MQSeries® Integrator

IBM

ESQL Reference

Version 2.0.2

SC34-5923-00

MQSeries® Integrator

IBM

ESQL Reference

Version 2.0.2

SC34-5923-00

Note!

Before using this information and the product it supports, be sure to read the general information under “Appendix D. Notices” on page 133.

First Edition (April 2001)

This edition applies to IBM® MQSeries Integrator Version 2.0.2 and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Figures

Tables

About this book

Much of the information in this book was previously contained in *MQSeries Integrator Using the Control Center*.

This book describes how to use the ESQL expressions that are necessary for configuring message nodes in MQSeries Integrator.

MQSeries messages can be manipulated, constructed, and reformatted by nodes in the message flow, using a specialized form of standard database Structured Query Language (SQL). This specialized form is known as Extended SQL, or ESQL, and supports MQSeries Integrator processing of the message structure. This means that although you do not have to define the message structure to the Control Center, you do have to understand the definition to be able to construct valid ESQL for message manipulation.

If you have not used ESQL before, or do not have a good understanding of message structure, you are recommended to read "Chapter 1. Basic message structure" on page 1.

For more information about message nodes see *MQSeries Integrator Using the Control Center*.

Many of the examples provided in this book are based on the example XML message in "Message referenced in examples" on page 113.

A glossary and bibliography are provided at the end of this book.

Who this book is for

This book is intended for anyone who wants to create or modify MQSeries Integrator message flows using message nodes.

What you need to know to understand this book

You need to have read and understood the general introduction to all aspects of MQSeries Integrator in *MQSeries Integrator Introduction and Planning* and *MQSeries Integrator Using the Control Center*.

Terms used in this book

All references in this book to MQSeries Integrator are to MQSeries Integrator Version 2 unless otherwise stated.

All references in this book to Windows NT® are also applicable to Windows® 2000 unless otherwise stated. MQSeries Integrator components that are installed and operated on Windows NT can also be installed and operated on Windows 2000.

Chapter 1. Basic message structure

MQSeries Integrator provides a message brokering function that can transform messages from one format to another. The brokers that manage these transformations need to interpret the structure and content of the messages they receive to perform the full range of transformation functions available with MQSeries Integrator.

This chapter describes

- The structure of a message within MQSeries Integrator. See "Message structure" for more information.
- The message types supported by MQSeries Integrator. See "Supported message types" on page 12 for more information.
- How those messages are handled. See "How a message is interpreted" on page 13 for more information

Message structure

The following terms are used in message definition:

Message

- Information you want to send from one place to another
- One or more elements of data, also known as fields
- A structured collection of bits and bytes

Message Set

- A collection of messages
- A central repository or dictionary of message definitions
- The message data associated with a business project

Element of data

- A piece of business information
- A data type
- Part of a C or COBOL structure

When a broker retrieves a message from an MQSeries queue, its first task is to pass the data to a message parser (described in "Parsers" on page 4). This reads the string of bits and converts them to a tree format (described in"Tree format"). The tree format is easier to understand and manipulate, but contains identical content to the bits from which it is formed. When a broker delivers a message to a recipient, the message is converted back into a bitstream.

Tree format

A message tree is made up of a number of *elements*. At the top of the tree is the *root*: this has no *parent* and no *siblings*. The root is parent to a number of *child* elements. Each child must have a parent, it can have zero or more siblings (with which it shares its parent), and it can have zero or more children.

The tree structure of a message is shown in Figure 1 on page 2. The message root has two children, ElementA1 and ElementA2 (which are therefore siblings sharing a single parent). The child ElementA1 has three children (ElementB1, ElementB2, and ElementB3) and ElementB2 has a further child ElementC1.

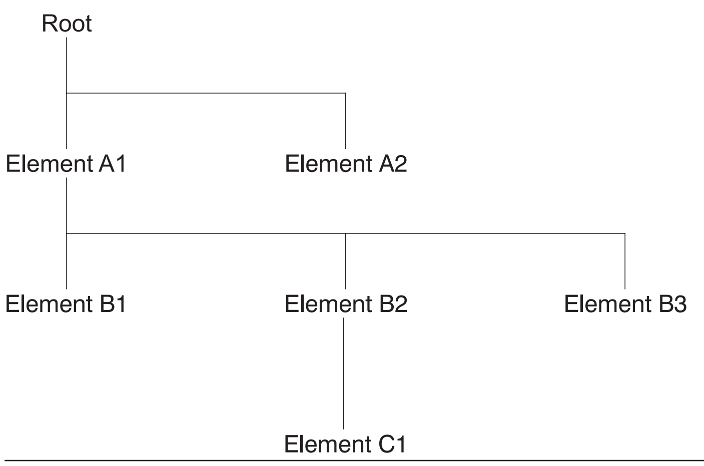


Figure 1. A message tree structure

When a message from an MQSeries message queue is parsed, three hierarchical trees are created:
- Message tree
- Destination List tree
- Exception List tree

The tree structure is independent of any message format. Provided you understand the structure of the message trees, you will be able to manipulate data or change the format of your message using MQSeries message nodes and ESQL.

This tree structure is explored further in the *MQSeries Integrator Programming Guide*.

Correlation names

Within ESQL, each of these trees is referred to by a correlation name.

The correlation name for the message tree is **Root**.

The correlation name for the Destination List tree is **DestinationList**.

The correlation name for the Exception List tree is **ExceptionList**.

These correlation names are used within the Database and Filter nodes.

Within the Compute node, there are two sets of trees: input and output, which are referenced using the following correlation identifiers:
- InputRoot
- InputDestinationList
- InputExceptionList
- OutputRoot
- OutputDestinationList
- OutputExceptionList

″Root″, ″DestinationList″ and ″ExceptionList″ are not valid correlation identifiers in a Compute node.

Each of these correlation names is described in “Initial correlation names” on page 55 .

Note: If you find that the tree structure is not clear when you look at one of your own messages, then you can use the Trace node as described in “Using a trace to view a message structure” on page 115 to make it easier to read.

Referring to simple fields in a message

You refer to fields in a message using a *field reference*. A field reference has a very similar format and meaning to a path in a file system. In its simplest form, a field reference consists of a period-separated sequence of identifiers. These identify the path in the message tree to get to the field you want. The simplest form of identifier is a sequence of alphanumeric characters, the first of which must be an alphabetic character.

You can also refer to specific elements that share a type and name by using the bracketed index value after the name. You can use an integer value or the defined constant `LAST`, for example

```
Body.Invoice.InvoiceNo
```

is equivalent to

```
Root.*[LAST].Invoice.InvoiceNo
```
[1]

The index numbering starts at value 1, and numbering is assigned from left to right. Using the example “Message referenced in examples” on page 113 this can be demonstrated as follows:

```
Body.Invoice.Purchases.Item[2].Author = 'Don Chamberlin'
```

Using quotes in the field reference

If you need to refer to fields with periods or spaces in their names, you must use double quotes around the reference:

```
   Body.Message."Companies on Wall Street"."mycompany.com"
```

If you need to refer to fields that contain double quotes, you must use two sets of double quotes around the reference:

```
   Body.Message.""hello""
```

If you need to refer to fields that have the same name as an ESQL keyword, shown in “Reserved words used in ESQL” on page 110, you must use double quotes around the reference:

```
   Body.Message."Set"
```

Using MQSeries constants in message headers

You can reference and update the fields within the MQSeries headers that are associated with each message. Every message has at least an MQMD, and most messages have one or more additional headers, for example the MQIIH (IMS/ESA bridge header).

You can use the defined MQSeries constants to test and assign values to the message header fields, both in their symbolic form, and as defined values. For example, to set the message type field, you can specify:

1. For an explanation of the use of * in this example, see “Anonymous field names” on page 61.

Symbolic constants

```
SET OutputRoot.MQMD.MsgType = MQMT_DATAGRAM;
```

You can achieve the same effect using the numeric value of MQMT_DATAGRAM:

```
SET OutputRoot.MQMD.MsgType = 8;
```

When these constants are used in ESQL within the nodes, the syntax checker accepts their use. However, if you use the *Check message flow* facility, the use of MQSeries constants might generate errors at this stage, with a message to indicate that the constants are not defined. You can ignore these messages; the constants will be processed correctly at run-time.

Table 1 tells you where you can find the definitions of the MQSeries constants for the MQSeries headers supported by MQSeries Integrator.

Table 1. MQSeries constants references

Header	Description	Reference
MQPCF	PCF header comprising: • MQCFH Command format header • MQCFIN PCF integer parameter • MQCFST PCF string parameter • MQCFIL PCF integer list parameter • MQCFSL PCF string list parameter	*MQSeries Programmable System Management*
MQCIH	CICS® bridge	*MQSeries Application Programming Reference*
MQDLH	Dead letter	*MQSeries Application Programming Reference*
MQIIH	IMS bridge	*MQSeries Application Programming Reference*
MQMD	Message descriptor	*MQSeries Application Programming Reference*
MQMDE	MQMD extension	*MQSeries Application Programming Reference*
MQRFH	Rules and formats	*MQSeries Publish/Subscribe User's Guide*
MQRFH2	Rules and formats version 2	*MQSeries Integrator Programming Guide*
MQRMH	Reference message	*MQSeries Application Programming Reference*
MQSAPH SMQ_BMH	SAP R/3 Link headers	*MQSeries link for R/3 User's Guide*
MQWIH	Workload information	*MQSeries Application Programming Reference*

CodedCharSetId, Encoding, and data conversion

The broker supports the MQSeries constants for CodedCharSetId and Encoding fields, and follows the MQSeries architecture rules for supporting MQCCSI_INHERIT and MQCCSI_DEFAULT as necessary when generating a MQSeries message.

Parsers

Parsers are used by brokers to validate incoming messages, and to build outgoing messages.

When you build an output message, you must specify the format of each element in the message either implicitly, by copying the entire input message, or by specifying the parser names in the ESQL. The parser name is specified in a SET statement, also known as an assignment, for example the parser in the following statement is generic XML:

```
SET OutputRoot.XML = InputBody;
```

The parsers used in MQSeries Integrator Version 2.0, are
- Properties
- MQMD and MQMDE
- Other header parsers as shown in “Appendix C. MQSeries message header parsers” on page 119
- Message body parsers
  - XML
  - NEONMSG
  - BLOB
  - MRM, including several wire formats:
    - CWF
    - XML
    - PDF
  - JMSMap
  - JMSStream

When a message is retrieved by a message flow, its constituent parts are passed to the correct parser for interpretation (unless interpretation is not required: for example if a whole message is copied).

