OpenEdge Development:
Messaging and ESB
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.

Third party acknowledgements — See the “Third party acknowledgements” section on page Preface–8.

November 2008

For the latest documentation updates see the OpenEdge Product Documentation category on PSDN (http://www.psdn.com/library/kbcategory.jspa?categoryID=129).
# Contents

1. **OpenEdge Applications in the Sonic Environment**. .......................... 1–1
   - Sonic messaging and integration systems ................................. 1–2
     - SonicMQ ........................................................................ 1–2
     - Sonic ESB ..................................................................... 1–2
   - OpenEdge Adapters for Sonic integration ................................. 1–3
     - OpenEdge Adapter for SonicMQ ...................................... 1–3
     - OpenEdge Adapter for Sonic ESB .................................... 1–5
   - Security considerations ....................................................... 1–6
     - BrokerConnect security ................................................. 1–6
     - OpenEdge Adapter for Sonic ESB security ....................... 1–7
   - Where to find detailed information ...................................... 1–8
     - OpenEdge Adapter for SonicMQ documentation ................. 1–8
     - OpenEdge Adapter for Sonic ESB documentation .............. 1–8

2. **Introducing the OpenEdge Adapter for SonicMQ** ............................ 2–1
   - Accessing the OpenEdge Adapter for SonicMQ ....................... 2–2
   - Understanding the OpenEdge Adapter for SonicMQ ............... 2–3
     - Unified domain for JMS sessions ..................................... 2–3
     - Exchanging messages ..................................................... 2–4
     - Understanding ABL–JMS object model ............................ 2–4
   - Introduction to messaging .................................................. 2–6
     - Point-to-Point (PTP) messaging .................................... 2–6
     - Publish-and-Subscribe messaging .................................. 2–8
     - Comparing PTP and Pub/Sub messaging .......................... 2–10
   - Messages and message types .............................................. 2–11

---

**Preface** ........................................................................................................ 1–1
### 3. Understanding the Messaging Models

**Using PTP messaging**
- Creating a session procedure and connecting to a SonicMQ Broker ............................................. 3–2
- Sending messages to a PTP queue ........................................................................................................ 3–2
- Receiving messages from a PTP queue ............................................................................................... 3–3
- Receiving a reply .................................................................................................................................. 3–3
- Temporary queues ................................................................................................................................. 3–4
- Deleting objects ..................................................................................................................................... 3–4
- Methods unique to Point-to-Point messaging .................................................................................... 3–4

**Using PUB/SUB messaging**
- Creating a session procedure and connecting to a SonicMQ Broker .................................................. 3–5
- Publishing a message to a Pub/Sub topic ............................................................................................. 3–5
- Subscribing to a Pub/Sub topic and receiving messages ..................................................................... 3–6
- Receiving a reply .................................................................................................................................. 3–6
- Durable subscriptions ............................................................................................................................ 3–7
- Temporary topic ...................................................................................................................................... 3–7
- Deleting objects ..................................................................................................................................... 3–7
- Methods unique to Pub/Sub messaging ............................................................................................... 3–8

### 4. Implementing Messaging

**Managing connections and sessions**
- Creating a JMS session .......................................................................................................................... 4–2
- Deleting a JMS session ............................................................................................................................ 4–2
- Connection options ................................................................................................................................. 4–3
- Managing fail-over support .................................................................................................................. 4–4
- Setting and getting JMS connection and session attributes ............................................................... 4–7
- Connecting to the OpenEdge Adapter for SonicMQ ............................................................................... 4–8
- Load balancing ...................................................................................................................................... 4–8
- Client persistence .................................................................................................................................... 4–8
- Fault tolerance ....................................................................................................................................... 4–9
- Establishing session control ................................................................................................................ 4–10
- Accessing message delivery parameters ............................................................................................ 4–11
- Request/Reply ........................................................................................................................................ 4–12
- Message selectors .................................................................................................................................... 4–12

**Externally managed connections**
- Using serialized connection objects .................................................................................................... 4–13
- Finding administered objects in JNDI or proprietary directories ...................................................... 4–20
- Setting the CLASSPATH ...................................................................................................................... 4–22

**Working with messages**
- Message life cycle ................................................................................................................................... 4–23
- Creating, populating, and accessing messages ..................................................................................... 4–24
- Sending messages to a queue ............................................................................................................... 4–34
- Publishing messages to a topic ............................................................................................................. 4–34
- Clearing messages ................................................................................................................................... 4–34
- Deleting messages ................................................................................................................................... 4–35
- Accessing message header properties ................................................................................................ 4–35
- Accessing message properties .............................................................................................................. 4–36

**Consuming messages**
- Creating a Message Consumer object .................................................................................................. 4–37
- Creating a message handler process ................................................................................................... 4–38
- Setting reply properties ....................................................................................................................... 4–38
- Receiving messages from a queue ....................................................................................................... 4–38
- Subscribing to a topic .......................................................................................................................... 4–39
- Terminating the Message Consumer object .......................................................................................... 4–39
- Processing messages ........................................................................................................................... 4–39
- Controlling flow of messages ................................................................................................................ 4–39
## Contents

- Reusing messages ........................................ 4–40
- Message-reception issues ............................... 4–40
- Reply mechanisms ......................................... 4–41
- Transaction and recovery procedures ................. 4–43
  - Message acknowledgement, forwarding, and recovery 4–44
- Error and condition handling ............................ 4–46

### 5. Programming for the OpenEdge Adapter for SonicMQ with the ABL–JMS API

- PTP message example ...................................... 5–1
  - Creating a PTP session procedure ...................... 5–2
  - Connecting to the broker ............................... 5–2
  - Creating a Message Consumer .......................... 5–3
  - Preparing to receive messages ........................ 5–3
  - Sending messages to the queue ......................... 5–3
  - Receiving messages from the queue .................... 5–3
  - Deleting a message ...................................... 5–4
  - Summary .................................................. 5–4
- Pub/Sub messaging example ............................... 5–6
  - Creating a Pub/Sub session procedure .............. 5–6
  - Connecting to the broker ............................... 5–6
  - Creating a Message Subscriber ....................... 5–7
  - Subscribing to a topic .................................. 5–7
  - Publishing to a topic .................................... 5–7
  - Consuming messages from a topic ..................... 5–7
  - Deleting messages ....................................... 5–8
  - Summary .................................................. 5–8
- Programming scenarios .................................... 5–10
  - Using JMS 1.1 unified domain model ................. 5–10
  - Using ServerConnect and ClientConnect ............ 5–11
  - Client persistence ...................................... 5–12
  - Enhanced XML support .................................. 5–14
  - TempTableMessage ...................................... 5–21
  - DataSetMessage ........................................ 5–24
  - Fault tolerance ......................................... 5–28

### 6. Guidelines for Using and Programming for the OpenEdge Adapter for Sonic ESB

- Native Invocation methodology ......................... 6–1
  - Creating an invocation file ............................ 6–2
  - Declarative invocation files ............................ 6–4
  - Non-declarative invocation files ...................... 6–9
  - Import Native Invocation files into Sonic Workbench 6–10
  - ABL file drag and drop .................................. 6–16
  - Testing an ESB Process containing OpenEdge Native Invocation Services 6–23
- Web Services Invocation methodology ................. 6–28
  - Differences between the Web Services Invocation methodology in the OpenEdge Adapter for Sonic ESB and the WSA 6–28

- ESBOEGEN .................................................. 6–24
Service definition considerations ........................................... 6–31
Session models ................................................................. 6–31
Sonic message handling run-time parameters ......................... 6–33

A. **ABL–JMS API Reference** ................................................. A–1
   Session objects .......................................................... A–2
   jmssession.p .............................................................. A–2
   ptpsession.p .............................................................. A–3
   pubsubsession.p ........................................................ A–3
   Methods in the Session objects ...................................... A–4
   Methods in the Message Consumer objects ......................... A–6
   Methods in the Message objects ..................................... A–7
   acknowledgeAndForward procedure ................................ A–10
   addBytesPart procedure ............................................... A–11
   addMessagePart procedure .......................................... A–11
   addTextPart procedure ............................................... A–12
   appendText procedure ................................................. A–12
   beginSession procedure ............................................. A–13
   browseQueue procedure ............................................. A–14
   cancelDurableSubscription procedure ............................ A–15
   clearBody procedure ................................................ A–15
   clearProperties procedure ......................................... A–16
   commitReceive procedure ........................................... A–16
   commitSend procedure ............................................... A–16
   createBytesMessage procedure .................................... A–17
   createChangeStateConsumer procedure ........................... A–17
   createDataSetMessage procedure ................................ A–18
   createHeaderMessage procedure ................................ A–18
   createMapMessage procedure ...................................... A–19
   createMessageConsumer procedure ................................ A–20
   createMultipartMessage procedure ................................ A–20
   createRejectedMessageConsumer procedure ..................... A–21
   createStreamMessage procedure ................................ A–21
   createTemporaryQueue procedure ................................ A–22
   createTemporaryTopic procedure ................................ A–22
   createTempTableMessage procedure ................................ A–23
   createTextMessage procedure ..................................... A–23
   createXMLMessage procedure ..................................... A–23
   deleteConsumer procedure ......................................... A–24
   deleteMessage procedure .......................................... A–24
   deleteSaxWriter procedure ....................................... A–25
   deleteSession procedure ........................................... A–25
   deleteTemporaryQueue procedure ................................. A–26
   deleteTemporaryTopic procedure ................................ A–26
   endOfStream function ................................................ A–26
   getAdapterService function ........................................ A–27
   getApplicationContext function ................................ A–28
   getBytesCount function ............................................. A–28
   getBrokerURL function ............................................. A–28
   getBytesPartByID function ........................................... A–29
   getBytesPartByIndex function ..................................... A–30
   getBytesToRaw function ............................................. A–30
   getChar function ..................................................... A–31
   getCharCount function ............................................... A–31
   getCharProperty function .......................................... A–31
   getClientID function ................................................ A–32
<table>
<thead>
<tr>
<th>Function Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>getClientPersistence function</td>
<td>A–32</td>
</tr>
<tr>
<td>getClientTransactionBufferSize function</td>
<td>A–33</td>
</tr>
<tr>
<td>getConnectionID function</td>
<td>A–33</td>
</tr>
<tr>
<td>getConnectionID function</td>
<td>A–34</td>
</tr>
<tr>
<td>getConnectionMetaDataSet function</td>
<td>A–34</td>
</tr>
<tr>
<td>getConnectionUrl function</td>
<td>A–34</td>
</tr>
<tr>
<td>getContentTypeName function</td>
<td>A–35</td>
</tr>
<tr>
<td>getDataSet function</td>
<td>A–35</td>
</tr>
<tr>
<td>getDate function</td>
<td>A–36</td>
</tr>
<tr>
<td>getDateProperty function</td>
<td>A–36</td>
</tr>
<tr>
<td>getDatePropertySet function</td>
<td>A–37</td>
</tr>
<tr>
<td>getDateTime function</td>
<td>A–37</td>
</tr>
<tr>
<td>getDateTimeFunction</td>
<td>A–38</td>
</tr>
<tr>
<td>getDateTimeFunction</td>
<td>A–38</td>
</tr>
<tr>
<td>getDecimal function</td>
<td>A–39</td>
</tr>
<tr>
<td>getDecimalProperty function</td>
<td>A–39</td>
</tr>
<tr>
<td>getDefaultPersistency function</td>
<td>A–39</td>
</tr>
<tr>
<td>getDefaultPriority function</td>
<td>A–40</td>
</tr>
<tr>
<td>getDefaultTimeToLive function</td>
<td>A–40</td>
</tr>
<tr>
<td>getDestinationName function</td>
<td>A–40</td>
</tr>
<tr>
<td>getFaultTolerant function</td>
<td>A–41</td>
</tr>
<tr>
<td>getFaultTolerantReconnectTimeout function</td>
<td>A–41</td>
</tr>
<tr>
<td>getFlowToDisk function</td>
<td>A–42</td>
</tr>
<tr>
<td>getInt function</td>
<td>A–42</td>
</tr>
<tr>
<td>getIntProperty function</td>
<td>A–42</td>
</tr>
<tr>
<td>getInt64 function</td>
<td>A–43</td>
</tr>
<tr>
<td>getInt64Property function</td>
<td>A–43</td>
</tr>
<tr>
<td>getInitialConnectionTimeout function</td>
<td>A–44</td>
</tr>
<tr>
<td>getItemType function</td>
<td>A–45</td>
</tr>
<tr>
<td>getJMSCorrelationID function</td>
<td>A–45</td>
</tr>
<tr>
<td>getJMSCorrelationIDAsBytes function</td>
<td>A–46</td>
</tr>
<tr>
<td>getJMSDeliveryMode function</td>
<td>A–46</td>
</tr>
<tr>
<td>getJMSDestination function</td>
<td>A–47</td>
</tr>
<tr>
<td>getJMSExpiration function</td>
<td>A–47</td>
</tr>
<tr>
<td>getJMSMessageID function</td>
<td>A–47</td>
</tr>
<tr>
<td>getJMSPriority function</td>
<td>A–48</td>
</tr>
<tr>
<td>getJMSRedelivered function</td>
<td>A–48</td>
</tr>
<tr>
<td>getJMSReplyTo function</td>
<td>A–49</td>
</tr>
<tr>
<td>getJMSRepairFunction</td>
<td>A–49</td>
</tr>
<tr>
<td>getJMSRepairFunction</td>
<td>A–49</td>
</tr>
<tr>
<td>getJMSServerName function</td>
<td>A–50</td>
</tr>
<tr>
<td>getJMSTimestamp function</td>
<td>A–50</td>
</tr>
<tr>
<td>getJMSType function</td>
<td>A–50</td>
</tr>
<tr>
<td>getLoadBalancing function</td>
<td>A–50</td>
</tr>
<tr>
<td>getLocalStoreDirectory function</td>
<td>A–51</td>
</tr>
<tr>
<td>getLocalStoreSize function</td>
<td>A–51</td>
</tr>
<tr>
<td>getLocalStoreWaitTime function</td>
<td>A–52</td>
</tr>
<tr>
<td>getLogical function</td>
<td>A–52</td>
</tr>
<tr>
<td>getLogicalProperty function</td>
<td>A–53</td>
</tr>
<tr>
<td>getLongString function</td>
<td>A–53</td>
</tr>
<tr>
<td>getLongStringCP function</td>
<td>A–54</td>
</tr>
<tr>
<td>getLongText function</td>
<td>A–54</td>
</tr>
<tr>
<td>getLongTextCP function</td>
<td>A–55</td>
</tr>
<tr>
<td>getMapNames function</td>
<td>A–55</td>
</tr>
<tr>
<td>getMemoPtr function</td>
<td>A–56</td>
</tr>
<tr>
<td>getMessagePartByID function</td>
<td>A–56</td>
</tr>
<tr>
<td>getMessagePartByIndex function</td>
<td>A–57</td>
</tr>
<tr>
<td>getMessageType function</td>
<td>A–58</td>
</tr>
<tr>
<td>getNoAcknowledge function</td>
<td>A–58</td>
</tr>
<tr>
<td>Function/Procedure</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------</td>
<td>------</td>
</tr>
<tr>
<td>getPartCount function</td>
<td>A–58</td>
</tr>
<tr>
<td>getPassword function</td>
<td>A–59</td>
</tr>
<tr>
<td>getProcHandle function</td>
<td>A–59</td>
</tr>
<tr>
<td>getProcName function</td>
<td>A–59</td>
</tr>
<tr>
<td>getPropertyNames function</td>
<td>A–60</td>
</tr>
<tr>
<td>getPropertyType function</td>
<td>A–60</td>
</tr>
<tr>
<td>getReconnectInterval function</td>
<td>A–61</td>
</tr>
<tr>
<td>getReconnectTimeout function</td>
<td>A–61</td>
</tr>
<tr>
<td>getReplyAutoDelete function</td>
<td>A–62</td>
</tr>
<tr>
<td>getReplyPersistence function</td>
<td>A–62</td>
</tr>
<tr>
<td>getReplyPriority function</td>
<td>A–63</td>
</tr>
<tr>
<td>getReplyTimeToLive function</td>
<td>A–63</td>
</tr>
<tr>
<td>getReplyToDestinationType function</td>
<td>A–64</td>
</tr>
<tr>
<td>getReuseMessage function</td>
<td>A–64</td>
</tr>
<tr>
<td>getSaxWriter function</td>
<td>A–65</td>
</tr>
<tr>
<td>getSession function</td>
<td>A–65</td>
</tr>
<tr>
<td>getSelectorAtBroker function</td>
<td>A–66</td>
</tr>
<tr>
<td>getSequential function</td>
<td>A–66</td>
</tr>
<tr>
<td>getShutdownWaitFor function</td>
<td>A–66</td>
</tr>
<tr>
<td>getSingleMessageAcknowledgement function</td>
<td>A–67</td>
</tr>
<tr>
<td>getTempTable function</td>
<td>A–67</td>
</tr>
<tr>
<td>getText function</td>
<td>A–68</td>
</tr>
<tr>
<td>getTextPartByID function</td>
<td>A–68</td>
</tr>
<tr>
<td>getTextPartByIndex function</td>
<td>A–69</td>
</tr>
<tr>
<td>getTextSegment function</td>
<td>A–69</td>
</tr>
<tr>
<td>getTransactedReceive function</td>
<td>A–70</td>
</tr>
<tr>
<td>getTransactedSend function</td>
<td>A–70</td>
</tr>
<tr>
<td>getUser function</td>
<td>A–70</td>
</tr>
<tr>
<td>getX-Document function</td>
<td>A–71</td>
</tr>
<tr>
<td>hasReplyTo function</td>
<td>A–71</td>
</tr>
<tr>
<td>inErrorHandling function</td>
<td>A–71</td>
</tr>
<tr>
<td>inMessageHandling function</td>
<td>A–72</td>
</tr>
<tr>
<td>inQueueBrowsing function</td>
<td>A–72</td>
</tr>
<tr>
<td>inReplyHandling function</td>
<td>A–72</td>
</tr>
<tr>
<td>isFaultTolerant function</td>
<td>A–73</td>
</tr>
<tr>
<td>isMessagePart function</td>
<td>A–73</td>
</tr>
<tr>
<td>JMS-MAXIMUM-MESSAGES global variable</td>
<td>A–74</td>
</tr>
<tr>
<td>messageHandler procedure</td>
<td>A–75</td>
</tr>
<tr>
<td>moveToNext procedure</td>
<td>A–76</td>
</tr>
<tr>
<td>publish procedure</td>
<td>A–77</td>
</tr>
<tr>
<td>readBytesToRaw procedure</td>
<td>A–78</td>
</tr>
<tr>
<td>readChar function</td>
<td>A–78</td>
</tr>
<tr>
<td>readDate function</td>
<td>A–78</td>
</tr>
<tr>
<td>readDateTime function</td>
<td>A–79</td>
</tr>
<tr>
<td>readDateTime-TZ function</td>
<td>A–79</td>
</tr>
<tr>
<td>readDecimal function</td>
<td>A–80</td>
</tr>
<tr>
<td>readInt function</td>
<td>A–80</td>
</tr>
<tr>
<td>readInt64 function</td>
<td>A–81</td>
</tr>
<tr>
<td>readLogical function</td>
<td>A–81</td>
</tr>
<tr>
<td>readLongString function</td>
<td>A–82</td>
</tr>
<tr>
<td>readLongStringCP function</td>
<td>A–82</td>
</tr>
<tr>
<td>receiveFromQueue procedure</td>
<td>A–83</td>
</tr>
<tr>
<td>recover procedure</td>
<td>A–84</td>
</tr>
<tr>
<td>requestReply procedure</td>
<td>A–85</td>
</tr>
<tr>
<td>reset procedure</td>
<td>A–86</td>
</tr>
<tr>
<td>rollbackReceive procedure</td>
<td>A–87</td>
</tr>
<tr>
<td>rollbackSend procedure</td>
<td>A–88</td>
</tr>
</tbody>
</table>
sendToQueue procedure .................................................. A–89
setAdapterService procedure ............................................. A–90
setApplicationContext procedure ........................................ A–90
setBoolean procedure .................................................... A–91
setBooleanProperty procedure .......................................... A–91
setBrokerURL procedure .................................................. A–92
setByte procedure .......................................................... A–92
setByteProperty procedure ............................................... A–93
setBytesFromRaw procedure .............................................. A–93
setChar procedure .......................................................... A–94
setClientID procedure ..................................................... A–94
setClientPersistence procedure .......................................... A–95
setClientTransactionBufferSize procedure ........................ A–96
setConnectID procedure ................................................... A–97
setConnectionFile procedure ............................................. A–97
setConnectionURLs procedure ............................................ A–98
setDataSet procedure ...................................................... A–98
setDate procedure .......................................................... A–99
setDateProperty procedure ............................................... A–100
setDateTime procedure ................................................... A–101
setDateTimeProperty procedure ......................................... A–101
setDateTime-TZ procedure ................................................ A–102
setDateTimeTzProperty procedure ...................................... A–102
setDefaultPersistency procedure ...................................... A–103
setDefaultPriority procedure ........................................... A–103
setDefaultTimeToLive procedure ...................................... A–104
setDouble procedure ...................................................... A–104
setDoubleProperty procedure .......................................... A–105
setErrorHandle procedure ................................................ A–105
setFaultTolerant procedure .............................................. A–106
setFaultTolerantReconnectTimeout procedure .................... A–107
setFloat procedure ........................................................ A–107
setFloatProperty procedure ............................................. A–108
setFlowToDisk procedure ................................................ A–108
setInt procedure .......................................................... A–109
setIntProperty procedure ............................................... A–109
setInt64 procedure ........................................................ A–110
setInt64Property procedure ............................................. A–110
setInitialConnectionTimeout procedure .............................. A–111
setJMSCorrelationID procedure ......................................... A–111
setJMSCorrelationIDAsBytes procedure ............................. A–112
setJMSReplyTo procedure ................................................ A–112
setJMSServerName procedure ........................................... A–113
setJMSVersion procedure ................................................ A–113
setLoadBalancing procedure ............................................ A–114
setLocalStoreDirectory procedure ..................................... A–114
setLocalStoreFileSize procedure ...................................... A–115
setLocalStoreWaitTime procedure ..................................... A–115
setLong procedure ........................................................ A–116
setLongProperty procedure ............................................. A–116
setLongString procedure ................................................ A–117
setLongText procedure ................................................... A–117
setMemptr procedure ...................................................... A–118
setNoAcknowledge procedure .......................................... A–119
setNoErrorDisplay procedure .......................................... A–119
setPassword procedure ................................................... A–120
setPingInterval procedure ............................................... A–121
B. Messaging Examples ............................................. B–1
   Pub/Sub messaging examples ................................ B–2
      Publishing and subscribing with a TextMessage ........ B–3
      Publishing with message properties and subscribing selectively B–4
      Publishing with a reply handle, subscribing, and receiving an automatic reply B–6
      Publishing, receiving, and processing a StreamMessage B–9
      Publishing, receiving, and parsing an XMLMessage ...... B–11

setPrefetchCount procedure ........................................... A–121
setPrefetchThreshold procedure ................................. A–122
setReconnectInterval procedure ................................ A–122
setReconnectTimeout procedure ................................ A–123
setReplyAutoDelete procedure .................................. A–123
setReplyPersistence procedure ................................. A–124
setReplyPriority procedure ..................................... A–124
setReplyTimeToLive procedure ............................... A–125
setReplyToDeviceDestinationType procedure ................. A–125
setReuseMessage procedure .................................... A–126
setSaxReader procedure ......................................... A–126
setSelectorAtBroker procedure ............................... A–127
setSequential procedure ........................................ A–127
setShort procedure ................................................ A–128
setShortProperty procedure ..................................... A–128
setShutdownWaitFor procedure ............................... A–129
setSingleMessageAcknowledgement procedure ............. A–129
setString procedure ............................................. A–130
setStringProperty procedure ................................... A–130
setTempTable procedure ......................................... A–131
setText procedure ................................................. A–131
setTransactedReceive procedure ................................ A–132
setTransactedSend procedure .................................. A–132
setUser procedure ................................................ A–133
setX-Document procedure ....................................... A–133
startReceiveMessages procedure .......................... A–134
stopReceiveMessages procedure ............................. A–134
subscribe procedure .............................................. A–135
waitForMessages procedure ................................. A–136
writeBoolean procedure ........................................ A–137
writeByte procedure ............................................ A–137
writeBytesFromRaw procedure .................................. A–138
writeChar procedure ............................................. A–138
writeDate procedure ........................................... A–139
writeDateTime procedure ...................................... A–140
writeDateTime-TZ procedure ................................ A–141
writeDouble procedure ......................................... A–141
writeFloat procedure ......................................... A–142
writeInt procedure ................................................ A–142
writeInt64 procedure ........................................... A–143
writeLong procedure .......................................... A–143
writeLongString procedure .................................. A–144
writeShort procedure .......................................... A–144
writeString procedure .......................................... A–145
Publishing, subscribing, and receiving the customer table in a StreamMessage ........................................... B–16
Publishing and receiving a group of messages in a transaction .......... B–18
Installing an error handler to handle an asynchronous error .......... B–20
Installing an error handler for synchronous errors ....................... B–21
PTP messaging examples ............................................. B–22
Sending a message to a queue and receiving a message from a queue .................................................... B–22
Achieving scalable server architecture with PTP queuing .............. B–23
MultiPartMessage example ............................................. B–26
Gateway sample application ........................................... B–27
Application files ................................................................ B–27
Running the sample application ......................................... B–30

C. Sample Native Invocation ESB process ............................. C–1
   Develop ABL source .................................................... C–2
   Configure OpenEdge servers ....................................... C–3
   Annotate source with OpenEdge Architect ....................... C–4
   Build Native Invocation and r-code files ......................... C–6
   Create ESB process in Sonic Workbench ......................... C–8
   Test ESB process ....................................................... C–13

Index ............................................................................. Index–1
Figures

Figure 2–1: Gateway model .......................... 2–9
Figure 4–1: Connection parameter value decision flow .................. 4–15
Figure 6–1: Development flow: ABL source to ESB itinerary .......... 6–3
Figure 6–2: Internal procedure invocation .......................... 6–14
Figure 6–3: Session-managed invocation template .................. 6–15
Figure 6–4: ESB process Service tab detail .......................... 6–21
Figure 6–5: OpenEdge Invocation detail .......................... 6–22
Figure 6–6: Sonic Management Console showing OpenEdge Adapter for Sonic ESB installed .................. 6–29
Figure 6–7: Recommended options for OpenEdge Adapter for Sonic ESB services .......................... 6–32
### Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>Adapter features</td>
<td>1–4</td>
</tr>
<tr>
<td>2-1</td>
<td>PTP messaging tasks</td>
<td>2–6</td>
</tr>
<tr>
<td>2-2</td>
<td>PTP messaging features</td>
<td>2–7</td>
</tr>
<tr>
<td>2-3</td>
<td>Pub/Sub messaging tasks</td>
<td>2–8</td>
</tr>
<tr>
<td>2-4</td>
<td>Pub/Sub messaging features</td>
<td>2–8</td>
</tr>
<tr>
<td>2-5</td>
<td>Comparing features</td>
<td>2–10</td>
</tr>
<tr>
<td>2-6</td>
<td>Message parts</td>
<td>2–11</td>
</tr>
<tr>
<td>2-7</td>
<td>Message types</td>
<td>2–11</td>
</tr>
<tr>
<td>3-1</td>
<td>Unique PTP messaging methods</td>
<td>3–4</td>
</tr>
<tr>
<td>3-2</td>
<td>Unique Pub/Sub messaging methods</td>
<td>3–8</td>
</tr>
<tr>
<td>4-1</td>
<td>OpenEdge Adapter for SonicMQ connection options</td>
<td>4–4</td>
</tr>
<tr>
<td>4-2</td>
<td>Connection parameter examples</td>
<td>4–6</td>
</tr>
<tr>
<td>4-3</td>
<td>Managing fail-over support</td>
<td>4–7</td>
</tr>
<tr>
<td>4-4</td>
<td>Connection and session attributes</td>
<td>4–7</td>
</tr>
<tr>
<td>4-5</td>
<td>Managing client persistence</td>
<td>4–9</td>
</tr>
<tr>
<td>4-6</td>
<td>Managing fault tolerance</td>
<td>4–10</td>
</tr>
<tr>
<td>4-7</td>
<td>Setting session methods</td>
<td>4–10</td>
</tr>
<tr>
<td>4-8</td>
<td>Setting and getting delivery parameters</td>
<td>4–11</td>
</tr>
<tr>
<td>4-9</td>
<td>Connection file parameters</td>
<td>4–13</td>
</tr>
<tr>
<td>4-10</td>
<td>Methods for handling text messages</td>
<td>4–24</td>
</tr>
<tr>
<td>4-11</td>
<td>Methods for handling map messages</td>
<td>4–26</td>
</tr>
<tr>
<td>4-12</td>
<td>Methods for handling stream messages</td>
<td>4–27</td>
</tr>
<tr>
<td>4-13</td>
<td>Methods for handling bytes messages</td>
<td>4–28</td>
</tr>
<tr>
<td>4-14</td>
<td>Methods for handling multi-part messages</td>
<td>4–29</td>
</tr>
<tr>
<td>4-15</td>
<td>Methods for handling XML messages</td>
<td>4–29</td>
</tr>
<tr>
<td>4-16</td>
<td>Methods for handling DataSet messages</td>
<td>4–30</td>
</tr>
<tr>
<td>4-17</td>
<td>Methods for handling TempTable messages</td>
<td>4–31</td>
</tr>
<tr>
<td>4-18</td>
<td>JMS and ABL data types for storing data</td>
<td>4–32</td>
</tr>
<tr>
<td>4-19</td>
<td>JMS and ABL data types for extracting data</td>
<td>4–33</td>
</tr>
<tr>
<td>4-20</td>
<td>Accessing message header information</td>
<td>4–35</td>
</tr>
<tr>
<td>4-21</td>
<td>Accessing message properties</td>
<td>4–36</td>
</tr>
<tr>
<td>4-22</td>
<td>Methods for the message handler</td>
<td>4–38</td>
</tr>
<tr>
<td>4-23</td>
<td>Managing transaction attributes</td>
<td>4–43</td>
</tr>
<tr>
<td>4-24</td>
<td>Acknowledging and forwarding messages</td>
<td>4–44</td>
</tr>
<tr>
<td>6-2</td>
<td>Conversion of ABL parameter to ESBOE parameter</td>
<td>6–14</td>
</tr>
<tr>
<td>A-1</td>
<td>Methods in Session objects</td>
<td>A–4</td>
</tr>
<tr>
<td>A-2</td>
<td>Methods in Message Consumer Objects</td>
<td>A–6</td>
</tr>
<tr>
<td>A-3</td>
<td>Methods in Message Objects</td>
<td>A–7</td>
</tr>
</tbody>
</table>
Procedures

Creating a PTP session ............................................ 5–2
Connecting to the broker ........................................... 5–2
Creating a Message Consumer ................................. 5–3
Preparing to receive messages ............................... 5–3
Sending messages to the queue .......................... 5–3
Deleting the message ............................................. 5–4
Complete code for sending a message using a PTP session ............................... 5–4
Complete code for receiving a message using a PTP session ............................ 5–5
Creating a Pub/Sub session .................................... 5–6
Connecting to the broker ...................................... 5–6
Creating a Message Subscriber .......................... 5–7
Subscribing to a topic ........................................ 5–7
Publishing message to a topic .......................... 5–7
Deleting the message ............................................. 5–8
Complete code for publish a message using a Pub/Sub session ..................... 5–8
Complete code for receiving a message using a Pub/Sub session .................. 5–9
Using SMQConnect on a client .................................. 5–11
Client persistence example .................................... 5–13
Saxwriter message .................................................. 5–14
setSaxReader example ......................................... 5–15
saxSender.p ......................................................... 5–16
saxReceiver.p ....................................................... 5–17
setX-Document example ........................................... 5–19
g getX-Document example .......................................... 5–19
domSender.p .......................................................... 5–20
domReceiver.p ......................................................... 5–20
ttmgs_send.p ......................................................... 5–22
tmsg_recv.p .......................................................... 5–23
dmsg_send.p .......................................................... 5–25
dmsg_recv.p .......................................................... 5–26
Fault tolerance set up ................................................. 5–29
Fault tolerant example ............................................ 5–29
example2.p .............................................................. 5–3
example1.p .............................................................. 5–4
example4.p .............................................................. 5–4
example3.p .............................................................. 5–5
example5.p .............................................................. 5–6
example6.p .............................................................. 5–7
example7.p .............................................................. 5–8
example9.p .............................................................. 5–9
extmple8.p .............................................................. 5–10
example11.p ............................................................. 5–11
example10.p ............................................................. 5–12
example13.p ............................................................. 5–13
example12.p ............................................................. 5–14
example15.p ............................................................. 5–16
example14.p ............................................................. 5–17
example23.p ............................................................. 5–18
example22.p ............................................................. 5–19
example16.p ............................................................. 5–20
example17.p ............................................................. 5–21
example18.p ............................................................. 5–22
example19.p ............................................................. 5–23
<table>
<thead>
<tr>
<th>File</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>example20.p</td>
<td>B–24</td>
</tr>
<tr>
<td>example21.p</td>
<td>B–25</td>
</tr>
<tr>
<td>appDriver.p</td>
<td>B–28</td>
</tr>
<tr>
<td>JMSgateway.p</td>
<td>B–28</td>
</tr>
<tr>
<td>customers.p</td>
<td>B–29</td>
</tr>
</tbody>
</table>
This Preface contains the following sections:

- Purpose
- Audience
- Organization
- Using this manual
- Typographical conventions
- Examples of syntax descriptions
- OpenEdge messages
- Third party acknowledgements
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.

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

Chapter 1, “OpenEdge Applications in the Sonic Environment”

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.

Chapter 2, “Introducing the OpenEdge Adapter for SonicMQ”

Explains basic concepts and general considerations for integrating OpenEdge applications with the JMS messaging service provided by SonicMQ.

Chapter 3, “Understanding the Messaging Models”

Explains the Point-to-Point (PTP) and Publish/Subscribe (Pub/Sub) JMS messaging models.

Chapter 4, “Implementing Messaging”

Describes how an OpenEdge client exchanges messages using the JMS messaging models.

Chapter 5, “Programming for the OpenEdge Adapter for SonicMQ with the ABL–JMS API”

Provides instructions for using the ABL–JMS API to program applications for the SonicMQ environment.

Chapter 6, “Guidelines for Using and Programming for the OpenEdge Adapter for Sonic ESB”

Provides general programming guidelines for the OpenEdge Adapter for Sonic ESB using the Native Invocation methodology and the Web Service Invocation methodology. The chapter 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 chapter, for detailed information.
Preface

Appendix A, “ABL–JMS API Reference”

Provides an alphabetical API reference for the OpenEdge Adapter for SonicMQ.

Appendix B, “Messaging Examples”

Provides ABL code examples of Pub/Sub and PTP messaging, and a sample application.

Appendix C, “Sample Native Invocation ESB process”

Provides an example of creating an ESB process using the Native Invocation methodology of the OpenEdge Adapter for Sonic ESB.

