpdate"
    (DATE (DYNAMIC-FUNCTION('getChar':U IN hMessage, "updateDate")),
     DYNAMIC-FUNCTION('getInt':U IN hMessage, "custNum"),
     DYNAMIC-FUNCTION('getBytesToRaw':U IN hMessage, "rawCust").
  RUN deleteMessage IN hMessage.
END PROCEDURE.
/* Publish remotely a local CustUpdate event. */
PROCEDURE handleLocalEvent:
  DEFINE INPUT PARAMETER dValue AS DATE NO-UNDO.
  DEFINE INPUT PARAMETER hCustomer AS HANDLE NO-UNDO.
  DEFINE INPUT PARAMETER iCustNum AS INTEGER NO-UNDO.
  RUN setString IN outMessage ("updateDate", STRING(dValue)).
  RUN setInt IN outMessage ("custNum", iCustNum).
  RUN setBytesFromRaw IN outMessage ("rawCust", bufferToRaw(hCustomer)).
  RUN publish IN hPubSubSession ("CustUpdate", outMessage, ?, ?, ?).
END PROCEDURE.
PROCEDURE deleteGateway:
  RUN deleteMessage IN outMessage.
  RUN deleteSession IN hPubSubSession.
  DELETE OBJECT THIS-PROCEDURE.
END PROCEDURE.
FUNCTION bufferToRaw RETURNS RAW (hBuffer AS HANDLE):
  /* Raw transfer variables */
  DEFINE VARIABLE rawBuf AS HANDLE NO-UNDO.
  DEFINE VARIABLE rawCust AS HANDLE NO-UNDO.
  ASSIGN
    rawBuf = BUFFER ttRaw:HANDLE
    rawCust = rawbuf:BUFFER-FIELD(1).
  hBuffer:RAW-TRANSFER(TRUE, rawCust).
  RETURN rawCust:BUFFER-VALUE.
END FUNCTION.
The procedure customers.p updates customer records from a specified country while keeping the other records identical to the master copy, as shown:

customers.p

/* customers.p: Manages customer records of a specified country and keeps the other records identical to the master copy. */
DEFINE VARIABLE hTTCust AS HANDLE NO-UNDO.
DEFINE VARIABLE hBuffer AS HANDLE NO-UNDO.
DEFINE TEMP-TABLE ttCustomer LIKE customer.
DEFINE BUFFER custtUpd FOR ttCustomer.
/* Getting a handle to a dynamic buffer. */
hTTCust = TEMP-TABLE ttCustomer:HANDLE.
hBuffer = hTTCust:DEFAULT-BUFFER-HANDLE.
/* Subscribes to CustUpdate events. */
SUBSCRIBE TO "CustUpdate" ANYWHERE RUN-PROCEDURE "updateCustFromRaw".
PROCEDURE loadCustomers:
  FOR EACH customer NO-LOCK:
    CREATE custt.
    BUFFER-COPY customer TO ttCustomer.
  END.
END PROCEDURE.
/* Updates a customer from the "correct" country, displays customers from other countries. */
PROCEDURE updateCustInteractive.
  DEFINE INPUT PARAMETER custNum AS INTEGER NO-UNDO.
  DEFINE INPUT PARAMETER custCountry AS CHARACTER NO-UNDO.
  FIND ttCustomer WHERE ttCustomer.custnum = custNum.
  IF ttCustomer.country = custCountry THEN DO:
    UPDATE ttCustomer WITH 2 COL.
    PUBLISH "CustUpdate" (TODAY, custNum, hBuffer).
  END.
  ELSE
    DISPLAY ttCustomer WITH 2 COL.
  END.
PROCEDURE updateCustFromRaw:
  DEFINE INPUT PARAMETER dValue AS DATE NO-UNDO.
  DEFINE INPUT PARAMETER custNum AS INTEGER NO-UNDO.
  DEFINE INPUT PARAMETER rawCust AS RAW NO-UNDO.
  FIND custtUpd WHERE custtUpd.custnum = custNum.
  RAW-TRANSFER rawCust TO custtUpd.
  MESSAGE custNum VIEW-AS ALERT-BOX TITLE "customer updated".
END PROCEDURE.

Running the sample application

The following steps describe how to run the gateway sample application.