With the exception of the MQMD, which must be the first header, the order of the headers preceding the message body is not important: the parser for each header processes that header independently. However, the fields are parsed in a particular order that is governed by the parser: you cannot predict or rely on the order chosen.

Message tree fields in ESQL have data types based on their MQSeries structure description as described in the mapping below. There are some exceptions to this, and these are noted.
- MQLONG types are represented as INTEGER.
- MQCHAR and MQCHARn are represented as CHARACTER.
- MQBYTE and MQBYTEn are represented as BLOB.
- Date and time fields are represented as:
  - TIMESTAMP if the field can be converted to a valid TIMESTAMP
  - CHARACTER if the field cannot be converted to a valid TIMESTAMP
- The Expiry field in the MQMD is a special case:
  - If it is set to -1 (unlimited) it is converted to an integer
  - If it is not set to -1, it is converted to a TIMESTAMP

Fields in MQSeries structures, excluding MQRFH2 folders, are represented by name and value elements. MQRFH2 folder names are represented by name only elements. Structure length fields and structure identifier fields are not visible, and are filled in with appropriate values by the broker.

Other fields that are updated by the broker include all format fields and domain fields in those parsers that support them.

Maintaining header integrity

The broker ensures that the integrity of the headers that precede a message body is maintained. The format of each part of the message is defined by the Format field in the immediately preceding header:

- The format of the first header is known, because this must be MQMD.
- The format of the next (second) part of the message, which might be another header, or the message body, is set in the Format field in the MQMD.
- If a third part of the message exists, its format is defined in the format field of the second part of the message.

This process is repeated as many times as is required by the number of headers that precede the message body. You do not have to populate these fields yourself: the broker handles this sequence for you.

If the body parser is not understood by MQSeries, the current Format field is checked. If it currently contains a registered parser name, it is set to MQFMT_NONE. The domain field is always updated. These actions might result in information explicitly stored by an SQL expression being replaced by the broker.

The properties parser

Every message has a set of standard properties that you can manipulate in the message flow nodes in the same way as any other property. The majority of these fields map to fields in the supported MQSeries headers and are passed to the appropriate parser when a message is delivered from one node to another.

If no parser is capable of receiving the property, the property is kept in the Properties parser until such time as a suitable parser can be found. If the message is converted to a bitstream, for example in an output node, any properties remaining solely in the property parser are discarded.

All messages generated by IBM supplied nodes provide a properties folder for the message as the first child of the root. It is not a requirement for a message to have a properties folder, although it is recommended. If you are using your own (plug-in) nodes, the interface provided does not automatically generate the properties folder for a message: if you want one in a message, you must create the folder yourself.

Having transmitted properties to each appropriate parser, the properties parser requests the values back from the owning parser. This ensures that the cached values of the properties are consistent with the message on entry and exit from each node. The state within any given node is dependant on the behavior of the node.

Properties parser

The parser name is "Properties". The standard properties are:

- MessageDomain
- MessageSet
- MessageType
- MessageFormat
- Encoding
- CodedCharSetId
- Transactional
- Persistence
- CreationTime
- ExpirationTime
- Priority
- Topic (this field contains a list)

Note: `InputRoot.Properties` can be coded as `InputProperties`.

Exception and destination list tree structure

A tree representation is used in a broker to represent the MQSeries message. Within the broker this tree representation is supplemented by two additional trees:

- The destination list tree

  This represents the destinations to which a message is sent.
- The exception list tree

  This represents the exception conditions that have occurred while processing that message.

A message being processed within the broker consists of three separate syntax element trees:

- The destination list tree
- The exception list tree
- The message tree

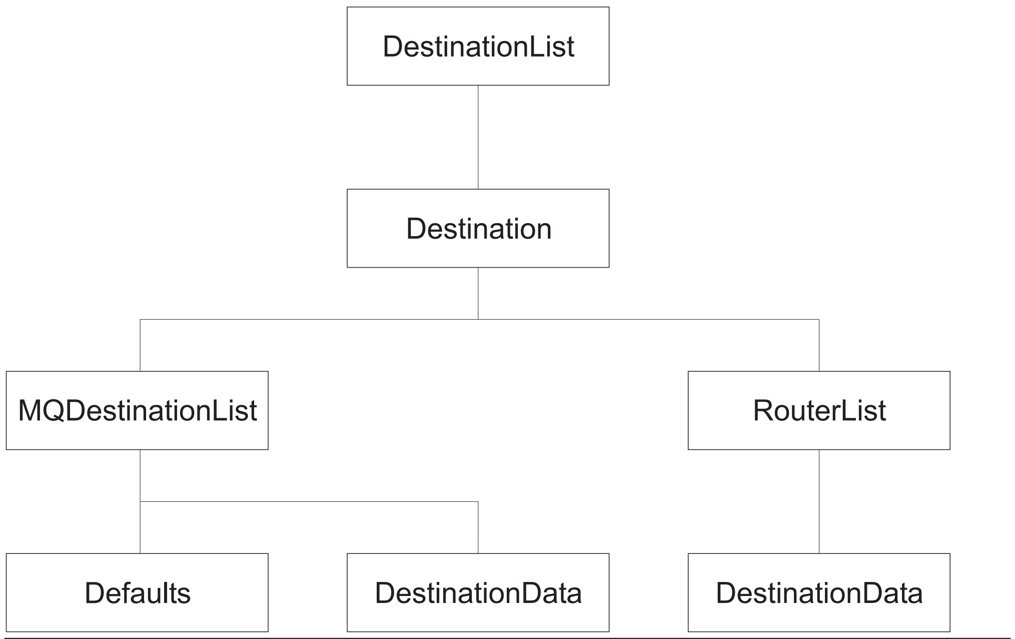
You can query and manipulate each of these trees in much the same way in Filter, Database, and Compute nodes. Elements can be created, examined, or even copied from one tree to another. The destination and exception list trees only exist within the broker and are not in the MQSeries message.

The following sections describe the structure of the destination and exception list trees.

Destination lists

A destination list tree describes a list of internal and external destinations to which a message will be sent. Output nodes can be configured to examine this list and send the message to the given destinations. Alternatively, they can be configured to send messages to a fixed destination. In this case, the destination list has no effect on broker operations and can be empty (that is, consist of a Destination List element only).

The destination list tree has the following structure:



The root of the tree is called “DestinationList”. The tree has a single name element called “Destination”: this is the first and only child of DestinationList. The Destination element consists of a number of children that indicate the transport types to which the message will be directed (the Transport identifiers). Each element is a single name element, for example, ″MQDestinationList″ or RouterList.

Note: In the Compute node there are two Destination List trees.

- An input tree: use InputDestinationList to refer to the root of this tree.
- An output tree: use OutputDestinationList to refer to the root of this tree.

The Transport identifier element might contain an element called “Defaults”. If it does, this must be in the first child and contains a set of name-value elements that give default values for the message destination and its put options.

The element that identifies the transport might also contain a number of elements called “DestinationData”. Each of these contains a set of name-value elements that defines a message destination and its put options.

- For MQDestinationList, the set of elements that define a destination comprises:
  queueManagerName
  queueName
  transactionMode
  persistenceMode
  newMsgId
  newCorrelId
  segmentationAllowed
  alternateUserAuthority

  All of these elements have a data type of CHARACTER. See the description of the MQOutput node in the online help for their descriptions and valid values.

You can access the online help from the Help menu in the Control Center taskbar or by highlighting an MQOutput node, right clicking, and selecting Help.

- For RouterList, the set of elements that define destination have a single entry, labelName.

Exception lists

If no exception conditions occur while you are processing a message, the exception list associated with that message consists of a root element only. This is, in effect, an empty list of exceptions.

If an exception condition occurs, message processing is suspended and an exception is thrown. Control is passed back to a higher level, that is, an enclosing catch block. An exception list is built to describe the failure condition, and then the whole message, together with the destination list and the newly-populated exception list, is propagated through an exception handling message flow path.

Exception handling paths start at a failure terminal (most message processing nodes have these), the catch terminal of an MQInput node, or the catch terminal of a TryCatch node, but are no different in principle from a normal message flow path. Such a flow consists of a set of interconnected message flow nodes defined by the designer of message flow. The exception handling paths differ in detail. For example, they might examine the exception list to determine the nature of the error, and so be able to make an appropriate response.

The message and destination list that are propagated to the exception handling message flow path are those in effect at the start of the exception path, not necessarily those in effect when the exception is thrown. Figure 2 illustrates this point:

- A message (M1) and destination list (D1) are being processed by a message flow. They are passed through the TryCatch node to Compute1.
- Compute1 updates the message and destination list and propagates a new message (M2) and destination list (D2) to the next node, Compute2.
- An exception is thrown in Compute2. The exception is propagated back to the TryCatch node, but the message and destination list are not. Therefore the exception handling path starting at point **A** has access to the first message and destination list, M1 and D1.
- If there had been no TryCatch node in the message flow, and the failure terminal of Compute2 had been connected (point **B**), the message and destination list M2 and D2 would have been propagated to the node connected to that failure terminal.

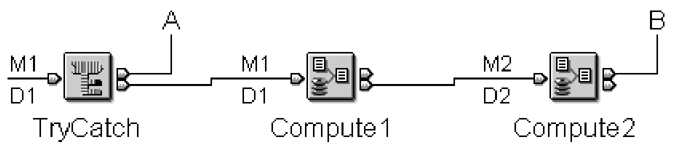


Figure 2. Message and destination list for an exception

The root of the Exception List tree is called "ExceptionList", and the tree itself consists of a set of one or more exception descriptions. Each exception description consists of a name element whose name is one of the following:
- RecoverableException
- ParserException
- ConversionException
- UserException
- DatabaseException

Note: In the Compute node there are two Exception List trees.
- An input tree: use InputExceptionList to refer to the root of this tree.
- An output tree: use OutputExceptionList to refer to the root of this tree.

These name elements contain children that take the form of a number of name-value elements that give details of the exception and zero or more name elements whose name is "Insert". The NLS (National Language Support) message number identified in a name-value element in turn identifies an MQSeries Integrator error message. All error messages are defined in detail in *MQSeries Integrator Messages*. The Insert values are used to replace the variables within this message, and provide further detail of the precise cause of the exception.

The name-value elements within the exception list are shown in Table 2 on page 12.