Using this manual

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 category on PSDN http://www.psdn.com/library/kbcategory.jspa?categoryID=129.

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 manual uses the following typographical 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>SMALL, BOLD CAPITAL LETTERS</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 simultaneous 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 sequential key sequence: you press and release the first key, then press another key. For example, ESCAPE H.</td>
</tr>
</tbody>
</table>

**Syntax:**

| Fixed width                     | A fixed-width font is used in syntax statements, code examples, system output, and filenames. |
| Fixed-width italics             | Fixed-width italics indicate variables in syntax statements. |
| Fixed-width bold                | Fixed-width bold indicates variables with special emphasis. |
| UPPERCASE fixed width           | Uppercase words are ABL keywords. Although these are always shown in uppercase, you can type them in either uppercase or lowercase in a procedure. |

This icon (three arrows) introduces a multi-step procedure.

This icon (one arrow) introduces a single-step procedure.

Period (.) or colon (:)

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.
Examples of syntax descriptions

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

Syntax

<table>
<thead>
<tr>
<th>Convention</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>
<tr>
<td></td>
<td>A vertical bar indicates a choice.</td>
</tr>
<tr>
<td>...</td>
<td>Ellipses indicate repetition: you can choose one or more of the preceding items.</td>
</tr>
</tbody>
</table>

FOR EACH Customer:
    DISPLAY Name.
END.

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

Syntax

<table>
<thead>
<tr>
<th>Convention</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>
<tr>
<td></td>
<td>A vertical bar indicates a choice.</td>
</tr>
<tr>
<td>...</td>
<td>Ellipses indicate repetition: you can choose one or more of the preceding items.</td>
</tr>
</tbody>
</table>

INITIAL [ constant [ , constant ] ]

In this example, the outer (small) brackets are part of the language, and the inner (large) brackets denote an optional item:
A called external procedure must use braces when referencing compile-time arguments passed by a calling procedure, as shown in this example:

**Syntax**

\[
\{ \ &\text{argument-name} \ \}
\]

In this example, EACH, FIRST, and LAST are optional, but you can choose only one of them:

**Syntax**

\[
\text{PRESELECT } [ \ \text{EACH} \ | \ \text{FIRST} \ | \ \text{LAST} ] \ record-phrase
\]

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

**Syntax**

\[
\text{MAXIMUM}( \ \text{expression} , \ \text{expression} [ , \ \text{expression} ] \ldots )
\]

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

**Syntax**

\[
\text{MESSAGE} \ \{ \ \text{expression} \ | \ \text{SKIP} \ [ (n) ] \} \ldots
\]

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

**Syntax**

\[
\{ \ \text{include-file} \ [ \ \text{argument} \ | \ \&\text{argument-name} = "argument-value" ] \ldots \}
\]

### 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, \text{WITH} is followed by six optional items:

**Syntax**

\[
\text{WITH} \ [ \ \text{ACCUM} \ max-length \ ] \ [ \ \text{expression} \ DOWN \ ]
\]

\[
[ \ \text{CENTERED} \ ] \ [ \ n \ \text{COLUMNS} \ ] \ [ \ \text{SIDE-LABELS} \ ]
\]

\[
[ \ \text{STREAM-IO} \ ]
\]
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

| ASSIGN   | { [ FRAME frame ] { field [ = expression ] } [ WHEN expression ] } ... | { record [ EXCEPT field ... ] } |

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-install-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.

Third party acknowledgements

OpenEdge includes software developed by the Apache Software Foundation (http://www.apache.org/). Copyright © 1999 The Apache Software Foundation. All rights reserved (Xerces C++ Parser (XML) and Xerces2 Java Parser (XML)); Copyright © 1999-2002 The Apache Software Foundation. All rights reserved (Xerces Parser (XML)); and Copyright © 2000-2003 The Apache Software Foundation. All rights reserved (Ant). The names “Apache,” “Xerces,” “ANT,” and “Apache Software Foundation” must not be used to endorse or promote products derived from this software without prior written permission. Products derived from this software may not be called “Apache”, nor may “Apache” appear in their name, without prior written permission of the Apache Software Foundation. For written permission, please contact apache@apache.org. Software distributed on an “AS IS” basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License agreement that accompanies the product.


OpenEdge includes DataDirect products for the Microsoft SQL Server database which contains a licensed implementation of the Microsoft TDS Protocol.
OpenEdge includes software authored by David M. Gay. Copyright © 1991, 2000, 2001 by Lucent Technologies (dtoa.c); Copyright © 1991, 1996 by Lucent Technologies (g_fmt.c); and Copyright © 1991 by Lucent Technologies (rnd_prod.s). Permission to use, copy, modify, and distribute this software for any purpose without fee is hereby granted, provided that this entire notice is included in all copies of any software which is or includes a copy or modification of this software and in all copies of the supporting documentation for such software. THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED WARRANTY. IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.

OpenEdge includes software authored by David M. Gay. Copyright © 1998-2001 by Lucent Technologies All Rights Reserved (decsrtod.c; strtodg.c); Copyright © 1998, 2000 by Lucent Technologies All Rights Reserved (decsrtodf.c; strtord.c); Copyright © 1998 by Lucent Technologies All Rights Reserved (dmisc.c; gdtoa.h; gethex.c; gmisc.c; sum.c); Copyright © 1998, 1999 by Lucent Technologies All Rights Reserved (gdtoa.c; misc.c; smisc.c; ulp.c); Copyright © 1998-2000 by Lucent Technologies All Rights Reserved (gdtoaimp.h); Copyright © 2000 by Lucent Technologies All Rights Reserved (hd_init.c). Full copies of these licenses can be found in the installation directory, in the c:/OpenEdge/licenses folder. Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that the copyright notice and this permission notice and warranty disclaimer appear in supporting documentation, and that the name of Lucent or any of its entities not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. LUCENT DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL LUCENT OR ANY OF ITS ENTITIES BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

OpenEdge includes software developed by IBM. Copyright © 1995-2003 International Business Machines Corporation and others. All rights reserved. Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, provided that the above copyright notice(s) and this permission notice appear in all copies of the Software and that both the above copyright notice(s) and this permission notice appear in supporting documentation. Software distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License agreement that accompanies the product. Except as contained in this notice, the name of a copyright holder shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization of the copyright holder. OpenEdge includes ICU software 1.8 and later - Copyright © 1995-2003 International Business Machines Corporation and others All rights reserved. Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, provided that the above copyright notice(s)
and this permission notice appear in all copies of the Software and that both the above copyright
notice(s) and this permission notice appear in supporting documentation.

OpenEdge includes Infragistics NetAdvantage for .NET v2008 Vol 1 CLR 2.0. Copyright © 1996-2008 Infragistics, Inc. All rights reserved.

OpenEdge includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/). Copyright © 1998-2007 The OpenSSL Project. All rights reserved. This product includes cryptographic software written by Eric Young (eay@cryptsoft.com). This product includes software written by Tim Hudson (tjh@cryptsoft.com). Copyright © 1995-1998 Eric Young (eay@cryptsoft.com) All rights reserved. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact openssl-core@openssl.org. Products derived from this software may not be called "OpenSSL" nor may "OpenSSL" appear in their names without prior written permission of the OpenSSL Project. Software distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License agreement that accompanies the product.

OpenEdge includes code licensed from RSA Security, Inc. Some portions licensed from IBM are available at http://oss.software.ibm.com/icu4j/.

OpenEdge includes the RSA Data Security, Inc. MD5 Message-Digest Algorithm. Copyright ©1991-2, RSA Data Security, Inc. Created 1991. All rights reserved.


OpenEdge includes Sonic software, which includes software developed by Apache Software Foundation (http://www.apache.org/). Copyright © 1999-2000 The Apache Software Foundation. All rights reserved. The names “Ant”, “Axis”, “Xalan,” “FOP,” “The Jakarta Project”, “Tomcat”, “Xerces” and/or “Apache Software Foundation” must not be used to endorse or promote products derived from the Product without prior written permission. Any product derived from the Product may not be called “Apache”, nor may “Apache” appear in their name, without prior written permission. For written permission, please contact apache@apache.org.

OpenEdge includes Sonic software, which includes software Copyright © 1999 CERN - European Organization for Nuclear Research. Permission to use, copy, modify, distribute and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. CERN makes no representations about the suitability of this software for any purpose. It is provided "as is" without expressed or implied warranty.

OpenEdge includes Sonic software, which includes software developed by ExoLab Project (http://www.exolab.org/). Copyright © 2000 Intalio Inc. All rights reserved. The names “Castor” and/or “ExoLab” must not be used to endorse or promote products derived from the Products without prior written permission. For written permission, please contact info@exolab.org. Exolab, Castor and Intalio are trademarks of Intalio Inc.
OpenEdge includes Sonic software, which includes the JMX Technology from Sun Microsystems, Inc. Use and Distribution is subject to the Sun Community Source License available at http://sun.com/software/communitysource.

OpenEdge includes Sonic software, which includes software developed by the ModelObjects Group (http://www.modelobjects.com). Copyright © 2000-2001 ModelObjects Group. All rights reserved. The name “ModelObjects” must not be used to endorse or promote products derived from this software without prior written permission. Products derived from this software may not be called “ModelObjects”, nor may “ModelObjects” appear in their name, without prior written permission. For written permission, please contact djacobs@modelobjects.com.

OpenEdge includes Sonic software, which includes code licensed from Mort Bay Consulting Pty. Ltd. The Jetty Package is Copyright © 1998 Mort Bay Consulting Pty. Ltd. (Australia) and others.

OpenEdge includes Sonic software, which includes files that are subject to the Netscape Public License Version 1.1 (the “License”); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.mozilla.org/NPL/. Software distributed under the License is distributed on an “AS IS” basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License. The Original Code is Mozilla Communicator client code, released March 31, 1998. The Initial Developer of the Original Code is Netscape Communications Corporation. Portions created by Netscape are Copyright © 1998-1999 Netscape Communications Corporation. All Rights Reserved.

OpenEdge includes Sonic software, which includes software developed by the University Corporation for Advanced Internet Development http://www.ucaid.edu Internet2 Project. Copyright © 2002 University Corporation for Advanced Internet Development, Inc. All rights reserved. Neither the name of OpenSAML nor the names of its contributors, nor Internet2, nor the University Corporation for Advanced Internet Development, Inc., nor UCAID may be used to endorse or promote products derived from this software and products derived from this software may not be called OpenSAML, Internet2, UCAID, or the University Corporation for Advanced Internet Development, nor may OpenSAML appear in their name without prior written permission of the University Corporation for Advanced Internet Development. For written permission, please contact opensaml@opensaml.org.

OpenEdge includes software Copyright © 2003-2006, Terence Parr All rights reserved. Neither the name of the author nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission. Software distributed on an “AS IS” basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License agreement that accompanies the product.

OpenEdge includes the UnixWare platform of Perl Runtime authored by Kiem-Phong Vo and David Korn. Copyright © 1991, 1996 by AT&T Labs. Permission to use, copy, modify, and distribute this software for any purpose without fee is hereby granted, provided that this entire notice is included in all copies of any software which is or includes a copy or modification of this software and in all copies of the supporting documentation for such software. THIS SOFTWARE IS BEING PROVIDED “AS IS”, WITHOUT ANY EXPRESS OR IMPLIED WARRANTY. IN PARTICULAR, NEITHER THE AUTHORS NOR AT&T LABS MAKE ANY REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
OpenEdge includes software developed by Vermont Creative Software. Copyright © 1988-1991 by Vermont Creative Software.

OpenEdge includes software developed by the World Wide Web Consortium. Copyright © 1994-2002 World Wide Web Consortium, (Massachusetts Institute of Technology, European Research Consortium for Informatics and Mathematics, Keio University). All rights reserved. This work is distributed under the W3C® Software License [http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231] in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

OpenEdge includes XML Tools, which includes versions 8.9 of the Saxon XSLT and XQuery Processor from Saxonica Limited (http://www.saxonica.com/) which are available from SourceForge (http://sourceforge.net/projects/saxon/). The Original Code of Saxon comprises all those components which are not explicitly attributed to other parties. The Initial Developer of the Original Code is Michael Kay. Until February 2001 Michael Kay was an employee of International Computers Limited (now part of Fujitsu Limited), and original code developed during that time was released under this license by permission from International Computers Limited. From February 2001 until February 2004 Michael Kay was an employee of Software AG, and code developed during that time was released under this license by permission from Software AG, acting as a "Contributor". Subsequent code has been developed by Saxonica Limited, of which Michael Kay is a Director, again acting as a "Contributor". A small number of modules, or enhancements to modules, have been developed by other individuals (either written especially for Saxon, or incorporated into Saxon having initially been released as part of another open source product). Such contributions are acknowledged individually in comments attached to the relevant code modules. All Rights Reserved. The contents of the Saxon files are subject to the Mozilla Public License Version 1.0 (the "License"); you may not use these files except in compliance with the License. You may obtain a copy of the License at http://www.mozilla.org/MPL/ and a copy of the license can also be found in the installation directory, in the c:/OpenEdge/licenses folder. Software distributed under the License is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License.

OpenEdge includes XML Tools, which includes Xs3P v1.1.3. The contents of this file are subject to the DSTC Public License (DPL) Version 1.1 (the "License"); you may not use this file except in compliance with the License. A copy of the license can be found in the installation directory, in the c:/OpenEdge/licenses folder. Software distributed under the License is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License. The Original Code is xs3p. The Initial Developer of the Original Code is DSTC. Portions created by DSTC are Copyright © 2001, 2002 DSTC Pty Ltd. All rights reserved.
OpenEdge Applications in the Sonic Environment

This chapter briefly describes the Sonic messaging environment and provides an overview of the available options for integrating OpenEdge® applications in such an environment, as outlined in the following sections:

- Sonic messaging and integration systems
- OpenEdge Adapters for Sonic integration
- Security considerations
- Where to find detailed information

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.
OpenEdge Applications in the Sonic Environment

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, is 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
- OpenEdge Adapter for Sonic ESB

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, acknowledgement, 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 Applications in the Sonic Environment

- **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.

For information on the OpenEdge Adapter for SonicMQ architecture, see *OpenEdge Getting Started: Application and Integration Services*.

**OpenEdge Adapter for SonicMQ operation**

Table 1–1 shows the operation features for each connection mode.

**Table 1–1: 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” section on page 4–2 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*


- **Management and configuration (Server Connect and BrokerConnect only)** — See *OpenEdge Application Server: Administration*

**OpenEdge Adapter for Sonic ESB 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 Chapter 6, “Guidelines for Using and Programming for the OpenEdge Adapter for Sonic ESB”

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*
Introducing the OpenEdge Adapter for SonicMQ

This chapter provides an introduction to accessing SonicMQ messaging services from OpenEdge®. The following sections describe:

- Accessing the OpenEdge Adapter for SonicMQ
- Understanding the OpenEdge Adapter for SonicMQ
- Introduction to messaging
- Messages and message types

Programming instructions are provided in Chapter 5, “Programming for the OpenEdge Adapter for SonicMQ with the ABL-JMS API.”

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 http://www.java.sun.com.
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
Understanding the OpenEdge Adapter for SonicMQ

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

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

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

**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:

```abl
RUN jms/ptpsession.p PERSISTENT SET ptpsession (adapterConnection).
```

The following ABL code starts a JMS session using topics:

```abl
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:

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

**Note:** See the “Connection options” section on page 4–4 for valid values for `adapterConnection`.

2–3
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
- Message Consumer objects
- Message objects

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 the “Managing connections and sessions” section on page 4–2.
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 the “Consuming messages” section on page 4–37.

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 the “Messages and message types” section on page 2–11.
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** (Single sender and receiver of a message)
- **Publish-and-Subscribe messaging** (Single sender and multiple receivers of a message)
- **Comparing PTP and Pub/Sub messaging**

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 from the queue.

Sending and receiving messages

Table 2–1 describes the tasks performed to send and receive messages using PTP messaging.

### Table 2–1: 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(^1)</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>

1. Before starting your session, queues must be defined.
PTP messaging options and features

Table 2–2 describes the features of the PTP messaging model.

Table 2–2: 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>
<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 the “Using PTP messaging” section on page 3–2.

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 Appendix B, “Messaging Examples.”
Introducing the OpenEdge Adapter for SonicMQ

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

Table 2–3 describes the tasks performed to send and receive messages using PTP messaging.

Table 2–3: 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>
<tr>
<td>5.</td>
<td>Subscriber</td>
<td>Consumes the message</td>
</tr>
<tr>
<td>6.</td>
<td>Subscriber</td>
<td>Acknowledges message receipt to the broker</td>
</tr>
<tr>
<td>7.</td>
<td>Message broker</td>
<td>Deletes the message after all subscribers acknowledged it</td>
</tr>
</tbody>
</table>

Pub/Sub messaging options and features

Table 2–4 describes the features of the Pub/Sub messaging model.

Table 2–4: 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 the “Using PUB/SUB messaging” section on page 3–5.
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 a Figure 2–1, 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 Appendix B, “Messaging Examples”).

Figure 2–1: Gateway model
Comparing PTP and Pub/Sub messaging

There are several distinguishing characters between the two message models. Table 2–5 compares PTP and Pub/Sub messaging.

<table>
<thead>
<tr>
<th></th>
<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>
<td></td>
</tr>
<tr>
<td>The receiver can browse the queue of undelivered messages.</td>
<td>Receivers only see delivered messages.</td>
<td></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>
<td></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>
<td></td>
</tr>
<tr>
<td>Message consumers can use message selectors to filter messages.</td>
<td>Message consumers can use message selectors to filter messages.</td>
<td></td>
</tr>
<tr>
<td>Support request/reply.</td>
<td>Support request/reply.</td>
<td></td>
</tr>
<tr>
<td>Queues permit Message Consumers to receive messages sent while disconnected.</td>
<td>Durable subscriptions permit Message Consumers to receive messages sent while disconnected.</td>
<td></td>
</tr>
</tbody>
</table>
Messages and message types

A Message is the package of information sent from a producer to a receiver through the Message Consumer. Table 2–6 describes the parts of a message.

Table 2–6: Message parts

<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. Table 2–7 lists the SonicMQ message types and content of each.

Table 2–7: Message types

<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” section on page 4–23.
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.

This chapter includes the following sections:

- Using PTP messaging
- Using PUB/SUB messaging

Programming instructions are provided in Chapter 5, “Programming for the OpenEdge Adapter for SonicMQ with the ABL–JMS API.”
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:

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.

**Note:** PTP messaging requires a queue.

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 jms/jmssession.p or jms/ptpsession.p persistently to instantiate the appropriate Session object and calls the `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 `deleteSession` procedure 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 sends a message to a queue:

1. The application obtains a handle to the PTP Session object.
2. The application creates a message by calling one of the following procedures from the Session object: `createBytesMessage` procedure, `createDataSetMessage` procedure, `createHeaderMessage` procedure, `createMapMessage` procedure, `createMultipartMessage` procedure, `createStreamMessage` procedure, `createTempTableMessage` procedure, `createXMLMessage` procedure.
3. The application populates the header fields, properties, and body of the message.

4. The application calls the sendToQueue procedure in the Session object with the message handle and the name of a queue as input parameters.

5. 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.

3. The application calls the receiveFromQueue procedure in the Session object with the name of a queue and the Message Consumer handle as input parameters.

4. The application executes a WAIT–FOR statement (or a waitForMessages procedure call) 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 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 a waitForMessages procedure call), 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.

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` and `deleteTemporaryQueue procedure`.

Deleting objects

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

- The PTP Session object calls the `deleteSession procedure`.
- The Message Object calls the `deleteMessage procedure`.
- The Message Consumer object calls the `deleteConsumer procedure`.

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

Table 3–1 lists the unique methods for Point-to-Point messaging.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sendToQueue procedure</code></td>
<td>Sends a message to a queue</td>
</tr>
<tr>
<td><code>receiveFromQueue procedure</code></td>
<td>Receives messages from a queue</td>
</tr>
<tr>
<td><code>browseQueue procedure</code></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” section on page 5–2.
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 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 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, createDataSetMessage procedure, createHeaderMessage procedure, createMapMessage procedure, createMultipartMessage procedure, createStreamMessage procedure, createTempTableMessage procedure, or createXMLMessage procedure.
3. The application populates the header fields, properties, and body of the message.
4. The application calls the publish procedure 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.
3. The application calls the startReceiveMessages procedure 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 a waitForMessages procedure call) 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 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 a waitForMessages procedure call), 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 or the subscribe procedure 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 and deleteTemporaryTopic procedure.

Deleting objects

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

- The PTP Session object calls the deleteSession procedure.
- The Message object calls the deleteMessage procedure.
- The Message Consumer object calls the deleteConsumer procedure.

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

Table 3–2 lists the unique methods for Pub/Sub messaging.

Table 3–2: Unique Pub/Sub messaging methods

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

For an example, see the “Pub/Sub messaging example” section on page 5–6.
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. This chapter includes the following sections:

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

Programming instructions are provided in Chapter 5, “Programming for the OpenEdge Adapter for SonicMQ with the ABL–JMS API.”

For an alphabetical API reference, see Appendix A, “ABL–JMS API Reference.”
Managing connections and sessions

In order to use the messaging capabilities of JMS and SonicMQ, 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
- Deleting a JMS session
- Connection options
- Managing fail-over support
- Setting and getting JMS connection and session attributes
- Connecting to the OpenEdge Adapter for SonicMQ
- Load balancing
- Client persistence
- Fault tolerance
- Establishing session control
- Accessing message delivery parameters
- Request/Reply
- Message selectors
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.

**Note:** Session attributes cannot be modified after calling the `beginSession` procedure.

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 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 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. Table 4–1 lists and explains the valid formats for expressing these options.

**Table 4–1: OpenEdge Adapter for SonicMQ connection options (1 of 3)**

<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><code>-DirectConnect</code></td>
<td>If included, causes the <code>-H</code> and <code>-S</code> parameters to be interpreted as the network address and TCP/IP port number of a specific OpenEdge Adapter for SonicMQ. If the <code>-DirectConnect</code> switch is omitted, the <code>-H</code> and <code>-S</code> parameters are interpreted as the network address and UDP port number of a NameServer.</td>
</tr>
<tr>
<td><code>-ssl</code></td>
<td>If included, specifies a secure connection to an SSL-enabled OpenEdge Adapter for SonicMQ. For more information, see <em>OpenEdge Getting Started: Core Business Services</em>.</td>
</tr>
</tbody>
</table>
Managing connections and sessions

Table 4–1: OpenEdge Adapter for SonicMQ connection options

<table>
<thead>
<tr>
<th>Connection option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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 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. <strong>Note:</strong> The service name of the OpenEdge Adapter for SonicMQ is a required part of the -URL parameter. Therefore, if the application subsequently calls setAdapterService procedure, the call is ignored.</td>
</tr>
<tr>
<td>-nohostverify</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>
</tbody>
</table>
Table 4–2 shows several examples of valid `adapterConnection` parameters.

<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>&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>&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>&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>&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>RUN jms/jmssession.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. Table 4–3 lists the methods for managing broker connections.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Getting</th>
</tr>
</thead>
<tbody>
<tr>
<td>setConnectionURLs procedure</td>
<td>getConnectionURLs function</td>
</tr>
<tr>
<td>setSequential procedure</td>
<td>getSequential function</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. Table 4–4 lists the methods for handling connection and session attributes.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Getting</th>
</tr>
</thead>
<tbody>
<tr>
<td>setAdapterService procedure</td>
<td>getAdapterService function</td>
</tr>
<tr>
<td>setJMSCommServerName procedure</td>
<td>getJMSCommServerName function</td>
</tr>
<tr>
<td>setBrokerURL procedure</td>
<td>getBrokerURL function</td>
</tr>
<tr>
<td>setPingInterval procedure</td>
<td>getConnectionMetaData function</td>
</tr>
<tr>
<td>setUser procedure</td>
<td>getUser function</td>
</tr>
<tr>
<td>setPassword procedure</td>
<td>getPassword function</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.

**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 and the `getLoadBalancing` function.

**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.

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

**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.
Table 4–5 lists the methods for managing client persistence.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Getting</th>
</tr>
</thead>
<tbody>
<tr>
<td>setLocalStoreDirectory procedure</td>
<td>getLocalStoreDirectory function</td>
</tr>
<tr>
<td>setLocalStoreSize procedure</td>
<td>getLocalStoreSize function</td>
</tr>
<tr>
<td>setReconnectTimeout procedure</td>
<td>getReconnectTimeout function</td>
</tr>
<tr>
<td>setReconnectInterval procedure</td>
<td>getReconnectInterval function</td>
</tr>
<tr>
<td>setClientPersistence procedure</td>
<td>getClientPersistence function</td>
</tr>
<tr>
<td>createRejectedMessageConsumer procedure</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getLocalStoreDirectory function</td>
</tr>
<tr>
<td></td>
<td>getLocalStoreSize function</td>
</tr>
<tr>
<td></td>
<td>getReconnectTimeout function</td>
</tr>
<tr>
<td></td>
<td>getReconnectInterval function</td>
</tr>
<tr>
<td></td>
<td>getClientPersistence function</td>
</tr>
</tbody>
</table>

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

**Note:** Creating a Rejected Message Listener is also optional. This listener notifies the client when a message is rejected during playback.

The caller must ensure that the connections to the machine and port number are correct. It is possible for messages to be lost if an incorrect broker is specified. Although the messages will be persisted to disk, they will never be sent since there will never be a broker to connect to.

**Note:** Client persistence does not support Message Consumers and transacted sessions.

For an example of client persistence, see the “Client persistence” section on page 5–12.

**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 Brokers 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.

Table 4–6 lists the methods for managing fault tolerant connections.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Getting</th>
</tr>
</thead>
<tbody>
<tr>
<td>setFaultTolerant procedure</td>
<td>isFaultTolerant function</td>
</tr>
<tr>
<td>setClientTransactionBufferSize procedure</td>
<td>getClientTransactionBufferSize function</td>
</tr>
<tr>
<td>setInitialConnectionTimeout procedure</td>
<td>getClientTransactionBufferSize function</td>
</tr>
<tr>
<td>setFaultTolerantReconnectTimeout procedure</td>
<td>getInitialConnectionTimeout function</td>
</tr>
<tr>
<td>createChangeStateConsumer procedure</td>
<td>getFaultTolerantReconnectTimeout function</td>
</tr>
</tbody>
</table>

Establishing session control

Table 4–7 lists the methods an application uses to manage session control.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>startReceiveMessages procedure</td>
<td>Starts receiving messages after creating a new session or after calling</td>
</tr>
<tr>
<td></td>
<td>stopReceiveMessages procedure</td>
</tr>
<tr>
<td>stopReceiveMessages procedure</td>
<td>Causes the SonicMQ Broker to stop delivering messages to the OpenEdge client</td>
</tr>
<tr>
<td>deleteSession procedure</td>
<td>Closes a session and its underlying connection and deletes the session</td>
</tr>
<tr>
<td>getConnectionID function</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 publish procedure call, the sendToQueue procedure call, or the requestReply procedure call. These values cannot be changed after the beginSession procedure is called.

Table 4–8 lists the methods for setting and getting delivery parameters.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Getting</th>
</tr>
</thead>
<tbody>
<tr>
<td>setDefaultPriority procedure</td>
<td>getDefaultPriority function</td>
</tr>
<tr>
<td>setDefaultTimeToLive procedure</td>
<td>getDefaultTimeToLive function</td>
</tr>
<tr>
<td>setDefaultPersistency procedure</td>
<td>getDefaultPersistency function</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 JMS-MAXIMUM-MESSAGES global variable definition.

Note: If you exceed the limit on the JMS-MAXIMUM-MESSAGES global variable, 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 and the setDefaultPersistency procedure.
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.

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 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 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 and getSelectorAtBroker function. 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
- Finding administered objects in JNDI or proprietary directories

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 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:

```
RUN jms/jmssession.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. Table 4–9 lists the connection parameters that can be set for each connection type.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Connection type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Broker-side selectors</td>
<td>All¹</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>
</tbody>
</table>

¹ Not all broker-side selectors can be set for all connection types.
Parameter values are fixed when the `beginSession` procedure is called. When the `beginSession` procedure 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.
Figure 4–1 depicts the process for determining the parameter value for each parameter during execution of the `beginSession` procedure.

![Connection parameter value decision flow](image)

**Figure 4–1:**  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:

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:

![Connection Configuration](image)

**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 the “Setting the CLASSPATH” section on page 4–22.) 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.
Externally managed connections