To run the gateway sample application:

1. Start the JMS server and the OpenEdge Adapter for SonicMQ BrokerConnect.
2. Start two or more OpenEdge clients. Each OpenEdge client calls, as shown:

```
RUN appDriver.p country-name
```

Each instance should be connected to the Sports database and should start up with a different country.
3. Update a customer record with one client and watch the others display an ALERT-BOX with the cust-num field of the modified customer.

4. Display the modified customer record at each application instance. All the copies are identical.
Sample Native Invocation ESB process

This appendix demonstrates the creation and testing of a simple ESB process using the Native Invocation methodology. In this example, the ABL procedure GetCustName.p is called from an ESB process, providing a customer number as input, and returning the customer name retrieved from a sports2000 database.

This example relies on the default settings. Consult your Sonic documentation for details on creating more complex ESB processes that do not rely on defaults.

For details, see the following topics:

• Develop ABL source
• Configure OpenEdge servers
• Annotate source with OpenEdge Architect
• Build Native Invocation and r-code files
• Create ESB process in Sonic Workbench
• Test ESB process
Develop ABL source

The first step in developing this example is to write the ABL procedure GetCustName.p. The procedure requires an INTEGER parameter as input and a CHARACTER parameter as output. The code is as follows:

```
DEF INPUT PARAM customerNumber AS INTEGER.
DEF OUTPUT PARAM customerName AS CHAR.
IF CONNECTED("sports2000") THEN DO:
    FIND FIRST customer WHERE custNum = customerNumber NO-LOCK NO-ERROR.
    IF AVAILABLE customer THEN
        customerName = Name.
    ELSE
        customerName = "No record".
    MESSAGE "CustomerName = " customerName SKIP.
END.
```

Save the .p file.

Configure OpenEdge servers

A successful compile of your source requires a database connection. Successful testing of your ESB process requires an AppServer connection and a database connection. Start a database server and AppServer using Progress Explorer.

Start your database first, then configure your AppServer broker, esbbroker1, to connect its agents to the database, as shown:

Once configured, start esbbroker1.
Annotate source with OpenEdge Architect

Annotate your source in OpenEdge Architect to provide the information required to create your native invocation file.

Note: There are other methods of creating native invocation files, such as ProxyGen, but creating them from annotated source is the preferred method. See ProxyGen section for more details.

To annotate your source:

1. Start OpenEdge Architect and open your source file.

   Note: Your source file must be in a project. For information on projects, see the OpenEdge Architect online help.

2. Right-click in the source window and select Source > Add Annotation, as shown:

3. The Add Annotation dialog box appears:
4. Select ESB Annotation - Main annotation from the drop-down in the Select annotation or enter annotation in text editor section, and then select your file.

5. Click Finish to add the annotation. The annotated source follows:

```plaintext
@openapi.openedge.export FILE(type="ESB", esboeFilename="%FILENAME%", useReturnValue="false", writeDataSetBeforeImage="false", executionMode="external").
DEF INPUT PARAM customerNumber AS INTEGER.
DEF OUTPUT PARAM customerName AS CHAR.
IF CONNECTED("sports2000") THEN DO:
  FIND FIRST customer WHERE custNum = customerNumber NO-LOCK NO-ERROR.
  IF AVAILABLE customer THEN
    customerName = Name.
  ELSE
    customerName = "No record".
  MESSAGE "CustomerName = " customerName SKIP.
END.
```
Build Native Invocation and r-code files

Once you have annotated your source, you build your native invocation (.esboe) file and r-code.

To build the .esboe file and r-code:

1. From the main menu chose OpenEdge > Admin > Database Administration to start Database Administration. Connect to your Sports2000 database.

2. From the Resources tab, select your procedure file. Right-click and select OpenEdge > Generate Sonic ESB Invocation Files, as shown:
OpenEdge compiles your source, saving it to r-code, and generates the native invocation file. Success is indicated by the following output in the **Console** window:

```
>GetCustName (C:\OE_work_dir\workspace\ESB_process_example\GetCustName.esboe)
>GetCustName.p (C:\OE_work_dir\workspace\ESB_process_example\GetCustName.r)
```

Proceed to Sonic Workbench to complete this example.

---

**Create ESB process in Sonic Workbench**

Once you have developed your ABL procedure and created a native invocation file, you create an ESB process in Sonic workbench.