Exception and destination list tree structure

Table 2. Exception list name-value elements

Name		Type	Description
File[1]		String	C++ source file name
Line[1]		Integer	C++ source file line number
Function[1]		String	C++ source function name
Type[2]		String	Source object type
Name[2]		String	Source object name
Label[2]		String	Source object label
Text[1]		String	Additional text
Catalog[3]		String	NLS message catalog name[4]
Severity[3]		Integer	1=information 2=warning 3=error
Number[3]		Integer	NLS message number[4]
Insert[3]	Type	Integer	The data type of the value: 0=Unknown 1=Boolean 2=Integer 3=FLOAT 4=Decimal 5=Character 6=Time 7=GMT Time 8=Date 9=Timestamp 10=GMT Timestamp 11=Interval 12=BLOB 13=Bit Array 14=Pointer
	Text	String	The data value

Notes:

1. The File, Line, Function, and Text elements should not be used for exception handling decision making. These elements ensure that information can be written to a log for use by IBM service personnel.
2. The Type, Name, and Label elements define the object (usually a Message Flow node) that was processing the message when the exception condition occurred.
3. The Catalog, Severity, and Number elements define an NLS message: the Insert elements that contain the two name-value elements shown define the inserts into that NLS message.
4. NLS message catalog name and NLS message number refer to a translatable message catalog and message number.

The exception description structure can be both repeated and nested to produce an exception list tree. In this tree:

- The depth (that is, the number of parent-child steps from the root) represents increasingly detailed information for the same exception.
- The width of the tree represents the number of separate exception conditions that occurred before processing was abandoned. You will find that this number is usually one, and results in an exception tree that consists of a number of exception descriptions connected as children of each other.

Supported message types

The messages supported by MQSeries Integrator are of three broad types, identified by a property of the message called the message domain:

1. A message can be *unstructured*: its message domain must be set to BLOB.
2. A message can be *self-defining*: its message domain must be set to XML.

   Two additional domains are included in this category to support JMS messages: the domain JMSMap can be used for jms_map messages and the domain JMSStream can be used for jms_stream messages.
3. A message can be *predefined* as in an MRM message.

How a message is interpreted

When the message arrives in a broker, it is removed from the input queue by the MQInput node defined in the message flow that processes messages from this queue. It must be processed by an appropriate parser to decode the physical structure and create the logical structure.

The MQInput node determines what to do with each message:
- If the message has an MQRFH or MQRFH2 header following the MQMD header, the domain identified in the MQRFH2 header is used to decide which root message parser is invoked.
- If the message does not have an MQRFH or MQRFH2 header, but the properties of the MQInput node indicate the domain of the message, the parser specified by the node property is invoked.
- If the message has a valid MQMD, but the message body cannot be recognized, the message cannot be interpreted or parsed, and it is handled as a binary object (BLOB). See "Working with unstructured messages in the BLOB domain" on page 28 for more information about these messages.

Each message received must have an MQMD header, and can have zero or more additional headers. MQSeries Integrator provides a parser for each of the following MQSeries headers:
- MQCFH
- MQCIH
- MQDLH
- MQIIH
- MQMD
- MQMDE
- MQRFH
- MQRFH2
- MQRMH
- MQSAPH
- MQWIH
- SMQ_BMH

Further details about the support for these parsers is given in "Appendix C. MQSeries message header parsers" on page 119.

MQSeries Integrator also supports the use of additional parsers. You can create a message parser using a defined programming interface. This interface and the techniques you must employ to create your own "plug-in" parsers are described in the *MQSeries Integrator Programming Guide*. If you use your own parser, you must set up your MQInput node properties to identify your parser.

Self-defining messages in the XML domain

The message carries the information about its content and structure within the message. Its definition is not held anywhere else.

When a self-defining message is received by the broker, it is handled by the XML parser, and a tree is created according to the XML definitions contained within that message.

A self-defining message is also known as a *generic XML message*. It does not have a recorded format.

Self-defining messages in the XML domain

A self-defining message can be handled by every IBM-supplied message processing node. The whole message can be stored in a database, and headers can be added to or removed from the message as it passes through the message flow.

The NEON domain

The NEON message domain supports predefined messages, but the domain of the message must be set to either

- `NEON`: The deprecated parser available in earlier versions of MQSeries Integrator. This parser can be processed in the NEONFormatter and NEONRules nodes.

  or

- `NEONMSG`: A plugin NEONMSG parser with enhanced functionality introduced with MQSeries Integrator Version 2.0.2. This parser can be processed in the NEONMap, NEONRulesEvaluation and NEONTransform nodes.

Both parsers can be used with MQSeries Integrator Version 2.0.2 (the NEON parser has been retained to provide backwards compatibility) but parsers can only be used with the nodes described above. That is, the new parser will only work with the new nodes, and the old parser will only work with the old nodes.

The message must be defined using the NEONRules and NEONFormatter Support for MQSeries Integrator graphical utilities that are supplied with MQSeries Integrator Version 2.0.2. You can create new messages and use existing messages defined to the NEONMSG domain.

A NEONMSG message can be handled by every IBM-supplied message processing node. The whole message can be stored in a database, and headers can be added to or removed from the message as it passes through the message flow. The NEONFormatterMap node can be used to transform a NEONMSG message. No other node can manipulate the message contents.

For further information about working with these messages, refer to the NEONRules and NEONFormatter Support for MQSeries Integrator User's Guideand *MQSeries Integrator Using the Control Center*.

The NEON parser domain becomes NEONMSG in MQSeries Integrator Version 2.0.2. NEON still exists, but only for compatibility with previous releases.

The MRM domain

This section describes messages in the MRM domain and how to work with them.

Predefined messages in the MRM domain

The MRM domain supports predefined messages that must have the message domain set to **MRM**. It must be defined to the *Message Repository Manager*, a component of the Configuration Manager. You can define messages to the MRM domain using the Control Center (Message Sets view). The MRM maintains these messages in the message repository.

You can also predefine a message to the MRM in the XML message domain (the domain is defined as a message set property, as described in "Message set properties" on page 23). If you define a message to the XML domain, you can use all the facilities available to MRM domain messages to manipulate and reference the message in the nodes within your message flows in the Control Center. However, you are not expected to assign these message sets to a broker, nor to deploy them. Because the domain is set to XML, the XML parser is invoked by the broker and does not reference any external message definition.

An MRM message can be handled by all the IBM-supplied message processing nodes except the NEON nodes.

The whole message, or parts of the message, can be stored in a database, and headers can be added to or removed from the message as it passes through the message flow. The message can be manipulated using ESQL defined within all message processing nodes that support manipulation (for example, compute and filter).

You can also transform any message in the MRM domain into any other format defined to the MRM using ESQL. This includes code page and encoding conversion: this capability provides a significant benefit as the message structure has already been provided in MQSeries Integrator and no MQSeries data conversion exits are required.

MRM Padding characters: In MQSeries Integrator, all MRM padding characters are subject to character conversion, regardless of whether they are defined as SPACE, specified character, hexadecimal value or numeric. If you are converting a message to a different code page, you need to ensure that the converted value of the padding character is valid for this code page. For example, when converting from ASCII to code page ’500’, if you have specified the numeric ’8’ as your padding character, this will be converted from 0x08 to 0x16. The ASCII and EBCDIC representations of ’back space’.

There is currently a restriction that the value of your padding character should not be greater than 0x7F. Also you should note that if you enter a hexadecimal or numeric value, it is considered to be the character represented by that number in UTF-8.

For a fuller discussion of data conversion considerations, see Chapter 9, Planning your MQSeries Integrator network, in *MQSeries Integrator Introduction and Planning*.

Messages with a message domain of MRM have three other characteristics for further classification:

1. Message format

   Three message formats are supported by the MRM:

   a. A message can have a message format of CWF (Custom Wire Format).

      These messages are MRM representations of legacy data structures created in the C or COBOL programming language, and imported into the MRM using the Control Center facilities. See “Importing legacy formats” on page 27 for details of how to complete this task.

      You can also create new messages using this format.

   b. A message can have a message format of PDF (predefined format).

      This is a specialized format used predominantly in the finance industry. It does not have any connection with the Portable Document Format defined by Adobe (also known as PDF).

      If you already use messages of this format, you can continue to use them and process them by specifying this format in the definitions.

   c. A message can have a message format of XML.

These messages are represented as XML documents. They conform to an XML DTD (Document Type Definition) that can be generated by the Control Center for documentation purposes.

2. Message set

   This identifies the message set to which each message belongs. This is specified as the message set *identifier*, not the message set name. When you define a message in the MRM message domain, you must define a message set that contains it. A message set can contain one or more related messages.
3. Message type

   The message type identifies the message definition within the set. It is the unique identifier for each message of this particular content and format.

An overview of the message definition process

The message definition process is managed by the MRM.

When you create or modify message definitions using the Control Center, the MRM stores them in the *message repository*, a set of tables in a database created and maintained by the Configuration Manager. [2]

Each message definition is created within, and belongs to, a *message set*, which is simply an organizational grouping of related messages. A message set includes the definitions of one or more related messages, typically those used by a single application. You construct each message using a set of building blocks, known as *message components*, some of which are supplied with MQSeries Integrator (the simple types) and some of which you define using the Control Center (the compound types).

So, for example, for a banking application you could define simple elements, such as Account Number and Account Balance, then include those simple elements in a compound element, such as Customer Record. The Account Number, Account Balance, and Customer Record elements would all be reusable by other message definitions within the same message set. The components of a message are described in detail in "The components of a message definition" on page 17.

You must *assign* message sets to those brokers that need to understand them. When you *deploy* message set assignments in the broker domain, the MRM constructs a *message dictionary* from each message set (one dictionary plus one CWF descriptor file for each set) and sends it to each broker that needs access to the message definitions.

The message model

The MQSeries Integrator message model provides a platform and language independent way of defining logical messages that represent structured business information.

In this message model, a message definition comprises separate, reusable *message components*. The relationship between components is defined as being either a *member* relationship or a *reference* relationship.

Reference relationship: A reference relationship is a defining relationship between two components. For example, an element component of type STRING has a reference relationship to an element length component that defines the length of the element.

2. The configuration repository and message repository are implemented using an IBM DB2 Universal Database® for Windows NT.

Reference relationships are always mandatory.

Member relationship: A member relationship is a parent-child relationship between two components. For example, a (parent) compound type has a member relationship with one or more (child) elements. The member relationship is always expressed as an attribute of the parent, not of the child.

Member relationships are always optional.