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;
    }
}
```

**Notes:** 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 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 the “Setting the CLASSPATH” section on page 4–22.
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
- Creating, populating, and accessing messages
- Sending messages to a queue
- Publishing messages to a topic
- Clearing messages
- Deleting messages
- Accessing message header properties
- Accessing message properties

Message life cycle

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

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, createDataSetMessage procedure, createHeaderMessage procedure, createMapMessage procedure, createMultipartMessage procedure, createStreamMessage procedure, createTempTableMessage procedure, or createXMLMessage procedure.

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 to delete the 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 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
- HeaderMessage
- MapMessage
- StreamMessage
- BytesMessage
- MultipartMessage
- XMLMessage
- DataSetMessage
- TempTableMessage
- Java Object messages
- Message size limits
- Storing and extracting data

TextMessage

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

Table 4–10 lists the methods for handling text messages.

Table 4–10: 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</td>
<td>setText procedure</td>
<td>endOfStream function</td>
</tr>
<tr>
<td></td>
<td>setLongText procedure</td>
<td>getCharCount function</td>
</tr>
<tr>
<td></td>
<td>appendText procedure</td>
<td>getText function</td>
</tr>
<tr>
<td></td>
<td>reset procedure</td>
<td>getTextSegment function</td>
</tr>
<tr>
<td></td>
<td>clearBody procedure</td>
<td>getLongText function</td>
</tr>
<tr>
<td></td>
<td></td>
<td>getLongTextCP function</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 the “Read-only and write-only modes” section on page 4–33.

**Note:** The 8K segment size is guaranteed. An OpenEdge application need not use the endOfStream function for messages smaller than 8K, since there is only one segment. For information about code page conversions and text size limits, see the “XML code page encoding” section on page B–15.

**HeaderMessage**

A HeaderMessage is a header-only message type that handles bodyless JMS messages. Use the createHeaderMessage procedure to handle header messages. See the “Accessing message header properties” section on page 4–35 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.

Table 4–11 lists the methods for handling map messages.

**Table 4–11: 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</td>
<td>setBoolean procedure</td>
<td>getMapNames function</td>
</tr>
<tr>
<td>procedure</td>
<td>setByte procedure</td>
<td>getItemType function</td>
</tr>
<tr>
<td></td>
<td>setBytesFromRaw procedure</td>
<td>getBytesToRaw function</td>
</tr>
<tr>
<td></td>
<td>setChar procedure</td>
<td>getChar function</td>
</tr>
<tr>
<td></td>
<td>setDate procedure</td>
<td>getDate function</td>
</tr>
<tr>
<td></td>
<td>setDateTime procedure</td>
<td>getDateTime function</td>
</tr>
<tr>
<td></td>
<td>setDateTime-TZ procedure</td>
<td>getDateTime-TZ function</td>
</tr>
<tr>
<td></td>
<td>setDouble procedure</td>
<td>getDecimal function</td>
</tr>
<tr>
<td></td>
<td>setFloat procedure</td>
<td>getInt function</td>
</tr>
<tr>
<td></td>
<td>setInt procedure</td>
<td>getInt64 function</td>
</tr>
<tr>
<td></td>
<td>setInt64 procedure</td>
<td>getLogical function</td>
</tr>
<tr>
<td></td>
<td>setLong procedure</td>
<td>getLongString function</td>
</tr>
<tr>
<td></td>
<td>setLongString procedure</td>
<td>getLongStringCP function</td>
</tr>
<tr>
<td></td>
<td>setShort procedure</td>
<td></td>
</tr>
<tr>
<td></td>
<td>setString procedure</td>
<td></td>
</tr>
<tr>
<td></td>
<td>clearBody procedure</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.

Table 4–12 lists the methods for handling stream messages.

Table 4–12: 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</td>
<td>writeBoolean procedure</td>
<td>endOfStream function</td>
</tr>
<tr>
<td></td>
<td>writeByte procedure</td>
<td>moveToNext procedure</td>
</tr>
<tr>
<td></td>
<td>writeBytesFromRaw procedure</td>
<td>readBytesToRaw procedure</td>
</tr>
<tr>
<td></td>
<td>writeChar procedure</td>
<td>readChar function</td>
</tr>
<tr>
<td></td>
<td>writeDate procedure</td>
<td>readDate function</td>
</tr>
<tr>
<td></td>
<td>writeDateTime procedure</td>
<td>readDateTime function</td>
</tr>
<tr>
<td></td>
<td>writeDateTime-TZ procedure</td>
<td>readDateTime-TZ function</td>
</tr>
<tr>
<td></td>
<td>writeDouble procedure</td>
<td>readDecimal function</td>
</tr>
<tr>
<td></td>
<td>writeFloat procedure</td>
<td>readInt function</td>
</tr>
<tr>
<td></td>
<td>writeInt procedure</td>
<td>readInt64 function</td>
</tr>
<tr>
<td></td>
<td>writeInt64 procedure</td>
<td>readLogical function</td>
</tr>
<tr>
<td></td>
<td>writeLong procedure</td>
<td>readLongString function</td>
</tr>
<tr>
<td></td>
<td>writeLongString procedure</td>
<td>readLongStringCP function</td>
</tr>
<tr>
<td></td>
<td>writeShort procedure</td>
<td></td>
</tr>
<tr>
<td></td>
<td>writeString procedure</td>
<td></td>
</tr>
<tr>
<td></td>
<td>reset procedure</td>
<td></td>
</tr>
<tr>
<td></td>
<td>clearBody procedure</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.

Table 4–13 lists the methods for handling bytes messages.

Table 4–13: 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>createBytesMessage</td>
<td>setMemptr procedure</td>
<td>readBytesToRaw procedure</td>
</tr>
<tr>
<td>procedure</td>
<td>writeBytesFromRaw procedure</td>
<td>setMemptr function</td>
</tr>
<tr>
<td></td>
<td>reset procedure</td>
<td>eofStream function</td>
</tr>
<tr>
<td></td>
<td>clearBody procedure</td>
<td>getBytesCount function</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.

Notes: 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 the “Publishing, subscribing, and receiving an XML document in a BytesMessage” section on page B–13.

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 the “XML code page encoding” section on page B–15. For an example, see the “Publishing, receiving, and parsing an XMLMessage” section on page B–11.

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.
Table 4–14 lists the methods for handling multi-part messages.

### Table 4–14: 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</td>
<td>addMessagePart procedure</td>
<td>isMessagePart function</td>
</tr>
<tr>
<td>procedure</td>
<td>addBytesPart procedure</td>
<td>getBytesPartByIndex function</td>
</tr>
<tr>
<td></td>
<td>addTextPart procedure</td>
<td>getBytesPartByID function</td>
</tr>
<tr>
<td></td>
<td>appendText procedure</td>
<td>getContentType,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>getPartCount</td>
</tr>
<tr>
<td></td>
<td></td>
<td>getTextPartByIndex function</td>
</tr>
<tr>
<td></td>
<td></td>
<td>getMemptr function</td>
</tr>
<tr>
<td></td>
<td></td>
<td>getMessagePartByIndex function</td>
</tr>
<tr>
<td></td>
<td></td>
<td>clearBody procedure</td>
</tr>
</tbody>
</table>

**Notes:** 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. Table 4–15 lists the methods for handling XML messages.

### Table 4–15: 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</td>
<td>setText procedure</td>
<td>endOfStream function</td>
</tr>
<tr>
<td></td>
<td>setLongText procedure</td>
<td>getCharCount function</td>
</tr>
<tr>
<td></td>
<td>appendText procedure</td>
<td>getText function</td>
</tr>
<tr>
<td></td>
<td>reset procedure</td>
<td>getTextSegment function</td>
</tr>
<tr>
<td></td>
<td>setX-Document procedure</td>
<td>getLongText function</td>
</tr>
<tr>
<td></td>
<td>setSaxReader procedure</td>
<td>getLongTextCP function</td>
</tr>
<tr>
<td></td>
<td>deleteSaxWriter procedure</td>
<td>getSaxWriter function</td>
</tr>
<tr>
<td></td>
<td>clearBody procedure</td>
<td>getX-Document function</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 `-cpinternal` startup parameter). For more information on the `-cpinternal` startup parameter, see *OpenEdge Deployment: Startup Command and Parameter Reference*.

The ABL–JMS implementation automatically converts the text to 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 the “XML code page encoding” section on page B–15.

**DataSetMessage**

A `DataSetMessage` is a message type whose body contains a ProDataSet. The information sent and received is based upon the existing `XMLMessage`. ABL has built in support to transform a ProDataSet into XML. Table 4–16 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><code>createDataSetMessage</code></td>
<td><code>setDataSet</code> procedure</td>
<td><code>getDataSet</code> function</td>
</tr>
<tr>
<td></td>
<td><code>reset</code> procedure</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>clearBody</code> procedure</td>
<td></td>
</tr>
</tbody>
</table>

For an example of a `DataSetMessage`, see the “`DataSetMessage`” section on page 5–24.
**TempTableMessage**

A TempTableMessage is a message type whose body contains a temp-table. The information sent and received is based upon the existing XMLMessage. ABL has built-in support to transform a temp-table into XML. Table 4–17 lists the methods for handling TempTable messages.

**Table 4–17: 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>createTempTableMessage procedure</td>
<td>setTempTable procedure reset procedure clearBody procedure</td>
<td>getTempTable function</td>
</tr>
</tbody>
</table>

For an example of a TempTableMessage, see the “TempTableMessage” section on page 5–21.

**Java Object messages**

The ABL–JMS API 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 TextMessage with the header of the Java Object message and a text body with the string “ObjectMessage: Not Supported.” (For more information, see the “Error and condition handling” section on page 4–46.)

**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 the “XML code page encoding” section on page B–15.

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 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 extracts a value (for example, an `INTEGER` value from a `StreamMessage`) and converts it to a `String` value.

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

Table 4–18 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>
<tr>
<td>DECIMAL</td>
<td>float</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>double</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>String</td>
</tr>
<tr>
<td>LONGCHAR</td>
<td>String</td>
</tr>
<tr>
<td>A single CHARACTER</td>
<td>char</td>
</tr>
<tr>
<td>RAW</td>
<td>byte array</td>
</tr>
<tr>
<td>MEMPTR</td>
<td>byte array (only with BytesMessage)</td>
</tr>
<tr>
<td>DATE</td>
<td>String</td>
</tr>
<tr>
<td>DATETIME</td>
<td>String</td>
</tr>
<tr>
<td>DATETIME-TZ</td>
<td>String</td>
</tr>
</tbody>
</table>
Table 4–19 maps the available conversions from JMS data types to ABL data types for data extraction.

### Table 4–19: 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** puts the cursor before the first item of the message and transfers it to read-only mode.

**Note:** The **publish procedure**, **sendToQueue procedure**, and **requestReply procedure** call the **reset procedure** implicitly.

The message is received by the receiver in read-only mode. The **clearBody procedure** 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 deletes all data from the message body.
- The clearProperties procedure 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 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 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. The clearProperties procedure deletes all header properties (but not the JMS-predefined header fields).
Deleting messages

An application explicitly deletes a message using the `deleteMessage` procedure. 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.

Table 4–20 lists the methods to access message header information.

Table 4–20: Accessing message header information

<table>
<thead>
<tr>
<th>Set header methods</th>
<th>Get header methods</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>setJMSReplyTo</code> procedure</td>
<td><code>getJMSReplyTo</code> function</td>
</tr>
<tr>
<td><code>setReplyToDestinationType</code> procedure</td>
<td><code>getReplyToDestinationType</code> function</td>
</tr>
<tr>
<td><code>setJMSCorrelationID</code> procedure</td>
<td><code>getJMSCorrelationID</code> function</td>
</tr>
<tr>
<td><code>setJMSCorrelationIDAsBytes</code> procedure</td>
<td><code>getJMSCorrelationIDAsBytes</code> function</td>
</tr>
<tr>
<td><code>setJMSType</code> procedure</td>
<td><code>getJMSType</code> function</td>
</tr>
<tr>
<td><code>getJMSDestination</code> function</td>
<td><code>getJMSDeliveryMode</code> function</td>
</tr>
<tr>
<td><code>getJMSRedelivered</code> function</td>
<td><code>getJMSExpiration</code> function</td>
</tr>
<tr>
<td><code>getMessageType</code> function</td>
<td><code>getJMSPriority</code> function</td>
</tr>
<tr>
<td><code>getJMSMessageID</code> function</td>
<td><code>hasReplyTo</code> function</td>
</tr>
<tr>
<td><code>getJMSDeliveryMode</code> function</td>
<td></td>
</tr>
<tr>
<td><code>getJMSTimestamp</code> function</td>
<td></td>
</tr>
<tr>
<td><code>getJMSExpiration</code> function</td>
<td></td>
</tr>
<tr>
<td><code>getJMSPriority</code> function</td>
<td></td>
</tr>
<tr>
<td><code>hasReplyTo</code> function</td>
<td></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. Table 4–21 lists the methods to access message properties.

Table 4–21: Accessing message properties

<table>
<thead>
<tr>
<th>Setting message properties</th>
<th>Getting message properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>setBooleanProperty procedure</td>
<td>getCharProperty function</td>
</tr>
<tr>
<td>setByteProperty procedure</td>
<td>getDateProperty function</td>
</tr>
<tr>
<td>setDateProperty procedure</td>
<td>getDateProperty function</td>
</tr>
<tr>
<td>setDateTimeProperty procedure</td>
<td>getDateProperty function</td>
</tr>
<tr>
<td>setDateTimeTzProperty procedure</td>
<td>getDateProperty function</td>
</tr>
<tr>
<td>setDoubleProperty procedure</td>
<td>getDateProperty function</td>
</tr>
<tr>
<td>setFloatProperty procedure</td>
<td>getDateProperty function</td>
</tr>
<tr>
<td>setIntProperty procedure</td>
<td>getIntProperty function</td>
</tr>
<tr>
<td>setInt64Property procedure</td>
<td>getInt4Property function</td>
</tr>
<tr>
<td>setLongProperty procedure</td>
<td>getLogicalProperty function</td>
</tr>
<tr>
<td>setShortProperty procedure</td>
<td>getPropertyType function</td>
</tr>
<tr>
<td>setStringProperty procedure</td>
<td>getPropertyNames function</td>
</tr>
<tr>
<td>clearProperties procedure</td>
<td></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 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
- Creating a message handler process
- Setting reply properties
- Receiving messages from a queue
- Subscribing to a topic
- Terminating the Message Consumer object
- Processing messages
- Controlling flow of messages
- Reusing messages
- Message-reception issues
- Reply mechanisms

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.

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 \texttt{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 \texttt{deleteConsumer} procedure 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 \texttt{messageHandler} procedure. The OpenEdge application passes context to the message handler using the \texttt{setApplicationContext} procedure.

**Accessing message handler information**

Table 4–22 lists procedures for getting message handler properties and type of message being handled.

<table>
<thead>
<tr>
<th>Getting message handler properties</th>
<th>Type of message handled</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{getApplicationContext} function</td>
<td>\texttt{inErrorHandling} function</td>
</tr>
<tr>
<td>\texttt{getDestinationName} function</td>
<td>\texttt{inMessageHandling} function</td>
</tr>
<tr>
<td>\texttt{getProcHandle} function</td>
<td>\texttt{inQueueBrowsing} function</td>
</tr>
<tr>
<td>\texttt{getProcName} function</td>
<td>\texttt{inReplyHandling} function</td>
</tr>
<tr>
<td>\texttt{getReplyAutoDelete} function</td>
<td></td>
</tr>
<tr>
<td>\texttt{getReplyPersistency} function</td>
<td></td>
</tr>
<tr>
<td>\texttt{getReplyPriority} function</td>
<td></td>
</tr>
<tr>
<td>\texttt{getReplyTimeToLive} function</td>
<td></td>
</tr>
<tr>
<td>\texttt{getSession} function</td>
<td></td>
</tr>
</tbody>
</table>

**Setting reply properties**

The Message Consumer sets the reply properties by using the \texttt{setReplyPriority} procedure, \texttt{setReplyTimeToLive} procedure, \texttt{setReplyPersistency} procedure, and \texttt{setReplyAutoDelete} procedure.

**Receiving messages from a queue**

In the PTP domain, applications receive messages from a queue. The application calls the \texttt{receiveFromQueue} procedure in \texttt{ptpsession.p} or \texttt{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 \texttt{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 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` 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` in `pubsubsession.p` or `jmssession.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` 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` 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`.

Processing messages

To control message processing use the `waitForMessages procedure`.

Controlling flow of messages

The application controls the flow of messages to the SonicMQ client from a queue using the `setPrefetchCount procedure` and the `setPrefetchThreshold procedure`.

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.
Implementing Messaging

Reusing messages

The application sets message reuse using the `setReuseMessage` procedure and `getReuseMessage` function.

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 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 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.

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 is called, message reception is canceled and the Message Consumer object is deleted.

**Notes:** To delete a durable subscription (in the Pub/Sub domain), the `cancelDurableSubscription` procedure 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.
Consuming messages

When a Message Consumer object is used for receiving replies through the requestReply procedure call, it can be used many times; there is no need to create one for every call. The deleteSession procedure 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 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. 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.
Implementing Messaging

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 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 the “Request/Reply” section on page 4–12 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 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 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 and the getReplyToDestinationType function, both of which support the CHARACTER values topic and queue.

The setReplyToDestinationType procedure can be called if the OpenEdge application calls the setUMSReplyTo 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

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

Table 4–23 lists the methods available for controlling the execution and recovery of transactions.

Table 4–23: Managing transaction attributes

<table>
<thead>
<tr>
<th>Setting</th>
<th>Rolling back</th>
</tr>
</thead>
<tbody>
<tr>
<td>commitSend procedure</td>
<td>rollbackSend procedure</td>
</tr>
<tr>
<td>commitReceive procedure</td>
<td>rollbackReceive procedure</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, the setTransactedReceive procedure, 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 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 in a Session object, all such messages are discarded.

Transacted receiving

When an application calls the commitReceive procedure in a Session object, all messages that have been received with the current transaction are acknowledged. When an application calls the rollbackReceive procedure in a Session object, all such messages are re-received (yielding the same effect as calling the recover procedure 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 the “Error and condition handling” section on page 4–46.

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.

Table 4–24 lists the methods the application uses to set message acknowledgement.

Table 4–24: Acknowledging and forwarding messages

<table>
<thead>
<tr>
<th>Setting</th>
<th>Getting</th>
</tr>
</thead>
<tbody>
<tr>
<td>setSingleMessageAcknowledgement procedure</td>
<td>getSingleMessageAcknowledgement function</td>
</tr>
<tr>
<td>setNoAcknowledge procedure</td>
<td>getNoAcknowledge function</td>
</tr>
<tr>
<td>acknowledgeAndForward procedure</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 in the Session object. If the recover procedure is called on a session stopped by the stopReceiveMessages procedure, 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 Appendix A, “ABL–JMS API Reference”).

- 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 and `setNoErrorDisplay` procedure.

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 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 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 to get a list of properties in the error message header. See Appendix B, “Messaging Examples” for an example.
Implementing Messaging

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.

### 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:

```xml
<java-exception>::<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 beginSession procedure, which creates the connection to the OpenEdge Adapter for SonicMQ and the JMS server, reports connection failures synchronously by calling:

```abel
RETURN ERROR <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:

```xml
<java-exception>::<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 setPingInterval procedure (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 ERROR, STOP, and 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 setApplicationContext procedure call to pass an ABL procedure handle to the Message Consumer object. The message handler can obtain the procedure handle by calling the getApplicationContext function in the Message Consumer object and can then make the appropriate internal procedure calls.

- As mentioned in the “Message acknowledgement, forwarding, and recovery” section on page 4–44, the message handler can call the setNoAcknowledge procedure 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, deleteConsumer procedure, and recover procedure. 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 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.
This chapter contains instructions for accessing SonicMQ messaging from the ABL (Advanced Business Language) through the ABL–JMS API provided by the OpenEdge® Adapter for SonicMQ.

This chapter includes the following sections:

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

For an alphabetical API reference, see Appendix A, “ABL–JMS API Reference.”
PTP message example

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

- Creating a PTP session procedure
- Connecting to the broker
- Creating a Message Consumer
- Preparing to receive messages
- Sending messages to the queue
- Receiving messages from the queue
- Deleting a message

Creating a PTP session procedure

In the Creating a PTP session example, the application creates a session object by calling ptpsession.p persistently.

Creating a PTP session

```abl
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.

Connecting to the broker

In the Connecting to the broker example, the OpenEdge application connects to the SonicMQ Broker to begin exchanging messages.

Connecting to the broker

```abl
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 shown in the Creating a Message Consumer.

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

Preparing to receive messages

In the Preparing to receive messages example, the OpenEdge application begins listening on the queue and prepares to receive messages from the queue.

```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 starts receipt of incoming messages.

Sending messages to the queue

In the Sending messages to the queue example, the application sends a message to the queue using the sendToQueue procedure.

```plaintext
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 Deleting the message example.

Deleting the message

```abl
RUN deleteMessage IN hMessage.
```

Summary

The Complete code for sending a message using a PTP session sample summarizes the steps for sending a message.

Complete code for sending a message using a PTP session

```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.
```
The Complete code for receiving a message using a PTP session sample summarizes the steps for receiving a message.

**Complete code for receiving a message using a PTP session**

```plaintext
/* 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).
/* 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
- Connecting to the broker
- Creating a Message Subscriber
- Subscribing to a topic
- Publishing to a topic
- Consuming messages from a topic
- Deleting messages
- Summary

Creating a Pub/Sub session procedure

In the Creating a Pub/Sub session example, the application creates a session object by calling `pubsubsession.p` persistently.

**Creating a Pub/Sub session**

```
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.

Connecting to the broker

In the Connecting to the broker example, the OpenEdge application connects to the SonicMQ Broker to begin exchanging messages.

**Connecting to the broker**

```
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 Creating a Message Subscriber example.

```
/* 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 Subscribing to a topic 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 Publishing message to a topic 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 Deleting the message example.

Deleting the message

RUN deleteMessage IN hMessage.

Summary

The Complete code for publish a message using a Pub/Sub session sample summarizes the steps for sending a message.

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

```abl
/* 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.
```
The Complete code for receiving a message using a Pub/Sub session sample summarizes the steps for receiving a message.

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

```plaintext
/* 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 ("ltcp://machinename:2506").
RUN beginSession IN hPubSubSession.

/* Subscribes to the newTopic topic. Received messages are handled by the
myintproc internal procedure. */
RUN createMessageConsumer IN hPubSubSession
    (THIS-PROCEDURE, "myintproc", OUTPUT hConsumer).

/* 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.

/* 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
- Using ServerConnect and ClientConnect
- Client persistence
- Enhanced XML support
- TempTableMessage
- DataSetMessage
- Fault tolerance

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/jmssession.p PERSISTENT SET hSession (adapterConnection).
```

Notes: The ModChat example demonstrates using the jmssession 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” section on page Preface–7. For an alphabetical API reference, see Appendix A, “ABL–JMS API Reference.”
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 the “Client persistence” section on page 4–8. For more information on fault tolerance, see the “Fault tolerance” section on page 4–9.

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

In the Using SMQConnect on a client example, the application creates a session procedure by calling jmsession.p persistently specifying the -SMQConnect connection parameter.

### Using SMQConnect on a client

```
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.
Using ServerConnect

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:

   ![Screenshot of SonicMQ ServerConnect settings](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 the “Client persistence” section on page 4–8.
The **Client persistence example** code sample shows how to set up client persistence.

### Client persistence example

```plaintext
DEFINE VARIABLE hSession AS HANDLE NO-UNDO.
DEFINE VARIABLE rejectedMsgH AS HANDLE NO-UNDO.

/* Run adapter as symbiotic process */
RUN jms/jmssession.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.
   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 hMessagePart AS HANDLE NO-UNDO.
   DEFINE VARIABLE iNumParts AS INTEGER 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, 1) = TRUE THEN DO:
      contentType = DYNAMIC-FUNCTION("getMessagePartByIndex":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 the “OpenEdge messages” section on page Preface–7.

The Saxwriter message example shows how to use the SAX-WRITER object.

Saxwriter message

```abl
DEFINE VARIABLE hSaxWriter AS HANDLE NO-UNDO.
DEFINE VARIABLE hSession AS HANDLE NO-UNDO.
DEFINE VARIABLE hMesg AS HANDLE NO-UNDO.

RUN jms/jmsession.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").
```
The setSaxReader example code sample shows how to use the setSaxReader procedure:

**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 saxSender.p example shows how to use the SAX-WRITER object.

saxSender.p

```abl
DEFINE VARIABLE hdl1 AS HANDLE NO-UNDO.
DEFINE VARIABLE hdl2 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.*/
hd12 = DYNAMIC-FUNCTION('GetSaxWriter':u IN hMesg, ?).
hd12: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 hdl1.
hd11:SET-INPUT-SOURCE("FILE", "personal.xml").
hd11:SAX-PARSE().
DELETE OBJECT hd11.

hd12: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 (hd12).

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

saxReceiver.p

```plaintext
DEFINE VARIABLE hd12 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 hMsg 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).
  hdl2:START-DOCUMENT().
  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 iNumChars 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:
  DEFINE INPUT PARAMETER namespaceURI AS CHARACTER NO-UNDO.
  DEFINE INPUT PARAMETER cLocalName AS CHARACTER NO-UNDO.
  DEFINE INPUT PARAMETER qName AS CHARACTER NO-UNDO.

  hdl2:END-ELEMENT(cLocalName, namespaceURI).
END.
The setX-Document example code sample shows how to use the setX-Document procedure.

**setX-Document example**

```
CREATE X-DOCUMENT hdl1.
hdl1: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 getX-Document example sample code shows how to use getX-Document function.

**getX-Document example**

```
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 domSender.p example shows how to use X-DOCUMENT object to send a message.

**domSender.p**

```abl
DEFINE VARIABLE hdl1 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).

/* Load the XML file. */
CREATE X-DOCUMENT hdl1.
hdll:LOAD("file", "personal.xml", FALSE).

/* The Adapter function SetX-Document will copy the XML from the X-DOCUMENT into the XML message. */
RUN setX-Document IN hMesg(hdl1).
DELETE OBJECT hdl1.

/* 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.
```

The X-DOCUMENT object reads an XMLMessage from a queue and writes it to a LONGCHAR. The domReceiver.p example shows how to use X-DOCUMENT object to receive a message.

**domReceiver.p**

```abl
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.
```
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 the “OpenEdge messages” section on page Preface–7.

For an alphabetical API reference, see Appendix A, “ABL–JMS API Reference.”
The `ttmsgs_send.p` example shows sending a `TempTableMessage`.

```abl
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 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, ?, ?, ?).
ENDIF
RUN deleteMessage IN hMessage.
DELETE 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, ?, ?, ?).
ENDIF
RUN deleteMessage IN hMessage.
RUN deleteSession IN hSession.
```
The ttmsg_recv.p example shows receiving a TempTableMessage.

### ttmsg_recv.p

```plaintext
DEFINE VARIABLE hMsgConsumer AS HANDLE NO-UNDO.
DEFINE VARIABLE hMsg 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 qh1 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.
```

5–23
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 the “OpenEdge messages” section on page Preface–7.

For an alphabetical API reference, see Appendix A, “ABL–JMS API Reference.”
The `dsmsg_send.p` 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 Name
  INDEX Comments 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.
END. /* FOR EACH Order */
END. /* FOR EACH Customer */

hds1 = DATASET myds:HANDLE.

/* Uncomment to write XML to a file
*/
```
Programming for the OpenEdge Adapter for SonicMQ with the ABL–JMS API

**dsmgs_send.p**

```abl
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 `dsmgs_recv.p` example shows receiving a DataSetMessage.

**dsmgs_recv.p**

```abl
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 hSession 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.
  /* hAutoReply is not used in this example */
  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
    "nspace-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:
        MESSAGE
        "name:        " hBuf:BUFFER-FIELD(ix):NAME skip
        "type:        " hBuf:BUFFER-FIELD(ix):DATA-TYPE skip
        "extent:      " hBuf:BUFFER-FIELD(ix):EXTENT skip
        "decimals:    " hBuf:BUFFER-FIELD(ix):DECIMALS skip
        "xmltype:     " hBuf:BUFFER-FIELD(ix):XML-DATA-TYPE skip
        "xmlnodetype: " hBuf:BUFFER-FIELD(ix):XML-NODE-TYPE skip
        "initial:     " hBuf:BUFFER-FIELD(ix):INITIAL skip
        "format:      " hBuf:BUFFER-FIELD(ix):FORMAT skip
        "help:        " hBuf:BUFFER-FIELD(ix):HELP skip
        VIEW-AS ALERT-BOX.
      END.
    END.
  END.
END.
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, the getFaultTolerant function, the isFaultTolerant function, the setConnectionURLs procedure, the setFaultTolerantReconnectTimeout procedure, the getFaultTolerantReconnectTimeout function, the setInitialConnectionTimeout procedure, the getInitialConnectionTimeout function, the setClientTransactionBufferSize procedure, the getClientTransactionBufferSize function, and the createChangeStateConsumer procedure. 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
- Example of a “ChangeState” handler (optional)

**Example of setting up fault tolerance**

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

**Fault tolerance set up**

```plaintext
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 after to calling the beginSession procedure.

The Fault tolerant example code sample shows how to use the createChangeStateConsumer procedure.

**Fault tolerant example**

```plaintext
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 will be "active", "reconnecting", "failed", or "closed" */
    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 chapter 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 manuals, as indicated in this chapter.

The chapter includes the following sections:

- 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 Appendix C, “Sample Native Invocation ESB process.”

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. Figure 6–1 graphically depicts the creation of an invocation .esboe file, following the declarative or non-declarative approach.

![Development flow: ABL source to ESB itinerary](image-url)
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. Table 6–1 describes the annotation name-value pairs.

Table 6–1: Annotation name-value pairs

<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>
</tbody>
</table>
### Annotation name-value pairs (2 of 2)

<table>
<thead>
<tr>
<th>Name</th>
<th>Default value</th>
<th>Mandatory/Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>useReturnValue</td>
<td>false</td>
<td>Optional</td>
<td>Specifies whether a return string is generated or not for a procedure. Specify <code>useReturnValue=&quot;true&quot;</code> to generate a return string. If <code>useReturnValue</code> 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 <code>writeDataSetBeforeImage=&quot;true&quot;</code> to write the before-image data. Specify <code>writeDataSetBeforeImage=&quot;false&quot;</code> to not write the before-image data. If no ProDataSet parameters exist, <code>writeDataSetBeforeImage</code> 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 the “Annotating ABL in OpenEdge Architect” section on page 6–7. 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:

```
/* 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.

@openapi.openedge.export(type="ESB")
PROCEDURE barr.
DEFINE INPUT PARAM abc AS INT.
DEFINE OUTPUT PARAM xyz 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:

```
/* 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 the “Configuring OpenEdge Architect for ESB annotations” section on page 6–25 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 the “Annotation syntax” section on page 6–4.

  **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.

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.

**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.

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**.

**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:

   ![New ESB Annotation Dialog Box](image)

   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 the “Configuring OpenEdge Architect for ESB annotations” section on page 6–25.

- 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” section on page 6–24 for details.

Note: You can generate .esboe files from unnannotated ABL code with ProxyGen. See the “Non-declarative invocation files” section on page 6–9 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.

**Note:** The following steps are not intended as a detailed tutorial for Sonic Workbench. For details about Sonic Workbench, consult your Sonic documentation.

**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. Chose 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 chose 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:

   ![Response Mapping Tab](image)

   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:

\[
\text{<ABLparameter>_<ABLdatatype>_<ABLparameter-type>}
\]
Table 6–2 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 6–2: 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.

Figure 6–2 shows the three invocation files in a sample process.

![Image of Internal procedure invocation](image)

**Figure 6–2: Internal procedure invocation**

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.

Figure 6–3 shows the 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:

![Wizard for selecting ABL procedure](image)

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.

   **Note:** 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.

   c. 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.
d. 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 appears:

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 the “Annotating ABL in OpenEdge Architect” section on page 6–7 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:

![Diagram of ESB process]

**Note:** 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. Figure 6–4 shows the Service tab detail.

![ESB process Service tab detail](image)

Figure 6–4: 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 Figure 6–5.

**Figure 6–5: OpenEdge Invocation detail**

**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 Step 2, 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:

**Syntax**

```
esboegen [-source directory] [-esboe directory | -archive filename] [-rcode directory] [-recurse] [files]
```

- **-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
- Generating ESB invocation files

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 Start → Programs → 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 box](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 dialog box](image)

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.
Figure 6–6 shows the OpenEdge services in the Sonic Management Console.

![Sonic Management Console showing OpenEdge Adapter for Sonic ESB installed](image)

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.
Figure 6–7 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 6–7:** 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.

This appendix includes the following sections:

- Session objects
- Methods in the Session objects
- Methods in the Message Consumer objects
- Methods in the Message objects

For more information on using this information in context, see Chapter 5, “Programming for the OpenEdge Adapter for SonicMQ with the ABL–JMS API.”
Session objects

The following are the Session objects:

- jmssession.p
- ptpsession.p
- pubsubsession.p

For information on the methods available for the Session objects, see the “Methods in the Session objects” section on page A–4.

jmssession.p

Starts a JMS session using the unified domain to access both queues and topics in one JMS session object.

Syntax

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

Parameters

`adapterConnection`

Desired adapter connection options. For more information, see the “Connection options” section on page 4–4.

Applies to

Session objects

See also

jmssession.p, ptpsession.p, pubsubsession.p

For more information, see the “Methods in the Session objects” section on page A–4.

ptpsession.p

Starts a JMS session using queues.

Syntax