To create an ESB process:

1. Start your Sonic Domain Manager if it is not already running.
2. Start Sonic Workbench.
3. Create a new project or open an existing project, then create a new ESB process by selecting **File > New > ESB Process**. The **New ESB Process** dialog box opens:

![New ESB Process dialog box](image)

- **Enter or select the parent folder:**
  - Sample ESB_process

- **File name:**
  - Get_cust_name

- **Advanced >>**
  - [ ] Wrap an existing ESB process as a web service operation.
4. Name your process and click **Finish**. An empty process appears:

![Sonic Workbench interface](image)

5. Select **File > Import**, then select **File System** and click **Next** to open the **Import** dialog box. Browse to the directory where your native invocation file resides. Select your invocation file and specify your project in the **Into folder** field as shown:
6. Click Finish to import your native invocation file.

7. From the Pallete tab, select OpenEdge Native Services. Add the template to the process and add a name, as shown:
8. Add the native invocation file to the process by dragging the file from the Navigator window onto the process.

9. If you wish to examine the process details, right click on the service and select Open. The details appear, as shown:
10. Save your process, and chose Yes when prompted to upload the process after saving.

**Test ESB process**

Your example process is now created and ready to be tested.

To test your ESB process:

1. In Sonic Workbench, switch to the **Container** tab and start the OpenEdge development container `dev_OpenEdgeTest`, as shown:
2. Create a test scenario by clicking **Create Scenario**. The **Create/Edit Scenario** dialog box appears:
3. Create the scenario by:
   
a) Giving the scenario a useful name in the **Scenario Name** field.
   
b) Specifying **Literal** in the **File/Literal** column for the default input.
   
c) Specifying an integer in the **Scenario Test Value** column (1 in this example).
   
d) Click **OK** to save the scenario.

4. Run the scenario by clicking the run icon next to the scenario. The results of running the scenario appear in the **Output** window:

   In this scenario, a value of "1" was sent to the ABL procedure GetCustName, and it returned "Lift Tours", demonstrating a successful execution of the ABL procedure by the Sonic ESB process.
Index

4GL
   Publish and Subscribe mechanism 356

A

ABL
   Native invocation drag and drop 146
   object model, ABL–JMS 36
   publish-and-subscribe mechanism 53
   transactions 102
ABL transactions 102
ABL-JMS object model
   Message Consumer objects 37
   Message objects 37
   Session objects 37
Accessing message handler
   methods 97
Accessing message header properties 94
Accessing message properties 95
Accessing messages 85
acknowledgeAndForward procedure 177
Acknowledgement of messages
   automatic 102
   commitReceive method 183
   lazy 102
   preventing 103
Adapter
   defined 35
   Understanding how it works 35
addBytesPart procedure 178
addMessagePart procedure 178
addTextPart procedure 179
Administered objects, JMS
   and OpenEdge Adapter for SonicMQ 82
AdminServerPlugins.properties file
   jvmArgs property 91
Appending text 85
appendText procedure 180
Application files gateway sample 356
AppServer
   session models 159
Asynchronous
   ASYNC completion procedures 99
   call events 99
   conditions 104
   error handling 90, 106
   error messages 37, 96
   error reporting 103
   reply 73, 264
Asynchronous error example 350
Automatic message acknowledgment 102

B
beginSession procedure 180
boolean data type
   mapped to ABL 91
Broker URL property 270
browseQueue procedure 181
Browsing
   messages on queue 97
byte array data type
   mapped to ABL 91
byte data type
   mapped to ABL 91
Bytes correlation ID, JMS
   getting 219
BytesMessage
   example 344
   handling 88
   parse 344
   Process BytesMessage 344
   publish and subscribe 344
   receive 344
   XML document 344

C
cancelDurableSubscription procedure 182
char data type
   mapped to ABL 91
CHARACTER data type 91, 346
Character limit
   text 85
Characters
   code-page encoding 89
clearBody 93
clearBody procedure 182
Clearing message body 94
Clearing message properties 94–95
Clearing messages 94
clearProperties 93
clearProperties procedure 183
Client ID
   identifier 319
   setting 273
Client persistence 70–71
Client persistence example 117
Code page example 346
commitReceive procedure 183
commitSend procedure 184
Comparing PTP and Pub/Sub 53
Concatenating text 85
Index