The components of a message definition: The components of a message definition are described in the remainder of this section. For each component, the reference and member relationships are identified.

Message component: The message component defines both the business meaning and the format of a single unit of information to be exchanged between applications.

- A message component has a reference relationship to a single type component (a compound type) that defines the content of the message. It can also have a reference relationship to an element qualifier.
- A message component has no member relationships.

Once a message component has been created, the reference of the type component cannot be changed.

Element component: The element component defines both the business meaning and the format of a single unit of information within a message.

- An element component has a reference relationship to a single type component (a simple type or a compound type) that defines the content of the element.

  An element component also has a reference relationship to an element length component, if the element is of simple type STRING.
- An element component can have a member relationship to one or more (child) element valid value components, which must have the same type as the element.

Once an element has been created, the identifiers of the type and element length components to which it refers cannot be changed.

Type component: The type component defines the format or content of a message or an element. A type can be a *simple* type or a *compound* type.

Simple type
: Is a basic data type supported by the run-time message parsers. The simple types are STRING, INTEGER, FLOAT, BOOLEAN, and BINARY. The simple types are created automatically when you create a message set.

Compound type
: Is a structure made up of one or more element components.

- A type component has no reference relationships.
- A compound type component has member relationships to one or more (child) elements.

Element length component: The element length component defines a maximum length value that completes the definition of any element of the simple type STRING.

- An element length component has no reference relationships.
- An element length component has no member relationships.

Category component: The category component groups messages within a message set, typically by business function. The extraction and generation functions of the MRM can produce their output by category.

- A category component has no reference relationships.
- A category component can have member relationships to one or more (child) messages.

Element valid value component: The element valid value component defines either a single value, or a range of values. One or more element valid value components can be associated with an element, or with an element qualifier, or both. One element valid value can define the default value of an element.

- An element valid value component has a reference relationship to a type component that defines the content of any element component to which the value may apply. (In other words, a valid value of type STRING can only be applied to an element of type STRING, or to its element qualifier.)
- An element valid value component has no member relationships.

Element qualifier component: An element qualifier component provides additional information that qualifies the definition of an element component. An element qualifier component can be associated with a specific element component within a specific message component to qualify its use in that message component only. For example, an element qualifier can specify that a specific element is mandatory in a specific message, even though it is optional elsewhere.

- An element qualifier component has a required reference relationship to an element component. The element qualifier component can be used to qualify the use of its related element in a specific instance.
- An element qualifier component can have member relationships to zero or more element valid value components, that must have the same type as the referenced element. If valid values are present for an element qualifier in a message, they apply to the related element within the specific message only (overriding those defined for the element). One such element valid value can denote a default value.

Figure 3 on page 19 shows all possible components of a message definition and summarizes the relationships between them.

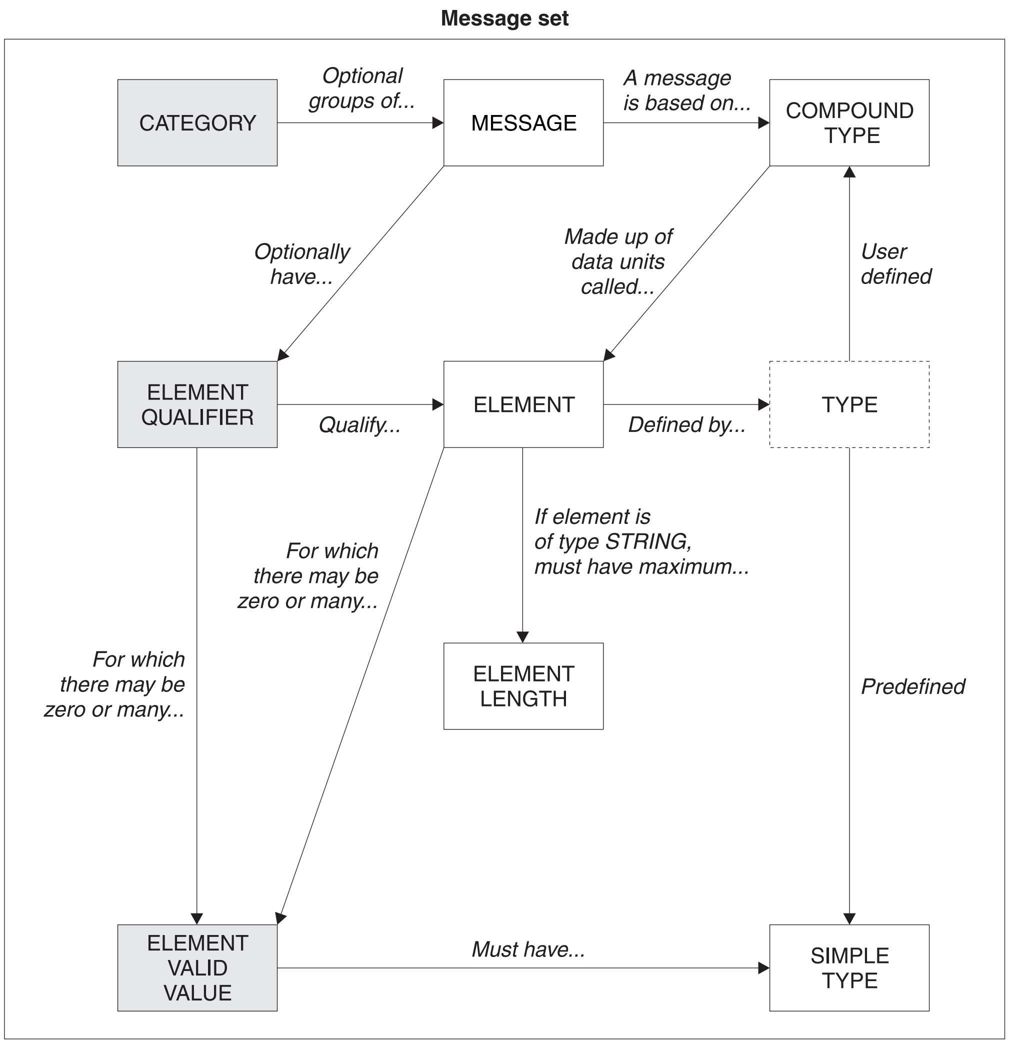


Figure 3. The components of a message

Component identifiers and names: Each component of a message definition has an identifier and a name.

Component identifier

Identifies a component uniquely within a message set. No two components in a message set can have the same identifier, and no two components of the same class (for example, two elements or two categories) can have identifiers that differ only by case. For example, you cannot define an element with the identifier "ADDRESS" and an element with the identifier "address" in the same message set.

In the case of element components, the element identifier is used in application programs to access data values assigned to the element.

An identifier must begin within an alphabetic character (`A-Z`, `a-z`). The remainder of the value, up to a maximum of 254 characters, can contain alphanumeric (`A-Z`, `a-z` and `0–9`), underscore (_),

	and period (.) characters. Other characters, including space characters, are not valid.
	You cannot change the identifier of a component.
Component name	Is a descriptive name for a component. It is typically the full name of a component (for example, “Street Name” or “Account Number Length”), in contrast to the component identifier, which is often an abbreviated name and subject to environmental conventions.
	You can change the name of a component (using the **Rename** action).

An example message definition: To illustrate the concepts introduced in this section, consider this example of a simple message:

Some items to note about this message:

- The top-level elements HomeAddress and WorkAddress have the same substructure, which you can define by creating a compound type component called Address that contains the common elements Line, Country, and ZipCode. The compound type Address is referenced by the top-level elements HomeAddress and WorkAddress.
- The elements Line, Country, and ZipCode all reference the simple type STRING, which is created by default when a message set is created. These elements must also reference an element length component.

If you create a message definition from the bottom up (that is, starting with the lowest-level components and working up to the top of the hierarchy), you are guaranteed to create a referenced component before you create the component that contains the reference.

The components of our example AddressesMessage would be created in the following order:

1. Simple type STRING (created by default when the message set is created)
2. Element length components, in any order:
   a. Maximum Length 50
   b. Maximum Length 20

3. Element components, in any order:
   a. Element Line, referencing simple type STRING and element length Maximum Length 50.
   b. Element Country, referencing simple type STRING and element length Maximum Length 50.
   c. Element Zip Code, referencing simple type STRING and element length Maximum Length 20.
4. Compound type component Address, with member relationships to the following child elements:
   - Element component Line
   - Element component Country
   - Element component Zip Code
5. Element components, in any order:
   a. Element Home Address, referencing compound type Address
   b. Element Work Address, referencing compound type Address
6. Compound type component HomeAndWork, with member relationships to the following child elements:
   - Element Home Address
   - Element Work Address
7. Message component AddressesMessage, referencing compound type HomeAndWork.

Clearly, before you use the Control Center to define your messages, you need to have done the data analysis that will enable you to create complete and accurate definitions in an efficient manner.

Message sets: A message set contains the definitions of one or more messages, plus the definitions of the components that make up those messages. A typical message set contains the definitions of all messages required by a single application. The run-time message dictionary provided by the MRM to the run-time message parsers contains definitions for all messages in a single message set.

In common with the components of a message, message sets must have a name. They must also have a level number that identifies this version of the message set. Message set properties, related to the data model layers, are described in "Message set properties" on page 23.

The data model layers

So far, we have discussed the concepts underlying the MRM's message model. However, the message set contains additional "layers" of information that support related MRM functions. These are:
- The documentation layer
- The C language layer
- The COBOL language layer
- The run-time layer
- The Custom Wire Format layer

These layers of information are visible to you in the Properties pane of the **Message Sets** view, as described in *MQSeries Integrator Using the Control Center*. They are described in more detail in the following sections.

The documentation layer: When you define each message component and each message set, you have the opportunity to provide a short description or a long description, or both, of that component or message set. You are recommended to use these description fields to describe the business meaning of the object, and to record any business rules that govern their use.

The documentation extractors of the MRM include this information in generated documentation. For more information about generating documentation from the MRM, see *MQSeries Integrator Using the Control Center*.

The C language layer: The MRM can generate C header files from the message definitions you create that can be used in messaging applications developed in C language. You specify values for properties of the components to support this function. These properties are mandatory. Although you might never wish to generate language bindings, the message set is incomplete if these properties are missing.

For the category component, you specify:

Category Header File Name
Provides the name of the header file into which structure definitions for all messages in this category are generated.

Include in Main Header
Specifies whether this header file is included from the main header file for the message set.