```abl
RUN jms/ptpsession.p PERSISTENT SET ptpsession (adapterConnection).
```

Parameters

`adapterConnection`

Desired adapter connection options. For more information, see the “Connection options” section on page 4–4.

Applies to

Session objects

See also

jmssession.p, ptpsession.p, pubsubsession.p

For more information, see the “Methods in the Session objects” section on page A–4.
pubsubsession.p

Starts a JMS session using topics.

**Syntax**

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

**Parameters**

`adapterConnection`

Desired adapter connection options. For more information, see the “Connection options” section on page 4–4.

**Applies to**

Session objects

**See also**

jmssession.p, ptpsession.p, pubsubsession.p

For more information, see the “Methods in the Session objects” section on page A–4.
Methods in the Session objects

Table A–1 lists the methods in the Session objects.

<table>
<thead>
<tr>
<th>Method in the Session objects</th>
<th>Method in the Session objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>beginSession procedure</td>
<td>browseQueue procedure</td>
</tr>
<tr>
<td>cancelDurableSubscription procedure</td>
<td>commitReceive procedure</td>
</tr>
<tr>
<td>commitSend procedure</td>
<td>createBytesMessage procedure</td>
</tr>
<tr>
<td>createChangeStateConsumer procedure</td>
<td>createDataSetMessage procedure</td>
</tr>
<tr>
<td>createHeaderMessage procedure</td>
<td>createMapMessage procedure</td>
</tr>
<tr>
<td>createMessageConsumer procedure</td>
<td>createMultipartMessage procedure</td>
</tr>
<tr>
<td>createRejectedMessageConsumer procedure</td>
<td>createStreamMessage procedure</td>
</tr>
<tr>
<td>createTemporaryQueue procedure</td>
<td>createTemporaryTopic procedure</td>
</tr>
<tr>
<td>createTempTableMessage procedure</td>
<td>createTextMessage procedure</td>
</tr>
<tr>
<td>createXMLMessage procedure</td>
<td>deleteSession procedure</td>
</tr>
<tr>
<td>deleteTemporaryQueue procedure</td>
<td>deleteTemporaryTopic procedure</td>
</tr>
<tr>
<td>getAdapterService function</td>
<td>getBrokerURL function</td>
</tr>
<tr>
<td>getClientID function</td>
<td>getClientPersistence function</td>
</tr>
<tr>
<td>getClientTransactionBufferSize function</td>
<td>getConnectID function</td>
</tr>
<tr>
<td>getConnectionID function</td>
<td>getConnectionMetaData function</td>
</tr>
<tr>
<td>getConnectionURLs function</td>
<td>getDiFaultTolerantPersistency function</td>
</tr>
<tr>
<td>getDefaultPriority function</td>
<td>getDiFaultTolerantReconnectTimeout function</td>
</tr>
<tr>
<td>getFaultTolerant function</td>
<td>getDiFaultTolerantReconnectTimeout function</td>
</tr>
<tr>
<td>getFlowToDisk function</td>
<td>getInitialConnectionTimeout function</td>
</tr>
<tr>
<td>getJMSName function</td>
<td>getLoadBalancing function</td>
</tr>
<tr>
<td>getLocalStoreDirectory function</td>
<td>getLocalStoreSize function</td>
</tr>
<tr>
<td>getLocalStoreWaitTime function</td>
<td>getPassword function</td>
</tr>
<tr>
<td>getReconnectInterval function</td>
<td>getReconnectTimeout function</td>
</tr>
<tr>
<td>getSelectorAtBroker function</td>
<td>getSequential function</td>
</tr>
<tr>
<td>getShutdownWaitFor function</td>
<td>getSingleMessageAcknowledgement function</td>
</tr>
<tr>
<td>getTransactedReceive function</td>
<td>getTransactedSend function</td>
</tr>
</tbody>
</table>
Table A–1: Methods in Session objects

<table>
<thead>
<tr>
<th>Method</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>getUser function</td>
<td>isFaultTolerant function</td>
</tr>
<tr>
<td>publish procedure</td>
<td>receiveFromQueue procedure</td>
</tr>
<tr>
<td>recover procedure</td>
<td>requestReply procedure</td>
</tr>
<tr>
<td>rollbackReceive procedure</td>
<td>rollbackSend procedure</td>
</tr>
<tr>
<td>sendToQueue procedure</td>
<td>setAdapterService procedure</td>
</tr>
<tr>
<td>setBrokerURL procedure</td>
<td>setClientID procedure</td>
</tr>
<tr>
<td>setClientPersistence procedure</td>
<td>setClientTransactionBufferSize procedure</td>
</tr>
<tr>
<td>setConnectID procedure</td>
<td>setConnectionFile procedure</td>
</tr>
<tr>
<td>setConnectionURLs procedure</td>
<td>setDefaultPersistency procedure</td>
</tr>
<tr>
<td>setDefaultPriority procedure</td>
<td>setDefaultTimeToLive procedure</td>
</tr>
<tr>
<td>setFaultTolerant procedure</td>
<td>setFaultTolerantReconnectTimeout procedure</td>
</tr>
<tr>
<td>setInitialConnectionTimeout procedure</td>
<td>setErrorHandler procedure</td>
</tr>
<tr>
<td>setFlowToDisk procedure</td>
<td>setJMServerName procedure</td>
</tr>
<tr>
<td>setLoadBalancing procedure</td>
<td>setLocalStoreDirectory procedure</td>
</tr>
<tr>
<td>setLocalStoreSize procedure</td>
<td>setLocalStoreWaitTime procedure</td>
</tr>
<tr>
<td>setNoErrorDisplay procedure</td>
<td>setReconnectInterval procedure</td>
</tr>
<tr>
<td>setReconnectDisplay procedure</td>
<td>setPassword procedure</td>
</tr>
<tr>
<td>setPingInterval procedure</td>
<td>setPrefetchCount procedure</td>
</tr>
<tr>
<td>setPrefetchThreshold procedure</td>
<td>setSelectorAtBroker procedure</td>
</tr>
<tr>
<td>setSequential procedure</td>
<td>setShutdownWaitFor procedure</td>
</tr>
<tr>
<td>setSingleMessageAcknowledgement procedure</td>
<td>setTransactedReceive procedure</td>
</tr>
<tr>
<td>setTransactedSend procedure</td>
<td>setUser procedure</td>
</tr>
<tr>
<td>startReceiveMessages procedure</td>
<td>stopReceiveMessages procedure</td>
</tr>
<tr>
<td>subscribe procedure</td>
<td>waitForMessages procedure</td>
</tr>
</tbody>
</table>
Methods in the Message Consumer objects

Table A–2 lists the methods in the Message Consumer objects.

Table A–2: Methods in Message Consumer Objects

<table>
<thead>
<tr>
<th>Method Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>acknowledgeAndForward procedure</td>
</tr>
<tr>
<td>deleteConsumer procedure</td>
</tr>
<tr>
<td>getApplicationContext function</td>
</tr>
<tr>
<td>getDestinationName function</td>
</tr>
<tr>
<td>getNoAcknowledge function</td>
</tr>
<tr>
<td>getProcHandle function</td>
</tr>
<tr>
<td>getProcName function</td>
</tr>
<tr>
<td>getReplyAutoDelete function</td>
</tr>
<tr>
<td>getReplyPersistency function</td>
</tr>
<tr>
<td>getReplyPriority function</td>
</tr>
<tr>
<td>getReplyTimeToLive function</td>
</tr>
<tr>
<td>getReuseMessage function</td>
</tr>
<tr>
<td>getSession function</td>
</tr>
<tr>
<td>inErrorHandling function</td>
</tr>
<tr>
<td>inMessageHandling function</td>
</tr>
<tr>
<td>inReplyHandling function</td>
</tr>
<tr>
<td>setApplicationContext procedure</td>
</tr>
<tr>
<td>setReplyPersistency procedure</td>
</tr>
<tr>
<td>setReplyPriority procedure</td>
</tr>
<tr>
<td>setReplyTimeToLive procedure</td>
</tr>
<tr>
<td>setReuseMessage procedure</td>
</tr>
</tbody>
</table>

For more information, see the “Message Consumer objects” section on page 2–5.
Methods in the Message objects

Table A–3 lists the methods in the Message objects.

**Note:** Not all methods are in all message types.

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Method Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>addBytesPart procedure</td>
<td>addMessagePart procedure</td>
</tr>
<tr>
<td>addTextPart procedure</td>
<td>appendText procedure</td>
</tr>
<tr>
<td>clearBody procedure</td>
<td>clearProperties procedure</td>
</tr>
<tr>
<td>deleteMessage procedure</td>
<td>deleteSaxWriter procedure</td>
</tr>
<tr>
<td>endOfStream function</td>
<td>getBytesCount function</td>
</tr>
<tr>
<td>getBytesPartByID function</td>
<td>getBytesPartByIndex function</td>
</tr>
<tr>
<td>getBytesToRaw function</td>
<td>getChar function</td>
</tr>
<tr>
<td>getCharCount function</td>
<td>getCharProperty function</td>
</tr>
<tr>
<td>getConnectionURLs function</td>
<td>getContentType</td>
</tr>
<tr>
<td>getDataSet function</td>
<td>getDate function</td>
</tr>
<tr>
<td>getDateProperty function</td>
<td>getDateTime function</td>
</tr>
<tr>
<td>getDateTimeProperty function</td>
<td>getDateTime-TZ function</td>
</tr>
<tr>
<td>getDateTimeTzProperty function</td>
<td>getDecimal function</td>
</tr>
<tr>
<td>getDecimalProperty function</td>
<td>getInt function</td>
</tr>
<tr>
<td>getIntProperty function</td>
<td>getInt64 function</td>
</tr>
<tr>
<td>getInt64Property function</td>
<td>getItemType function</td>
</tr>
<tr>
<td>getJMSCorrelationID function</td>
<td>getJMSCorrelationIDAsBytes function</td>
</tr>
<tr>
<td>getJMSDeliveryMode function</td>
<td>getJMSDestination function</td>
</tr>
<tr>
<td>getJMSExpiration function</td>
<td>getJMSMessageID function</td>
</tr>
<tr>
<td>getJMSExpiration function</td>
<td>getJMSRedelivered function</td>
</tr>
<tr>
<td>getJMSReplyTo function</td>
<td>getJMSTimestamp function</td>
</tr>
<tr>
<td>getJMSType function</td>
<td>getLoadBalancing function</td>
</tr>
<tr>
<td>getLogical function</td>
<td>getLogicalProperty function</td>
</tr>
<tr>
<td>getLongString function</td>
<td>getLongStringCP function</td>
</tr>
<tr>
<td>getLongText function</td>
<td>getLongTextCP function</td>
</tr>
<tr>
<td>getMapNames function</td>
<td>getMemptr function</td>
</tr>
<tr>
<td>Method in Message Objects</td>
<td></td>
</tr>
<tr>
<td>------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>getMessagePartByID function</td>
<td></td>
</tr>
<tr>
<td>getMessagePartByIndex function</td>
<td></td>
</tr>
<tr>
<td>getMessageType function</td>
<td></td>
</tr>
<tr>
<td>getPartCount function</td>
<td></td>
</tr>
<tr>
<td>getPropertyNames function</td>
<td></td>
</tr>
<tr>
<td>getPropertyType function</td>
<td></td>
</tr>
<tr>
<td>getReplyToDestinationType function</td>
<td></td>
</tr>
<tr>
<td>getSaxWriter function</td>
<td></td>
</tr>
<tr>
<td>getSequential function</td>
<td></td>
</tr>
<tr>
<td>getTempTable function</td>
<td></td>
</tr>
<tr>
<td>getText function</td>
<td></td>
</tr>
<tr>
<td>getTextPartByIndex function</td>
<td></td>
</tr>
<tr>
<td>getTextSegment function</td>
<td></td>
</tr>
<tr>
<td>getX-Document function</td>
<td></td>
</tr>
<tr>
<td>hasReplyTo function</td>
<td></td>
</tr>
<tr>
<td>isMessagePart function</td>
<td></td>
</tr>
<tr>
<td>messageHandler procedure</td>
<td></td>
</tr>
<tr>
<td>moveToNext procedure</td>
<td></td>
</tr>
<tr>
<td>readBytesToRaw procedure</td>
<td></td>
</tr>
<tr>
<td>readChar function</td>
<td></td>
</tr>
<tr>
<td>readDateTime function</td>
<td></td>
</tr>
<tr>
<td>readDateTime-TZ function</td>
<td></td>
</tr>
<tr>
<td>readDecimal function</td>
<td></td>
</tr>
<tr>
<td>readInt function</td>
<td></td>
</tr>
<tr>
<td>readInt64 function</td>
<td></td>
</tr>
<tr>
<td>readLogical function</td>
<td></td>
</tr>
<tr>
<td>readLongString function</td>
<td></td>
</tr>
<tr>
<td>readLongStringCP function</td>
<td></td>
</tr>
<tr>
<td>reset procedure</td>
<td></td>
</tr>
<tr>
<td>setBoolean procedure</td>
<td></td>
</tr>
<tr>
<td>setBooleanProperty procedure</td>
<td></td>
</tr>
<tr>
<td>setByte procedure</td>
<td></td>
</tr>
<tr>
<td>setByteProperty procedure</td>
<td></td>
</tr>
<tr>
<td>setBytesFromRaw procedure</td>
<td></td>
</tr>
<tr>
<td>setChar procedure</td>
<td></td>
</tr>
<tr>
<td>setDataSet procedure</td>
<td></td>
</tr>
<tr>
<td>setDate procedure</td>
<td></td>
</tr>
<tr>
<td>setDateProperty procedure</td>
<td></td>
</tr>
<tr>
<td>setDateTime procedure</td>
<td></td>
</tr>
<tr>
<td>setDateTimeProperty procedure</td>
<td></td>
</tr>
<tr>
<td>setDateTime-TZ procedure</td>
<td></td>
</tr>
<tr>
<td>setDateTimeTzProperty procedure</td>
<td></td>
</tr>
<tr>
<td>setDouble procedure</td>
<td></td>
</tr>
<tr>
<td>setDoubleProperty procedure</td>
<td></td>
</tr>
<tr>
<td>setFloat procedure</td>
<td></td>
</tr>
<tr>
<td>setFloatProperty procedure</td>
<td></td>
</tr>
<tr>
<td>setInt procedure</td>
<td></td>
</tr>
<tr>
<td>setInt64 procedure</td>
<td></td>
</tr>
<tr>
<td>setInt64Property procedure</td>
<td></td>
</tr>
<tr>
<td>setIntProperty procedure</td>
<td></td>
</tr>
<tr>
<td>setJMSCorrelationID procedure</td>
<td></td>
</tr>
<tr>
<td>setJMSCorrelationIDAsBytes procedure</td>
<td></td>
</tr>
<tr>
<td>setJMSReplyTo procedure</td>
<td></td>
</tr>
<tr>
<td>setJMSType procedure</td>
<td></td>
</tr>
<tr>
<td>setLong procedure</td>
<td></td>
</tr>
<tr>
<td>setLongProperty procedure</td>
<td></td>
</tr>
<tr>
<td>setLongString procedure</td>
<td></td>
</tr>
<tr>
<td>setLongText procedure</td>
<td></td>
</tr>
</tbody>
</table>
### Table A–3: Methods in Message Objects

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Method Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>setMemptr procedure</td>
<td>setNoErrorDisplay procedure</td>
</tr>
<tr>
<td>setReplyToDestinationType procedure</td>
<td>setSaxReader procedure</td>
</tr>
<tr>
<td>setSequential procedure</td>
<td>setShort procedure</td>
</tr>
<tr>
<td>setShortProperty procedure</td>
<td>setSingleMessageAcknowledgement procedure</td>
</tr>
<tr>
<td>setString procedure</td>
<td>setStringProperty procedure</td>
</tr>
<tr>
<td>setTempTable procedure</td>
<td>setText procedure</td>
</tr>
<tr>
<td>setX-Document procedure</td>
<td>writeBoolean procedure</td>
</tr>
<tr>
<td>writeByte procedure</td>
<td>writeBytesFromRaw procedure</td>
</tr>
<tr>
<td>writeChar procedure</td>
<td>writeDate procedure</td>
</tr>
<tr>
<td>writeDateTime procedure</td>
<td>writeDateTime-TZ procedure</td>
</tr>
<tr>
<td>writeDouble procedure</td>
<td>writeFloat procedure</td>
</tr>
<tr>
<td>writeInt procedure</td>
<td>writeInt64 procedure</td>
</tr>
<tr>
<td>writeLong procedure</td>
<td>writeLongString procedure</td>
</tr>
<tr>
<td>writeShort procedure</td>
<td>writeString procedure</td>
</tr>
</tbody>
</table>

For more information, see the “Message objects” section on page 2–5.
acknowledgeAndForward procedure

Forwards and acknowledges a message in a single operation.

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, setSingleMessageAcknowledgement procedure, getSingleMessageAcknowledgement function, setNoAcknowledge procedure, getNoAcknowledge function

For more information, see the “Message acknowledgement, forwarding, and recovery” section on page 4–44 and the “Single-message acknowledgement” section on page 4–45.
addBytesPart procedure

Adds any arbitrary part to a MultipartMessage.

Syntax

```plaintext
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, addMessagePart procedure, addTextPart procedure, isMessagePart function, getPartCount function, getContentType, getMessagePartByID function, getMessagePartByIndex function, writeBytesFromRaw procedure, readBytesToRaw procedure, setMemptr procedure, getMemptr function

For more information, see the “MultipartMessage” section on page 4–28 and the “MultiPartMessage example” section on page B–26.

addMessagePart procedure

Adds a SonicMQ message to a MultipartMessage.

Syntax

```plaintext
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, addBytesPart procedure, addMessagePart procedure, addTextPart procedure, isMessagePart function, getPartCount function, getContentType, getMessagePartByID function, getMessagePartByIndex function, writeBytesFromRaw procedure, readBytesToRaw procedure, setMemptr procedure, getMemptr function

For more information, see the “MultipartMessage” section on page 4–28 and the “MultiPartMessage example” section on page B–26.
addTextPart procedure

Adds a text part to a MultipartMessage.

Syntax

```abl
PROCEDURE addTextPart.
DEFINE INPUT PARAMETER charString AS CHARACTER.
DEFINE INPUT PARAMETER contentTypeString AS CHARACTER.
DEFINE INPUT PARAMETER contentIDString AS CHARACTER.
```

Applies to Message objects

Note The method resembles the addBytesPart procedure except that it takes a CHARACTER string instead of a MEMPTR.

See also createMultipartMessage procedure, addBytesPart procedure, addMessagePart procedure, isMessagePart function, getContentType, getPartCount function, getMessagePartByID function, getMessagePartByIndex function, writeBytesFromRaw procedure, readBytesToRaw procedure, setMemptr procedure, getMemptr function

For more information, see the “MultipartMessage” section on page 4–28 and the “MultiPartMessage example” section on page B–26.

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 the setText procedure.
- This procedure is not needed when using the setLongText procedure, which writes LONGCHAR data.

See also createTextMessage procedure, setText procedure, setLongText procedure, appendText procedure, endOfStream function, getCharCount function, getText function, getTextSegment function, getLongText function, getLongTextCP function

For more information, see the “TextMessage” section on page 4–24.
beginSession procedure

Connects to the OpenEdge Adapter for SonicMQ and starts a JMS connection and session.

Syntax

```
PROCEDURE beginSession.
```

Applies to
Session objects

Notes
- If the beginSession procedure 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, getSession function, deleteSession procedure

For more information, see the “Connecting to the OpenEdge Adapter for SonicMQ” section on page 4–8 and the “Establishing session control” section on page 4–10.
browseQueue procedure

Allows applications to view messages in a queue without consuming them.

Syntax

```
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 the `startReceiveMessages` procedure to browse messages on a queue.

- This procedure executes remotely (sends a message to the OpenEdge Adapter for SonicMQ).

See also

- `browseQueue` procedure, `receiveFromQueue` procedure, `sendToQueue` procedure

For more information, see the “Sending messages to a queue” section on page 4–34, the “Receiving messages from a queue” section on page 4–38, and the “Methods unique to Point-to-Point messaging” section on page 3–4.

For an example, see the “PTP message example” section on page 5–2.
cancelDurableSubscription procedure

Cancels a durable subscription.

**Syntax**

```plaintext
PROCEDURE cancelDurableSubscription.
DEFINE INPUT PARAMETER subscriptionName AS CHARACTER.
```

**Parameters**

`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 the `deleteConsumer` procedure first to delete the Message Consumer.
- This procedure executes remotely (sends a message to the OpenEdge Adapter for SonicMQ).

**See also**

`cancelDurableSubscription` procedure, `publish` procedure, `subscribe` procedure

For more information, see the “Publishing messages to a topic” section on page 4–34, the “Subscribing to a topic” section on page 4–39, the “Durable subscriptions” section on page 4–39, and the “Methods unique to Pub/Sub messaging” section on page 3–8.

For an example, see the “Pub/Sub messaging example” section on page 5–6.

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**

```plaintext
PROCEDURE clearBody.
```

**Applies to**

Message objects

**Note**

The `clearBody` procedure transfers a `StreamMessage`, `TextMessage`, `XMLMessage`, `BytesMessage`, `TempTableMessage`, or `DataSetMessage` to write-only mode.

**See also**

`clearProperties` procedure

For more information, see the “Accessing message properties” section on page 4–36.
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

For more information, see the “Accessing message properties” section on page 4–36.

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, commitReceive procedure, rollbackSend procedure, rollbackReceive procedure, recover procedure

For more information, see the “Transaction and recovery procedures” section on page 4–43.

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, commitReceive procedure, rollbackSend procedure, rollbackReceive procedure, recover procedure

For more information, see the “Transaction and recovery procedures” section on page 4–43.
createBytesMessage procedure

Creates a new BytesMessage.

Syntax

PROCEDURE createBytesMessage.  
DEFINE OUTPUT PARAMETER messageHandle AS HANDLE.

Applies to

Session objects

See also

createBytesMessage procedure, setMemptr procedure, writeBoolean procedure, writeByte procedure, writeBytesFromRaw procedure, writeChar procedure, writeDate procedure, writeDateTime procedure, writeDateTime-TZ procedure, writeDouble procedure, writeFloat procedure, writeInt procedure, writeLong procedure, writeLongString procedure, writeShort procedure, writeString procedure, getBytesCount function, endOfStream function, moveToNext procedure, readBytesToRaw procedure, readChar function, readDate function, readDateTime function, readDateTime-TZ function, readDecimal function, readInt function, readLogical function, readLongString function, readLongStringCP function, getMemptr function

For more information, see the “BytesMessage” section on page 4–28.

For an example, see the “Publishing, subscribing, and receiving an XML document in a BytesMessage” section on page B–13.
createChangeStateConsumer procedure

Creates a “Change State” handle to contain the state of the SonicMQ Broker connection changes.

Syntax

```abl
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 the createChangeStateConsumer procedure after the beginSession procedure.
- 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, getFaultTolerant function, isFaultTolerant function, createChangeStateConsumer procedure, setFaultTolerantReconnectTimeout procedure, getFaultTolerantReconnectTimeout function, setInitialConnectionTimeout procedure, getInitialConnectionTimeout function, setClientTransactionBufferSize procedure, getClientTransactionBufferSize function

For more information, see the “Fault tolerance” section on page 4–9 and the “Fault tolerance” section on page 5–28.

createDataSetMessage procedure

Creates a new DataSetMessage.

Syntax

```abl
PROCEDURE createDataSetMessage.
DEFINE OUTPUT PARAMETER messageHandle AS HANDLE.
```

Applies to

Session objects

See also

createDataSetMessage procedure, dataSet procedure, dataSet function

For an example, see the “DataSetMessage” section on page 5–24.
createHeaderMessage procedure

Creates a new HeaderMessage.

Syntax

```plaintext
PROCEDURE createHeaderMessage.
DEFINE OUTPUT PARAMETER messageHandle AS HANDLE.
```

Applies to Session objects

See also For more information, see the “HeaderMessage” section on page 4–25.

createMapMessage procedure

Creates a new MapMessage.

Syntax

```plaintext
PROCEDURE createMapMessage.
DEFINE OUTPUT PARAMETER messageHandle AS HANDLE.
```

Applies to Session objects

See also createMapMessage procedure, setBoolean procedure, setByte procedure, setBytesFromRaw procedure, setChar procedure, setDate procedure, setDateTime procedure, setDateTime-TZ procedure, setDouble procedure, setFloat procedure, setInt procedure, setLong procedure, setLongString procedure, setShort procedure, setString procedure, getMapNames function, getItemType function, getBytesToRaw function, getChar function, getDate function, getDateTime function, getDateTime-TZ function, getDecimal function, getInt function, getLogical function, getLongString function, getLongStringCP function

For more information, see the “MapMessage” section on page 4–26.
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

Note

The application must pass the name of an internal procedure for handling messages and the handle to the internal procedure to the createMessageConsumer procedure.

See also

createMessageConsumer procedure, deleteConsumer procedure, messageHandler procedure, waitForMessages procedure

For more information see the “Message Consumer objects” section on page 2–5, the “Consuming messages” section on page 4–37, the “Terminating the Message Consumer object” section on page 4–39, and the “Creating a message handler process” section on page 4–38.

createMultipartMessage procedure

Creates a MultipartMessage.

Syntax

```
PROCEDURE createMultipartMessage.
DEFINE OUTPUT PARAMETER messageH AS HANDLE.
```

Applies to  
Session objects

See also  
addBytesPart procedure, addMessagePart procedure, addTextPart procedure, isMessagePart function, getContentType, getPartCount function, getMessagePartByID function, getMessagePartByIndex function, writeBytesFromRaw procedure, readBytesToRaw procedure, setMemptr procedure, getMemptr function

For more information, see the “MultipartMessage” section on page 4–28 and the “MultiPartMessage example” section on page B–26.
**createRejectedMessageConsumer procedure**

Creates a Message Consumer to handle all rejected messages.

### Syntax

```plaintext
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 the `createRejectedMessageConsumer` procedure after the `beginSession` procedure.

### See also

- `createRejectedMessageConsumer` procedure, `setClientPersistence` procedure,
- `getClientPersistence` function, `setLocalStoreDirectory` procedure, `getLocalStoreDirectory` function,
- `setLocalStoreSize` procedure, `getLocalStoreSize` function, `setLocalStoreWaitTime` procedure,
- `getLocalStoreWaitTime` function, `setReconnectTimeout` procedure,
- `getReconnectTimeout` function, `setReconnectInterval` procedure, `getReconnectInterval` function

For more information, see the “Client persistence” section on page 4–8 and the “Client persistence” section on page 5–12.

**createStreamMessage procedure**

Creates a new `StreamMessage`.

### Syntax

```plaintext
PROCEDURE createStreamMessage.
DEFINE OUTPUT PARAMETER messageHandle AS HANDLE.
```

### Applies to

Session objects

### See also

- `createStreamMessage` procedure, `writeBoolean` procedure, `writeByte` procedure,
- `writeBytesFromRaw` procedure, `writeChar` procedure, `writeDate` procedure, `writeDateTime` procedure,
- `writeDateTime-TZ` procedure, `writeDouble` procedure, `writeFloat` procedure,
- `writeInt` procedure, `writeLong` procedure, `writeLongString` procedure, `writeShort` procedure,
- `writeString` procedure, `endOfStream` function, `moveToNext` procedure, `readBytesToRaw` procedure,
- `readChar` function, `readDate` function, `readDateTime` function, `readDateTime-TZ` function,
- `readDecimal` function, `readInt` function, `readLogical` function, `readLongString` function,
- `readLongStringCP` function

For more information, see the “StreamMessage” section on page 4–27.
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 the beginSession procedure has not yet been called.
- An error is returned if the temporary queue cannot be created.

See also createTemporaryQueue procedure, deleteTemporaryQueue procedure, createTemporaryTopic procedure, deleteTemporaryTopic procedure

For more information, see the “Temporary queues” section on page 3–4 and the “Temporary topic” section on page 3–7.

createTemporaryTopic procedure

Creates a temporary topic on the SonicMQ Broker and returns the topic name.

Syntax

```
PROCEDURE createTemporaryTopic.
DEFINE OUTPUT PRARMETER tname AS CHARACTER.
```

Applies to Session objects

Notes

- An error is returned if the beginSession procedure has not yet been called.
- An error is returned if the temporary topic cannot be created.

See also createTemporaryQueue procedure, deleteTemporaryQueue procedure, createTemporaryTopic procedure, deleteTemporaryTopic procedure

For more information, see the “Temporary queues” section on page 3–4 and the “Temporary topic” section on page 3–7.
createTempTableMessage procedure

Creates a new TempTableMessage.

Syntax

PROCEDURE createTempTableMessage.
DEFINE OUTPUT PARAMETER messageHandle AS HANDLE.

Applies to Session objects

See also createTempTableMessage procedure, setTempTable procedure, getTempTable function

For more information, see the “TempTableMessage” section on page 5–21.

createTextMessage procedure

Creates a new TextMessage object.

Syntax

PROCEDURE createTextMessage.
DEFINE OUTPUT PARAMETER messageHandle AS HANDLE.

Applies to Session objects

See also createTextMessage procedure, setText procedure, setLongText procedure, appendText procedure, endOfStream function, getCharCount function, getText function, getTextSegment function, getLongText function, getLongTextCP function

For more information, see the “TextMessage” section on page 4–24 and the “StreamMessage” section on page 4–27.

createXMLMessage procedure

Creates a new XMLMessage.

Syntax

PROCEDURE createXMLMessage.
DEFINE OUTPUT PARAMETER messageHandle AS HANDLE.

Applies to Session objects

See also createXMLMessage procedure, setX-Document procedure, getX-Document function, setSaxReader procedure, getSaxWriter function, deleteSaxWriter procedure

For more information, see the “XMLMessage” section on page 4–29.
deleteConsumer procedure

Deletes the Message Consumer object.

Syntax

```plaintext
PROCEDURE deleteConsumer.
```

Applies to Message Consumer objects

Notes

- In the Pub/Sub domain, the `deleteConsumer procedure` cancels the subscription. In the PTP domain, the `deleteConsumer procedure` removes the association with a queue.
- This procedure executes remotely (sends a message to the OpenEdge Adapter for SonicMQ).

See also `createMessageConsumer procedure`, `deleteConsumer procedure`, `messageHandler procedure`, `waitForMessages procedure`

For more information see the “Message Consumer objects” section on page 2–5, the “Consuming messages” section on page 4–37, the “Terminating the Message Consumer object” section on page 4–39, and the “Creating a message handler process” section on page 4–38.

deleteMessage procedure

Deletes a message (including a TempTableMessage and DataSetMessage) and deallocates all of its memory and resources.

Syntax

```plaintext
PROCEDURE deleteMessage.
```

Applies to Message objects

See also For more information, see the “Deleting messages” section on page 4–35.
deleteSaxWriter procedure

Terminates the SAX-WRITER and deletes the SAX-WRITER handle created by the `getSaxWriter` function.

**Syntax**

```
PROCEDURE deleteSaxWriter.
INPUT PARAMETER hd1 AS HANDLE.
```

**Applies to** Message objects

**Notes**

- This procedure calls the `setLongText` procedure 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 the `setLongText` procedure, and by `DELETE OBJECT`.

**See also** `createXMLMessage` procedure, `setX-Document` procedure, `getX-Document` function, `setSaxReader` procedure, `getSaxWriter` function, `deleteSaxWriter` procedure

For more information, see the “XMLMessage” section on page 4–29.

deleteSession procedure

Closes a session and its underlying connection and deletes the session procedure.

**Syntax**

```
PROCEDURE deleteSession.
```

**Applies to** Session objects

**Note** This procedure executes remotely (sends a message to the OpenEdge Adapter for SonicMQ).

**See also** `beginSession` procedure, `getSession` function, `deleteSession` procedure

For more information, see the “Establishing session control” section on page 4–10.
deleteTemporaryQueue procedure

Deletes the temporary queue on the SonicMQ Broker.

Syntax

```apl
PROCEDURE deleteTemporaryQueue.
DEFINE INPUT PARAMETER qname AS CHAR.
```

Applies to
Session objects

Notes
• Any Message Consumers for the temporary queue must be deleted with the `deleteConsumer procedure` before calling this procedure.
• An error is returned if the `beginSession procedure` has not yet been called.
• An error is returned if the temporary queue cannot be deleted.

See also
`createTemporaryQueue procedure`, `deleteTemporaryQueue procedure`, `createTemporaryTopic procedure`, `deleteTemporaryTopic procedure`

For more information, see the “Temporary queues” section on page 3–4 and the “Temporary topic” section on page 3–7.

deleteTemporaryTopic procedure

Deletes the temporary topic on the SonicMQ Broker.

Syntax

```apl
PROCEDURE deleteTemporaryTopic.
DEFINE INPUT PARAMETER tname AS CHARACTER.
```

Applies to
Session objects

Notes
• Any Message Consumers for the temporary topic must be deleted with the `deleteConsumer procedure` before calling this procedure.
• An error is returned if the `beginSession procedure` has not yet been called.
• An error is returned if the temporary topic cannot be deleted.

See also
`createTemporaryQueue procedure`, `deleteTemporaryQueue procedure`, `createTemporaryTopic procedure`, `deleteTemporaryTopic procedure`

For more information, see the “Temporary queues” section on page 3–4 and the “Temporary topic” section on page 3–7.
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 the endOfStream function if it used the getMemptr function for extracting the data.

• The endOfStream function is also used with the TextMessage, StreamMessage, and BytesMessage message types.

See also createTextMessage procedure, setText procedure, setLongText procedure, appendText procedure, endOfStream function, getCharCount function, getText function, getTextSegment function, getLongText function, getLongTextCP function

For more information, see the “TextMessage” section on page 4–24.

createBytesMessage procedure, setMemptr procedure, writeBoolean procedure, writeByte procedure, writeBytesFromRaw procedure, writeChar procedure, writeDate procedure, writeDateTime procedure, writeDateTime-TZ procedure, writeDouble procedure, writeFloat procedure, writeInt procedure, writeLong procedure, writeLongString procedure, writeShort procedure, writeString procedure, getBytesCount function, endOfStream function, moveToNext procedure, readBytesToRaw procedure, readChar function, readDate function, readDateTime function, readDateTime-TZ function, readDecimal function, readInt function, readLogical function, readLongString function, readLongStringCP function, getMemptr function

For more information, see the “BytesMessage” section on page 4–28.

For an example, see the “Publishing, subscribing, and receiving an XML document in a BytesMessage” section on B–13.

createStreamMessage procedure, writeBoolean procedure, writeByte procedure, writeBytesFromRaw procedure, writeChar procedure, writeDate procedure, writeDateTime procedure, writeDateTime-TZ procedure, writeDouble procedure, writeFloat procedure, writeInt procedure, writeLong procedure, writeLongString procedure, writeShort procedure, writeString procedure, endOfStream function, moveToNext procedure, readBytesToRaw procedure, readChar function, readDate function, readDateTime function, readDateTime-TZ function, readDecimal function, readInt function, readLogical function, readLongString function, readLongStringCP function

For more information, see the “StreamMessage” section on page 4–27.
getAdapterService function

Returns the service name under which the OpenEdge Adapter for SonicMQ BrokerConnect is registered with the NameServer.

Syntax

FUNCTION getAdapterService RETURNS CHARACTER.

Applies to  Session objects (for BrokerConnect only)

Note  If the setAdapterService procedure was not called, Null is returned.

See also  setAdapterService procedure, getAdapterService function

For more information, see the “Setting and getting JMS connection and session attributes” section on page 4–7.

getApplicationContext function

Returns application context information.

Syntax

FUNCTION getApplicationContext RETURNS HANDLE.

Applies to  Message Consumer objects

See also  setApplicationContext procedure, getApplicationContext function

For more information, see the “Accessing message handler information” section on page 4–38 and the “Creating a message handler process” section on page 4–38.

getBrokerURL function

Returns the value set by the preceding the setBrokerURL procedure.

Syntax

FUNCTION getBrokerURL RETURNS CHARACTER.

Applies to  Session objects

Note  If the setBrokerURL procedure was not called, Null is returned.

See also  setBrokerURL procedure, getBrokerURL function

For more information, see the “Setting and getting JMS connection and session attributes” section on page 4–7.
getBytesCount function

Returns the number of bytes in a BytesMessage.

Syntax

FUNCTION getBytesCount RETURNS INTEGER.

Applies to
Message objects

See also
createBytesMessage procedure, setMemptr procedure, writeBoolean procedure, writeByte procedure, writeBytesFromRaw procedure, writeChar procedure, writeDate procedure, writeDateTime procedure, writeDateTime-TZ procedure, writeDouble procedure, writeFloat procedure, writeInt procedure, writeLong procedure, writeLongString procedure, writeShort procedure, writeString procedure, getBytesCount function, endOfStream function, moveToNext procedure, readBytesToRaw procedure, readChar function, readDate function, readDateTime function, readDateTime-TZ function, readDecimal function, readInt function, readLogical function, readLongString function, readLongStringCP function, getMemptr function

For more information, see the “BytesMessage” section on page 4–28.

For an example, see the “Publishing, subscribing, and receiving an XML document in a BytesMessage” section on page B–13.

getBytesPartByID function

Retrieves a bytes part and returns the content type as a CHARACTER string.

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, addBytesPart procedure, addMessagePart procedure, addTextPart procedure, isMessagePart function, getPartCount function, getMessagePartByID function, getMessagePartByIndex function, writeBytesFromRaw procedure, readBytesToRaw procedure, setMemptr procedure, getMemptr function

For more information, see the “MultipartMessage” section on page 4–28 and the “MultiPartMessage example” section on page B–26.
getBytesPartByIndex function

Retrieves a bytes part and returns the content type as a CHARACTER string.

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, setMemptr procedure, writeBoolean procedure, writeByte procedure, writeBytesFromRaw procedure, writeChar procedure, writeDate procedure, writeDateTime procedure, writeDateTime-TZ procedure, writeDouble procedure, writeFloat procedure, writeInt procedure, writeLong procedure, writeLongString procedure, writeShort procedure, writeString procedure, getBytesCount function, endOfStream function, moveToNext procedure, readBytesToRaw procedure, readChar function, readDate function, readDateTime function, readDateTime-TZ function, readDecimal function, readInt function, readLogical function, readLongString function, readLongStringCP function, getMemptr function

For more information, see the “BytesMessage” section on page 4–28.

For an example, see the “Publishing, subscribing, and receiving an XML document in a BytesMessage” section on page B–13.

For more information, see the “MultipartMessage” section on page 4–28 and the “MultiPartMessage example” section on page B–26.

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, setBoolean procedure, setByte procedure, setBytesFromRaw procedure, setChar procedure, setDate procedure, setDateTime procedure, setDateTime-TZ procedure, setDouble procedure, setFloat procedure, setInt procedure, setLong procedure, setLongString procedure, setShort procedure, setString procedure, getMapNames function, getItemType function, getBytesToRaw function, getChar function, getDate function, getDate function, getDate function, getDate function, getDateTime function, getDateTime-TZ function, getDecimal function, getInt function, getLogical function, getLongString function, getLongStringCP function

For more information, see the “MapMessage” section on page 4–26.
getChar function

Gets an item of any data type except byte from a MapMessage.

Syntax

```
FUNCTION getChar RETURNS CHARACTER (itemName AS CHARACTER).
```

Applies to Message objects

See also createMapMessage procedure, setBoolean procedure, setByte procedure, setBytesFromRaw procedure, setChar procedure, setDate procedure, setDateTime procedure, setDateTime-TZ procedure, setDouble procedure, setFloat procedure, setInt procedure, setLong procedure, setLongString procedure, setShort procedure, setString procedure, getMapNames function, getItemType function, getBytesToRaw function, getChar function, getDate function, getDateTime function, getDateTime-TZ function, getDecimal function, getInt function, getLogical function, getLongString function, getLongStringCP function

For more information, see the “MapMessage” section on page 4–26.

getCharCount function

Returns the total number of characters in a message.

Syntax

```
FUNCTION getCharCount RETURNS INTEGER.
```

Applies to Message objects

See also createTextMessage procedure, setText procedure, setLongText procedure, appendText procedure, endOfStream function, getCharCount function, getText function, getTextSegment function, getLongText function, getLongTextCP function

For more information, see the “TextMessage” section on page 4–24.

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

For more information, see the “Accessing message properties” section on page 4–36.
getClientID function

Returns the client ID value for the SonicMQ Broker connection.

Syntax

FUNCTION getClientID RETURNS CHARACTER.

Applies to

Session objects

Note

If the setClientID procedure was not called, Null is returned.

See also

setClientID procedure, getClientID function

For more information, see the “Setting and getting JMS connection and session attributes” section on page 4–7.

getClientPersistence function

Returns the state of client persistence.

Syntax

FUNCTION getClientPersistence RETURNS LOGICAL.

Applies to

Session objects

Notes

• Only applicable for Client Persistence.

• The default is FALSE.

See also

createRejectedMessageConsumer procedure, setClientPersistence procedure,
getClientPersistence function, setLocalStoreDirectory procedure, getLocalStoreDirectory
function, setLocalStoreSize procedure, getLocalStoreSize function, setLocalStoreWaitTime
procedure, getLocalStoreWaitTime function, setReconnectTimeout procedure,
getReconnectTimeout function, setReconnectInterval procedure, getReconnectInterval
function

For more information, see the “Client persistence” section on page 4–8 and the “Client
persistence” section on page 5–12.
**getClientTransactionBufferSize function**

Returns the client buffer size in bytes for Fault Tolerant transacted messages in memory.

**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, getFaultTolerant function, isFaultTolerant function, createChangeStateConsumer procedure, setFaultTolerantReconnectTimeout procedure, getFaultTolerantReconnectTimeout function, setInitialConnectionTimeout procedure, getInitialConnectionTimeout function, setClientTransactionBufferSize procedure, getClientTransactionBufferSize function

For more information, see the “Fault tolerance” section on page 4–9 and the “Fault tolerance” section on page 5–28.

**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 the setConnectID procedure.
- Returns UNKNOWN when called before the setConnectID procedure.

**See also**

setConnectID procedure
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 the beginSession procedure.

See also

For more information, see the “Establishing session control” section on page 4–10.

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” section on page 4–7.

getConnectionURLs function

Returns a comma-separated list of SonicMQ Broker URLs that the client will try to connect to.

Syntax

FUNCTION getConnectionURLs RETURNS CHARACTER.

Applies to

Session objects

See also

setConnectionURLs procedure, getConnectionURLs function, setSequential procedure, getSequential function

For more information, see the “Managing fail-over support” section on page 4–7.
getContentType

Retrieves the content type of the message part corresponding to the index in a MultipartMessage.

**Syntax**

```
FUNCTION getContentType RETURNS CHARACTER (INPUT iIndex AS INTEGER).
```

**Applies to** Message objects

**See also** createMultipartMessage procedure, addBytesPart procedure, addMessagePart procedure, addTextPart procedure, isMessagePart function, getPartCount function, getMessagePartByIndex function, writeBytesFromRaw procedure, readBytesToRaw procedure, setMemptr procedure, getMemptr function

For more information, see the “MultipartMessage” section on page 4–28 and the “MultiPartMessage example” section on page B–26.