Condition handling 103
Configuring and administering the Generic JMS Adapter 41
Connection attributes 70
Connection options 67
Connection parameter examples 67
Connections
  attributes 70, 205
  failures 106
Consuming 102
Consuming messages
  creating 96
  handler 97
  Message acknowledgement
    automatic 102
    preventing 103
    single 103
  Message recovery 102–103
  processing states 99
  reception issues 98–99
  Reply mechanisms 100
  Reply properties 97
  scope 99
  synchronous reception 100
  Transacted sessions
    illegal calls 102
    receiving 102
    recover 102
Correlation ID
  getting 218
createBytesMessage procedure 184
createChangeStateConsumer procedure 185
createDataSetMessage procedure 186
createHeaderMessage procedure 186
createMapMessage procedure 186
createMessageConsumer procedure 187
createMultipartMessage procedure 188
createRejectedMessageConsumer procedure 188
createStreamMessage procedure 189
createTemporaryQueue procedure 190
createTemporaryTopic procedure 190
createTempTableMessage procedure 191
createTextMessage procedure 191
createXMLMessage procedure 192
Creating
  Message Consumers 37, 187
  message headers 94
  messages 93
  publishers 94
  Session objects 37, 65, 110, 113, 116
  subscribers 98
  TextMessage 85
  XMLMessage 89
Creating Message Consumer 96
Creating message handler 97
Creating messages 85

D
Data extraction methods
  Java and ABL 91
Data types
  CHARACTER 91, 346
  extraction in Java and ABL 91
  storage in Java and ABL 91
  in StreamMessage 87
  INTEGER 91
  LOGICAL 91
  MEMPTR 88
  RAW 88
DataSetMessage
  handling 90
DATE data type 91
Date values
  in map messages 217
  reading from stream messages 253
DATETIME data type 91
DATETIME-TZ data type 91
DECIMAL data type 91
deleteConsumer procedure 192
deleteMessage procedure 193
deleteSaxWriter procedure 193
deleteSession procedure 194
deleteTemporaryQueue procedure 194
deleteTemporaryTopic procedure 195
Deleting
  data from message body 93
  durable subscriptions 99
  Message Consumer 98
  Message Consumer objects 99
  replies 100
  sessions 67
  temporary destination 73, 264
Deleting Message Consumer 98
Deleting messages 94
Delivery parameters 72
Deployment
  serialized connection objects 74, 77
Destination
  temporary 73, 264
  -DirectConnect connection parameter 67
Disabling calls to non-supported APIs 46
Discardable messages 73
domain unification 116
double data type
  mapped to ABL 91
Durable subscriptions
  deleting 99

E
Encoding XML example 346
endOfStream function 195
Enhanced XML support 118
Error handling 30
Error handler example 350–351
Error handling SonicMQ Adapter 350–351
Errors handling 69, 103
synchronously reported 104
ESBOE file 134
ESBOEGEN utility 153
Example BytesMessage 344
Example gateway sample
application files 356
running 359
Example Multipart message 356
Example StreamMessage 347
Example TextMessage
asynchronous error 350
error handler 350–351
synchronous error 351
Exchanging messages 36
Expiration time
of messages, getting 220

F
Fault tolerance
Replicated brokers 72
Fault tolerance example 131
float data type
mapped to ABL 91
Functions, ABL
getCharCount 85
getJMSReplyTo 100
getLongText 85
getMemptr 88
getPropertyNames 105
getReplyToDestinationType 100
getText 85
ggetTextSegment 85
moveToNext 253
readBytesToRaw 88
readChar 91

G
Gateway model
ABL publish and subscribe 53
Gateway sample application
application files 356
running 359
Gateway sample application files 356
Gateway sample running 359
generic JMS adapter
locating the connection factory 43