For the element component, you specify:

C Language Name
Provides the name used for this element as a field within C structure definitions. By default, the element identifier is used.

For the type component, you specify:

C Language Name
Provides the name for the C structure definition that is generated for the type. By default, the type identifier is used.

File Name
Provides a name for a header file to be generated containing a structure definition for the type. This value is optional, and is not usually specified: the structure definitions for type components appear only in the category header files.

The COBOL language layer: The MRM can generate COBOL copy books from the message definitions you create that can be used in messaging applications developed in COBOL language. You specify values for properties of the components to support this function. These properties are mandatory. Although you might never wish to generate language bindings, the message set is incomplete if these properties are missing.

For the category component, you specify:

Category Copy Book Name
Provides the name of the copy book file into which structure definitions for all messages in this category are generated.

For the message component, you specify:

COBOL Language Name
Provides the name used for the COBOL structure definition that is generated for the message. By default, the message identifier is used.

Message Copy Book Name
Provides the name of the copy book file into which the structure definition for the message is generated.

For the element component, you specify:

COBOL Language Name
Provides the name used for this element as a field within COBOL structure definitions. By default, the element identifier is used.

For the type component, you specify:

COBOL Language Name
Provides the name for the COBOL structure definition that is generated for the type. By default, the type identifier is used.

Structure Copy Book Name
Provides a copy book file name into which the structure definition for the type is generated.

The Custom Wire Format layer: The CWF layer defines additional information that is used to define the mapping between logical messages and legacy message formats defined by applications that use data structure features of languages such as C and COBOL to populate the message structure. This information is used to produce a wire format descriptor that can be used by a run-time message parser.

You specify some of the following properties for each element that is a child in a type. For example, if the logical type is string, the physical type packed decimal is not applicable. Similarly, if the logical type is float, and the physical type is extended decimal, and the signed field is set to True, then the sign orientation field is applicable. The properties that might be applicable are:
- Physical type
- Length (Count or Value Of)
- Signed and Sign Orientation
- Skip count
- Byte alignment
- String justification
- Padding character
- Virtual decimal point
- Repeat (Count or Value Of)

Message set properties

The properties of a message set are displayed on several tabs in the Properties pane of the Message Sets view. The tabs are:

Basic tab (identified by message set name)
This tab defines basic properties for the message set. These are:

Level Is a numeric value that identifies the version of the message set. If you are creating a second (or subsequent) version, you must set this property to a value higher than the highest existing level number.

See “Message set versioning” on page 26 for more information.

Finalized
Indicates if the message set has been finalized (true) or not (false).

Freeze Time Stamp
Indicates the date and time when the message set was frozen. If this is not set, the message set has never been frozen.

Identifier
Is the identifier by which the message is known (used in addition to its name). It is a unique value, and is automatically allocated when the message set is created.

Base Message Set
Is the base message set from which this message set's definition is derived. All components defined in the base message set are also defined in this message set.

C Language tab
The MRM can generate C header files from the message definitions you create that can be used in messaging applications developed in C language.

This tab defines properties that identify names for header files generated from this message set using the **Message Sets —> Generate** command from the Control Center. These properties are mandatory.

Main Header File Name
Is the name of the generated header file that contains C structure definitions of the messages in this message set.

Orphan Header File Name
Is the name of the generated header file that contains definitions of C structures (types) that are not used by any message in this message set.

COBOL Language tab
The MRM can generate COBOL copy books from the message definitions you create that can be used in messaging applications developed in COBOL language.

This tab defines the property Main Copy Book Name that identifies the name of the main copy book generated from this message set by using the **Message Sets —> Generate** command from the Control Center. These properties are mandatory.

Custom Wire Format tab
The CWF defines additional information that is used to define the mapping between logical messages and legacy message formats defined by applications that use data structure features of languages such as C and COBOL to populate the message structure. This information is used to produce a wire format descriptor that can be used by a run-time message parser.

The properties on this tab are:

Custom Wire Format Identifier	You are recommended to use the default value of CWF. You can extend this identifier to eight characters if you choose, but the first three characters must always be CWF. You cannot specify embedded blanks or special characters.
Byte Alignment Pad	This defines the default padding character used for this message set. The default is 0.

Boolean True and False Values
Boolean True and False Values define the True and False values to be used by every element of type Boolean in this message set. The defaults are 00000001 and 00000000.

Boolean True and False Values must be the same length, and can be between 1 and 4 bytes long. They must be defined in half-byte values and you must specify an even number (to define a number of whole bytes). For example, if you want your Boolean values to be ASCII characters Y and N, you would enter the two characters 54 in the True field and the two characters 46 in the False field.

Run Time tab
Defines the property parser that identifies the message domain for this message set, and therefore the parser that the broker invokes to interpret the messages. Five options are available:

- MRM. This is the default and usual case.
- XML. You can set the domain and parser to XML if you want to define the message set for easier manipulation and reference in the message flows in which it is used (as described in *MQSeries Integrator Using the Control Center*).
- NEONMSG
- JMSMAP
- JMSSTREAM

Description tab
Includes a short description and a long description of the message set. Both are optional properties.

When you define each message component and each message set, you have the opportunity to provide a short description or a long description, or both, of that component or message set. You are recommended to use these description fields to describe the business meaning of the object, and to record any business rules that govern their use.

The documentation extractors of the MRM include this information in generated documentation. For more information about generating documentation from the MRM, see *MQSeries Integrator Using the Control Center*.

Message set states: The state of a message set varies in line with development, testing, and production cycles.

The states of a message set are:

Normal
If a message set is not locked (checked out), frozen, or finalized, it is considered to be in normal working state (but this state is not specified in the view). It can be checked out, updated, and checked in. This state is not explicitly stated in the Control Center: it is inferred by the message not being locked, frozen, or finalized.

	Note: When you create a new message set, it is automatically checked in, then checked out, and you will see the key icon appear against the new message set. A message set is never shown in new state (with the new icon against it). However, components of the message set (for example, an element) do appear as new when they are first created.
Locked	The state of a message set while it is checked out (locked) by a Control Center user. A message set must be in this state before you can change any of its properties. A message set must also be locked before you can freeze it.
Frozen	The state of a message set that is not expected to change (for example, on entry to a test phase). Neither the message set itself nor its contents can be changed while it is in this state, nor can they be checked out. A message set can be unfrozen by selecting **Unfreeze** if subsequent change is required. Frozen state is indicated by the existence of a Freeze date in the message properties. An attempt to freeze a message set fails if any component of the message set is checked out or if any of the message definitions it contains is incomplete. A message set must move to the frozen state from the locked state, and both state changes must be requested by the same user.
Finalized	A message set and its components in this state cannot be changed or checked out. Finalized state is indicated by the message property Finalized set to true. An attempt to finalize a message set fails if any component of the message set is checked out or if any of the message definitions it contains is incomplete. A message set can move to the finalized state from any other state. However, if it moves from the locked state to the finalized state, both state changes must be requested by the same user.

Once a message set is finalized, no further changes can be made to its contents. However, you can create a new message set based on the finalized message set, within which you can define new messages. You can also make limited changes to the existing messages in the new message set. For more information, see “Message set versioning”.

Message set versioning: A message set can be based on another message set, provided that the message set on which it is based has been finalized. You might want to use this facility to maintain separate versions of a message set, reflecting the evolution of a message set through maintenance and other fixes.

When a message set is based on another message set, it contains a copy of the complete contents of the base message set. Within the new message set, new messages can be defined, and limited modifications can be performed on existing messages. A separate run-time dictionary is produced for the new message set.

A message set can have the same name as another message set in the same message repository if:
- The new message set is based on the message set of the same name.
- The message set on which it is based has a higher level number than any other message set with the same name in the same repository.
- The level number of the new message set is one higher than that of the message set on which it is based.

Creating a message set with the same name as an existing message set: You can create a message set with the same name as the existing message set.

To create a new version:
- Finalize the existing message set
- Remove the existing message set from you workspace
- Create the new message set by:
  - Specifying the same name
  - Specifying the next highest number for the level (increment the old level by 1)
  - Selecting the existing message (used as the base message set) from the drop-down list

When the new message set has been created successfully, you can add the existing message set back into your workspace.

Importing legacy formats

The MRM provides C and COBOL language importers, which you can use to help you create a message set containing message definitions that originate from legacy applications. Such applications are typically those that use C or COBOL data structures to populate messages. The source code of those applications must be available to the import function of the MRM.

The import function parses the source code files, isolates the data structure definitions, and creates logical definitions that correspond to those data structures. It also sets the appropriate CWF properties to define the mapping between the logical definitions and the physical message format, as defined by the C or COBOL data structures.

A compound type is created for each data structure, and elements and element lengths are created for each field within the data structure. For more detailed information about the way in which C and COBOL data structures are interpreted by the MRM language importers, see *MQSeries Integrator Using the Control Center*.

A report is generated by the import function that describes all the definitions that have been created. It includes information about errors or conflicts within the definitions. You can elect to produce this report without committing any changes to the message set. You are recommended to do this and check the report before running the complete import process.

When the import process is complete, you need only to create a message component for each compound type that defines a complete message; all other components are created automatically. However, you are recommended to review your message definition, and edit it if necessary, to ensure that it meets your needs.

See *MQSeries Integrator Using the Control Center* for details of how to import these structures.

Note: You cannot import message sets created by another Control Center user into your Control Center session. This function is only supported by the message set import and export command (**mqsiimpexpmsgset**), which exports (and imports) directly from the Configuration Manager. This is described in the *MQSeries Integrator Administration Guide*.

Generating MRM message set Document Type Definitions (DTDs)

A broker accesses a message set definition in a message dictionary (each message set is deployed in a separate dictionary). Client applications cannot access message dictionaries. They must use one of following two options for accessing the definitions used by the broker.

- You can generate an XML Document Type Definition (DTD) from the message set within the message repository.
- If you have created the MRM definitions by importing C or COBOL data structures, your applications can continue to use those data structures.

For information about either of these tasks, see *MQSeries Integrator Using the Control Center*.

Authorization to work with Messages

To perform any of the tasks described in this chapter, you must:

- Have the correct Control Center user role, which can be one of:
  - **Message flow and message set developer**
  - **All roles**

  For information about setting your user role, see *MQSeries Integrator Using the Control Center*.
- Be a member of the MQSeries Integrator group **mqbrdevt**

The BLOB domain

This section describes messages in the BLOB domain and how to work with them.