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, setDataSet procedure, getDataSet function

For an example, see the “DataSetMessage” section on page 5–24.
**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, setBoolean procedure, setByte procedure, setBytesFromRaw procedure, setChar procedure, setDateTime procedure, setDateTime-TZ procedure, setDouble procedure, setFloat procedure, setInt procedure, setLong procedure, setLongString procedure, setShort procedure, setString procedure, getMapNames function, getItemType function, getBytesToRaw function, getChar function, getDate function, getDateTime function, getDateTime-TZ function, getDecimal function, getInt function, getLogical function, getLongString function, getLongStringCP function

For more information, see the “MapMessage” section on page 4–26.

**getDateTimeProperty function**

Returns a date value with no time or time zone information.

**Syntax**

```
FUNCTION getDateTimeProperty 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** setDateTimeProperty procedure, getDateTimeProperty function

For more information, see the “Accessing message properties” section on page 4–36.
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, setBoolean procedure, setByte procedure, setBytesFromRaw procedure, setChar procedure, setDateTime procedure, setDateTime-TZ procedure, setDouble procedure, setFloat procedure, setInt procedure, setLong procedure, setLongString procedure, setShort procedure, setString procedure, getMapNames function, getItemType function, getBytesToRaw function, getChar function, getDateTime function, getDateTime-TZ function, getDecimal function, getInt function, getLogical function, getLongString function, getLongStringCP function

For more information, see the “MapMessage” section on page 4–26.

generate function

Returns a date-time property with no time zone information.

Syntax

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, getDateTimeProperty function

For more information, see the “Accessing message properties” section on page 4–36.
getDateTime-TZ function

Returns a date-time value including time zone information.

Syntax

```abl
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, setBoolean procedure, setByte procedure, setBytesFromRaw procedure, setChar procedure, setDate procedure, setDateTime procedure, setDateTime-TZ procedure, setDouble procedure, setFloat procedure, setInt procedure, setLong procedure, setLongString procedure, setShort procedure, setString procedure, getMapNames function, getItemType function, getBytesToRaw function, getChar function, getDate function, getDateTime function, getDateTime-TZ function, getDecimal function, getInt function, getLogical function, getLongString function, getLongStringCP function

For more information, see the “MapMessage” section on page 4–26.

getDateTimeTzProperty function

Returns a date-time property including time zone information.

Syntax

```abl
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, getDateTimeTzProperty function

For more information, see the “Accessing message properties” section on page 4–36.
getDecimal function

Gets any numeric item from a MapMessage.

Syntax

FUNCTION getDecimal RETURNS DECIMAL (itemName AS CHARACTER).

Applies to  Message objects

See also  createMapMessage procedure, setBoolean procedure, setByte procedure, setBytesFromRaw procedure, setChar procedure, setDate procedure, setDateTime procedure, setDateTime-TZ procedure, setDouble procedure, setFloat procedure, setInt procedure, setLong procedure, setLongString procedure, setShort procedure, setString procedure, getMapNames function, getItemType function, getBytesToRaw function, getChar function, getDate function, getDateTime function, getDateTime-TZ function, getDecimal function, getInt function, getLogical function, getLongString function, getLongStringCP function

For more information, see the “MapMessage” section on page 4–26.

getDecimalProperty function

Returns any numeric message property.

Syntax

FUNCTION getDecimalProperty RETURNS DECIMAL (propertyName AS CHARACTER).

Applies to  Message objects

See also  For more information, see the “Accessing message properties” section on page 4–36.

getDefaultPersistency function

Returns the value specified by the setDefaultPersistency procedure.

Syntax

FUNCTION getDefaultPersistency RETURNS CHARACTER.

Applies to  Session objects

Note  If the setDefaultPersistency procedure was not called, PERSISTENT is returned.

See also  setDefaultPersistency procedure, getDefaultPersistency function, setDefaultPriority procedure, getDefaultPriority function, setDefaultTimeToLive procedure, getDefaultTimeToLive function

For more information, see the “Accessing message delivery parameters” section on page 4–11.
getDefaultPriority function

Returns the value specified by the setDefaultPriority procedure.

Syntax

FUNCTION getDefaultPriority RETURNS INTEGER.

Applies to

Session objects

Note

If the setDefaultPriority procedure was not called, 4 is returned.

See also

setDefaultPersistency procedure, getDefaultPersistency function, setDefaultPriority procedure
getPriority function, setDefaultTimeToLive procedure, getDefaultTimeToLive function

For more information, see the “Accessing message delivery parameters” section on page 4–11.

getDefaultTimeToLive function

Returns the value specified by the setDefaultTimeToLive procedure.

Syntax

FUNCTION getDefaultTimeToLive RETURNS DECIMAL.

Applies to

Session objects

Note

If the setDefaultTimeToLive procedure was not called, UNKNOWN is returned.

See also

setDefaultPersistency procedure, getDefaultPersistency function, setDefaultPriority procedure
getPriority function, setDefaultTimeToLive procedure, getDefaultTimeToLive function

For more information, see the “Accessing message delivery parameters” section on page 4–11.

getDestinationName function

Returns the name of the destination that messages arrive from when the Message Consumer was passed to the subscribe procedure or the receiveFromQueue procedure.

Syntax

FUNCTION getDestinationName RETURNS CHARACTER.

Applies to

Message Consumer objects

See also

For more information, see the “Accessing message handler information” section on page 4–38.
getFaultTolerant function

Returns the current Fault Tolerant setting.

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, getFaultTolerant function, isFaultTolerant function,
createChangeStateConsumer procedure, setFaultTolerantReconnectTimeout procedure,
getFaultTolerantReconnectTimeout function, setInitialConnectionTimeout procedure,
getInitialConnectionTimeout function, setClientTransactionBufferSize procedure,
getClientTransactionBufferSize function

For more information, see the “Fault tolerance” section on page 4–9 and the “Fault tolerance”
section on page 5–28.

getFaultTolerantReconnectTimeout function

Returns the Fault Tolerant reconnection timeout.

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, getFaultTolerant function, isFaultTolerant function,
createChangeStateConsumer procedure, setFaultTolerantReconnectTimeout procedure,
getFaultTolerantReconnectTimeout function, setInitialConnectionTimeout procedure,
getInitialConnectionTimeout function, setClientTransactionBufferSize procedure,
getClientTransactionBufferSize function

For more information, see the “Fault tolerance” section on page 4–9 and the “Fault tolerance”
section on page 5–28.
getFlowToDisk function

This function returns the current flow-to-disk setting.

Syntax

FUNCTION getFlowToDisk RETURNS INTEGER.

Applies to    Session objects
See also      setFlowToDisk procedure

For more information on this SonicMQ feature see the Flow to Disk section in the chapter for SonicMQ Client Sessions in 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, setBoolean procedure, setByte procedure, setBytesFromRaw procedure, setChar procedure, setDate procedure, setDateTime procedure, setDateTime-TZ procedure, setDouble procedure, setFloat procedure, setInt procedure, setLong procedure, setLongString procedure, setShort procedure, setString procedure, getMapNames function, getItemType function, getBytesToRaw function, getChar function, getDate function, getDateTime function, getDateTime-TZ function, getDecimal function, getInt64 function, getLogical function, getLongString function, getLongStringCP function

For more information, see the “MapMessage” section on page 4–26.

getIntProperty function

Returns int, short, and byte message properties.

Syntax

FUNCTION getIntProperty RETURNS INTEGER (propertyName AS CHARACTER).

Applies to    Message objects
See also      setIntProperty procedure, getIntProperty function, setInt64Property procedure, getIntProperty function, clearProperties procedure

For more information, see the “Accessing message properties” section on page 4–36.
getInt64 function

Gets INT64 items from a MapMessage.

**Syntax**

FUNCTION getInt64 RETURNS INT64 (itemName AS CHARACTER).

**Applies to** Message objects

**See also** createMapMessage procedure, setBoolean procedure, setByte procedure, setBytesFromRaw procedure, setChar procedure, setDate procedure, setDateTime procedure, setDateTime-TZ procedure, setDouble procedure, setFloat procedure, setInt procedure, setInt64 procedure, setLong procedure, setLongString procedure, setShort procedure, setString procedure, getMapNames function, getItemType function, getBytesToRaw function, getChar function, getDate function, getDateTime function, getDateTime-TZ function, getDecimal function, getInt function, getLogical function, getLongString function, getLongStringCP function

For more information, see the “MapMessage” section on page 4–26.

getInt64Property function

Returns int64 message properties.

**Syntax**

FUNCTION getInt64Property RETURNS INT64 (propertyName AS CHARACTER).

**Applies to** Message objects

**See also** setInt64Property procedure, setIntProperty procedure, getIntProperty function, clearProperties procedure

For more information, see the “Accessing message properties” section on page 4–36.
**getInitialConnectionTimeout function**

Returns the initial Fault Tolerant reconnection timeout in seconds.

**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, getFaultTolerant function, isFaultTolerant function,  
createChangeStateConsumer procedure, setFaultTolerantReconnectTimeout procedure,  
getFaultTolerantReconnectTimeout function, setInitialConnectionTimeout procedure,  
getInitialConnectionTimeout function, setClientTransactionBufferSize procedure,  
getClientTransactionBufferSize function

For more information, see the “Fault tolerance” section on page 4–9 and the “Fault tolerance”  
section on page 5–28.
getItemType function

Returns the data type of an item in a MapMessage.

Syntax

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 — The getItemType function 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 — The getItemType function returns the datetime-tz value for all date items.

See also createMapMessage procedure, setBoolean procedure, setByte procedure, setBytesFromRaw procedure, setChar procedure, setDate procedure, setDateTime procedure, setDateTime-TZ procedure, setDouble procedure, setFloat procedure, setInt procedure, setLong procedure, setLongString procedure, setShort procedure, setString procedure, getMapNames function, getItemType function, getBytesToRaw function, getChar function, getDateTime function, getDateTime-TZ function, getDecimal function, getFloat function, getLogical function, getLongString function, getLongStringCP function

For more information, see the “MapMessage” section on page 4–26.

getJMSCorrelationID function

Returns the correlation ID.

Syntax

FUNCTION getJMSCorrelationID RETURNS CHARACTER.

Applies to Message objects

Note This value is an application-defined correlation ID, typically the ID of the message replied to.

See also setJMSCorrelationID procedure, getJMSCorrelationID function

For more information, see the “Accessing message header properties” section on page 4–35.
getJMSCorrelationIDAsBytes function

Returns a proprietary (JMS-provider-dependent) correlation ID.

Syntax

FUNCTION getJMSCorrelationIDAsBytes RETURNS RAW.

Applies to Message objects

Note When accessing SonicMQ, the bytesCorrelationID field can be used for storing application-defined values.

See also setJMSCorrelationIDAsBytes procedure, getJMSCorrelationIDAsBytes function

For more information, see the “Accessing message header properties” section on page 4–35.

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, getJMSRedelivered function, getType function, getJMSMessageID function, getJMSDeliveryMode function, getJMSExpiration function, getJMSPriority function

For more information, see the “Accessing message header properties” section on page 4–35.
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, getJMSRedelivered function, getMessageType function, getJMSMessageID function, getJMSDeliveryMode function, getJMSTimestamp function, getJMSExpiration function, getJMSPriority function

For more information, see the “Accessing message header properties” section on page 4–35.

getJMSExpiration function

Returns the expiration time (GMT).

**Syntax**

```
FUNCTION getJMSExpiration RETURNS DECIMAL.
```

**Applies to** Message objects

**See also** getJMSDestination function, getJMSRedelivered function, getMessageType function, getJMSMessageID function, getJMSDeliveryMode function, getJMSTimestamp function, getJMSExpiration function, getJMSPriority function

For more information, see the “Accessing message header properties” section on page 4–35.

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, getJMSRedelivered function, getMessageType function, getJMSMessageID function, getJMSDeliveryMode function, getJMSTimestamp function, getJMSExpiration function, getJMSPriority function

For more information, see the “Accessing message header properties” section on page 4–35.
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, getJMSRedelivered function, getMessageType function, getJMSMessageID function, getJMSDeliveryMode function, getJMSTimestamp function, getJMSExpiration function, getJMSPriority function

For more information, see the “Accessing message header properties” section on page 4–35.

getJMSRedelivered function

Returns TRUE (at the receiver side) if this is not the first delivery of this message.

Syntax

FUNCTION getJMSRedelivered RETURNS LOGICAL.

Applies to Message objects

Note 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, getJMSRedelivered function, getMessageType function, getJMSMessageID function, getJMSDeliveryMode function, getJMSTimestamp function, getJMSExpiration function, getJMSPriority function

For more information, see the “Accessing message header properties” section on page 4–35.
getJMSReplyTo function

Returns the reply destination.

Syntax

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 the getReplyToDestinationType function if both a queue destination and a topic destination might be stored in the received message.

See also setJMSReplyTo procedure, getJMSReplyTo function, hasReplyTo function, setReplyToDestinationType procedure, getReplyToDestinationType function

For more information, see the “Accessing message header properties” section on page 4–35.

getJMSServerName function

Returns the value set by the preceding the setJMSServerName procedure.

Syntax

FUNCTION getJMSServerName RETURNS CHARACTER.

Applies to Session objects

Note

If the setJMSServerName procedure is not called, the Unknown value (?) is returned.

See also setJMSServerName procedure, getJMSServerName function

For more information, see the “Setting and getting JMS connection and session attributes” section on page 4–7.
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, getJMSRedelivered function, getMessageType function, getJMSMessageID function, getJMSDeliveryMode function, getJMSTimestamp function, getJMSExpiration function, getJMSPriority function

For more information, see the “Accessing message header properties” section on page 4–35.

getJMSType function

Returns a proprietary (JMS-provider-dependent) type name.

Syntax

FUNCTION getJMSType RETURNS CHARACTER.

Applies to  Message objects

Note  When accessing SonicMQ, the JMSType field can be used for storing application-defined values.

See also  setJMSType procedure, getJMSType function

For more information, see the “Accessing message header properties” section on page 4–35.

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, getLoadBalancing function

For more information, see the “Load balancing” section on page 4–8.
getLocalStoreDirectory function

Returns the directory that will be used by the adapter to persist messages.

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 the setLocalStoreDirectory procedure.

• If the setLocalStoreDirectory procedure was never called, the value is the Unknown value (?).

See also

createRejectedMessageConsumer procedure, setClientPersistence procedure,
getClientPersistence function, setLocalStoreDirectory procedure, getLocalStoreDirectory function, setLocalStoreSize procedure, getLocalStoreSize function, setLocalStoreWaitTime procedure, getLocalStoreWaitTime function, setReconnectTimeout procedure, getReconnectTimeout function, setReconnectInterval procedure, getReconnectInterval function

For more information, see the “Client persistence” section on page 4–8 and the “Client persistence” section on page 5–12.

getLocalStoreSize function

Returns the maximum size of the local store in kilobytes.

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

createRejectedMessageConsumer procedure, setClientPersistence procedure,
getClientPersistence function, setLocalStoreDirectory procedure, getLocalStoreDirectory function, setLocalStoreSize procedure, getLocalStoreSize function, setLocalStoreWaitTime procedure, getLocalStoreWaitTime function, setReconnectTimeout procedure, getReconnectTimeout function, setReconnectInterval procedure, getReconnectInterval function

For more information, see the “Client persistence” section on page 4–8 and the “Client persistence” section on page 5–12.
getLocalStoreWaitTime function

Returns the wait interval before Client Persistence begins.

Syntax

```plaintext
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

createRejectedMessageConsumer procedure, setClientPersistence procedure, getClientPersistence function, setLocalStoreDirectory procedure, getLocalStoreDirectory function, setLocalStoreSize procedure, getLocalStoreSize function, setLocalStoreWaitTime procedure, getLocalStoreWaitTime function, setReconnectTimeout procedure, getReconnectTimeout function, setReconnectInterval procedure, getReconnectInterval function

For more information, see the “Client persistence” section on page 4–8 and the “Client persistence” section on page 5–12.

getLogical function

Returns a boolean item by name from a MapMessage.

Syntax

```plaintext
FUNCTION getLogical RETURNS LOGICAL (itemName AS CHARACTER).
```

Applies to

Message objects

See also

createMapMessage procedure, setBoolean procedure, setByte procedure, setBytesFromRaw procedure, setChar procedure, setDate procedure, setDateTime procedure, setDateTime-TZ procedure, setDouble procedure, setFloat procedure, setInt procedure, setLong procedure, setLongString procedure, setShort procedure, setString procedure, getMapNames function, getItemType function, getBytesToRaw function, getChar function, getDate function, getDateTime function, getDateTime-TZ function, getDecimal function, getInt function, getLogical function, getLongString function, getLongStringCP function

For more information, see the “MapMessage” section on page 4–26.
**getLogicalProperty function**

Returns a boolean message property.

**Syntax**

```
FUNCTION getLogicalProperty RETURNS LOGICAL (propertyName AS CHARACTER).
```

**Applies to** Message objects

**See also** clearProperties procedure

For more information, see the “Accessing message properties” section on page 4–36.

**getLongString function**

Returns a LONGCHAR item by name from a MapMessage.

**Syntax**

```
FUNCTION getLongString RETURNS LONGCHAR (itemName AS CHARACTER).
```

**Applies to** Message objects

**Note**

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 the getLongTextCP function.

**See also** createMapMessage procedure, setBoolean procedure, setByte procedure, setBytesFromRaw procedure, setChar procedure, setDate procedure, setDateTime procedure, setDateTime-TZ procedure, setDouble procedure, setFloat procedure, setInt procedure, setLong procedure, setLongString procedure, setShort procedure, setString procedure, getMapNames function, getItemType function, getBytesToRaw function, getChar function, getDate function, getDateTime function, getDateTime-TZ function, getDecimal function, getInt function, getLogical function, getLongFunction, getLongStringCP function

For more information, see the “MapMessage” section on page 4–26.
**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, setBoolean procedure, setByte procedure, setBytesFromRaw procedure, setChar procedure, setDate procedure, setDateTime procedure, setDateTime-TZ procedure, setDouble procedure, setFloat procedure, setInt procedure, setLong procedure, setLongString procedure, setShort procedure, setString procedure, getMapNames function, getItemType function, getBytesToRaw function, getChar function, getDate function, getDateTime function, getDateTime-TZ function, getDecimal function, getInt function, getLogical function, getLongString function, getLongStringCP function

For more information, see the “MapMessage” section on page 4–26.

**getLongText function**

Returns all text in a TextMessage.

**Syntax**

FUNCTION getLongText RETURNS LONGCHAR.

**Applies to** Message objects

**Notes**
- Implicitly calls the reset procedure.
- 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 the getLongTextCP function.

**See also**
createTextMessage procedure, setText procedure, setLongText procedure, appendText procedure, endOfStream function, getCharCount function, getText function, getTextSegment function, getLongText function, getLongTextCP function

For more information, see the “TextMessage” section on page 4–24.
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).
```

Applies to

Message objects

Notes

• Implicitly calls the reset procedure.

• The LONGCHAR data returned is converted to the code page specified by the `code_page` parameter.

See also

createTextMessage procedure, setText procedure, setLongText procedure, appendText procedure, endOfStream function, getCharCount function, getText function, getTextSegment function, getLongText function, getLongTextCP function

For more information, see the “TextMessage” section on page 4–24.

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, setBoolean procedure, setByte procedure, setBytesFromRaw procedure, setChar procedure, setDate procedure, setDateTime procedure, setDateTime-TZ procedure, setDouble procedure, setFloat procedure, setInt procedure, setLong procedure, setLongString procedure, setShort procedure, setString procedure, getMapNames function, getItemType function, getBytesToRaw function, getChar function, getDate function, getDateTime function, getDateTime-TZ function, getDecimal function, getInt function, getLogical function, getLongString function, getLongStringCP function

For more information, see the “MapMessage” section on page 4–26.
getMemptr function

Returns a reference to a MEMPTR variable that contains exactly all the bytes of a BytesMessage.

Syntax

```
FUNCTION getMemptr RETURNS MEMPTR.
```

Applies to

Message objects

Notes

- This function implicitly calls the reset procedure.
- If the message was in a write-only mode, it will be in a read-only/reset mode after the call.
- The getMemptr function does not create a copy of the MEMPTR variable; it returns a reference to the data maintained by the Message object.
- The deleteMessage procedure 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, setMemptr procedure, writeBoolean procedure, writeByte procedure, writeBytesFromRaw procedure, writeChar procedure, writeDate procedure, writeDateTime procedure, writeDateTime-TZ procedure, writeDouble procedure, writeFloat procedure, writeInt procedure, writeLong procedure, writeLongString procedure, writeShort procedure, writeString procedure, getContentType, getBytesCount function, endOfStream function, moveToNext procedure, readBytesToRaw procedure, readChar function, readDate function, readDateTime function, readDateTime-TZ function, readDecimal function, readInt function, readLogical function, readLongString function, readLongStringCP function, getMemptr function

For more information, see the see the “StreamMessage” section on page 4–27 and the “BytesMessage” section on page 4–28.

For an example, see the “Publishing, subscribing, and receiving an XML document in a BytesMessage” section on page B–13.
getMessagePartByID function

Retrieves a message part corresponding to the given contentID.

Syntax

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 the deleteMessage procedure 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, addBytesPart procedure, addMessagePart procedure,
addTextPart procedure, isMessagePart function, getContentType, getPartCount function,
getMessagePartByIndex function, getTextPartByID function, writeBytesFromRaw procedure,
readBytesToRaw procedure, setMemptr procedure, getMemptr function

For more information, see the “MultipartMessage” section on page 4–28 and the

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 the deleteMessage procedure 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, addBytesPart procedure, addMessagePart procedure,
addTextPart procedure, isMessagePart function, getPartCount function, getMessagePartByID
function, getMessagePartByIndex function, writeBytesFromRaw procedure, readBytesToRaw
procedure, setMemptr procedure, getMemptr function

For more information, see the “MultipartMessage” section on page 4–28 and the
**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, getJMSRedelivered function, getMessageType function, getJMSMessageID function, getJMSDeliveryMode function, getJMSTimestamp function, getJMSExpiration function, getJMSPriority function

For more information, see the “Accessing message header properties” section on page 4–35.

**getNoAcknowledge function**

Returns TRUE if the setNoAcknowledge procedure was called.

**Syntax**

FUNCTION getNoAcknowledge RETURNS LOGICAL.

**Applies to** Message Consumer objects

**See also** acknowledgeAndForward procedure, setSingleMessageAcknowledgement procedure, getSingleMessageAcknowledgement function, setNoAcknowledge procedure, getNoAcknowledge function

For more information, see the “Message acknowledgement, forwarding, and recovery” section on page 4–44 and the “Single-message acknowledgement” section on page 4–45.

**getPartCount function**

Returns the number of parts in a MultipartMessage.

**Syntax**

FUNCTION getPartCount RETURNS INTEGER.

**Applies to** Message objects

**See also** createMultipartMessage procedure, addBytesPart procedure, addMessagePart procedure, addTextPart procedure, isMessagePart function, getContentType, getMessagePartByID function, getMessagePartByIndex function, writeBytesFromRaw procedure, readBytesToRaw procedure, setMemptr procedure, getMemptr function

For more information, see the “MultipartMessage” section on page 4–28 and the “MultiPartMessage example” section on page B–26.
**getPassword function**

Returns the value set by the preceding the `setPassword` procedure.

**Syntax**

```plaintext
FUNCTION getPassword RETURNS CHARACTER.
```

**Applies to** Session objects

**Note** If the `setPassword` procedure was not called, The Unknown value (?) is returned.

**See also** `setPassword` procedure, `getPassword` function

For more information, see the “Setting and getting JMS connection and session attributes” section on page 4–7.

**getProcHandle function**

Returns the handle to a procedure that contains the name of an internal procedure for handling messages.

**Syntax**

```plaintext
FUNCTION getProcHandle RETURNS HANDLE.
```

**Applies to** Message Consumer objects

**See also** `getProcName` function, `getProcHandle` function

For more information, see the “Accessing message handler information” section on page 4–38.

**getProcName function**

Returns the name of the internal procedure for handling messages.

**Syntax**

```plaintext
FUNCTION getProcName RETURNS CHARACTER.
```

**Applies to** Message Consumer objects

**See also** `getProcName` function, `getProcHandle` function

For more information, see the “Accessing message handler information” section on page 4–38.
getPropertyNames function

Returns a comma-separated list of the properties of a message.

Syntax

```FUNCTION getPropertyNames RETURNS CHARACTER.```

Applies to Message objects

See also clearProperties procedure

For more information, see the “Accessing message properties” section on page 4–36.

getPropertyType function

Returns the message property’s data type.

Syntax

```FUNCTION getPropertyType RETURNS CHARACTER (propertyName AS CHARACTER).```

Parameters

- `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

For more information, see the “Accessing message properties” section on page 4–36.
getReconnectInterval function

Returns the interval for reconnection attempts in seconds.

Syntax

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
createRejectedMessageConsumer procedure, setClientPersistence procedure,
getClientPersistence function, setLocalStoreDirectory procedure, getLocalStoreDirectory
function, setLocalStoreSize procedure, getLocalStoreSize function, setLocalStoreWaitTime
procedure, getLocalStoreWaitTime function, setReconnectTimeout procedure,
getReconnectTimeout function, setReconnectInterval procedure, getReconnectInterval
function

For more information, see the “Client persistence” section on page 4–8 and the “Client
persistence” section on page 5–12.

getReconnectTimeout function

Returns the timeout for reconnection attempts in minutes.

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.

See also
createRejectedMessageConsumer procedure, setClientPersistence procedure,
getClientPersistence function, setLocalStoreDirectory procedure, getLocalStoreDirectory
function, setLocalStoreSize procedure, getLocalStoreSize function, setLocalStoreWaitTime
procedure, getLocalStoreWaitTime function, setReconnectTimeout procedure,
getReconnectTimeout function, setReconnectInterval procedure, getReconnectInterval
function

For more information, see the “Client persistence” section on page 4–8 and the “Client
persistence” section on page 5–12.
**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** setReplyPersistency procedure, getReplyPersistency function, setReplyPriority procedure, getReplyPriority function, setReplyTimeToLive procedure, getReplyTimeToLive function, setReplyAutoDelete procedure, getReplyAutoDelete function

For more information, see the “Accessing message handler information” section on page 4–38 and the “Setting reply properties” section on page 4–38.

**getReplyPersistency function**

Returns the value for message persistency.

**Syntax**

```FUNCTION getReplyPersistency RETURNS CHARACTER.```  

**Applies to** Message Consumer objects  

**Note** If the setReplyPersistency procedure was not called, PERSISTENT is returned.

**See also** setReplyPersistency procedure, getReplyPersistency function, setReplyPriority procedure, getReplyPriority function, setReplyTimeToLive procedure, getReplyTimeToLive function, setReplyAutoDelete procedure, getReplyAutoDelete function

For more information, see the “Accessing message handler information” section on page 4–38 and the “Setting reply properties” section on page 4–38.
getReplyPriority function

Returns the priority of the reply messages.

Syntax

```
FUNCTION getReplyPriority RETURNS INTEGER.
```

Applies to Message Consumer objects

Note If the setReplyPriority procedure was not called, the returned value is 4.

See also setReplyPersistency procedure, getReplyPersistency function, setReplyPriority procedure, getReplyPriority function, setReplyTimeToLive procedure, getReplyTimeToLive function, setReplyAutoDelete procedure, getReplyAutoDelete function

For more information, see the “Accessing message handler information” section on page 4–38 and the “Setting reply properties” section on page 4–38.

getReplyTimeToLive function

Returns the time to live value (in milliseconds) of the reply messages.

Syntax

```
FUNCTION getReplyTimeToLive RETURNS DECIMAL.
```

Applies to Message Consumer objects

Note If the setReplyTimeToLive procedure was not called, UNKNOWN is returned.

See also setReplyPersistency procedure, getReplyPersistency function, setReplyPriority procedure, getReplyPriority function, setReplyTimeToLive procedure, getReplyTimeToLive function, setReplyAutoDelete procedure, getReplyAutoDelete function

For more information, see the “Accessing message handler information” section on page 4–38 and the “Setting reply properties” section on page 4–38.
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 the getReplyToDestinationType function 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 setUMSReplyTo procedure, getJMSReplyTo function, hasReplyTo function, setReplyToDestinationType procedure, getReplyToDestinationType function

For more information, see the “Accessing message header properties” section on page 4–35.

getReuseMessage function

Returns the value set by the setReuseMessage procedure.

Syntax

FUNCTION getReuseMessage RETURNS LOGICAL.

Applies to Message Consumer objects

Note

TRUE if the setReuseMessage procedure was called; if not, it returns FALSE.

See also setReuseMessage procedure, getReuseMessage function

For more information, see the “Reusing messages” section on page 4–40.
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 the deleteSaxWriter procedure.

• Possible errors are returned by CREATE-SAX-WRITER or SET-OUTPUT-DESTINATION.

See also  createXMLMessage procedure, setX-Document procedure, getX-Document function, setSaxReader procedure, getSaxWriter function, deleteSaxWriter procedure

For more information, see the “XMLMessage” section on page 4–29.

getSession function

Returns a handle to the session.

Syntax

FUNCTION getSession RETURNS HANDLE.

Applies to  Message Consumer objects

See also  beginSession procedure, getSession function, deleteSession procedure

For more information, see the “Accessing message handler information” section on page 4–38.
getSelectorAtBroker function

Gets the broker selector setting.

Syntax

```FUNCTION getSelectorAtBroker RETURNS LOGICAL.```

Applies to Session objects

Note The default is FALSE.

See also setSelectorAtBroker procedure, getSelectorAtBroker function

For more information, see the “Message selectors” section on page 4–12.

getSequential function

Returns a LOGICAL value indicating how a fail-over list is used.

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.
- The `getSequential` function returns TRUE if connection attempts are sequential and FALSE if connection attempts are random.

See also setConnectionURLs procedure, getConnectionURLs function, setSequential procedure, getSequential function

For more information, see the “Managing fail-over support” section on page 4–7.

getShutdownWaitFor function

Returns a LOGICAL value indicating the current value of the shutdown WAIT-FOR flag.

Syntax

```FUNCTION getShutdownWaitFor RETURNS LOGICAL.```

Applies to Session objects

Notes

- The `getShutdownWaitFor` function returns YES if the `setShutdownWaitFor` procedure was never called.

See also `setShutdownWaitFor` procedure
getSingleMessageAcknowledgement function

Returns a logical value indicating whether a client session is configured to use single-message acknowledgement.

Syntax

FUNCTION getSingleMessageAcknowledgement RETURNS LOGICAL.

Applies to Session objects

Notes

• The getSingleMessageAcknowledgement function returns TRUE if the client session is configured to use single-message acknowledgement.

• The getSingleMessageAcknowledgement function returns FALSE if the client session is not so configured.

See also acknowledgeAndForward procedure, setSingleMessageAcknowledgement procedure, getSingleMessageAcknowledgement function, setNoAcknowledge procedure, getNoAcknowledge function

For more information, see the “Message acknowledgement, forwarding, and recovery” section on page 4–44 and the “Single-message acknowledgement” section on page 4–45.

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, setTempTable procedure, getTempTable function

For more information, see the “TempTableMessage” section on page 5–21.
**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 the `reset procedure`.

**See also**  
`createTextMessage procedure`, `setText procedure`, `setLongText procedure`, `appendText procedure`, `endOfStream function`, `getCharCount function`, `getText function`, `getTextSegment function`, `getLongText function`, `getLongTextCP function`

For more information, see the “TextMessage” section on page 4–24.

**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 the `getBytesPartByID function`.

**See also**  
`getBytesPartByID function`, `getMessagePartByID function`, `getTextPartByID function`

For more information, see the “MultipartMessage” section on page 4–28.
getTextPartByIndex function

Retrieves a text part and returns the content type as a CHARACTER string.

Syntax

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 the
  getBytesPartByIndex function.

See also getBytesPartByIndex function, getMessagePartByIndex function, getTextPartByIndex
  function

For more information, see the “MultipartMessage” section on page 4–28.

getTextSegment function

Returns the next text segment when handling large messages in read-only mode.

Syntax

FUNCTION getTextSegment RETURNS CHARACTER.

Applies to Message objects

Note

As an alternative to retrieving multiple text segments with the getTextSegment function, you
  can use the getLongText function to retrieve LONGCHAR data in a single operation.

See also createTextMessage procedure, setText procedure, setLongText procedure, appendText
  procedure, endOfStream function, getCharCount function, getText function, getTextSegment
  function, getLongText function, getLongTextCP function

For more information, see the “TextMessage” section on page 4–24.
getTransactedReceive function

Returns the value set by the preceding the setTransactedReceive procedure.

Syntax

```
FUNCTION getTransactedReceive RETURNS LOGICAL.
```

Applies to
Session objects

Note
If the setTransactedReceive procedure was not called, FALSE is returned.

See also
setTransactedReceive procedure, getTransactedReceive function, setTransactedSend procedure, getTransactedSend function

For more information, see the “Setting and getting JMS connection and session attributes” section on page 4–7.

getTransactedSend function

Returns the value set by the preceding the setTransactedSend procedure.

Syntax

```
FUNCTION getTransactedSend RETURNS LOGICAL.
```

Applies to
Session objects

Note
If the setTransactedSend procedure was not called, FALSE is returned.

See also
For more information, see the “Setting and getting JMS connection and session attributes” section on page 4–7.

getUser function

Returns the value set by the preceding the setUser procedure.

Syntax

```
FUNCTION getUser RETURNS CHARACTER.
```

Applies to
Session objects

Note
If the setUser procedure was not called, the Unknown value (?) is returned.

See also
setUser procedure, getUser function

For more information, see the “Setting and getting JMS connection and session attributes” section on page 4–7.
**getX-Document function**

Parses an XML document from a SonicMQ XMLMessage into an X-DOCUMENT.