Generic JMS adapter
prerequisites 33
updating the jmsProvider.properties File 42
updating the jvmargs property 42
getAdapterService function 196
getApplicationContext function 197
getBrokerURL function 197
getBytesCount function 198
getBytesPartByID function 198
getBytesPartByIndex function 199
getBytesToRaw function 200
getChar function 200
getCharCount function 201
getCharProperty function 201
clientID function 202
clientPersistence function 202
getClientTransactionBufferSize function 203
getConnectionID function 204
getConnectionMetaData function 205
getConnectionURLs function 205
dataSet function 206
getDateTime function 207
getDateTimeProperty function 207
getDateTimeTime function 208
getDateTimeTimeZone function 209
getDateTimeTimeProperty function 209
getDateTimeTimeZoneProperty function 210
getDecimal function 210
getDecimalProperty function 211
ggetDefaultPersistency function 211
ggetDefaultPriority function 212
ggetDefaultTimeToLive function 212
gestinationName function 213
getFaultTolerant function 213
getFaultTolerantReconnectTimeout function 214
getFlowToDisk function 214
getAddressionTimeout function 217
getInt function 215
getInt64 function 216
getIntlProperty function 215
getItemType function 217
getJMSCorrelationID function 218
getJMSCorrelationIDAsBytes function 219
getJMSDeliveryMode function 219
getJMSDestination function 220
getJMSExpiration function 220
getJMSMessageID function 221
getJMSPriority function 221
getJMSRedelivered function 222
getJMSReplyTo function 222
getJMSQueueName function 223
getJSTimestamp function 223
getJSTime function 224
getLoadBalancing function 224
getLocalStoreDirectory function 225
getLocalStoreSize function 225
Load balancing 70
Locating connection factory
   using the AdminServerPlugins.properties file 43
locating the connection factory 43
Locating the connection factory
   using JNDI 43
LOGICAL data type 91
long data type
   mapped to ABL 91
LONGCHAR data type 91, 180, 246

M

Managing connections
   examples 67
      option 67
      setting and getting 70
      starting 70
Managing session
   Client persistence 71
   delivery parameters 72
Managing sessions
   Client persistence 70
   Creating multiple sessions 66
   Creating session 66
Deleting sessions 67
   delivery parameters 72
   discardable 73
Fault tolerance 71–72
Load balancing 70
Message selectors 74
   number of messages 73
   Replicated brokers 72
   request and reply 73
   Serialized connection objects 74
   setting 72
   setting methods 72
   Storing undeliverable messages 71
MapMessage
   handling 86
Mapping
   ABL data types to JMS data types 91
Maximum number of messages 73
MEMPTR data type 91
MEMPTR variable 88
Message
   properties 95
Message acknowledgement
      automatic 102
      preventing 103
      single 103
Message acknowledgement and recovery 102
Message Consumer
   deleting 98
Message Consumer objects
   cancellation of reception 99
Message Consumer scope 99
Message delivery parameters
   methods 72
Message handler
   accessing 97
   methods 97
Message handlers
   errors and conditions 106
      getting properties 97
Message headers
   accessing 94–95
   properties 94
Message life cycle 84
Message processing states 99
Message properties
   accessing 95
   clearing 95
Message reception issues
   starting and stopping 99
Message recovery 103
Message selectors 74
Message size limits 91
Message types 54
messageHandler procedure 252
Messages
   accessing 85
      acknowledgement 102, 183
      acknowledgement, automatic 102
      browsing on queue 97
BytesMessage
      handling 88
      clearBody 93
      clearBody and clearProperties 93
      clearing 94
      clearProperties 93
Consuming
      creating 96
         handler 97
      creating 85
      data 91
DataSetMessage
      handling 90
      deleting 94
      discardable 73
      durable subscriptions 98
      expiration time, getting 220
      header properties 94
HeaderMessage 86
   life cycle 84
Load balancing 70
MapMessage
   handling 86
MultipartMessage
   maximum number 73
   modes 93
MultipartMessage
   handling 88
Messages (continued)
persistency values 281
populating 85
preventing acknowledgement 103
priority values 221, 282
priority, default 282,
processes 320
properties 95
publishing 94
queue browsing 97
queues 93, 97
read only 93
receiving 97
reception issues 98
reception, synchronous 100
recovery 103
redelivery 265
Reply properties 97
request and reply 73
reuse 98
sending 93
setting and getting delivery parameters 72
setting maximum messages 73
setting maximum number 73
size limits 91
starting and stopping reception 99
Storing and extracting data 91
StreamMessage
handling 87
subscribing 98
TempTableMessage
handling 90
TextMessage
handling 85
time to live value 254, 266
topics 94, 98
write only 93
XMLMessage 89
Messaging
Client persistence 70–71
Fault tolerance 71–72
Message selectors 74
Replicated brokers 72
Storing undeliverable messages 71
Messaging example
PTP 109
Pub/Sub 113
scalable server architecture 354
Messaging examples
PTP messaging examples 354
Pub/Sub messaging examples 333, 352
Messaging introduction
Comparison 53
Message types 54
PTP 49
PTP features 50
Messaging introduction (continued)
PTP queuing 51
PTP send and receive 50
Pub/Sub features 52
Pub/Sub integration 53
Pub/Sub messaging 51
Pub/Sub send and receive 51
Methods in Message Consumer objects 173
Methods in the Message objects 174
Methods in the Session objects 171
Methods, 4GL
X–DOC:SAVE 347
Methods, ABL
appendText 85
beginSession 66, 106
browseQueue 97
cancelDurableSubscription 99
clearBody 93
clearProperties 93
commitReceive 102
deleteMessage 84
deleteSession 67, 99
cetAddress 100
getAddress 97
recover 102–103
requestReply 73, 93, 99–100
reset 93
rollbackReceive 102
sendToQueue 100
setAdapterService 267
setApplicationContext 106
setClientID 319
setJMSReplyTo 100
setLongText 85
setMemptr 88
setNoAcknowledge 102–103, 106
setNoErrorDisplay 104
setPingInterval 106
setReplyAutoDelete 100
setReplyToDestinationType 100
setText 85
setTransactedReceive 101
setTransactedSend 101
stopReceiveMessages 99
subscribe 98
waitForMessages 37, 99, 107
writeBytesFromRaw 88
writeShort 91
Modes
read-only 93
write-only 85, 93, 182
moveToNext procedure 253
Multipart message example 356
MultipartMessage
handling 88
N
Native Invocation file 134
Native Invocation Methodology 133
-nohostverify connection parameter 67
Non-supported APIs
non-supported methods 46
NumberFormatException error message 271