Unstructured messages in the BLOB domain

An unstructured message must have a message domain of *BLOB*. It has no known (or defined) structure. These messages can be processed and routed by MQSeries Integrator, but the manipulation that you can perform is very limited.

You can perform some simple manipulation at the message level, and take other actions on the whole message.

Working with unstructured messages in the BLOB domain

The structure and format of an unstructured message are not recognized or understood. Therefore the broker cannot perform any validation on messages in this domain, and a message in the BLOB domain cannot be manipulated within a message flow, except at the message level. For example, you can work with a substring of the message (for example, the 10th to 20th characters) but you cannot work at the field or element level, as these structures are not known.

You can, however, store the full message in a database, you can route the message according to topic (derived from the header), and you can add or remove headers from the message.

Chapter 2. ESQL Overview

This chapter introduces the following:
- What is ESQL
- Four main Database SQL statements
- The three main IBM message nodes, and how they map to Database SQL
- “Case sensitivity of ESQL syntax” on page 33
- “Order of processing in ESQL” on page 33
- “Nulls in Filter and Compute expressions” on page 33
- “Nulls in Boolean expressions” on page 34
- Basic MQSeries Integrator message structure (generic XML)

What is ESQL?

ESQL is similar to Database SQL, is based upon the SQL3 standard, and has been created to process MQSeries Integrator Version 2.0 messages.

MQSeries Integrator Version 2.0 provides message nodes, also known as IBM primitive nodes, to manipulate the content and flow of messages and databases. The functions and parameters of each of these nodes are described in *MQSeries Integrator Using the Control Center*, and in the online help.

Some of the more complex message nodes need to be adapted, by the user, to meet specific needs. This "tailoring" of the messages nodes, is done through ESQL programming language.

Main data manipulation statements in Database SQL

There are four main data manipulation statements in Database SQL:
- SELECT `columnnames` FROM `tablename` WHERE `columnname=value`
- INSERT INTO `tablename (columnnames)` VALUES `(column values)`
- UPDATE `tablename` SET `columnname=value` WHERE `columnname=value`
- DELETE FROM `tablename` WHERE `columnname=value`.

Comparison of main IBM primitive nodes and Database SQL statements

Each of the Database SQL statements can be related to the three principal MQSeries Integrator message nodes:
- Filter
- Compute
- Database

The MQSeries Integrator Version 2.0 **Filter node** is like an IF statement in C or COBOL, and is equivalent to the `WHERE` clause of the ESQL `SELECT` statement. It does not update input or output. The result of a Filter expression can be:
- `True`: If an expression evaluates to `TRUE` then the message will be propagated to the true terminal.
- `False`: If an expression evaluates to `FALSE` then the message will be propagated to the false terminal.

- `Failure`: If a failure is detected in computation, for example attempting to CAST an invalid value to a DATE, then the message will be propagated to the failure terminal.
- `Unknown`: If a path reference is made to an element that does not exist, the computation attempts to resolve an equation with an unknown state and the message will be propagated to the unknown terminal.

An output terminal for each of these results is on the Filter node.

For example a Filter expression looks like:

```
Body.Invoice.Customer.FirstName ='Andrew'
```

evaluates to true in the sample message described in “Message referenced in examples” on page 113.

Notes:

1. `Body` is known as a correlation name
2. `Body` is the MQSeries message (last child of root) and could be specified as either `Root.<parser name>` or `Root.*[LAST]`
3. When using the Filter node, a single expression without a semicolon is used
4. Single quotes are used when comparing string values
5. Periods delimit the levels of the message
6. Nested tags must be fully qualified in outer tags

The MQSeries Integrator Version 2.0 **Compute node** is like the SET part of Database SQL UPDATE. This node manipulates messages, which it processes one at a time. Compute nodes read input messages, and build output messages. A graphical user interface is provided for you to copy all or part of a message from input to output.

Notes:

1. Compute node copies all or part of an input message to an output message, for example:
   ```
   SET OutputRoot = InputRoot;
   ```
2. SET assigns a value to a variable, or copies part of a message tree
3. Semicolon statement terminator is mandatory
4. OutputRoot and InputRoot are correlation names
5. `OutputRoot` is the entire output message
6. `InputRoot` is the entire input message

The MQSeries Integrator Version 2.0 **Database node** allows you to modify a database. Data from an input message is substituted into an ESQL expression which in turn modifies the rows in a database.

Notes:

1. The Database node inserts, updates, or deletes row(s) in a database table
2. Semicolon is mandatory
3. The node cannot change the message being propagated to the output terminal
4. Database is a correlation name
5. The Database node allows all ESQL computations and functionality except manipulating the message body

Case sensitivity of ESQL syntax

The following text describes when the case in which ESQL statements are specified is significant.

- Correlation names are case sensitive

  See "Correlation names" on page 2 for a description of correlation names.
- Parser names are case sensitive
- ESQL language words are not case sensitive
- References to elements in a path are frequently case sensitive. This depends on the parser. All parsers supplied by MQSeries Integrator Version 2.0 are case sensitive. For example, here are some case sensitive path names:

```
InputRoot.Properties.MessageSet
InputRoot.Properties.MessageType
InputRoot.Properties.MessageFormat
InputRoot.Properties.Encoding
InputRoot.Properties.CodedCharSetId
InputRoot.Properties.Transactional
InputRoot.Properties.Persistence
InputRoot.Properties.CreationTime
InputRoot.Properties.ExpirationTime
InputRoot.Properties.Priority
InputRoot.Properties.Topic
```

Order of processing in ESQL

Standard ESQL precedence is:

- Expressions in brackets
- Prefix operators (e.g. minus)
- Multiplication and division
- Concatenation
- Addition and subtraction
- Operations at the same precedence level are applied from left to right

Nulls in Filter and Compute expressions

To explain how nulls operate in ESQL, let us consider the Filter node expression:

```
Body.Wrong.Field >123
```

The field, Body.Wrong.Field, does not exist in the message so it evaluates to `NULL`. The comparison of `NULL` with `123` results in `NULL`. The Filter expression evaluates this to `unknown` and is propagated to the `Unknown` terminal. This result might not be desirable. A better approach would be to test whether the field exists first:

```
Body.Wrong.Field IS NOT NULL
```

An example of how the same approach can be used for the Compute node expression follows:

```
If InputBody.Wrong.Field IS NOT NULL THEN
SET OutputRoot.XML.Wrong.Field = InputBody.Wrong.Field;
ELSE
SET OutputRoot.XML.Wrong.Field = ' ';
END IF;
```

Nulls in Boolean expressions

The table below illustrates the results of AND and OR operations on components ″P″ and ″Q″ in ESQL:

Table 3.

Value of P	Value of Q	P AND Q	P OR Q
T	T	T	T
F	T	F	T
F	F	F	F
U	T	U	T
U	F	F	U
U	U	U	U

Note: In the table, ″T″ indicates the expression is true, ″F″ is false, and ″U″ is unknown.

Chapter 3. ESQL Concepts

This chapter introduces the following:

- "Data types"
- "CASTs" on page 39
- "Predicates" on page 53

Data types

Within a broker, all elements of a message map to a data type. Within the Compute and Database nodes of the broker, it is possible to use intermediate variables to help process a message. Each of the intermediate variables must be declared with a data type before use.

It is not always possible to predict the data type that will result from evaluating an expression. This is because expressions are "compiled" without reference to any kind of message schema, and so some type errors will be not be caught until run-time.

This chapter describes all the data types supported by ESQL. If you would like to see a full list of the different supported data types, please go to "Data types used in ESQL" on page 106.

Numbers

DECIMAL

Decimal numbers have a precision and scale. You cannot define precision and scale when declaring a DECIMAL, as they are assigned automatically. It is only possible to specify precision and scale when casting to a DECIMAL. When casting to DECIMAL, if the precision and scale:

- are not specified, the output defaults to the precision and scale of the input
- are specified, but precision is too small for input, then a run-time error is generated
- are specified, but scale is smaller than input, then truncation occurs

Precision is the total number of digits of a number, and scale is the number of digits to the right of the decimal point.

- The minimum precision is 1
- The maximum precision is 31
- The minimum scale is 0
- The maximum scale is 30

To declare a decimal variable D:

```
DECLARE D DECIMAL;
```

INTEGER, INT

The INTEGER data type stores numbers using 64-bit binary precision; This gives a range of values between -9223372036854775808 and 9223372036854775807. In addition to the normal integer literal format, integer literals can be written in hexadecimal notation, for example `0x1234abcd`.

The hexadecimal letters A to F can be written in uppercase or lowercase, as can the 'x' after the initial zero.

Note: If a literal of this form is too large to be represented as an integer, it is represented as a decimal.

FLOAT
A value of the FLOAT data type is a 64 bit approximation of a real number. A float literal can be defined using the scientific notation, as in `6.6260755e-34`, or as a simple number as in `1.2`.

The case of the "e" is not significant so "E" can be used instead if necessary. "E" identifies the exponent of the number (10 to the power of).

For more information about numeric functions, please see “Numeric functions” on page 82.

Strings

Strings can be character strings, byte strings, or bit strings.

- A string literal of any type must be enclosed in single quotes.
- If you want to include a single quote within a character string literal, you must use another single quote as an escape character.

  For example, the assignment SET X=’he’’was’’’ puts the value `he'was'` into X.

CHARACTER, CHAR
Character string.

BLOB BLOB is a byte string. A byte string is a series of 8-bit bytes that is used to represent arbitrary binary data. A byte string literal is defined using a string of hexadecimal digits, as in the following example:

```
SET X = X'0123456789ABCDEF'
```

There must be an even number of digits in the string, because two digits are required to define each byte. Each digit can be one of the hexadecimal digits. The hexadecimal letters can be specified in uppercase or lowercase.

BIT A bit string is a series of bits used to represent arbitrary binary data that does not contain an exact number of bytes. Bit string literals are defined in a similar way to byte string literals, for example:

```
B'0100101001'
```

Any number of digits, which must be either 0 or 1, can be specified.

For more information about string functions, please see “String manipulation functions” on page 78.

Datetime types

The `DATE`, `GMTTIME`, `GMTTIMESTAMP`, `TIME`, and `TIMESTAMP`, data types are collectively known as **Datetime** data types.

| **DATE** The format of `DATE` data type is the word `DATE` followed by a space, followed by a date in single quotes in the form ’yyyy-mm-dd’. For example:

```
DECLARE MyDate DATE;
SET MyDate = DATE '2000-02-29';
```

| Leading zeroes in the year, month, and day must not be omitted.