**Syntax**

```
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 the `getLongText` function and `xdoc:LOAD` will be returned.

**See also**

`createXMLMessage` procedure, `setX-Document` procedure, `getX-Document` function, `setSaxReader` procedure, `getSaxWriter` function, `deleteSaxWriter` procedure

For more information, see the “XMLMessage” section on page 4–29.

**hasReplyTo function**

Returns TRUE if the JMSreplyTo header was set.

**Syntax**

```
FUNCTION hasReplyTo RETURNS LOGICAL.
```

**Applies to** Message objects

**See also**

`setJMSReplyTo` procedure, `getJMSReplyTo` function, `hasReplyTo` function, `setReplyToDestinationType` procedure, `getReplyToDestinationType` function

For more information, see the “Accessing message header properties” section on page 4–35.

**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, `inMessageHandling` function, `inQueueBrowsing` function, `inReplyHandling` function

For more information, see the “Accessing message handler information” section on page 4–38.
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, inMessageHandling function, inQueueBrowsing function, inReplyHandling function

For more information, see the “Accessing message handler information” section on page 4–38.

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, inMessageHandling function, inQueueBrowsing function, inReplyHandling function

For more information, see the “Accessing message handler information” section on page 4–38.

inReplyHandling function

Returns TRUE when called from a message handler if an application is handling a reply message.

Syntax

```
FUNCTION inReplyHandling RETURNS LOGICAL.
```

Applies to  
Message Consumer objects

See also  
inErrorHandling function, inMessageHandling function, inQueueBrowsing function, inReplyHandling function

For more information, see the “Accessing message handler information” section on page 4–38.
isFaultTolerant function

Determines if the SonicMQ Broker connected supports Fault Tolerance.

Syntax

FUNCTION isFaultTolerant RETURNS LOGICAL

Applies to

Session objects

Notes

• Only applicable for Fault Tolerant connections.
• This function must be called after the beginSession procedure is called.

See also

setFaultTolerant procedure, getFaultTolerant function, isFaultTolerant function, createChangeStateConsumer procedure, setFaultTolerantReconnectTimeout procedure, getFaultTolerantReconnectTimeout function, setInitialConnectionTimeout procedure, getInitialConnectionTimeout function, setClientTransactionBufferSize procedure, getClientTransactionBufferSize function

For more information, see the “Fault tolerance” section on page 4–9 and the “Fault tolerance” section on page 5–28.

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

Note

If the isMessagePart function returns TRUE, use message-part access methods.

See also

createMultipartMessage procedure, addBytesPart procedure, addMessagePart procedure, addTextPart procedure, isMessagePart function, getContentType, getPartCount function, getMessagePartByID function, getMessagePartByIndex function, writeBytesFromRaw procedure, readBytesToRaw procedure, setMemptr procedure, getMemptr function

For more information, see the “MultipartMessage” section on page 4–28 and the “MultiPartMessage example” section on page B–26.
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” section on page 4–11.
messageHandler procedure

Handles incoming JMS and error messages.

Syntax

```plaintext
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` (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, deleteConsumer procedure, messageHandler procedure, waitForMessages procedure

For more information see the “Message Consumer objects” section on page 2–5, the “Consuming messages” section on page 4–37, the “Terminating the Message Consumer object” section on page 4–39, and the “Creating a message handler process” section on page 4–38.
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 the reset procedure 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.
- The moveToNext procedure function cannot precisely determine certain data types. It is important to be aware of the following limitations:
  - **String values** — The moveToNext procedure 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** — The moveToNext procedure 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** — The moveToNext procedure 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, writeBoolean procedure, writeByte procedure, writeBytesFromRaw procedure, writeChar procedure, writeDate procedure, writeDateTime procedure, writeDateTime-TZ procedure, writeDouble procedure, writeFloat procedure, writeInt procedure, writeLong procedure, writeLongString procedure, writeShort procedure, writeString procedure, endOfStream function, moveToNext procedure, readBytesToRaw procedure, readChar function, readDate function, readDateTime function, readDateTime-TZ function, readDecimal function, readInt function, readInt64 function, readLogical function, readLongString function, readLongStringCP function

For more information, see the “StreamMessage” section on page 4–27.
publish procedure

Publishes a message to a topic.

Syntax

```plaintext
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, publish procedure, subscribe procedure

For more information, see the “Publishing messages to a topic” section on page 4–34, the “Subscribing to a topic” section on page 4–39, the “Durable subscriptions” section on page 4–39, and the “Methods unique to Pub/Sub messaging” section on page 3–8.

For an example, see the “Pub/Sub messaging example” section on page 5–6.
**readBytesToRaw procedure**

Returns byte array data from the body of a `StreamMessage` or a `BytesMessage`.

**Syntax**

```plaintext
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, `getContentType`, `writeBoolean` procedure, `writeByte` procedure, `writeBytesFromRaw` procedure, `writeChar` procedure, `writeDate` procedure, `writeDateTime` procedure, `writeDateTime-TZ` procedure, `writeDouble` procedure, `writeFloat` procedure, `writeInt` procedure, `writeLong` procedure, `writeLongString` procedure, `writeShort` procedure, `writeString` procedure, `endOfStream` function, `moveToNext` procedure, `readBytesToRaw` procedure, `readChar` function, `readDate` function, `readDateTime` function, `readDateTime-TZ` function, `readDecimal` function, `readInt` function, `readLogical` function, `readLongString` function, `readLongStringCP` function

For more information, see the “StreamMessage” section on page 4–27 and the “BytesMessage” section on page 4–28.

**readChar function**

Returns any message data segment except bytes data from the body of a `StreamMessage`.

**Syntax**

```plaintext
FUNCTION readChar RETURNS CHARACTER.
```

**Applies to** Message objects

**Note**

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, `writeBoolean` procedure, `writeByte` procedure, `writeBytesFromRaw` procedure, `writeChar` procedure, `writeDate` procedure, `writeDateTime` procedure, `writeDateTime-TZ` procedure, `writeDouble` procedure, `writeFloat` procedure, `writeInt` procedure, `writeLong` procedure, `writeLongString` procedure, `writeShort` procedure, `writeString` procedure, `endOfStream` function, `moveToNext` procedure, `readBytesToRaw` procedure, `readChar` function, `readDate` function, `readDateTime` function, `readDateTime-TZ` function, `readDecimal` function, `readInt` function, `readLogical` function, `readLongString` function, `readLongStringCP` function

For more information, see the “StreamMessage” section on page 4–27.
**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, writeBoolean procedure, writeByte procedure, writeBytesFromRaw procedure, writeChar procedure, writeDate procedure, writeDateTime procedure, writeDateTime-TZ procedure, writeDouble procedure, writeFloat procedure, writeInt procedure, writeLong procedure, writeLongString procedure, writeShort procedure, writeString procedure, endOfStream function, moveToNext procedure, readBytesToRaw procedure, readChar function, readDate function, readDateTime function, readDateTime-TZ function, readDecimal function, readInt function, readLogical function, readLongString function, readLongStringCP function

For more information, see the “StreamMessage” section on page 4–27.

**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, writeBoolean procedure, writeByte procedure, writeBytesFromRaw procedure, writeChar procedure, writeDate procedure, writeDateTime procedure, writeDateTime-TZ procedure, writeDouble procedure, writeFloat procedure, writeInt procedure, writeLong procedure, writeLongString procedure, writeShort procedure, writeString procedure, endOfStream function, moveToNext procedure, readBytesToRaw procedure, readChar function, readDate function, readDateTime function, readDateTime-TZ function, readDecimal function, readInt function, readLogical function, readLongString function, readLongStringCP function

For more information, see the “StreamMessage” section on page 4–27.
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, writeBoolean procedure, writeByte procedure, writeBytesFromRaw procedure, writeChar procedure, writeDate procedure, writeDateTime procedure, writeDateTime-TZ procedure, writeDouble procedure, writeFloat procedure, writeInt procedure, writeLong procedure, writeLongString procedure, writeShort procedure, writeString procedure, endOfStream function, moveToNext procedure, readBytesToRaw procedure, readChar function, readDate function, readDateTime function, readDateTime-TZ function, readDecimal function, readInt function, readLogical function, readLongString function, readLongStringCP function

For more information, see the “StreamMessage” section on page 4–27.

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, writeBoolean procedure, writeByte procedure, writeBytesFromRaw procedure, writeChar procedure, writeDate procedure, writeDateTime procedure, writeDateTime-TZ procedure, writeDouble procedure, writeFloat procedure, writeInt procedure, writeLong procedure, writeLongString procedure, writeShort procedure, writeString procedure, endOfStream function, moveToNext procedure, readBytesToRaw procedure, readChar function, readDate function, readDateTime function, readDateTime-TZ function, readDecimal function, readInt function, readLogical function, readLongString function, readLongStringCP function

For more information, see the “StreamMessage” section on page 4–27.
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, 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, eofStream function, moveToNext procedure, 
readBytesToRaw procedure, readChar function, readDate function, readDateTime function, 
readDateTime-TZ function, readDecimal function, readInt64 function, readLogical function, 
readLongString function, readLongStringCP function

For more information, see the “StreamMessage” section on page 4–27.

readInt64 function

Returns INT64 data from the body of a StreamMessage.

Syntax

FUNCTION readInt64 RETURNS INT64.

 Applies to Message objects

See also createStreamMessage 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, eofStream function, moveToNext procedure, 
readBytesToRaw procedure, readChar function, readDate function, readDateTime function, 
readDateTime-TZ function, readDecimal function, readInt64 function, readLogical function, 
readLongString function, readLongStringCP function

For more information, see the “StreamMessage” section on page 4–27.
**readLogical function**

Returns boolean data from the body of a `StreamMessage`.

**Syntax**

```ABL
FUNCTION readLogical RETURNS LOGICAL.
```

**Applies to** Message objects

**See also**
- `createStreamMessage` procedure
- `writeBoolean` procedure
- `writeByte` procedure
- `writeBytesFromRaw` procedure
- `writeChar` procedure
- `writeDate` procedure
- `writeDateTime` procedure
- `writeDateTime-TZ` procedure
- `writeDouble` procedure
- `writeFloat` procedure
- `writeInt` procedure
- `writeLong` procedure
- `writeLongString` procedure
- `writeShort` procedure
- `writeString` procedure
- `endOfStream` function
- `moveToNext` procedure
- `readBytesToRaw` procedure
- `readChar` function
- `readDate` function
- `readDateTime` function
- `readDateTime-TZ` function
- `readDecimal` function
- `readFloat` function
- `readInt` function
- `readLogical` function
- `readLongString` function
- `readLongStringCP` function

For more information, see the “StreamMessage” section on page 4–27.

---

**readLongString function**

Returns String data from the body of a `StreamMessage`.

**Syntax**

```ABL
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 the `readLongStringCP` function.

**See also**
- `createStreamMessage` procedure
- `writeBoolean` procedure
- `writeByte` procedure
- `writeBytesFromRaw` procedure
- `writeChar` procedure
- `writeDate` procedure
- `writeDateTime` procedure
- `writeDateTime-TZ` procedure
- `writeDouble` procedure
- `writeFloat` procedure
- `writeInt` procedure
- `writeLong` procedure
- `writeLongString` procedure
- `writeShort` procedure
- `writeString` procedure
- `endOfStream` function
- `moveToNext` procedure
- `readBytesToRaw` procedure
- `readChar` function
- `readDate` function
- `readDateTime` function
- `readDateTime-TZ` function
- `readDecimal` function
- `readFloat` function
- `readInt` function
- `readLogical` function
- `readLongString` function
- `readLongStringCP` function

For more information, see the “StreamMessage” section on page 4–27.
readLongStringCP function

Returns LONGCHAR data from the body of a StreamMessage and converts it to the specified code page.

**Syntax**

```
FUNCTION readLongStringCP RETURNS LONGCHAR (code_page as CHARACTER).
```

**Applies to** Message objects

**Note** The LONGCHAR data returned is converted to the code page specified by the `code_page` parameter.

**See also**

createStreamMessage procedure, writeBoolean procedure, writeByte procedure, writeBytesFromRaw procedure, writeChar procedure, writeDate procedure, writeDateTime procedure, writeDateTime-TZ procedure, writeDouble procedure, writeFloat procedure, writeInt procedure, writeLong procedure, writeLongString procedure, writeShort procedure, writeString procedure, endOfStream function, moveToNext procedure, readBytesToRaw procedure, readChar function, readDate function, readDateTime function, readDateTime-TZ function, readDecimal function, readInt function, readLogical function, readLongString function, readLongStringCP function

For more information, see the “StreamMessage” section on page 4–27.
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, receiveFromQueue procedure, sendToQueue procedure

For more information, see the “Sending messages to a queue” section on page 4–34, the “Receiving messages from a queue” section on page 4–38, and the “Methods unique to Point-to-Point messaging” section on page 3–4.

For an example, see the “PTP message example” section on page 5–2.
recover procedure

Redelivers all unacknowledged messages received up to that point in the current session.

Syntax

```procedural
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 instead.

- This procedure executes remotely (sends a message to the OpenEdge Adapter for SonicMQ).

See also

commitSend procedure, commitReceive procedure, rollbackSend procedure, rollbackReceive procedure, recover procedure

For more information, see the “Transaction and recovery procedures” section on page 4–43.
**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**

```abl
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 the `requestReply` procedure 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” section on page 4–12.
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

```language
PROCEDURE reset.
```

Applies to Message objects

Notes

- Sending the message causes an implicit call to the reset procedure.
- The message becomes read-only and arrives at the receiver in a reset state.

See also For more information, see the “TextMessage” section on page 4–24, the “StreamMessage” section on page 4–27, and the “BytesMessage” section on page 4–28.

rollbackReceive procedure

Starts redelivering the messages received up to that point in the current transaction.

Syntax

```language
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, commitReceive procedure, rollbackSend procedure, rollbackReceive procedure, recover procedure

For more information, see the “Transaction and recovery procedures” section on page 4–43.
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, commitReceive procedure, rollbackSend procedure, rollbackReceive procedure, recover procedure

For more information, see the “Message acknowledgement, forwarding, and recovery” section on page 4–44.
sendToQueue procedure

Sends a message to a queue.

Syntax

```pascal
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, receiveFromQueue procedure, sendToQueue procedure

For more information, see the “Sending messages to a queue” section on page 4–34, the “Receiving messages from a queue” section on page 4–38, and the “Methods unique to Point-to-Point messaging” section on page 3–4.

For an example, see the “PTP message example” section on page 5–2.
setAdapterService procedure

Specifies the service name under which the OpenEdge Adapter for SonicMQ BrokerConnect is registered with the NameServer.

Syntax

```abl
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 the `setAdapterService` procedure 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, `getAdapterService` function

For more information, see the “Setting and getting JMS connection and session attributes” section on page 4–7.

setApplicationContext procedure

Passes context to the message handler.

Syntax

```abl
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, `getApplicationContext` function

For more information, see the “Accessing message handler information” section on page 4–38 and the “Creating a message handler process” section on page 4–38.
**setBoolean procedure**

Converts data to the JMS boolean data type in a MapMessage.

**Syntax**

```
PROCEDURE setBoolean.
DEFINE INPUT PARAMETER itemName AS CHARACTER.
DEFINE INPUT PARAMETER value AS LOGICAL.
```

**Applies to** Message objects

**Note** An Unknown value (?) is considered FALSE.

**See also** createMapMessage procedure, setBoolean procedure, setByte procedure, setBytesFromRaw procedure, setChar procedure, setDate procedure, setDateTime procedure, setDateTime-TZ procedure, setDouble procedure, setFloat procedure, setInt procedure, setLong procedure, setLongString procedure, setShort procedure, setString procedure, getMapNames function, getItemType function, getBytesToRaw function, getChar function, getDate function, getDateTime function, getDateTime-TZ function, getDecimal function, getInt function, getLogical function, getLongString function, getLongStringCP function

For more information, see the “MapMessage” section on page 4–26.

**setBooleanProperty procedure**

Sets a boolean message property.

**Syntax**

```
PROCEDURE setBooleanProperty.
DEFINE INPUT PARAMETER propertyName AS CHARACTER.
DEFINE INPUT PARAMETER propertyValue AS LOGICAL.
```

**Applies to** Message objects

**Note** An Unknown value (?) is considered a FALSE value.

**See also** clearProperties procedure

For more information, see the “Accessing message properties” section on page 4–36.
setBrokerURL procedure

Sets the value of the SonicMQ Broker URL.

Syntax

```
PROCEDURE setBrokerURL.
DEFINE INPUT PARAMETER brokerURL AS CHARACTER.
```

Parameters  

`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, getBrokerURL function

For more information, see the “Setting and getting JMS connection and session attributes” section on page 4–7.

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, setBoolean procedure, setByte procedure, setBytesFromRaw procedure, setChar procedure, setDate procedure, setDateTime procedure, setDateTime-TZ procedure, setDouble procedure, setFloat procedure, setInt procedure, setLong procedure, setLongString procedure, setShort procedure, setString procedure, getMapNames function, getItemType function, getBytesToRaw function, getChar function, getDate function, getDateTime function, getDateTime-TZ function, getDecimal function, getInt function, getLogical function, getLongString function, getLongStringCP function

For more information, see the “MapMessage” section on page 4–26.
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

Note  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

For more information, see the “Accessing message properties” section on page 4–36.

setBytesFromRaw procedure

Converts data in a MapMessage to the JMS byte data type.

Syntax

```plaintext
PROCEDURE setBytesFromRaw.
DEFINE INPUT PARAMETER itemName AS CHARACTER.
DEFINE INPUT PARAMETER values AS RAW.
```

Applies to  Message objects

See also  createMapMessage procedure, setBoolean procedure, setByte procedure, setBytesFromRaw procedure, setChar procedure, setDate procedure, setDateTime procedure, setDateTime-TZ procedure, setDouble procedure, setFloat procedure, setInt procedure, setLong procedure, setLongString procedure, setShort procedure, setString procedure, getMapNames function, getItemType function, getBytesToRaw function, getChar function, getDate function, getDateTime function, getDateTime-TZ function, getDecimal function, getInt function, getLogical function, getLongString function, getLongStringCP function

For more information, see the “MapMessage” section on page 4–26.
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

Note The number of characters in the char value must be one.

See also createMapMessage procedure, setBoolean procedure, setByte procedure, setBytesFromRaw procedure, setChar procedure, setDate procedure, setDateTime procedure, setDateTime-TZ procedure, setDouble procedure, setFloat procedure, setInt procedure, setLong procedure, setLongString procedure, setShort procedure, setString procedure, getMapNames function, getItemType function, getBytesToRaw function, getChar function, getDate function, getDateTime function, getDateTime-TZ function, getDecimal function, getInt function, getLogical function, getLongString function, getLongStringCP function

For more information, see the “MapMessage” section on page 4–26.

setClientID procedure

Sets the client ID value for the SonicMQ Broker connection and overwrites the default client ID set on the server side.

Syntax

```
PROCEDURE setClientID.
DEFINE INPUT PARAMETER clientID AS CHARACTER.
```

Parameters 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, the setClientID procedure overwrites the default client ID set on the server side.

See also setClientID procedure, getClientID function

For more information, see the “Setting and getting JMS connection and session attributes” section on page 4–7, the “Methods unique to Pub/Sub messaging” section on page 3–8, and the “Subscribing to a topic” section on page 4–39.
setClientPersistence procedure

Sets client persistence.

Syntax

```
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 the setClientID procedure to use client persistence.
- If the default is FALSE, client persistence is not enabled.
- You cannot change client persistence for an active session.
- Call the setClientPersistence procedure prior to calling the beginSession procedure.
- To change the value of client persistence, you must stop and restart the session for any changes to take effect.

See also
createRejectedMessageConsumer procedure, setClientPersistence procedure, getClientPersistence function, setLocalStoreDirectory procedure, getLocalStoreDirectory function, setLocalStoreSize procedure, getLocalStoreSize function, setLocalStoreWaitTime procedure, getLocalStoreWaitTime function, setReconnectTimeout procedure, getReconnectTimeout function, setReconnectInterval procedure, getReconnectInterval function

For more information, see the “Client persistence” section on page 4–8 and the “Client persistence” section on page 5–12.
**setClientTransactionBufferSize procedure**

Sets the SonicMQ client buffer size in bytes for Fault Tolerant transacted messages in memory.

**Syntax**

```apl
PROCEDURE setClientTransactionBufferSize.
DEFINE INPUT PARAMETER sz AS INTEGER.
```

**Applies to** Session objects (ClientConnect and ServerConnect only).

**Notes**

- Only applicable for Fault Tolerant connections.
- Call the `setClientTransactionBufferSize` procedure before the `beginSession` procedure 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, `getFaultTolerant` function, `isFaultTolerant` function,
- `createChangeStateConsumer` procedure, `setFaultTolerantReconnectTimeout` procedure,
- `getFaultTolerantReconnectTimeout` function, `setInitialConnectionTimeout` procedure,
- `getInitialConnectionTimeout` function, `setClientTransactionBufferSize` procedure,
- `getClientTransactionBufferSize` function

For more information, see the “Fault tolerance” section on page 4–9 and the “Fault tolerance” section on page 5–28.
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.

Note
Sets the connection ID between the Sonic client and Sonic broker. If not explicitly set, Sonic uses an internal value.

See also
getConnectID function

setConnectionFile procedure

Sets the connection file.

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 the beginSession procedure is called.

See also
For more information, see the “Using serialized connection objects” section on page 4–13.
setConnectionURLs procedure

Specifies a list of broker URLs for the client to try to connect to.

Syntax

```c
PROCEDURE setConnectionURLs.
DEFINE INPUT PARAMETER brokerList AS CHARACTER.
```

Parameters  
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 the setBrokerURL procedure.
- Call this procedure instead of the setBrokerURL procedure when there is a list of broker URLs.

See also  
setConnectionURLs procedure, getConnectionURLs function, setSequential procedure, getSequential function

For more information, see the “Managing fail-over support” section on page 4–7.

setDataSet procedure

Sets the DataSetMessage.

Syntax

```c
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 the setLongText procedure.

See also  
createDataSetMessage procedure, setDataSet procedure, getDataSet function

For an example, see the “DataSetMessage” section on page 5–24.
setDate procedure

Sets a date value as String data in a MapMessage.

Syntax

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, setBoolean procedure, setByte procedure, setBytesFromRaw procedure, setChar procedure, setDate procedure, setDateTime procedure, setDateTime-TZ procedure, setDouble procedure, setFloat procedure, setInt procedure, setLong procedure, setLongString procedure, setShort procedure, setString procedure, getMapNames function, getItemType function, getBytesToRaw function, getChar function, getDate function, getDateTime function, getDateTime-TZ function, getDecimal function, getInt function, getLogical function, getLongString function, getLongStringCP function

For more information, see the “MapMessage” section on page 4–26.
**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`, `getDateProperty function`

For more information, see the “Accessing message properties” section on page 4–36.
setDateTime procedure

Sets a date-time value as String data in a MapMessage.

Syntax

```plaintext
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  createMapMessage procedure, setBoolean procedure, setByte procedure, setBytesFromRaw procedure, setChar procedure, setDate procedure, setDateTime procedure, setDateTime-TZ procedure, setDouble procedure, setFloat procedure, setInt procedure, setLong procedure, setLongString procedure, setShort procedure, setString procedure, getMapNames function, getItemType function, getBytesToRaw function, getChar function, getDate function, getDateTime function, getDateTime-TZ function, getDecimal function, getInt function, getLogical function, getLongString function, getLongStringCP function

For more information, see the “MapMessage” section on page 4–26.

setDateTimeProperty procedure

Sets a date-time value.

Syntax

```plaintext
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, getDateTimeProperty function

For more information, see the “Accessing message properties” section on page 4–36.
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 DATETIME-TZ value.

See also  createMapMessage procedure, setBoolean procedure, setByte procedure, setBytesFromRaw procedure, setChar procedure, setDate procedure, setDateTime procedure, setDateTime-TZ procedure, setDouble procedure, setFloat procedure, setInt procedure, setLong procedure, setLongString procedure, setShort procedure, setString procedure, getMapNames function, getItemType function, getBytesToRaw function, getChar function, getDate function, getDateTime function, getDateTime-TZ function, getDecimal function, getInt function, getLogical function, getLongString function, getLongStringCP function

For more information, see the “MapMessage” section on page 4–26.

setDateTimeTzProperty procedure

Sets a date-time value, including time zone information.

Syntax

```
PROCEDURE setDateTimeTzProperty.
DEFINE INPUT PARAMETER propertyName AS CHARACTER.
DEFINE INPUT PARAMETER propertyValue AS DATETIME-TZ.
```

Applies to  Message objects

Notes

- The date-time value, including time zone information, is set as a string message property in the message header.
- The procedure writes value as a DATETIME-TZ value, adding default time and time zone information.

See also  setDateTimeTzProperty procedure, getDateTimeTzProperty function

For more information, see the “Accessing message properties” section on page 4–36.
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, getDefaultPersistency function, setDefaultPriority procedure, getDefaultPriority function, setDefaultTimeToLive procedure, getDefaultTimeToLive function

For more information, see the “Accessing message delivery parameters” section on page 4–11.

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.

See also setDefaultPersistency procedure, getDefaultPersistency function, setDefaultPriority procedure, getDefaultPriority function, setDefaultTimeToLive procedure, getDefaultTimeToLive function

For more information, see the “Accessing message delivery parameters” section on page 4–11.
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, getDefaultPersistency function, setDefaultPriority procedure, getDefaultPriority function, setDefaultTimeToLive procedure, getDefaultTimeToLive function

For more information, see the “Accessing message delivery parameters” section on page 4–11.

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.
```

**Applies to** Message objects

**See also** createMapMessage procedure, setBoolean procedure, setByte procedure, setBytesFromRaw procedure, setChar procedure, setDate procedure, setDateTime procedure, setDateTime-TZ procedure, setDouble procedure, setFloat procedure, setInt procedure, setLong procedure, setLongString procedure, setShort procedure, setString procedure, getMapNames function, getItemType function, getBytesToRaw function, getChar function, getDate function, getDateTime function, getDateTime-TZ function, getDecimal function, getInt function, getLogical function, getLongString function, getLongStringCP function

For more information, see the “MapMessage” section on page 4–26.
setDoubleProperty procedure

Sets a double message property.

Syntax

```
PROCEDURE setDoubleProperty.
DEFINE INPUT PARAMETER propertyName AS CHARACTER.
DEFINE INPUT PARAMETER propertyValue AS DECIMAL.
```

Applies to Message objects

See also clearProperties procedure

For more information, see the “Accessing message properties” section on page 4–36.

setErrorHandler procedure

Sets the error-handling procedure.

Syntax

```
PROCEDURE setErrorHandler.
DEFINE INPUT PARAMETER messageConsumer AS HANDLE.
```

Applies to Session objects

Notes

- The application must create the error-handling messageConsumer object and call the setErrorHandler procedure after calling the beginSession procedure. If the application does not call the setErrorHandler procedure, 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 the setErrorHandler procedure in the Session object. The setErrorHandler procedure 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).

- The getPropertyNames function can be used to get the list of properties in the error message header.

See also setErrorHandler procedure, setNoErrorDisplay procedure

For more information, see the “Error and condition handling” section on page 4–46. For an example, see the Appendix B, “Messaging Examples.”
setFaultTolerant procedure

Enables or disables Fault Tolerance for the session.

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 the beginSession procedure 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 the setConnectionURLs procedure 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.
- The setSequential procedure 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, getFaultTolerant function, isFaultTolerant function, createChangeStateConsumer procedure, setFaultTolerantReconnectTimeout procedure, getFaultTolerantReconnectTimeout function, setInitialConnectionTimeout procedure, getInitialConnectionTimeout function, setClientTransactionBufferSize procedure, getClientTransactionBufferSize function

For more information, see the “Fault tolerance” section on page 4–9 and the “Fault tolerance” section on page 5–28.
setFaultTolerantReconnectTimeout procedure

Sets a time limit in seconds on reconnection attempts.

**Syntax**

```plaintext
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 the `beginSession` procedure is called.
- The URLs specified in the `setConnectionURLs` procedure are used to attempt reconnection to a SonicMQ Broker.

**See also**  
`setFaultTolerant` procedure, `getFaultTolerant` function, `isFaultTolerant` function, `createChangeStateConsumer` procedure, `setFaultTolerantReconnectTimeout` procedure, `getFaultTolerantReconnectTimeout` function, `setInitialConnectionTimeout` procedure, `getInitialConnectionTimeout` function, `setClientTransactionBufferSize` procedure, `getClientTransactionBufferSize` function

For more information, see the “Fault tolerance” section on page 4–9 and the “Fault tolerance” section on page 5–28.

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, `setBoolean` procedure, `setByte` procedure, `setBytesFromRaw` procedure, `setChar` procedure, `setDate` procedure, `setDateTime` procedure, `setDateTime-TZ` procedure, `setDouble` procedure, `setFloat` procedure, `setInt` procedure, `setLong` procedure, `setLongString` procedure, `setShort` procedure, `setString` procedure, `getMapNames` function, `getItemType` function, `getBytesToRaw` function, `getChar` function, `getDate` function, `getDateTime` function, `getDateTime-TZ` function, `getDecimal` function, `getInt` function, `getLogical` function, `getLongString` function, `getLongStringCP` function

For more information, see the “MapMessage” section on page 4–26.
setFloatProperty procedure

Sets a float message property.

Syntax

```abl
PROCEDURE setFloatProperty.
DEFINE INPUT PARAMETER propertyName AS CHARACTER.
DEFINE INPUT PARAMETER propertyValue AS DECIMAL.
```

Applies to Message objects

See also clearProperties procedure

For more information, see the “Accessing message properties” section on page 4–36.

setFlowToDisk procedure

Enables or disables the SonicMQ flow-to-disk functionality.

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 the `beginSession procedure`.
- The SonicMQ broker only uses the flow-to-disk feature for pub/sub messages.

See also getFlowToDisk function

For more information on this SonicMQ feature see the Flow to Disk section in the chapter for SonicMQ Client Sessions in the SonicMQ Application Programming Guide.
setInt procedure

Converts data in a MapMessage to the JMS int data type.

Syntax

```assembler
PROCEDURE setInt.
DEFINe INPUT PARAMETER itemName AS CHARACTER.
DEFINe INPUT PARAMETER value AS INTEGER.
```

Applies to Message objects

See also createMapMessage procedure, setBoolean procedure, setByte procedure, setBytesFromRaw procedure, setChar procedure, setDateTime procedure, setDateTime-TZ procedure, setDouble procedure, setFloat procedure, setInt64 procedure, setLong procedure, setLongString procedure, setShort procedure, setString procedure, getMapNames function, getItemType function, getBytesToRaw function, getChar function, getDate function, getDateTime function, getDateTime-TZ function, getDecimal function, getInt function, getInt64 function, getLogical function, getLongString function, getLongStringCP function

For more information, see the “MapMessage” section on page 4–26.

setIntProperty procedure

Converts an int message property.

Syntax

```assembler
PROCEDURE setIntProperty.
DEFINe INPUT PARAMETER propertyName AS CHARACTER.
DEFINe INPUT PARAMETER propertyValue AS INTEGER.
```

Applies to Message objects

See also getIntProperty function, clearProperties procedure, setInt64Property procedure

For more information, see the “Accessing message properties” section on page 4–36.
**setInt64 procedure**

Converts data in a `MapMessage` to the JMS `long` data type.

**Syntax**

```apl
PROCEDURE setInt64.
DEFINE INPUT PARAMETER itemName AS CHARACTER.
DEFINE INPUT PARAMETER value AS INT64.
```

**Applies to** Message objects

**Note**
The Unknown value (?) is allowed.

**See also**
`createMapMessage procedure`, `setBoolean procedure`, `setByte procedure`, `setBytesFromRaw procedure`, `setChar procedure`, `setDate procedure`, `setDateTime procedure`, `setDateTime-TZ procedure`, `setDouble procedure`, `setFloat procedure`, `setInt procedure`, `setLong procedure`, `setLongString procedure`, `setShort procedure`, `setString procedure`, `getMapNames function`, `getItemType function`, `getBytesToRaw function`, `getChar function`, `getDate function`, `getDateTime function`, `getDateTime-TZ function`, `getDecimal function`, `getInt function`, `getInt64 function`, `getLogical function`, `getLongString function`, `getLongStringCP function`

For more information, see the “MapMessage” section on page 4–26.

---

**setInt64Property procedure**

Sets the value of a JMS message property to `INT64`.

**Syntax**

```apl
PROCEDURE setInt64Property.
DEFINE INPUT PARAMETER propertyName AS CHARACTER.
DEFINE INPUT PARAMETER propertyValue AS INT64.
```

**Applies to** Message objects

**Note**
The Unknown value (?) is allowed.

**See also**
`getInt64Property function`, `setIntProperty procedure`, `getIntProperty function`, `clearProperties procedure`

For more information, see the “Accessing message properties” section on page 4–36.
setInitialConnectionTimeout procedure

Sets a time limit in seconds on the initial Fault tolerant connection attempt to the SonicMQ Broker.

**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 the `beginSession` procedure is called.
- The URLs specified in the `setConnectionURLs` procedure 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, `getFaultTolerant` function, `isFaultTolerant` function, `createChangeStateConsumer` procedure, `setFaultTolerantReconnectTimeout` procedure, `getFaultTolerantReconnectTimeout` function, `setInitialConnectionTimeout` procedure, `getClientTransactionBufferSize` procedure, `getInitialConnectionTimeout` function

For more information, see the “Fault tolerance” section on page 4–9 and the “Fault tolerance” section on page 5–28.

setJMSCorrelationID procedure

Sets the correlation ID.

**Syntax**

```
PROCEDURE setJMSCorrelationID
DEFINE INPUT PARAMETER correlationID AS CHARACTER.
```

**Applies to** Message objects

**Note**

This value is application-defined; typically it is set to the ID of the message replied to.

**See also**

setJMSCorrelationID procedure, `getJMSCorrelationID` function

For more information, see the “Accessing message header properties” section on page 4–35.
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, getJMSCorrelationIDAsBytes function

For more information, see the “Accessing message header properties” section on page 4–35.

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.
- The `setReplyToDestinationType` procedure must be called to set the correct destination type.

See also setJMSReplyTo procedure, getJMSReplyTo function, hasReplyTo function, setReplyToDestinationType procedure, getReplyToDestinationType function

For more information, see the “Accessing message header properties” section on page 4–35.
**setJMSServerName procedure**

Specifies the JMS broker implementation, SonicMQ.

**Syntax**

```
PROCEDURE setJmsServerName.
DEFINE INPUT PARAMETER jmsServerName AS CHARACTER.
```

**Parameters**  
*jmsServerName*

Specifies the JMS broker implementation, SonicMQ.

**Applies to**  
Session objects

**Note**  
If set on the client, it overwrites the *jmsServerName* property set on the OpenEdge Adapter for SonicMQ side.

**See also**  
setJMSServerName procedure, getJMSServerName function

For more information, see the “Setting and getting JMS connection and session attributes” section on page 4–7.

**setJMSType procedure**

Sets the type name, which is proprietary (JMS-provider-dependent).

**Syntax**

```
PROCEDURE setJMSType
DEFINE INPUT PARAMETER typeName AS CHARACTER.
```

**Applies to**  
Message objects

**Note**  
When accessing SonicMQ, the JMSType field can be used for storing application-defined values.

**See also**  
setJMSType procedure, getJMSType function

For more information, see the “Accessing message header properties” section on page 4–35.
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 the beginSession procedure has already been called, an error is raised.

See also  setLoadBalancing procedure, getLoadBalancing function

For more information, see the “Load balancing” section on page 4–8.

setLocalStoreDirectory procedure

Sets the directory that will be used by the adapter to persist messages.

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 the beginSession procedure.

See also  createRejectedMessageConsumer procedure, setClientPersistence procedure, getClientPersistence function, setLocalStoreDirectory procedure, getLocalStoreDirectory function, setLocalStoreSize procedure, getLocalStoreSize function, setLocalStoreWaitTime procedure, getLocalStoreWaitTime function, setReconnectTimeout procedure, getReconnectTimeout function, setReconnectInterval procedure, getReconnectInterval function

For more information, see the “Client persistence” section on page 4–8 and the “Client persistence” section on page 5–12.
**setLocalStoreSize procedure**

Sets the maximum size of the local store in kilobytes.

**Syntax**

```plaintext
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 the `beginSession` procedure.