O
Object Messages 90
OpenEdge Adapter for Sonic ESB
Annotating with OpenEdge Architect 138
Annotation syntax examples 137
ESBOEGEN utility 153
Generating ESBOE files 140
Importing native invocation files into Sonic Workbench 141
Introduction 30
Native Invocation annotation syntax 136
Native Invocation file 134
Native Invocation file parameter mapping 144
Native Invocation methodology 133
OpenEdge Architect configurations 154
Sample process 361
Using a session-managed AppServer 145
Using ProxyGen to generate ESBOE files 140
Web Services Invocation methodology 156
OpenEdge Adapter for Sonic generic JMS-compliant messaging system
Introduction 33
OpenEdge Adapter for SonicMQ
 ABL-JMS object model 36
Accessing 39
Exchanging messages 36
Messaging 49
Understanding 35
Unified domain 35

P
Parse
 XMLMessage 342
Parse BytesMessage 344
Parse XMLMessage 342
parser, XML 346
Password
 broker login 301
-pf connection parameter 67
Ping interval
 SonicMQ broker 301
Populating messages 85
Prerequisites 33
Preventing message acknowledgment 103
Priority values, getting 221
Procedure files
 jmssession.p 37
 ptpsession.p 37
pubsubsession.p 37
Process StreamMessage 340
Process StreamMessage StreamMessage receive 347
Process XMLMessage 342
Processing messages 98
Programming errors 103
Programming example
 Code page 346
Programming examples
 Client persistence 117
ClientConnect 116
DataSetMessage 128
Encoding XML 346
Enhanced XML support 118
Fault tolerance 131
ServerConnect 116
SMQConnect 116
TempTableMessage 125
Properties selector, JMS 97–98
Provider attributes 205
PTP messaging example
 Connect broker 110
Create Message Consumer 110
Create session 110
Delete messages 111
Prepare receive 110
Receive messages 110–111
Send messages 111
Summary 112
PTP messaging examples 354
PTP messaging examples 352
ptpsession.p 37, 110, 116, 170
Pub/Sub messaging example
 Connect broker 113
Consume messages 114
Create Message Subscriber 113
Create session 113
Delete messages 114
Publish topic 114
Subscribe topic 114
Summary 115
Pub/Sub messaging examples 333, 352
Publish and receive
 TextMessage 349
Publish and subscribe
 BytesMessage 344
StreamMessage 340, 347
TextMessage 334, 336–337
XMLMessage 342
publish procedure 254
Publishing and receiving TextMessage 349
Publishing and subscribing with a BytesMessage 344
Publishing and subscribing with a StreamMessage 340, 347
Publishing and subscribing with a TextMessage 334, 336–337
Publishing and subscribing with a XMLMessage 342
Publishing messages to topic 94
Publishing with message properties and subscribing selectively 336
Publishing, JMS 359
pubsubsession.p 37, 113, 171