GMTTIME

The GMTTIME data type is very similar to TIME, except that the time values are interpreted as values in Greenwich Mean Time. GMTTIME values are defined in much the same way as Time values. For example:

```
DECLARE GetGmttime GMTTIME;
SET GetGmttime = GMTTIME '12:00:00';
```

GMTTIMESTAMP

As with the GMTTIME, the GMTTIMESTAMP data type is very similar to the TIMESTAMP data type, except that the values are interpreted as values in Greenwich Mean Time. GMTTIMESTAMP values are defined in much the same way as TIMESTAMP values, that is as

```
DECLARE GetGMTTimeStamp GMTTIMESTAMP;
SET GetGMTTimeStamp = GMTTIMESTAMP '1999-12-31 23:59:59.999999';
```

TIME The format of TIME data type is the word TIME followed by a space, followed by a time in single quotes in the form ’hh:mm:ss’. For example:

```
DECLARE MyTime;
SET MyTime = TIME '11:49:23.656';
```

Each of the hour, minute and second fields in a TIME literal must always be two digits. The exception is the optional fractional seconds field which, if present, can be up to 6 digits in length.

TIMESTAMP

The format of TIMESTAMP data type is the word TIMESTAMP followed by a space, followed by a timestamp in single quotes in the form ’yyyy-mm-dd hh:mm:ss’. For example:

```
DECLARE MyTimeStamp;
SET MyTimeStamp = TIMESTAMP '1999-12-31 23:59:59';
```

Each of the hour, minute and second fields in a TIMESTAMP literal must always be two digits. The exception is the optional fractional seconds field which, if present, can be up to 6 digits in length.

For more information about datetime functions, please see“Datetime functions” on page 86.

INTERVAL

Data type `INTERVAL`. An `INTERVAL` value represents an interval of time. There are two kinds of `INTERVAL` values:

- One that is specified in years and months.
- One that is specified in days, hours, minutes and seconds (including fractions of a second).

The split between months and days arises because the number of days in each month varies. An interval of one month and a day is not really meaningful, and certainly cannot be sensibly converted into an equivalent interval in numbers of days only.

An interval literal is defined by the following syntax:

```
INTERVAL <interval string> <interval qualifier>
```

The format of interval string and interval qualifier are defined by the following table:

Table 4. Format of interval strings and qualifiers

Interval qualifier	Interval string format	Example
YEAR	'<year>' or '<sign> <year>'	'10'
YEAR TO MONTH	'<year>-<month>' or '<sign> <year>-<month>'	'- 2-06'
MONTH	'<month>' or '<sign> <month>'	'18'
DAY	'<day>' or '<sign> <day>'	'-30'
DAY TO HOUR	'<day> <hour>' or <sign> <day> <hour>'	'1 02'
DAY TO MINUTE	'<day> <hour>:<minute>' or '<sign> <day> <hour>:<minute>'	'1 02:30'
DAY TO SECOND	'<day> <hour>:<minute>:<second>' or '<sign> <day> <hour>:<minute>:<second>'	'1 02:30:15' or '-1 02:30:15.333'
HOUR	'<hour>' or '<sign> <hour>'	'24'
HOUR TO MINUTE	'<hour>:<minute>' or '<sign> <hour>:<minute>'	'1:30'
HOUR TO SECOND	'<hour>:<minute>:<second>' or '<sign> <hour>:<minute>:<second>'	'1:29:59' or '1:29:59.333'
MINUTE	'<minute>' or '<sign> <minute>'	'90'
MINUTE TO SECOND	'<minute>:<second>' or '<sign> <minute>:<second>'	'89:59'
SECOND	'<second>' or '<sign> <second>'	'15' or '15.7'

Where an interval contains both a year and a month value, a hyphen is used between the two values. In this instance, the month value must be within the range `[0, 11]`.

If an interval contains a month value, and no year value, the month value is unconstrained.

A space is used to separate days from the rest of the interval.

If an interval contains more than one of HOUR, MINUTE, and SECOND, then a colon is needed between the values. The values of these fields are constrained as follows:

Field	Valid Values
HOUR	0-23
MINUTE	0-59
SECOND	0-59.999...

Some examples of valid interval values are:
- 72 hours
- 3 days and 23 hours
- 3600 seconds
- 90 minutes and 5 seconds

Some examples of invalid interval values are:
- 3 days and 36 hours

  A day field is specified, so the hours field is constrained to `[0,23]`.
- 1 hour and 90 minutes

  An hour field is specified, so minutes are constrained to `[0,59]`.

Here are some simple examples of interval literals:

```
INTERVAL '1' HOUR
INTERVAL '90' MINUTE
INTERVAL '1-06' YEAR TO MONTH
```

BOOLEAN

A `BOOLEAN` data type can have the values:
- True
- False
- Unknown

A valid filter expression must always return a Boolean value.

CASTs

CAST specifications

A CAST transforms the value of one data type into another data type. Please see "Data types used in ESQL" on page 106 which gives the full list of data types used in ESQL.

Casting

CAST statement

```
►►──CAST──(expression AS──┬─BIT─────────────────────────┬─)──────────────►◄
                          ├─BLOB────────────────────────┤
                          ├─BOOLEAN─────────────────────┤
                          ├─CHAR────────────────────────┤
                          ├─CHARACTER───────────────────┤
                          ├─DATE────────────────────────┤
                          ├─DECIMAL(precision, scale)───┤
                          ├─FLOAT───────────────────────┤
                          ├─GMTTIME─────────────────────┤
                          ├─GMTTIMESTAMP────────────────┤
                          ├─INT─────────────────────────┤
                          ├─INTEGER─────────────────────┤
                          ├─INTERVAL interval qualifier─┤
                          ├─TIME────────────────────────┤
                          └─TIMESTAMP───────────────────┘
```

Note: For *interval qualifier* formats, see Table 4 on page 38.

A CAST specification returns its first operand (source expression) in the type specified by the data type. More complicated conversions can be performed using user defined functions. In all cases if the source expression is NULL, the result will be NULL. If the evaluated source expression is not compatible with the target data type, or if the source expression is of the wrong format, a run-time error is generated.

Supported CASTs

A CAST is not supported between every combination of data types. Those that are supported are listed below, along with the effect of the CAST.

Table 5. Supported CASTs

Source data type	Target data type	Effect
CHARACTER	BOOLEAN	The character string is interpreted in the same way that a boolean literal is interpreted. That is, the character string must be one of the strings TRUE, FALSE, UNKNOWN (in any case combination).
CHARACTER	FLOAT	The character string is interpreted in the same way as a floating point literal is interpreted.
CHARACTER	DATE	The character string must conform to the rules for a date literal or for the date string. That is, the character string can be either DATE '1998-11-09' or 1998-11-09.
CHARACTER	DECIMAL	The character string is interpreted in the same way as an exact numeric literal is interpreted to form a temporary decimal result with a scale and precision defined by the format of the string. This is then converted into a decimal of the specified precision and scale, with a run-time error being generated if the conversion would result in loss of significant digits.
CHARACTER	INTEGER	The character string is interpreted in the same way as an integer literal is interpreted.

Table 5. Supported CASTs (continued)

Source data type	Target data type	Effect
CHARACTER	INTERVAL	The character string must conform to the rules for an interval literal with the same interval qualifier as specified in the CAST specification, or it must conform to the rules for an interval string that apply for the specified interval qualifier.
CHARACTER	TIME	The character string must conform to the rules for a time literal or for the time string. That is, the character string can be either TIME '09:24:15' or 09:24:15.
CHARACTER	TIMESTAMP	The character string must conform to the rules for a timestamp literal or for the timestamp string. That is, the character string can be either TIMESTAMP '1998-11-09 09:24:15' or 1998-11-09 09:24:15.
CHARACTER	GMTTIME	The character string must conform to the rules for a GMT time literal or for the time string. That is, the character string can be either GMTTIME '09:24:15' or 09:24:15.
CHARACTER	GMTTIMESTAMP	The character string must conform to the rules for a GMT timestamp literal or for the timestamp string. That is, the character string can be either GMTTIMESTAMP '1998-11-09 09:24:15' or 1998-11-09 09:24:15.
CHARACTER	BIT	The character string must conform to the rules for a bit string literal or to the rules for the contents of the bit string literal. That is, the character string can be of the form B'bbbbbbb' or bbbbbb (where 'b' can be either '0' or '1').
CHARACTER	BLOB	The character string must conform to the rules for a binary string literal or to the rules for the contents of the binary string literal. That is, the character string can be of the form X'hhhhhh' or hhhhhh (where 'h' can be any hexadecimal digit characters).
BOOLEAN	CHARACTER	If the source value is TRUE, the result is the character string 'TRUE'. If the source value is FALSE, the result is the character string 'FALSE'. Because the UNKNOWN boolean value is the same as the NULL value for booleans, the result will be the NULL character string value if the source value is UNKNOWN.
FLOAT	CHARACTER	The result is the shortest character string that conforms to the definition of an approximate numeric literal and whose mantissa consists of a single digit that is not '0', followed by a period and an unsigned integer, and whose interpreted value is the value of the float.

DATE	CHARACTER	The result is a string conforming to the definition of a date literal, whose interpreted value is the same as the source date value.
		
		For example:
		`CAST(DATE '1998-11-09' AS CHAR)`
		
		would return
		`DATE '1998-11-09'`

DECIMAL	CHARACTER	The result is the shortest character string that conforms to the definition of an exact numeric literal and whose interpreted value is the value of the decimal.

Casting

Table 5. Supported CASTs (continued)

Source data type	Target data type	Effect
INTEGER	CHARACTER	The result is the shortest character string that conforms to the definition of an exact numeric literal and whose interpreted value is the value of the integer.

INTERVAL	CHARACTER	The result is a string conforming to the definition of an interval literal, whose interpreted value is the same as the source interval value.
		
		For example:
		`CAST(INTERVAL '4' YEARS AS CHAR)`
		
		would return
		`INTERVAL '4' YEARS`

TIME	CHARACTER	The result is a string conforming to the definition of a time literal, whose interpreted value is the same as the source time value.
		
		For example:
		`CAST(TIME '09:24:15' AS CHAR)`
		
		would return
		`TIME '09:24:15'`

TIMESTAMP	CHARACTER	The result is a string conforming to the definition of a timestamp literal, whose interpreted value is the same as the source timestamp value.
		