**See also**
- `createRejectedMessageConsumer` procedure, `setClientPersistence` procedure, `getClientPersistence` function, `setLocalStoreDirectory` procedure, `getLocalStoreDirectory` function, `setLocalStoreSize` function, `getLocalStoreSize` function, `setLocalStoreWaitTime` procedure, `getLocalStoreWaitTime` function, `setReconnectTimeout` procedure, `getReconnectTimeout` function, `setReconnectInterval` procedure, `getReconnectInterval` function

For more information, see the “Client persistence” section on page 4–8 and the “Client persistence” section on page 5–12.

**setLocalStoreWaitTime procedure**

Sets the interval in seconds before Client Persistence begins.

**Syntax**

```plaintext
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 the `beginSession` procedure.

**See also**
- `createRejectedMessageConsumer` procedure, `setClientPersistence` procedure, `getClientPersistence` function, `setLocalStoreDirectory` procedure, `getLocalStoreDirectory` function, `setLocalStoreSize` procedure, `getLocalStoreSize` function, `getLocalStoreWaitTime` function, `setReconnectTimeout` procedure, `getReconnectTimeout` function, `setReconnectInterval` procedure, `getReconnectInterval` function

For more information, see the “Client persistence” section on page 4–8 and the “Client persistence” section on page 5–12.
**setLong procedure**

Converts long data in a MapMessage in Text and XML messages.

**Syntax**

```plaintext
PROCEDURE setLong.
DEFINE INPUT PARAMETER itemName AS CHARACTER.
DEFINE INPUT PARAMETER value AS DECIMAL.
```

**Applies to** Message objects

**Note** Any fractional part of the DECIMAL value is truncated.

**See also** createMapMessage procedure, setBoolean procedure, setByte procedure, setBytesFromRaw procedure, setChar procedure, setDate procedure, setDateTime procedure, setDateTime-TZ procedure, setDouble procedure, setFloat procedure, setInt procedure, setLong procedure, setLongString procedure, setShort procedure, setString procedure, getMapNames function, getItemType function, getBytesToRaw function, getChar function, getDate function, getDateTime function, getDateTime-TZ function, getDecimal function, getInt function, getLogical function, getLongString function, getLongStringCP function

For more information, see the “MapMessage” section on page 4–26.

**setLongProperty procedure**

Sets a long message property.

**Syntax**

```plaintext
PROCEDURE setLongProperty.
DEFINE INPUT PARAMETER propertyName AS CHARACTER.
DEFINE INPUT PARAMETER propertyValue AS DECIMAL.
```

**Applies to** Message objects

**Note** Any fractional part of the DECIMAL value is truncated.

**See also** clearProperties procedure

For more information, see the “Accessing message properties” section on page 4–36.
**setLongString procedure**

Sets String data in a MapMessage.

**Syntax**

```
PROCEDURE setLongString.
DEFINE INPUT PARAMETER itemName AS CHARACTER.
DEFINE INPUT PARAMETER value AS LONGCHAR.
```

**Applies to** Message objects

**See also**

createMapMessage procedure, setBoolean procedure, setByte procedure, setBytesFromRaw procedure, setChar procedure, setDate procedure, setDateTime procedure, setDateTime-TZ procedure, setDouble procedure, setFloat procedure, setInt procedure, setLong procedure, setLongString procedure, setShort procedure, setString procedure, getMapNames function, getItemType function, getBytesToRaw function, getChar function, getDate function, getDateTime function, getDateTime-TZ function, getDecimal function, getInt function, getLogical function, getLongString function, getLongStringCP function

For more information, see the “MapMessage” section on page 4–26.

**setLongText procedure**

Clears the message body and sets a new text or XML value of any length.

**Syntax**

```
PROCEDURE setLongText.
DEFINE INPUT PARAMETER textValue AS LONGCHAR.
```

**Applies to** Message objects

**Note** An error is returned if the Unknown value (?) is specified.

**See also**

createTextMessage procedure, setText procedure, setLongText procedure, appendText procedure, endOfStream function, getCharCount function, getText function, getTextSegment function, getLongText function, getLongTextCP function

For more information, see the “TextMessage” section on page 4–24.
**setMemptr procedure**

Sets the specified number of bytes from the MEMPTR variable starting at `startIndex` in a BytesMessage.

**Syntax**

```plaintext
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.
- The `setMemptr` procedure implicitly calls the `clearBody` procedure 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, `setMemptr` procedure, `writeBoolean` procedure, `writeByte` procedure, `writeBytesFromRaw` procedure, `writeChar` procedure, `writeDate` procedure, `writeDateTime` procedure, `writeDateTime-TZ` procedure, `writeDouble` procedure, `writeFloat` procedure, `writeInt` procedure, `writeLong` procedure, `writeLongString` procedure, `writeShort` procedure, `writeString` procedure, `getContentType` function, `getBytesCount` function, `endOfStream` function, `moveToNext` procedure, `readBytesToRaw` procedure, `readChar` function, `readDate` function, `readDateTime` function, `readDateTime-TZ` function, `readDecimal` function, `readInt` function, `readLogical` function, `readLongString` function, `readLongStringCP` function, `getMemptr` function

For more information, see the “BytesMessage” section on page 4–28.

For an example, see the “Publishing, subscribing, and receiving an XML document in a BytesMessage” section on page B–13.
**setNoAcknowledge procedure**

Instructs the ABL–JMS implementation not to acknowledge this message.

**Syntax**

```
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, setSingleMessageAcknowledgement procedure, getSingleMessageAcknowledgement function, setNoAcknowledge procedure, getNoAcknowledge function

For more information, see the “Message acknowledgement, forwarding, and recovery” section on page 4–44 and the “Single-message acknowledgement” section on page 4–45.

**setNoErrorDisplay procedure**

Turns the automatic display of synchronous errors and conditions on and off.

**Syntax**

```
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, the setNoErrorDisplay procedure must be called in the message itself to change the noDisplay property.
- Errors caused by method calls are automatically displayed.

**See also**

setErrorHandler procedure, setNoErrorDisplay procedure

For more information, see the “Error and condition handling” section on page 4–46.
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

```plaintext
PROCEDURE setPassword.  
DEFINE INPUT PARAMETER password AS CHARACTER.
```

### Parameters

- **password**
  
  Password value for the SonicMQ Broker login.

### Applies to

Session objects

### Note

If called, the `setPassword` procedure overwrites the default password property set on the OpenEdge Adapter for SonicMQ side.

### See also

- `setPassword` procedure, `getPassword` function
  
  For more information, see the “Setting and getting JMS connection and session attributes” section on page 4–7.
setPingInterval procedure

Specifies the interval in seconds for the JMS Adapter to actively ping the SonicMQ Broker so communication failure can be detected promptly.

Syntax

PROCEDURE setPingInterval.
DEFINE INPUT PARAMETER pingInterval AS INTEGER.

Parameters

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 the beginSession procedure is called.

See also

For more information, see the “Setting and getting JMS connection and session attributes” section on page 4–7, 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.

Syntax

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 message from a queue.
- If this procedure is called after the beginSession procedure is called, an error is raised.

See also

setPrefetchCount procedure, setPrefetchThreshold procedure

For more information, see the “Controlling flow of messages” section on page 4–39.
setPrefetchThreshold procedure

Determines when the SonicMQ client goes back to the broker for more messages.

Syntax

```abl
PROCEDURE setPrefetchThreshold.
DEFINE INPUT PARAMETER threshold AS INTEGER.
```

Parameters

*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 the `beginSession` procedure is called, an error is raised.

See also

`setPrefetchCount` procedure, `setPrefetchThreshold` procedure

For more information, see the “Controlling flow of messages” section on page 4–39.

setReconnectInterval procedure

Sets the interval in seconds between reconnect attempts.

Syntax

```abl
PROCEDURE setReconnectInterval.
DEFINE INPUT PARAMETER interval AS INTEGER.
```

Applies to

Session objects (ClientConnect and ServerConnect only)

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 the `beginSession` procedure.

See also

`createRejectedMessageConsumer` procedure, `setClientPersistence` procedure, `getClientPersistence` function, `setLocalStoreDirectory` procedure, `getLocalStoreDirectory` function, `setLocalStoreSize` procedure, `getLocalStoreSize` function, `setLocalStoreWaitTime` procedure, `getLocalStoreWaitTime` function, `setReconnectTimeout` procedure, `getReconnectTimeout` function, `setReconnectInterval` procedure, `getReconnectInterval` function

For more information, see the “Client persistence” section on page 4–8 and the “Client persistence” section on page 5–12.
setReconnectTimeout procedure

Sets the maximum amount of time in minutes that the client will attempt to reconnect to a broker.

Syntax

```plaintext
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 the beginSession procedure.

See also createRejectedMessageConsumer procedure, setClientPersistence procedure, getClientPersistence function, setLocalStoreDirectory procedure, getLocalStoreDirectory function, setLocalStoreSize procedure, getLocalStoreSize function, setLocalStoreWaitTime procedure, getLocalStoreWaitTime function, setReconnectTimeout procedure, getReconnectTimeout function, setReconnectInterval procedure, getReconnectInterval function

For more information, see the “Client persistence” section on page 4–8 and the “Client persistence” section on page 5–12.

setReplyAutoDelete procedure

Specifies whether all reply messages are to be automatically deleted.

Syntax

```plaintext
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 setReplyPersistency procedure, getReplyPersistency function, setReplyPriority procedure, getReplyPriority function, setReplyTimeToLive procedure, getReplyTimeToLive function, setReplyAutoDelete procedure, getReplyAutoDelete function

For more information, see the “Accessing message handler information” section on page 4–38 and the “Setting reply properties” section on page 4–38.
setReplyPersistency procedure

Sets the value for message persistency when the Message Consumer is passed to the requestReply procedure.

Syntax

```plaintext
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 setReplyPersistency procedure, getReplyPersistency function, setReplyPriority procedure, getReplyPriority function, setReplyTimeToLive procedure, getReplyTimeToLive function, setReplyAutoDelete procedure, getReplyAutoDelete function

For more information, see the “Accessing message handler information” section on page 4–38 and the “Setting reply properties” section on page 4–38.

setReplyPriority procedure

Sets the priority of the reply messages when the Message Consumer is passed to the requestReply procedure.

Syntax

```plaintext
PROCEDURE setReplyPriority.
DEFINe INPUT PARAMETER priority AS INTEGER.
```

Applies to Message Consumer objects

Notes

- The range of values is 0–9; the default is 4.
- This procedure can be called only once.

See also setReplyPersistency procedure, getReplyPersistency function, setReplyPriority procedure, getReplyPriority function, setReplyTimeToLive procedure, getReplyTimeToLive function, setReplyAutoDelete procedure, getReplyAutoDelete function

For more information, see the “Accessing message handler information” section on page 4–38 and the “Setting reply properties” section on page 4–38.
setReplyTimeToLive procedure

Sets the time to live value (in milliseconds) of the reply messages when the Message Consumer is passed to the requestReply procedure.

Syntax

```plaintext
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 setReplyPersistency procedure, getReplyPersistency function, setReplyPriority procedure, getReplyPriority function, setReplyTimeToLive procedure, getReplyTimeToLive function, setReplyAutoDelete procedure, getReplyAutoDelete function

For more information, see the “Accessing message handler information” section on page 4–38 and the “Setting reply properties” section on page 4–38.

setReplyToDestinationType procedure

Sets the type of the destination specified by the setJMSReplyTo procedure.

Syntax

```plaintext
PROCEDURE setReplyToDestinationType
DEFINE INPUT PARAMETER type AS CHARACTER.
```

Applies to Message objects

Notes

- The type value can be queue or topic.
- If the setReplyToDestinationType procedure 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 to set topic.

See also setJMSReplyTo procedure, getJMSReplyTo function, hasReplyTo function, setReplyToDestinationType procedure, getReplyToDestinationType function

For more information, see the “Accessing message header properties” section on page 4–35.
**setReuseMessage procedure**

Instructs the Message Consumer object not to create a new message for each received message.

**Syntax**

```plaintext
PROCEDURE setReuseMessage.
```

**Applies to** Message Consumer objects

**Notes**

- Calling the `setReuseMessage` procedure 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, `getReuseMessage` function

For more information, see the “Reusing messages” section on page 4–40.

**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**

```plaintext
PROCEDURE setSaxReader.
   INPUT PARAMETER hdl 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.

For more information, see the “XMLMessage” section on page 4–29.
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 the beginSession procedure.

See also setSelectorAtBroker procedure, getSelectorAtBroker function

For more information, see the “Message selectors” section on page 4–12.

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, getConnectionURLs function, setSequential procedure, getSequential function

For more information, see the “Managing fail-over support” section on page 4–7.
**setShort procedure**

Converts data to the JMS short data type in a MapMessage.

**Syntax**

```plaintext
PROCEDURE setShort.
DEFINE INPUT PARAMETER itemName AS CHARACTER.
DEFINE INPUT PARAMETER value AS INTEGER.
```

**Applies to** Message objects

**Note** The server returns a NumberFormatException message for a value overflow.

**See also** createMapMessage procedure, setBoolean procedure, setByte procedure, setBytesFromRaw procedure, setChar procedure, setDate procedure, setDateTime procedure, setDateTime-TZ procedure, setDouble procedure, setFloat procedure, setInt procedure, setLong procedure, setLongString procedure, setShort procedure, setString procedure, getMapNames function, getItemType function, getBytesToRaw function, getChar function, getDate function, getDateTime function, getDateTime-TZ function, getDecimal function, getInt function, getLogical function, getLongString function, getLongStringCP function

For more information, see the “MapMessage” section on page 4–26.

---

**setShortProperty procedure**

Sets a short message property.

**Syntax**

```plaintext
PROCEDURE setShortProperty.
DEFINE INPUT PARAMETER propertyName AS CHARACTER.
DEFINE INPUT PARAMETER propertyValue AS INTEGER.
```

**Applies to** Message objects

**Note** The server returns a NumberFormatException message for a value overflow.

**See also** clearProperties procedure

For more information, see the “Accessing message properties” section on page 4–36.
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 the deleteSession procedure is to execute the ABL statement WAIT-FOR PROCEDURE COMPLETE. The WAIT-FOR statement in the deleteSession procedure can cause an error if your application has an outstanding WAIT-FOR call. You can direct the deleteSession procedure to skip the WAIT-FOR call, by calling the setShutdownWaitFor procedure with a logical “NO” input parameter.

- If you direct the deleteSession procedure 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 the deleteSession procedure to execute the ABL statement WAIT-FOR PROCEDURE COMPLETE.

**See also** deleteSession procedure, getShutdownWaitFor function

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 the beginSession procedure. Otherwise, an error is raised.

**See also** acknowledgeAndForward procedure, setSingleMessageAcknowledgement procedure, getSingleMessageAcknowledgement function, setNoAcknowledge procedure, getNoAcknowledge function

For more information, see the “Message acknowledgement, forwarding, and recovery” section on page 4–44 and the “Single-message acknowledgement” section on page 4–45.
**setString procedure**

Converts data in a MapMessage to the JMS String data type.

**Syntax**

```abl
PROCEDURE setString.
DEFINE INPUT PARAMETER itemName AS CHARACTER.
DEFINE INPUT PARAMETER value AS CHARACTER.
```

**Applies to** Message objects

**See also**
createMapMessage procedure, setBoolean procedure, setByte procedure, setBytesFromRaw procedure, setChar procedure, setDate procedure, setDateTime procedure, setDateTime-TZ procedure, setDouble procedure, setFloat procedure, setInt procedure, setLong procedure, setLongString procedure, setShort procedure, setString procedure, getMapNames function, getItemType function, getBytesToRaw function, getChar function, getDate function, getDateTime function, getDateTime-TZ function, getDecimal function, getInt function, getLogical function, getLongString function, getLongStringCP function

For more information, see the “MapMessage” section on page 4–26.

**setStringProperty procedure**

Sets a String message property.

**Syntax**

```abl
PROCEDURE setStringProperty.
DEFINE INPUT PARAMETER propertyName AS CHARACTER.
DEFINE INPUT PARAMETER propertyValue AS CHARACTER.
```

**Applies to** Message objects

**See also** clearProperties procedure

For more information, see the “Accessing message properties” section on page 4–36.
**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 the setLongText procedure.

**See also**
createTempTableMessage procedure, setTempTable procedure, getTempTable function

For more information, see the “TempTableMessage” section on page 5–21.

**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 calls to append more text.
- As an alternative to concatenating multiple CHARACTER segments with the appendText procedure, use the setLongText procedure.

**See also**
createTextMessage procedure, setText procedure, setLongText procedure, appendText procedure, endOfStream function, getCharCount function, getText function, getTextSegment function, getLongText function, getLongTextCP function

For more information, see the “TextMessage” section on page 4–24.
setTransactedReceive procedure

Makes the session transacted for receiving.

Syntax

```
PROCEDURE setTransactedReceive.
```

Applies to
Session objects

Note
A session is not transacted by default.

See also
setTransactedReceive procedure, getTransactedReceive function, setTransactedSend procedure, getTransactedSend function

For more information, see the “Setting and getting JMS connection and session attributes” section on page 4–7.

setTransactedSend procedure

Makes the session transacted for sending.

Syntax

```
PROCEDURE setTransactedSend.
```

Applies to
Session objects

Note
A session is not transacted by default.

See also
setTransactedReceive procedure, getTransactedReceive function, setTransactedSend procedure, getTransactedSend function

For more information, see the “Setting and getting JMS connection and session attributes” section on page 4–7.
**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**

```plaintext
PROCEDURE setUser.
DEFINE INPUT PARAMETER user AS CHARACTER.
```

**Parameters**

*user*

SonicMQ Broker login.

**Applies to**

Session objects

**Note**

If called, the `setUser procedure` overwrites the default user property set on the OpenEdge Adapter for SonicMQ side.

**See also**

`setUser procedure`, `getUser function`

For more information, see the “Setting and getting JMS connection and session attributes” section on page 4–7.

**setX-Document procedure**

Copies an XML document specified by the handle parameter into the XMLMessage.

**Syntax**

```plaintext
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 the `setLongText procedure` to copy the `longchar` into the `XMLMessage`.
- Other errors include any error generated by `xdoc:SAVE`, any error generated by the `XMLMessage`, or by the `setLongText procedure`.

**See also**

`createXMLMessage procedure`, `setX-Document procedure`, `getX-Document function`, `setSaxReader procedure`, `getSaxWriter function`, `deleteSaxWriter procedure`

For more information, see the “XMLMessage” section on page 4–29.
**startReceiveMessages procedure**

Starts receiving messages after creating a new session or after calling the `stopReceiveMessages` procedure.

**Syntax**

```
PROCEDURE startReceiveMessages.
```

**Applies to** Session objects

**Notes**

- Messages can be sent without calling the `startReceiveMessages` procedure.
- This procedure executes remotely (sends a message to the OpenEdge Adapter for SonicMQ).

**See also** `startReceiveMessages` procedure, `stopReceiveMessages` procedure

For more information, see the “Establishing session control” section on page 4–10.

**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 the `startReceiveMessages` procedure 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 the `stopReceiveMessages` procedure was called might be received by the client after the `stopReceiveMessages` procedure call.
- This procedure executes remotely (sends a message to the OpenEdge Adapter for SonicMQ).

**See also** `startReceiveMessages` procedure, `stopReceiveMessages` procedure

For more information, see the “Establishing session control” section on page 4–10.
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` or an empty string, 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 the `setClientID` procedure 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, publish procedure, subscribe procedure

For more information, see the “Publishing messages to a topic” section on page 4–34, the “Subscribing to a topic” section on page 4–39, the “Durable subscriptions” section on page 4–39, and the “Methods unique to Pub/Sub messaging” section on page 3–8.

For an example, see the “Pub/Sub messaging example” section on page 5–6.

waitForMessages procedure

Waits and processes events as long as the user-defined function is TRUE.

Syntax

```abl
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, deleteConsumer procedure, messageHandler procedure, waitForMessages procedure

For more information see the “Message Consumer objects” section on page 2–5, the “Consuming messages” section on page 4–37, the “Terminating the Message Consumer object” section on page 4–39, and the “Creating a message handler process” section on page 4–38.
writeBoolean procedure

Writes boolean data to the body of a StreamMessage.

Syntax

```
PROCEDURE writeBoolean.
DEFINE INPUT PARAMETER value AS LOGICAL.
```

Applies to
Message objects

Note
An Unknown value (?) is considered FALSE.

See also
createStreamMessage procedure, writeBoolean procedure, writeByte procedure,
writeBytesFromRaw procedure, writeChar procedure, writeDate procedure, writeDateTime procedure, writeDateTime-TZ procedure, writeDouble procedure, writeFloat procedure, writeInt procedure, writeLong procedure, writeLongString procedure, writeShort procedure, writeString procedure, endOfStream function, moveToNext procedure, readBytesToRaw procedure, readChar function, readDate function, readDateTime function, readDateTime-TZ function, readDecimal function, readInt function, readLogical function, readLongString function, readLongStringCP function

For more information, see the “StreamMessage” section on page 4–27.

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, writeBoolean procedure, writeByte procedure,
writeBytesFromRaw procedure, writeChar procedure, writeDate procedure, writeDateTime procedure, writeDateTime-TZ procedure, writeDouble procedure, writeFloat procedure, writeInt procedure, writeLong procedure, writeLongString procedure, writeShort procedure, writeString procedure, endOfStream function, moveToNext procedure, readBytesToRaw procedure, readChar function, readDate function, readDateTime function, readDateTime-TZ function, readDecimal function, readInt function, readLogical function, readLongString function, readLongStringCP function

For more information, see the “StreamMessage” section on page 4–27.
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.

See also createStreamMessage procedure, getContentType, writeBoolean procedure, writeByte procedure, writeBytesFromRaw procedure, writeChar procedure, writeDate procedure, writeDateTime procedure, writeDateTime-TZ procedure, writeDouble procedure, writeFloat procedure, writeInt procedure, writeLong procedure, writeLongString procedure, writeShort procedure, writeString procedure, endOfStream function, moveToNext procedure, readBytesToRaw procedure, readChar function, readDate function, readDateTime function, readDateTime-TZ function, readDecimal function, readInt function, readLogical function, readLongString function, readLongStringCP function

For more information, see the “StreamMessage” section on page 4–27 and the “BytesMessage” section on page 4–28.

writeChar procedure

Writes char data to the body of a StreamMessage.

Syntax

```
PROCEDURE writeChar.
DEFINE INPUT PARAMETER value AS CHARACTER.
```

Applies to Message objects

Note

The number of characters in the char value must be one.

See also createStreamMessage procedure, writeBoolean procedure, writeByte procedure, writeBytesFromRaw procedure, writeChar procedure, writeDate procedure, writeDateTime procedure, writeDateTime-TZ procedure, writeDouble procedure, writeFloat procedure, writeInt procedure, writeLong procedure, writeLongString procedure, writeShort procedure, writeString procedure, endOfStream function, moveToNext procedure, readBytesToRaw procedure, readChar function, readDate function, readDateTime function, readDateTime-TZ function, readDecimal function, readInt function, readLogical function, readLongString function, readLongStringCP function

For more information, see the “StreamMessage” section on page 4–27.
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, writeBoolean procedure, writeByte procedure, writeBytesFromRaw procedure, writeChar procedure, writeDate procedure, writeDateTime procedure, writeDateTime-TZ procedure, writeDouble procedure, writeFloat procedure, writeInt procedure, writeLong procedure, writeLongString procedure, writeShort procedure, writeString procedure, endOfStream function, moveToNext procedure, readBytesToRaw procedure, readChar function, readDate function, readDateTime function, readDateTime-TZ function, readDecimal function, readInt function, readLogical function, readLongString function, readLongStringCP function

For more information, see the “StreamMessage” section on page 4–27.
**writeDateTime procedure**

Writes a date-time value as String data to the body of a StreamMessage.

**Syntax**

```plaintext
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, writeBoolean procedure, writeByte procedure, writeBytesFromRaw procedure, writeChar procedure, writeDate procedure, writeDateTime procedure, writeDateTime-TZ procedure, writeDouble procedure, writeFloat procedure, writeInt procedure, writeLong procedure, writeLongString procedure, writeShort procedure, writeString procedure, endOfStream function, moveToNext procedure, readBytesToRaw procedure, readChar function, readDate function, readDateTime function, readDateTime-TZ function, readDecimal function, readInt function, readLogical function, readLongString function, readLongStringCP function

For more information, see the “StreamMessage” section on page 4–27.
writeDateTime-TZ procedure

Writes a date-time value, including time zone information, as String data to the body of a StreamMessage.

Syntax

```
PROCEDURE writeDateTime-TZ.
DEFINE INPUT PARAMETER value AS DATETIME-TZ.
```

 Applies to Message objects

 Note An error is returned if the Unknown value (?) is specified.

See also createStreamMessage procedure, writeBoolean procedure, writeByte procedure, writeBytesFromRaw procedure, writeChar procedure, writeDate procedure, writeDateTime procedure, writeDateTime-TZ procedure, writeDouble procedure, writeFloat procedure, writeInt procedure, writeLong procedure, writeLongString procedure, writeShort procedure, writeString procedure, endOfStream function, moveToNext procedure, readBytesToRaw procedure, readChar function, readDate function, readDateTime function, readDateTime-TZ function, readDecimal function, readInt function, readLogical function, readLongString function, readLongStringCP function

For more information, see the “StreamMessage” section on page 4–27.

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, writeBoolean procedure, writeByte procedure, writeBytesFromRaw procedure, writeChar procedure, writeDate procedure, writeDateTime procedure, writeDateTime-TZ procedure, writeDouble procedure, writeFloat procedure, writeInt procedure, writeLong procedure, writeLongString procedure, writeShort procedure, writeString procedure, endOfStream function, moveToNext procedure, readBytesToRaw procedure, readChar function, readDate function, readDateTime function, readDateTime-TZ function, readDecimal function, readInt function, readLogical function, readLongString function, readLongStringCP function

For more information, see the “StreamMessage” section on page 4–27.
writeFloat procedure

Writes float data to the body of a StreamMessage.

Syntax

PROCEDURE writeFloat.
DEFINE INPUT PARAMETER value AS DECIMAL.

Applies to Message objects

Note The fractional part of the DECIMAL value is truncated.

See also createStreamMessage procedure, writeBoolean procedure, writeByte procedure, writeBytesFromRaw procedure, writeChar procedure, writeDate procedure, writeDateTime procedure, writeDateTime-TZ procedure, writeDouble procedure, writeFloat procedure, writeInt procedure, writeLong procedure, writeLongString procedure, writeShort procedure, writeString procedure, endOfStream function, moveToNext procedure, readBytesToRaw procedure, readChar function, readDate function, readDateTime function, readDateTime-TZ function, readDecimal function, readInt function, readLogical function, readLongString function, readLongStringCP function

For more information, see the “StreamMessage” section on page 4–27.

writeInt procedure

Writes int data to the body of a StreamMessage.

Syntax

PROCEDURE writeInt.
DEFINE INPUT PARAMETER value AS INTEGER.

Applies to Message objects

See also createStreamMessage procedure, writeBoolean procedure, writeByte procedure, writeBytesFromRaw procedure, writeChar procedure, writeDate procedure, writeDateTime procedure, writeDateTime-TZ procedure, writeDouble procedure, writeFloat procedure, writeInt64 procedure, writeLong procedure, writeLongString procedure, writeShort procedure, writeString procedure, endOfStream function, moveToNext procedure, readBytesToRaw procedure, readChar function, readDate function, readDateTime function, readDateTime-TZ function, readDecimal function, readInt function, readInt64 function, readLogical function, readLongString function, readLongStringCP function

For more information, see the “StreamMessage” section on page 4–27.
writeInt64 procedure

Writes INT64 data to the body of a StreamMessage.

**Syntax**

```
PROCEDURE writeInt64.
DEFINE INPUT PARAMETER value AS INT64.
```

**Applies to** Message objects

**Note** The Unknown value (?) is allowed.

**See also**
createStreamMessage procedure, writeBoolean procedure, writeByte procedure, writeBytesFromRaw procedure, writeChar procedure, writeDate procedure, writeDateTime procedure, writeDateTime-TZ procedure, writeDouble procedure, writeFloat procedure, writeInt procedure, writeLong procedure, writeLongString procedure, writeShort procedure, writeString procedure, endOfStream function, moveToNext procedure, readBytesToRaw procedure, readChar function, readDate function, readDateTime function, readDateTime-TZ function, readDecimal function, readInt function, readInt64 function, readLogical function, readLongString function, readLongStringCP function

For more information, see the “StreamMessage” section on page 4–27.

writeLong procedure

Writes long data to the body of a StreamMessage.

**Syntax**

```
PROCEDURE writeLong.
DEFINE INPUT PARAMETER value AS DECIMAL.
```

**Applies to** Message objects

**Note** The fractional part of the DECIMAL value is truncated.

**See also**
createStreamMessage procedure, writeBoolean procedure, writeByte procedure, writeBytesFromRaw procedure, writeChar procedure, writeDate procedure, writeDateTime procedure, writeDateTime-TZ procedure, writeDouble procedure, writeFloat procedure, writeInt procedure, writeLong procedure, writeLongString procedure, writeShort procedure, writeString procedure, endOfStream function, moveToNext procedure, readBytesToRaw procedure, readChar function, readDate function, readDateTime function, readDateTime-TZ function, readDecimal function, readInt function, readLogical function, readLongString function, readLongStringCP function

For more information, see the “StreamMessage” section on page 4–27.
writeLongString procedure

Writes String data of any length to the body of a StreamMessage.

Syntax

```
PROCEDURE writeLongString.
DEFINE INPUT PARAMETER value AS LONGCHAR.
```

Applies to

Message objects

See also

createStreamMessage procedure, writeBoolean procedure, writeByte procedure,
writeBytesFromRaw procedure, writeChar procedure, writeDate procedure, writeDateTime
procedure, writeDateTime-TZ procedure, writeDouble procedure, writeFloat procedure,
writeInt procedure, writeLong procedure, writeLongString procedure, writeShort procedure,
writeString procedure, endOfStream function, moveToNext procedure, readBytesToRaw
procedure, readChar function, readDate function, readDateTime function, readDateTime-TZ
function, readDecimal function, readInt function, readLogical function, readLongString
function, readLongStringCP function

For more information, see the “StreamMessage” section on page 4–27.

writeShort procedure

Writes short data to the body of a StreamMessage.

Syntax

```
PROCEDURE writeShort.
DEFINE INPUT PARAMETER value AS INTEGER.
```

Applies to

Message objects

Note

The server returns a NumberFormatException message for a value overflow.

See also

createStreamMessage procedure, writeBoolean procedure, writeByte procedure,
writeBytesFromRaw procedure, writeChar procedure, writeDate procedure, writeDateTime
procedure, writeDateTime-TZ procedure, writeDouble procedure, writeFloat procedure,
writeInt procedure, writeLong procedure, writeLongString procedure, writeShort procedure,
writeString procedure, endOfStream function, moveToNext procedure, readBytesToRaw
procedure, readChar function, readDate function, readDateTime function, readDateTime-TZ
function, readDecimal function, readInt function, readLogical function, readLongString
function, readLongStringCP function

For more information, see the “StreamMessage” section on page 4–27.
writeString procedure

Writes String data to the body of a StreamMessage.

Syntax

```sql
PROCEDURE writeString.
DEFINE INPUT PARAMETER value AS CHARACTER.
```

Applies to Message objects

See also createStreamMessage procedure, writeBoolean procedure, writeByte procedure, writeBytesFromRaw procedure, writeChar procedure, writeDate procedure, writeDateTime procedure, writeDateTime-TZ procedure, writeDouble procedure, writeFloat procedure, writeInt procedure, writeLong procedure, writeLongString procedure, writeShort procedure, writeString procedure, endOfStream function, moveToNext procedure, readBytesToRaw procedure, readChar function, readDate function, readDateTime function, readDateTime-TZ function, readDecimal function, readInt function, readLogical function, readLongString function, readLongStringCP function

For more information, see the “StreamMessage” section on page 4–27.
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.

This appendix includes the following sections:

- Pub/Sub messaging examples
- PTP messaging examples
- MultiPartMessage example
- Gateway sample application

For information on locating the examples, see the “OpenEdge messages” section on page Preface–7.

For an alphabetical API reference, see Appendix A, “ABL–JMS API Reference.”
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
- Publishing with message properties and subscribing selectively
- Publishing with a reply handle, subscribing, and receiving an automatic reply
- Publishing, receiving, and processing a StreamMessage
- Publishing, receiving, and parsing an XMLMessage
- Publishing, subscribing, and receiving an XML document in a BytesMessage
- Publishing, subscribing, and receiving the customer table in a StreamMessage
- Publishing and receiving a group of messages in a transaction
- Installing an error handler to handle an asynchronous error
- Installing an error handler for synchronous errors
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:

```
/* 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:

```pascal
example1.p

/* 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. 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 call).

**To run example3.p and example4.p:**

1. Run example4.p first so the subscriber is running before you publish, as shown:

```pascal
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).
```

2. Run example1.p to publish the TextMessage to a topic, as shown:
2. Run example3.p, as shown:

example3.p

```plaintext
/* Publishes a Text message with properties. */
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.").

/* Set the "FROM:" and the "TO:" properties */
RUN setStringProperty IN hMessage ("FROM", "Super Golf Center").
RUN setStringProperty IN hMessage ("TO", "Sub Par Golf").

/* Publish the message on to the golf topic */
RUN publish IN hPubSubSession ("GolfTopic", hMessage, ?, ?, ?).
```

example4.p

```plaintext
/* Publishes a Text message with properties. */
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.").

/* Set the "FROM:" and the "TO:" properties */
RUN setStringProperty IN hMessage ("FROM", "Super Golf Center").
RUN setStringProperty IN hMessage ("TO", "Sub Par Golf").

/* Publish the message on to the golf topic */
RUN publish IN hPubSubSession ("GolfTopic", hMessage, ?, ?, ?).
```

example4.p (2 of 2)

```plaintext
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.
```
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 directly. It can only reply to messages that have the JMSReplyTo header field. The procedure example7.p publishes using the requestReply procedure 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:

```plaintext
/* 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
(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 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.
```
2. Run `example6.p` to subscribe with explicit reply by calling the `publish` procedure directly, as shown:

```abl
/* 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.
```

```abl
/* 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.
```

```abl
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.
```
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:

```plaintext
/* 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.
```
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 Sportsdatabase.
2. Run example9.p to receive the StreamMessage containing the customer names and numbers, as shown:

example9.p

```assembly
/* 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.
```

example7.p

```assembly
/* Wait forever to receive messages since "u1" is never applied. */
WAIT-FOR u1 OF THIS-PROCEDURE.

PROCEDURE golfHandler:
    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 – we are not sending a response. */
    DISPLAY "reply text: "
    DYNAMIC-FUNCTION('getText':U IN hReply) FORMAT "X(30)".
    RUN deleteMessage IN hReply.
END PROCEDURE.
```
3. Run example8.p to publish a StreamMessage containing customer names and numbers, as shown:

```tpascal
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.

  RUN deleteMessage IN hMessage.
  APPLY "U1" TO THIS-PROCEDURE.
END PROCEDURE.
```

```tpascal
/* 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, ?, ?, ?).
```
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
/* 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.
```
2. Run example10.p to create and publish an XML Message with the data of 100 people, as shown:

```plaintext
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" encoding="ISO8859-1"?>').
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:

```
/* 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.
```
2. Run example12.p to publish the BytesMessage containing an XML document, as shown:

```plaintext
/* Publishes an XML document in a Bytes message */
DEFINE VARIABLE hDoc AS HANDLE NO-UNDO.
DEFINE VARIABLE hPubSubSession AS HANDLE NO-UNDO.
DEFINE VARIABLE hMesg AS HANDLE NO-UNDO.
DEFINE VARIABLE mValue AS MEMPTR NO-UNDO.

RUN jms/pubsubsession.p PERSISTENT SET hPubSubSession
   ("-H localhost -S 5162 ").
RUN setBrokerURL IN hPubSubSession ("localhost:2506").
RUN beginSession IN hPubSubSession.
RUN createBytesMessage IN hPubSubSession (OUTPUT hMesg).

CREATE X-DOCUMENT hDoc.
   hDoc:SAVE("memptr", mValue).

RUN setMemptr IN hMesg(mValue, ?, ?).
RUN publish IN hPubSubSession ("xmlTopic", hMesg, ?, ?, ?).

SET-SIZE(mValue) = 0.
RUN deleteMessage IN hMesg.
RUN deleteSession IN hPubSubSession.

/* The personal.xml document:
<?xml version="1.0" encoding='UTF-8' ?>
<personnel>
   <person id="Irving.Nigrini">
      <name><family>Nigrini</family> <given>Irving</given></name>
      <email>i.nigrini@subpargolf.com</email>
      <link manager="Thomas.Roy"/>
   </person>
   <person id="Jules.Nigrini">
      <name><family>Nigrini</family> <given>Jules</given></name>
      <email>j.nigrini@subpargolf.com</email>
      <link manager="Thomas.Roy"/>
   </person>
</personnel>*/
```
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 the “Publishing, subscribing, and receiving an XML document in a BytesMessage” section on page B–13.)