Q
Queue 49
Queue browsing 97
Queues
  browsing 97
  receiving messages 97
  sending messages 93

R
Read only modes 93
Read-only mode 93
readBytesToRaw procedure 256
readChar function 256
readDate function 257
readDateTime function 258
readDateTime-TZ function 258
readDecimal function 259
readLogical function 259
readLongString function 261
readLongStringCP function 262
Receive BytesMessage 344
Receive StreamMessage 340, 347
Receive table
  StreamMessage 347
Receive transaction
  TextMessage 349
Receive XML
  BytesMessage 344
Receive XMLMessage 342
receiveFromQueue procedure 262
Receiving messages to queue 97
Receiving table StreamMessage 347
Receiving transaction TextMessage 349
Receiving XML in BytesMessage 344
recover procedure 263
Recovery of messages 103
Redelivery of messages 265
Replies
  destination, getting 222
  queue for published message 100
  and SonicMQ Adapter 337
  second session for 101
Reply mechanisms 100
Reply properties 97
Reply queue for published message 100
Request and reply 73
requestReply procedure 264
reset procedure 265
Reuse of messages 98
rollbackReceive procedure 265
rollbackSend procedure 266
Rolling back transactions 222
Run-time conditions 103, 106
Running gateway sample 359

S
-S connection parameter 67
Scalable server architecture
  messaging example 354
Selector
  receiving with 97, 181, 262
  subscribing with 98, 336
Send and receive
  TextMessage 352, 354
Sending and receiving TextMessage 352, 354
Sending messages to queue 93
sendToQueue procedure 266
Serialized connection objects
  creating 77
  using 74
Services, Sonic ESB
  deployment 158
  management 158
  run-time properties 158
  session models 159
Session attributes 70
Session models
  for Sonic ESB services 159
Session objects
  jmssession.p 170
  ptpsession.p 170
  pubsubsession.p 171
  SonicMQ Adapter 37
Sessions
  and connections in JMS 65
  attributes 70
  creating in JMS 66
  second, for replies 101
  transacted for receiving 315
  transacted for sending 316
setAdapterService procedure 267
setApplicationContext procedure 268
setBoolean procedure 268
setBooleanProperty procedure 269
setBrokerURL procedure 270
setByte procedure 270
setByteProperty procedure 271
setBytesFromRaw procedure 271
setChar procedure 272
setClientID procedure 273
setClientPersistence procedure 273
setClientTransactionBufferSize procedure 274
setConnectionFile procedure 275
setConnectionURLs procedure 276
setDataSet procedure 277
setDate procedure 277
setDateProperty procedure 278
setDateTime procedure 279
setDateTime-TZ procedure 280
setDateTimeProperty procedure 280
setDefaultPersistency procedure 281
setDefaultPriority procedure 282
setDefaultTimeToLive procedure 283
setDouble procedure 283
setDoubleProperty procedure 284
setErrorHandler procedure 284
setFaultTolerant procedure 285
setFaultTolerantReconnectTimeout procedure 286
setFloat procedure 287
setFloatProperty procedure 287
setInitialConnectionTimeout procedure 290
setInt procedure 288
setInt64 procedure 289
setInt64Property procedure 290
setIntProperty procedure 289
setJMSCorrelationID procedure 291
setJMSCorrelationIDAsBytes procedure 292
setJMSReplyTo procedure 292
setJMSServerName procedure 293
setJMSType procedure 293
setLoadBalancing procedure 294
setLocalStoreDirectory procedure 294
setLocalStoreSize procedure 295
setLocalStoreWait procedure 296
setLong procedure 296
setLongProperty procedure 297
setLongString procedure 298
setLongText procedure 298
setMemptr procedure 299
setNoAcknowledge procedure 300
setPassword procedure 301
setPingInterval procedure 301
setPrefetchCount procedure 302
setPrefetchThreshold procedure 303
setReconnectInterval procedure 304
setReconnectTimeout procedure 304
setReplyAutoDelete procedure 305
setReplyPersistency procedure 306
setReplyPriority procedure 306
setReplyTimeToLive procedure 307
setReplyToDestinationType procedure 308
setReuseMessage procedure 308
setSaxReader procedure 309