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 example15.p 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:

```abl
/* 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
(TTHIS-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.
```

```
example15.p

```plaintext
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.

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.
```

3. Run example14.p, as shown:

example14.p

```plaintext
/* 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
/* 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.
```
2. Run `example22.p` to subscribe and receive messages from `example22.p` in a single transaction, as shown:

```plaintext
/* 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.
```

`example23.p` (2 of 2)

```plaintext
/* 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:

```
/* 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 "Disocnnectiong 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.
```

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

```/* 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.
```
Messaging Examples

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
- Achieving scalable server architecture with PTP queuing

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:

example19.p

```plaintext
/* 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
(TTHIS-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 example20.p 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:

```plaintext
example20.p

/* 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:

example21.p

/* 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
(THE-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", THE-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.

DEFINE VARIABLE replyText AS CHARACTER 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.
RUN hReply = replyMessage.
RUN setText IN hReply (replyText).

FUNCTION inWait RETURNS LOGICAL:
RETURN true.

END.

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 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 Appendix A, “ABL–JMS API Reference.”)

The following sections describe:

- Application files
- Running the sample application

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.

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.

4. When an application modifies a customer record, it publishes the new record through an ABL `PUBLISH CustUpdate` call.
5. 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.
6. 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.
7. 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:

**appDriver.p**

```plaintext
/* 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**

```plaintext
/* 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/pubsessession.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".
```
The procedure `customers.p` updates customer records from a specified country while keeping the other records identical to the master copy, as shown:

```plaintext
/* 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 custUpd FOR ttCustomer.

/* Getting a handle to a dynamic buffer. */
hTTCust = TEMP-TABLE ttCustomer:HANDLE.
hBuffer = hTTCust:DEFAULT-BUFFER-HANDLE.
```
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:

```plaintext
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.

```plaintext
/* 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.

/* Updates a customer record from a RAW value. */
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.
```
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. Creating and testing the process is detailed in the following sections:

- 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

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.
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:

```abl
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:

![Add Annotation dialog box](image)

Select **ESB Annotation - Main** annotation from the drop-down in the **Select annotation or enter annotation in text editor** section, and then and select your file.

Click **Finish** to add the annotation. The annotated source follows:

```openapi
@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 appears:
4. Name your process and click Finish. An empty process appears:
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:

![Import dialog box]

Click **Finish** to import your native invocation file.
6. From the Pallele tab, select OpenEdge Native Services. Add the template to the process and add a name, as shown:

![Diagram showing the process with a Pallele tab and added OpenEdge Native Services template.]

7. Add the native invocation file to the process by dragging the file from the Navigator window onto the process.
8. If you wish to examine the process details, right click on the service and select **Open**. The details appear, as shown:

![Service details](image)

9. 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:

![Create/Edit Scenario dialog box](image)

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:

![Output Window](image)

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

A

ABL
  Native invocation drag and drop 6–16
  object model, ABL–JMS 2–4
  publish-and-subscribe mechanism 2–9
  transactions 4–44

ABL transactions 4–44

ABL-JMS object model 2–4
  Message Consumer objects 2–5
  Message objects 2–5
  Session objects 2–4

Accessing message handler 4–38
  methods 4–38

Accessing message header properties 4–35

Accessing message properties 4–36

Accessing messages 4–24

acknowledgeAndForward procedure A–10

Acknowledgement of messages
  automatic 4–44
  commitReceive method A–16
  lazy 4–44
  preventing 4–44

addBytesPart procedure A–11

addMessagePart procedure A–11

addTextPart procedure A–12

Administered objects, JMS
  and OpenEdge Adapter for SonicMQ 4–20

AdminServerPlugins.properties file
  jvmArgs property 4–31

Appending text 4–25

appendText procedure A–12

Application files gateway sample B–27

AppServer
  session models 6–31

Asynchronous
  ASYNC completion procedures 4–41
  call events 4–41
  conditions 4–46
  error handling 4–31, 4–48
  error messages 2–5, 4–37
  error reporting 4–46
  reply 4–12, A–86

Asynchronous error example B–20

Automatic message acknowledgment 4–44

B

beginSession procedure A–13

boolean data type
  mapped to ABL 4–32, 4–33

Broker URL property A–92

browseQueue procedure A–14

Browsing
  messages on queue 4–39
byte array data type
  mapped to ABL 4–32, 4–33
byte data type
  mapped to ABL 4–32, 4–33
Bytes correlation ID, JMS
  getting A–46
BytesMessage 4–28
  example B–13
  handling 4–28
  parse B–13
  Process BytesMessage B–13
  publish and subscribe B–13
  receive B–13
  XML document B–13

C
cancelDurableSubscription procedure A–15
char data type
  mapped to ABL 4–32, 4–33
CHARACTER data type 4–32, 4–33, B–15
Character limit
  text 4–25
Characters
  code-page encoding 4–30
clearBody 4–34
clearBody procedure A–15
Clearing message body 4–34
Clearing message properties 4–34, 4–36
Clearing messages 4–34
clearProperties 4–34
clearProperties procedure A–16
Client ID
  identifier A–135
  setting A–94
Client persistence 4–8, 4–9
Client persistence example 5–12
Code page example B–15
commitReceive procedure A–16
commitSend procedure A–16
Comparing PTP and Pub/Sub 2–10
Concatenating text 4–25

Condition handling 4–46
Connection attributes 4–7
Connection options 4–4
Connection parameter examples 4–6
Connections
  attributes 4–7, A–34
  failures 4–48
Consuming 4–43
Consuming messages 4–37
  creating 4–37
  handler 4–38
  Message acknowledgement 4–44
    automatic 4–44
    preventing 4–44
    single 4–45
  Message recovery 4–44, 4–45
  processing states 4–41
  reception issues 4–40
  Reply mechanisms 4–41
  Reply properties 4–38
  scope 4–40
  synchronous reception 4–41
  Transacted sessions 4–43
    illegal calls 4–43
    receiving 4–43
    recover 4–43
    sending 4–43
Correlation ID
  getting A–45
–cpinternal startup parameter B–15, B–16
createBytesMessage procedure A–17
createChangeStateConsumer procedure A–18
createDataSetMessage procedure A–18
createHeaderMessage procedure A–19
createMapMessage procedure A–19
createMessageConsumer procedure A–20
createMultipartMessage procedure A–20
createRejectedMessageConsumer procedure A–21
createStreamMessage procedure A–21
createTemporaryQueue procedure A–22
createTemporaryTopic procedure A–22
createTempTableMessage procedure A–23
createTextMessage procedure A–23
createXMLMessage procedure A–23

Creating
Message Consumers 2–5, A–20
message headers 4–35
messages 4–33
publishers 4–34
Session objects 2–4, 4–2, 5–2, 5–6, 5–11
subscribers 4–39
TextMessage 4–25
XMLMessage 4–30

Creating Message Consumer 4–37
Creating message handler 4–38
Creating messages 4–24

D

Data extraction methods
Java and ABL 4–32

Data types
CHARACTER 4–32, B–15
extraction in Java and ABL 4–32, 4–33
in StreamMessage 4–27
INTEGER 4–32
LOGICAL 4–32
MEMPTR 4–28, 4–29
RAW 4–28, 4–29
storage in Java and ABL 4–32

DataSetMessage 4–30, 5–24
handling 4–30

DATE data type 4–32

Date values
in map messages A–45
reading from stream messages A–76
See also DATE, DATETIME, and DATETIME-TZ data types

DATETIME data type 4–32
DATETIME-TZ data type 4–32
DECIMAL data type 4–32
deleteConsumer procedure A–24
deleteMessage procedure A–24
deleteSaxWriter procedure A–24
deleteSession procedure A–25
deleteTemporaryQueue procedure A–26
deleteTemporaryTopic procedure A–26

Deleting
data from message body 4–34
durable subscriptions 4–40
Message Consumer 4–39
Message Consumer objects 4–40
objects 3–4, 3–7
replies 4–42
sessions 4–3
temporary destination 4–12, A–86

Deleting Message Consumer 4–39
Deleting messages 4–35
Delivery parameters 4–35

Deployment
serialized connection objects 4–13, 4–15
Destination 4–12, A–86
temporary 4–12, A–86
-DirectConnect connection parameter 4–4

Discardable messages 4–11
domain unification 5–10
double data type
mapped to ABL 4–32, 4–33
Durable subscriptions 3–7, 4–39, A–94
deleting 4–40

E

Encoding XML example B–15
endOfStream function A–27

Enhanced XML support 5–14

Error
handling 1–5

Error handler example B–20, B–21

Error handling
SonicMQ Adapter B–20, B–21

Errors
handling 4–7, 4–46
synchronously reported 4–46

ESBOE file 6–2
ESBOEGEN utility 6–24

Example BytesMessage B–13
Example gateway sample B–27, B–30
application files B–27
running B–30
Example Multipart message B–26
Example StreamMessage B–16
Example TextMessage B–18, B–20, B–21, B–22, B–23
  asynchronous error B–20
  error handler B–20, B–21
  synchronous error B–21
Exchanging messages 2–4
Expiration time
  of messages, getting A–47

F
Fault tolerance 4–9, 4–10
  Replicated brokers 4–10
Fault tolerance example 5–28
float data type
  mapped to ABL 4–32, 4–33
4GL
  Publish and Subscribe mechanism B–27
Functions, ABL
  getCharCount 4–25
  getJMSReplyTo 4–42
  getLongText 4–25
  getMemptr 4–28
  getPropertyNames 4–47
  getReplyToDestinationType 4–42
  getText 4–25
  getTextSegment 4–25
  moveToNext A–76
  readBytesToRaw 4–28
  readChar 4–32

G
Gateway model
  ABL publish and subscribe 2–9
Gateway sample application B–27, B–30
  application files B–27
  running B–30
Gateway sample application files B–27
Gateway sample running B–30
getAdapterService function A–28
getApplicationContext function A–28
getBrokerURL function A–28
getBytesCount function A–29
getBytesPartByIndex function A–30
getBytesToRaw function A–30
getChar function A–31
getCharCount function A–31
getCharProperty function A–31
getClientID function A–32
getClientPersistence function A–32
getClientTransactionBufferSize function A–33
getConnectionID function A–34
getConnectionMetaData function A–34
getConnectionURLs function A–34
getDataSet function A–35
getDate function A–36
getPropertyNames 4–47
getDateTime function A–37
getDateTimeProperty function A–37
getDateTimeTZ function A–38
getDateTimeTZProperty function A–38
getDecimal function A–39
getDecimalProperty function A–39
getDefaultPersistency function A–39
getDefaultPriority function A–40
getDefaultTimeToLive function A–40
getDestinationName function A–40
getFaultTolerant function A–41
getFaultTolerantReconnectTimeout function A–41
getFlowDisk function A–42
getInitialConnectionTimeout function A–44
getInt function A–42
getInt64 function A–43
getIntProperty function A–42
getItemType function A–45
getJMSCorrelationID function A–45
getJMSCorrelationIDAsBytes function A–46
getJMSDeliveryMode function A–46
getJMSDestination function A–47
getJMSExpiration function A–47
getJMSMessageID function A–47
getJMSPriority function A–48
getJMSRedelivered function A–48
getJMSReplyTo function A–49
getJMSServerName function A–49
getJMSTimestamp function A–50
getJMSType function A–50
getLoadBalancing function A–50
getLocalStoreDirectory function A–51
getLocalStoreSize function A–51
getLocalStoreWait procedure A–52
getLogical function A–52
getLogicalProperty function A–53
getLongString function A–53
getLongStringCP function A–54
getLongText function A–54
getLongTextCP function A–55
getMapNames function A–55
getMemptr function A–56
getMessagePartByID function A–57
getMessagePartByIndex function A–57
getMessageType function A–58
getPartCount function A–58
getPassword function A–59
getProcHandle function A–59
getProcName function A–59
getPropertyNames function A–60
getPropertyType function A–60
getReconnectInterval function A–61
getReconnectTimeout function A–61
getReplyAutoDelete function A–62
getReplyPersistency function A–62
getReplyPriority function A–63
getReplyTimeToLive function A–63
getReplyToDestinationType function A–64
getReuseMessage function A–64
getSaxWriter function A–65
getSelectorAtBroker function A–66
getSequential function A–66
getSession function A–65
getSingleMessageAcknowledgement function A–67
getTempTable function A–67
getText function A–68
getTextPartByID function A–68
gTextPartByIndex function A–69
gTextSegment function A–69
gTransactedReceive function A–70
gTransactedSend function A–70
gUser function A–70
gX-Document function A–71

H
-H connection parameter 4–4
hasReplyTo function A–71
HeaderText 4–25

I
I/O blocking state 4–41
illegal calls 4–43
inErrorHandling function A–71
inMessageHandling function A–72
inQueueBrowsing function A–72
inReplyHandling function A–72
int data type mapped to ABL 4–32, 4–33
INDEX

INDEX–6

INTERRUPTS 4–49
ISFAULTTOLERANT FUNCTION A–73
ISMESSAGEPART FUNCTION A–73

J

Java Message Service
broker
implementation, specifying A–113
Java Message Service (JMS)
See SonicMQ Adapter
administered objects 4–20
bytes correlation ID A–46
connection attributes 4–7, A–34
described 1–2
derror handling examples B–21
JNDI directory 4–20
maximum number of messages 4–11
message exchange steps 2–4
Multipart message example B–26
publish (procedure) B–6, B–7
queues, creating B–22
queues, receiving on B–22
StreamMessage B–9, B–16
transactions 4–44
XMLMessage B–11

Java Object messages 4–31

JMS. See Java Message Service

JMS and ABL data types
storing data 4–32
JMS message exchange 2–4

JMS messaging 3–1, 4–1
PTP 3–1, 4–1
JMS messaging
Pub/Sub 3–1, 4–1

JMS session 4–3

JMS session domain unification 5–10

JMS–MAXIMUM–MESSAGES 4–11

JMS–MAXIMUM–MESSAGES global
variable A–74

JMSReplyTo header field B–6

JMSSESSION 5–10

JMSSESSION.P 2–4, A–2

JVMARGS property 4–31
in AdminServerPlugins.properties file
4–31

L

Lazy acknowledgment 4–44

Life cycles
creating messages 4–23
Message Consumer object 4–37
receiving messages 4–23

Limits
RAW data 4–28, 4–29
Load balancing 4–8

LOGICAL data type 4–32

long data type
mapped to ABL 4–32

LONGCHAR data type 4–32, A–12, A–69

long data type
mapped to ABL 4–33

M

Managing connections 4–2
examples 4–6
option 4–4
setting and getting 4–7
starting 4–8

Managing session
Client persistence 4–9
delivery parameters 4–11

Managing sessions 4–2
Client persistence 4–8
Creating multiple sessions 4–3
Creating session 4–3
Deleting sessions 4–3
delivery parameters 4–11
discardable 4–11
Fault tolerance 4–9, 4–10
Load balancing 4–8
Message selectors 4–12
number of messages 4–11
Replicated brokers 4–10
request and reply 4–12
Serialized connection objects 4–13
setting 4–10
setting methods 4–10
Storing undeliverable messages 4–8
MapMessage 4–26
   handling 4–26
Mapping
   ABL data types to JMS data types 4–32, 4–33
Maximum number of messages 4–11
MEMPTR data type 4–32, 4–33
MEMPTR variable 4–28
Message
   properties 4–36
Message acknowledgement
   automatic 4–44
   preventing 4–44
   single 4–45
Message acknowledgement and recovery 4–44
Message Consumer
   deleting 4–39
Message Consumer objects 2–5
   cancellation of reception 4–40
Message Consumer scope 4–40
Message delivery parameters 4–11
   methods 4–11
Message handler 4–38
   accessing 4–38
   methods 4–38
Message handlers
   errors and conditions 4–49
   getting properties 4–38
Message headers
   accessing 4–35, 4–36
   properties 4–35
Message life cycle 4–23
Message processing states 4–41
Message properties
   accessing 4–36
   clearing 4–36
Message queues. See Queues.
Message reception issues 4–40
   starting and stopping 4–40
Message recovery 4–45
Message selectors 4–12
Message size limits 4–31
Message types 2–11
   BytesMessage 2–11
   HeaderMessage 2–11
   MapMessage 2–11
   MultiPartMessage 2–11
   StreamMessage 2–11
   TextMessage 2–11
   XMLMessage 2–11
messageHandler procedure A–75
Messages 2–11, 4–23
   accessing 4–24
   acknowledgement 4–44, A–16
   acknowledgement, automatic 4–44
   browsing on queue 4–39
   BytesMessage 2–11, 4–28
      handling 4–28
   clearBody 4–34
   clearBody and clearProperties 4–34
   clearing 4–34
   clearProperties 4–34
   Consuming 4–37
      creating 4–37
      handler 4–38
      creating 4–24
      data 4–32
   DataSetMessage 4–30
      handling 4–30
   deleting 3–4, 3–7, 4–35
   discardable 4–11
   durable subscriptions 4–39
   expiration time, getting A–47
   header properties 4–35
   headerMessage 2–11, 4–25
   life cycle 4–23
   Load balancing 4–8
   MapMessage 2–11, 4–26
      handling 4–26
   maximum number 4–11
   modes 4–33
   MultiPartMessage 2–11
   MultiPartMessage 4–28
      handling 4–29
   parts 2–11
   persistency values A–103
   populating 4–24
   preventing acknowledgement 4–44
   priority values A–48, A–103
   priority, default A–103
   processes A–136
   properties 4–36
   publishing 4–34
   queue browsing 4–39
   queues 4–34, 4–38
   read only 4–33
   receiving 4–38
   reception issues 4–40
   reception, synchronous 4–41
   recovery 4–45
   redelivery A–87
   Reply properties 4–38
request and reply 4–12
reuse 4–40
sending 4–34
setting and getting delivery parameters 4–11
setting maximum messages 4–11
setting maximum number 4–11
size limits 4–31
starting and stopping reception 4–40
Storing and extracting data 4–32
StreamMessage 2–11, 4–27
handling 4–27
subscribing 4–39
TempTableMessage 4–31
handling 4–31
TextMessage 2–11
handling 4–24
time to live value A–77, A–89
topics 4–34, 4–39
write only 4–33
XMLMessage 2–11, 4–29
Messaging 4–23
Client persistence 4–8, 4–9
Fault tolerance 4–9, 4–10
Message selectors 4–12
Replicated brokers 4–10
Storing undeliverable messages 4–8
Messaging example
PTP 5–2
Pub/Sub 5–6
scalable server architecture B–23
Messaging examples B–1 to B–30
PTP messaging examples B–22 to B–25
Pub/Sub messaging examples B–2 to B–22
Messaging introduction 2–6
Comparison 2–10
Message types 2–11
PTP 2–6
PTP features 2–7
PTP queuing 2–7
PTP send and receive 2–6
PTP tasks 2–6
Pub/Sub features 2–8
Pub/Sub integration 2–9
Pub/Sub messaging 2–8
Pub/Sub send and receive 2–8
Pub/Sub tasks 2–8
Messaging, See Java Message Service;
Sonic ESB Adapter; SonicMQ Adapter
Methods in Message Consumer objects A–6
Methods in the Message objects A–7
Methods in the Session objects A–4
Methods, 4GL
X–DOC:SAVE B–15
Methods, ABL
appendText 4–25
beginSession 4–3, 4–48
browseQueue 4–39
cancelDurableSubscription 4–40
clearBody 4–34
clearProperties 4–34
commitReceive 4–43
commitSend 4–43
createMessageConsumer 3–3, 3–6
deleteMessage 4–23
deleteSession 3–2, 3–5, 4–3, 4–41
getApplicationContext 4–49
publish 4–42
receiveFromQueue 4–38
recover 4–43, 4–45
requestReply 4–12, 4–33, 4–41, 4–42
reset 4–33
rollbackReceive 4–43
rollbackSend 4–43
sendToQueue 4–42
setAdapterService A–90
setApplicationContext 4–49
setClientID A–135
setJMSReplyTo 4–42
setLongText 4–25
setMemptr 4–28
setNoAcknowledge 4–43, 4–44, 4–49
setNoErrorDisplay 4–47
setPingInterval 4–48
setReplyAutoDelete 4–42
setReplyToDestinationType 4–42
setText 4–25
setTransactedReceive 4–43
setTransactedSend 4–43
stopReceiveMessages 4–40
subscribe 4–39
waitForMessages 2–5, 3–6, 4–41, 4–49
writeBytesFromRaw 4–28
writeShort 4–32
Modes
read-only 4–33
write-only 4–25, 4–33, A–15
moveToNext procedure A–76
Multipart message example B–26
MultipartMessage 4–28
handling 4–29
N
Native Invocation file 6–2
Native Invocation Methodology 6–2
-nohostverify connection parameter 4–5

NumberFormatException error message A–93

O

Object Messages 4–31

OpenEdge Adapter for Sonic ESB
Annotating with OpenEdge Architect 6–7
Annotation syntax examples 6–5
ESBOEGEN utility 6–24
Generating ESBOE files 6–9
Importing native invocation files into Sonic Workbench 6–10
Introduction 1–5
Native Invocation annotation syntax 6–4
Native Invocation file 6–2
Native Invocation file parameter mapping 6–13
Native Invocation methodology 6–2
OpenEdge Architect configurations 6–25
Sample process C–1
Using a session-managed AppServer 6–15
Using ProxyGen to generate ESBOE files 6–9
Web Services Invocation methodology 6–28

OpenEdge Adapter for SonicMQ
ABL-JMS object model 2–4
Accessing 2–2
Exchanging messages 2–4
Messaging 2–6
Understanding 2–3
Unified domain 2–3

P

Parse
XMLMessage B–11
Parse BytesMessage B–13
Parse XMLMessage B–11
parser, XML B–15

Password
broker login A–120
-pf connection parameter 4–6

Ping interval
SonicMQ broker A–121
Populating messages 4–24
Preventing message acknowledgment 4–44

Priority values, getting A–48

Procedure files
jmsession.p 2–4
ptpsession.p 2–4
pubsubsession.p 2–4

Process StreamMessage B–9, B–16

Process XMLMessage B–11

Processing messages 4–39

Programming errors 4–46

Programming example
Code page B–15

Programming examples 5–10
Client persistence 5–12
ClientConnect 5–11
DataSetMessage 5–24
Encoding XML B–15
Enhanced XML support 5–14
Fault tolerance 5–28
ServerConnect 5–11, 5–12
SMQConnect 5–11, 5–12
TempTableMessage 5–21

Properties selector, JMS 4–38, 4–39

Provider attributes A–34

PTP messaging 3–2
Connect broker 3–2
Create session 3–2
Deleting 3–4
Receiving from queue 3–3
receiving reply 3–3
Sending to queue 3–2
Temporary queues 3–4
Unique methods 3–4

PTP messaging example 5–2
Connect broker 5–2
Create Message Consumer 5–3
Create session 5–2
Delete messages 5–4
Prepare receive 5–3
Receive messages 5–3
Send messages 5–3
Summary 5–4

PTP messaging examples B–22 to B–25
ptpsession.p 2–4, 5–2, 5–11, A–2

Pub/Sub messaging 3–5
Connect broker 3–5
Create session 3–5
Deleting objects 3–7
Publish to topic 3–5
Receiving messages 3–6
Receiving reply 3–6
Index

Subscribe to topic 3–6
Temporary topic 3–7
Unique methods 3–8

Pub/Sub messaging example 5–6
Connect broker 5–6
Consume messages 5–7
Create Message Subscriber 5–7
Create session 5–6
Delete messages 5–8
Publish topic 5–7
Subscribe topic 5–7
Summary 5–8

Pub/Sub messaging examples B–2 to B–22

Publish and receive
TextMessage B–18

Publish and subscribe
BytesMessage B–13
StreamMessage B–9, B–16
TextMessage B–3, B–4, B–6
XMLMessage B–11
publish procedure A–77

Publishing and receiving TextMessage B–18

Publishing and subscribing with a
BytesMessage B–13

Publishing and subscribing with a
StreamMessage B–9, B–16

Publishing and subscribing with a
TextMessage B–3, B–4, B–6

Publishing and subscribing with a
XMLMessage B–11

Publishing messages to topic 4–34
Publishing with message properties and
subscribing selectively B–4

Publishing, JMS. See Java Message Service
pubsubsession.p 2–4, 5–6, A–3

R
Read only modes 4–33
readBytesToRaw procedure A–78
readChar function A–78
readDate function A–79
readDateTime function A–79
readDateTime-TZ function A–80
readDecimal function A–80
readInt function A–81
readLogical function A–82
readLongString function A–82
readLongStringCP function A–83
Read-only mode 4–33
Receive BytesMessage B–13
Receive StreamMessage B–9, B–16
Receive table
StreamMessage B–16
Receive transaction
TextMessage B–18
Receive XML
BytesMessage B–13
Receive XMLMessage B–11
receiveFromQueue procedure A–84

Receiving
reply 3–3

Receiving messages to queue 4–38
Receiving table StreamMessage B–16
Receiving transaction TextMessage B–18
Receiving XML in BytesMessage B–13
recover procedure A–85

Recovery of messages 4–45
Redelivery of messages A–87

Replies
destination, getting A–49
mechanisms 4–41
queue for published message 4–42
second session for 4–43
and SonicMQ Adapter B–6
Index

Reply mechanisms 4–41
Reply properties 4–38
Reply queue for published message 4–42
Request and reply 4–12
requestReply procedure A–86
reset procedure A–87
Reuse of messages 4–40
rollbackReceive procedure A–87
rollbackSend procedure A–88
Rolling back transactions A–48
Running gateway sample B–30
Run-time conditions 4–46, 4–48

S
-S connection parameter 4–4
Scalable server architecture
messaging example B–23
Selector
receiving with 4–38, A–14, A–84
subscribing with 4–39, B–4
Send and receive
TextMessage B–22, B–23
Sending
messages 3–3
to queue 3–2
Sending and receiving TextMessage B–22, B–23
Sending messages to queue 4–34
sendToQueue procedure A–89
Serialized connection objects 4–13, 4–15
creating 4–15
using 4–13
Services, Sonic ESB
deployment 6–29
management 6–29
run-time properties 6–30
session models 6–31
Session attributes 4–7
Session models
for Sonic ESB services 6–31
Session objects A–2
jmssession.p A–2
ptpsession.p A–2
pubsubsession.p A–3
SonicMQ Adapter 2–4
Sessions
and connections in JMS 4–2
attributes 4–7
creating in JMS 4–3
second, for replies 4–43
transacted for receiving A–132
transacted for sending A–132
setAdapterService procedure A–90
setApplicationContext procedure A–90
setBoolean procedure A–91
setBooleanProperty procedure A–91
setBrokerURL procedure A–92
setByte procedure A–92
setByteProperty procedure A–93
setBytesFromRaw procedure A–93
setChar procedure A–94
setClientID procedure A–94
setClientPersistence procedure A–95
setClientTransactionBufferSize procedure A–96
setConnectionFile procedure A–97
setConnectionURLs procedure A–98
setDataSet procedure A–98
setDate procedure A–99
setDateProperty procedure A–100
setDateTime procedure A–101
setDateTimeProperty A–101
setDateTime-TZ procedure A–102
setDateTimeTzProperty procedure A–102
setDefaultPersistency procedure A–103
setDefaultPriority procedure A–103
setDefaultTimeToLive procedure A–104
setDouble procedure A–104
setDoubleProperty procedure A–105
setErrorHandler procedure A–105
setFaultTolerant procedure A–106
setFaultTolerantReconnectTimeout procedure A–107
setFloat procedure A–107
setFloatProperty procedure A–108
setInitialConnectionTimeout procedure A–111
setInt procedure A–109
setInt64 procedure A–110
setInt64Property procedure A–110
setIntProperty procedure A–109
setJMSCorrelationID procedure A–111
setJMSCorrelationIDAsBytes procedure A–112
setJMSReplyTo procedure A–112
setJMSServerName procedure A–113
setJMSType procedure A–113
setLoadBalancing procedure A–114
setLocalStoreDirectory procedure A–114
setLocalStoreSize procedure A–115
setLocalStoreWait procedure A–115
setLong procedure A–116
setLongProperty procedure A–116
setLongString procedure A–117
setLongText procedure A–117
setMemptr procedure A–118
setNoAcknowledge procedure A–119
setPassword procedure A–120
setPingInterval procedure A–121
setPrefetchCount procedure A–121
setPrefetchThreshold procedure A–122
setReconnectInterval procedure A–122
setReconnectTimeout procedure A–123
setReplyAutoDelete procedure A–123
setReplyPersistency procedure A–124
setReplyPriority procedure A–124
setReplyTimeToLive procedure A–125
setReplyToDestinationType procedure A–125
setReuseMessage procedure A–126
setSaxReader procedure A–126
setSelectorAtBroker procedure A–127
setSequential procedure A–127, A–129
setShort procedure A–128
setShortProperty procedure A–128
setString procedure A–130
setStringProperty procedure A–130
setTempTable procedure A–131
setText procedure A–131
Setting and getting connection attributes 4–7
Setting and getting session attributes 4–7
Setting CLASSPATH 4–22
Setting reply properties 4–38
Setting session options 4–10
methods 4–10
setTransactedReceive procedure A–132
setTransactedSend procedure A–132
setUser procedure A–133
setX-Document procedure A–133
short data type mapped to ABL 4–32, 4–33
Single message acknowledgment 4–45
Sonic ESB described 1–2
SonicMQ See also SonicMQ Adapter; SonicMQ brokers broker connection 3–2 broker login A–120 described 1–2
Index–13

pinging broker A–121
server architecture, scaling B–23
user value for login A–133

SonicMQ Adapter
See also SonicMQ; SonicMQ brokers
See Java Message Service
Accessing 2–2
automatic reply mechanism B–6
connecting 4–8
defined 2–3
documentation references 1–8
error handling B–20, B–21
Exchanging messages between ABL and
JMS 2–4
licensing 1–5
overview 1–3
PTP messaging examples B–22 to B–26
replies B–6
session objects 2–4
Understanding how it works 2–3

SonicMQ brokers
See also SonicMQ; SonicMQ Adapter
publishing to topics 3–5, 4–34
receiving messages from queues 3–3
receiving messages to queues 4–38, 4–39
sending messages to queues 3–2, 4–34
subscribing to topics 3–6, 4–39

Starting connection to SonicMQ Adapter
4–8
Starting message reception 4–40
startReceiveMessages procedure A–134
Stopping message reception 4–40
stopReceiveMessages procedure A–134
Storing and extracting data 4–32
Storing undeliverable messages 4–8
StreamMessage 4–27
customer table B–16
example B–9, B–16
handling 4–27
process B–9, B–16
publish and subscribe B–9, B–16
receive B–9, B–16
String data type
mapped to ABL 4–32, 4–33
subscribe (procedure) 4–37
subscribe procedure A–135
Subscribing messages to topic 4–39
synchronous error example B–21
Synchronous message reception 4–41

T
Temporary destination 4–12, A–86
TempTableMessage 4–31, 5–21
handling 4–31
TextMessage
asynchronous error B–20
automatic reply B–6
error handler B–20, B–21
example B–3, B–4, B–6, B–18, B–20,
B–21, B–22, B–23
handling 4–24
publish and receive B–18
publish and subscribe B–3, B–4, B–6
queues B–22, B–23
receive group B–18
reply handle B–6
send and receive B–22, B–23
synchronous error B–21
transaction B–18
using queues B–22, B–23

time to live
for messages A–77, A–89
Topics 2–8, 3–5, 3–6, 4–12, 4–34, 4–39,
4–40, 4–42, 4–46, A–86
publishing messages 3–5, 4–34
subscribing 3–6, 4–39
Transacted receiving 4–43
Transacted sending 4–43
illegal calls 4–43
recover 4–43
setNoAcknowledge 4–43
Transacted sessions 4–43, 4–44
ABL transactions 4–44
receiving 4–43
sending 4–43
setNoAcknowledge 4–43
Transacted sessions, JMS 4–44
receiving 4–43
sending 4–43
Transactions
rolling back A–48

U
ubroker.properties file
default ping interval, specifying A–121
Unicode B–15
Unified domain 2–3
-URL connection parameter 4–5

Index–13
Using queues
   TextMessage B–22, B–23
UTF–8
   encoded XML document B–15
UTF–8 encoding B–15

W

WAIT–FOR state 1–5, 2–5, 4–37, 4–38, 4–49
WAIT–FOR statement 3–3, 3–6, 4–41, 4–49
waitForMessages procedure A–136
Web Service Definition (WSD) files
   usage 6–30
Web Service Mapping (WSM) files
   usage 6–30
Web Services Description Language (WSDL) files
   generating 1–5, 6–30
Write only modes 4–33
writeBoolean procedure A–137
writeByte procedure A–137
writeBytesFromRaw procedure A–138
writeChar procedure A–138
writeDate procedure A–139
writeDateTime procedure A–140
writeDateTime-TZ procedure A–141
writeDouble procedure A–141
writeFloat procedure A–142
writeInt procedure A–142
writeLong procedure A–143
writeLongString procedure A–144
Write-only mode 4–25, 4–33, A–15
writeShort procedure A–144
writeString procedure A–145
WSDL files. See Web Services Description Language files

X

XML
   code-page encoding B–15
   parser 4–30, B–15
XMLMessage 4–29
   example B–11
   parse B–11
   process B–11
   publish and subscribe B–11
   receive B–11