setSelectorAtBroker procedure 309
setSequential procedure 310
setShort procedure 310
setShortProperty procedure 311
setSingleMessageAcknowledgement procedure 312
setString procedure 313
setStringProperty procedure 313
setTempTable procedure 314
setText procedure 315
Setting and getting connection attributes 70
Setting and getting session attributes 70
Setting CLASSPATH 83
Setting reply properties 97
Setting session options 72
methods 72
setTransactedReceive procedure 315
setTransactedSend procedure 316
setUser procedure 316
setX-Document procedure 317
short data type 91
mapped to ABL 91
Single message acknowledgment 103
Sonic ESB 28
described 28
SonicESB Adapter 33
documentation references 33
SonicMQ 30
broker login 301
described 28
pinging broker 301
server architecture, scaling 354
user value for login 316
SonicMQ Adapter 36
Accessing 39
automatic reply mechanism 337
connecting 70
documentation references 32
error handling 350–351
Exchanging messages between ABL and JMS 36
licensing 30
overview 29
PTP messaging examples 352, 356
replies 337
session objects 37
SonicMQ brokers 38
publishing to topics 94
receiving messages to queues 97
sending messages to queues 93
subscribing to topics 98
Starting connection to SonicMQ Adapter 70
Starting message reception 99
startReceiveMessages procedure 317
Stopping message reception 99
stopReceiveMessages procedure 318
Storing and extracting data 91
Storing undeliverable messages 71
StreamMessage
  customer table 347
  example 340, 347
  handling 87
  process 340, 347
  publish and subscribe 340, 347
  receive 340, 347
String data type
  mapped to ABL 91
  subscribe (procedure) 96
  subscribe procedure 319
Subscribing messages to topic 98
synchronous error example 351
Synchronous message reception 100
T
Temporary destination 73, 264
TempTableMessage
  handling 90
TextMessage
  asynchronous error 350
  automatic reply 337
  error handler 350–351
  example 334, 336–337, 349–352, 354
  handling 85
  publish and receive 349
  publish and subscribe 334, 336–337
  queues 352, 354
  receive group 349
  reply handle 337
  send and receive 352, 354
  synchronous error 351
  transaction 349
  using queues 352, 354
Time to live
  for messages 254, 266
Topics
  publishing messages 94
  subscribing 98
Transacted receiving 102
Transacted sending
  illegal calls 102
  recover 102
  setNoAcknowledge 102
Transacted sessions
  ABL transactions 102
  receiving 102
  setNoAcknowledge 102
Transacted sessions, JMS
  receiving 102
Transactions
  rolling back 222
U
ubroker.properties file
  default ping interval, specifying 301
Unicode 346
Unified domain 35
Updating the jmsProvider.properties file 42
Updating the jvmargs property 42
Updating your ABL-JMS API 46
-URL connection parameter 67
Using JNDI 43
Using queues
  TextMessage 352, 354
Using the AdminServerPlugins.properties file 43
UTF-8
  encoded XML document 347
  UTF-8 encoding 347
W
WAIT–FOR state 30, 37, 96, 106
WAIT–FOR statement 99–100, 107
waitForMessages procedure 320
Web Service Definition (WSD) files
  usage 158
Web Service Mapping (WSM) files
  usage 158
Web Services Description Language (WSDL) files
  generating 31, 158
Web Services Description Language files 31
Write only modes 93
Write-only mode 85, 93, 182
writeBoolean procedure 321
writeByte procedure 321
writeBytesFromRaw procedure 322
writeChar procedure 323
writeDate procedure 323
writeDateTime procedure 324
writeDateTime-TZ procedure 325
writeDouble procedure 326
writeFloat procedure 326
writeInt procedure 327
writeLong procedure 328
writeLongString procedure 329
writeShort procedure 330
writeString procedure 330
X
XML
  code-page encoding 346
  parser 89, 346
XMLMessage
  example 342
  parse 342
  process 342
<table>
<thead>
<tr>
<th>XMLMessage (continued)</th>
<th>XMLMessage (continued)</th>
</tr>
</thead>
<tbody>
<tr>
<td>publish and subscribe 342</td>
<td>receive 342</td>
</tr>
</tbody>
</table>