		For example:
		`CAST(TIMESTAMP '1998-11-09 09:24:15' AS CHAR)`
		
		would return
		`TIMESTAMP '1998-11-09 09:24:15'`

GMTTIME	CHARACTER	The result is a string conforming to the definition of a gmttime literal whose interpreted value is the same as the source value. The result string will have the form GMTTIME 'hh:mm:ss'.
GMTTIMESTAMP	CHARACTER	The result is a string conforming to the definition of a gmttimestamp literal whose interpreted value is the same as the source value. The result string will have the form GMTTIMESTAMP 'yyyy-mm-dd hh:mm:ss'.
BIT	CHARACTER	The result is a string conforming to the definition of a bit string literal whose interpreted value is the same as the source value. The result string will have the form B'bbbbbb' (where b is either '0' or '1').
BLOB	CHARACTER	The result is a string conforming to the definition of a binary string literal whose interpreted value is the same as the source value. The result string will have the form X'hhhh' (where h is any hexadecimal digit character).
TIME	GMTTIME	The result value is the source value minus the local time zone displacement (as returned by LOCAL_TIMEZONE). The hours field is calculated modulo 24.
GMTTIME	TIME	The result value is source value plus the local time zone displacement (as returned by LOCAL_TIMEZONE). The hours field is calculated modulo 24.

Table 5. Supported CASTs (continued)

Source data type	Target data type	Effect
GMTTIMESTAMP	TIMESTAMP	The result value is source value plus the local time zone displacement (as returned by LOCAL_TIMEZONE).
TIMESTAMP	GMTTIMESTAMP	The result value is the source value minus the local time zone displacement (as returned by LOCAL_TIMEZONE).
INTEGER or DECIMAL	FLOAT	The number is converted, with rounding if necessary.
FLOAT	INTEGER or DECIMAL	If the conversion would not lead to loss of leading significant digits, the conversion will happen with the number being rounded as necessary. If the conversion would lead to loss of leading significant digits, a run-time error is generated. Loss of significant digits can occur when converting an approximate numeric value to an integer, or to a decimal whose precision is not sufficient.
INTEGER or DECIMAL	INTEGER or DECIMAL	If the conversion would not lead to loss of leading significant digits, the conversion will happen with the number being rounded as necessary. If the conversion would lead to loss of leading significant digits, a run-time error is generated. Loss of significant digits can occur when converting (say) a decimal to another decimal with insufficient precision, or an integer to a decimal with insufficient precision.
INTERVAL	INTERVAL	Year-month intervals are only convertible to year-month intervals, and day-second intervals are only convertible to day-second intervals. The conversion is done by converting the source interval into a scalar in units of the least significant field of the target interval qualifier. This value is then normalized into an interval with the target interval qualifier. For example, to convert an interval which has the qualifier MINUTE TO SECOND into an interval with the qualifier DAY TO HOUR, the source value is converted into a scalar in units of hours, and this value is then normalized into an interval with qualifier DAY TO HOUR.
INTERVAL	INTEGER or DECIMAL	If the interval value has a qualifier that has only one field, the result is an exact numeric with that value. If the interval has a qualifier with more than one field, such as YEAR TO MONTH, a run-time error is generated.
INTEGER or DECIMAL	INTERVAL	If the interval qualifier specified has only one field, the result will be an interval with that qualifier with the field equal to the value of the exact numeric. Otherwise a run-time error is generated.
TIME	TIMESTAMP	The result is a value whose date fields are taken from the current date, and whose time fields are taken from the source time value.
TIMESTAMP	TIME	The result is a value whose fields consist of the time fields of the source timestamp value.
TIMESTAMP	DATE	The result is a value whose fields consist of the date fields of the source timestamp value.

CAST expressions

CAST expressions are used often when dealing with generic XML messages: all fields in a generic XML message have string values, therefore to perform arithmetic calculations or datetime comparisons (for example), the string value of the field must first be cast into a value of the appropriate type.

For example, if you wanted to filter on trade messages where the date of the trade was today, you could write the following expression:

```
CAST(Body.Trade.Date AS DATE) = CURRENT_DATE
```

In this example, the string value of the `Date` field in the message is converted into a date value, and then compared with the current date.

Note: It is not always necessary to cast values between types. Some casts are done implicitly. For example, numbers are implicitly cast between the three numeric types for the purposes of comparison and arithmetic. Character strings are also implicitly cast to other data types for the purposes of comparison.

There are three situations in which a data value of one type is implicitly (that is, without an explicit CAST instruction) cast to another type. The behavior and restrictions of the implicit cast are the same as described above for explicit CAST, except where noted in the following sections.

Implicit CASTs for comparisons

The standard SQL comparison operators >, <, >=, <=, =, <> are supported for comparing two values in ESQL.

When the data types of the two values are not the same, one of them can be implicitly cast to the type of the other to allow the comparison to proceed. In the table below, the vertical axis represents the left hand operand, the horizontal axis represents the right hand operand.

An ″L″ means that the right hand operand is cast to the type of the left hand operand before comparison, an ″R″ means the opposite, an ″X″ means no implicit casting takes place, and a blank means that comparison between the values of the two data types is not supported.

Table 6. Implicit CASTs for comparison

	ukn	**bln**	**int**	**float**	**dec**	**char**	**time**	**gtm**	**date**	**ts**	**gts**	**ivl**	**blob**	**bit**
ukn														
bln		X				L								
int			X	R	R	L								
float			L	X	L	L								
dec			L	R	X	L								
chr		R	R	R	R	X	R	R	R	R	R	R[1]	R	R
tm						L	X	L						
gtm						L	R	X						
dt						L			X	R[2]	R[2]			
ts						L			L[2]	X	L			
gts						L			L[2]	R	X			
ivl						L[1]						X		
blb						L							X	
bit						L								X

Notes:

1. When casting from a character string to an interval, the character string must be of the format ″INTERVAL ′<values>′ <qualifier>″. The format ″<values>″, which is allowable for an explicit CAST, is not allowable here because no qualifier external to the string is supplied.
2. When casting from a DATE to a TIMESTAMP or GMTTIMESTAMP, the time portion of the TIMESTAMP is set to all zero values - ′00:00:00′. This is different to the behavior of the explicit CAST, which sets the time portion to the current time.

Numeric types

The comparison operators operate on all three numeric types.

Character strings

You cannot define an alternative collation order that, for example, collates upper and lowercase characters equally.

Note: When comparing character strings, trailing blanks are not significant so the comparison `'hello' = 'hello '` returns true.

Datetime values

Datetime values are compared in accordance with the natural rules of the Gregorian calendar and clock.

You can compare the time zone you are working in with the GMT time zone. The GMT time zone is converted into a local time zone based on the time zone difference between your local time zone and the GMT time specified.

When you compare your local time with the GMT time, the comparison is based on the difference at a given time on a given date.

Conversion is always based on the value of LOCAL_TIMEZONE. This is because GMTTimestamps are converted to local Timestamps only if it can be done unambiguously. Converting a local Timestamp to a GMTTimestamp has difficulties around the daylight saving cut-over time, and converting between times and GMT times (without date information) has to be done based on the LOCALTIMEZONE value, because you cannot specify which time zone difference to use otherwise.

Booleans

Boolean values can be compared using all or the normal comparison operators. The TRUE value is defined to be greater than the FALSE value. Comparing either value to the UNKNOWN boolean value (which is equivalent to NULL) returns an UNKNOWN result.

Intervals

Intervals are compared by converting the two interval values into intermediate representations, so that both intervals have the same interval qualifier. Year-month intervals can be compared only with other year-month intervals, and day-second intervals can be compared only with other day-second intervals.

For example, if an interval in minutes, such as `INTERVAL '120' MINUTE` is compared with an interval in days to seconds, such as `INTERVAL '0 02:01:00'`, the two intervals are first converted into values that have consistent interval qualifiers, which can then be compared. So, in this example, the first value could be converted into an interval in days to seconds, which will give `INTERVAL '0 02:00:00'` which can then be compared with the second value.

Comparing character strings with other types

If a character string is compared to a value of another type, MQSeries Integrator attempts to cast the character string into a value of the same data type as the other value.

For example, you could write an expression such as:

```
'1234' > 4567
```

The character string on the left would be converted into an integer before the comparison takes place. This behavior reduces some of the need for explicit CAST operators when comparing values derived from a generic XML message with literal values. (For details of explicit casts that are supported, see Table 5 on page 40.) It is this facility that allows you to write an expression such as:

```
Body.Trade.Quantity > 5000
```

In this example, the field reference on the left evaluates to the character string '1000' and, because this is being compared to an integer, that character string is converted into an integer before the comparison takes place.

Note that you must still check whether the price field that you want interpreted as a decimal is greater than a given threshold. You must make sure that the literal you compare it to is a decimal value and not an integer.

For example:

```
Body.Trade.Price > 100
```

would not have the desired effect, because the `Price` field would be converted into an integer, and that conversion would fail because the character string contains a decimal point. However, the following expression will succeed:

```
Body.Trade.Price > 100.00
```

Implicit CASTs for arithmetic operations

Normally the arithmetic operators (+, -, *, and /) operate on operands of the same data type, and return a value of the same data type as the operands. Cases where it is acceptable for the operands to be of different data types, or where the data type of the resulting value is different from the type of the operands, are in Table 7.

Table 7. Implicit CASTs for arithmetic operations

Left operand data type	Right operand data type	Supported operators	Result data type
INTEGER	FLOAT	+, -, *, /	FLOAT[1]
INTEGER	DECIMAL	+, -, *, /	DECIMAL[1]
INTEGER	INTERVAL	*	INTERVAL[4]
FLOAT	INTEGER	+, -, *, /	FLOAT[1]
FLOAT	DECIMAL	+, -, *, /	FLOAT[1]
FLOAT	INTERVAL	*	INTERVAL[4]
DECIMAL	INTEGER	+, -, *, /	DECIMAL[1]
DECIMAL	FLOAT	+, -, *, /	FLOAT[1]
DECIMAL	INTERVAL	*	INTERVAL[4]
TIME	TIME	-	INTERVAL[2]
TIME	GMTTIME	-	INTERVAL[2]
TIME	INTERVAL	+, -	TIME[3]
GMTTIME	TIME	-	INTERVAL[2]
GMTTIME	GMTTIME	-	INTERVAL[2]
GMTTIME	INTERVAL	+, -	GMTTIME[3]
DATE	DATE	-	INTERVAL[2]
DATE	INTERVAL	+, -	DATE[3]
TIMESTAMP	TIMESTAMP	-	INTERVAL[2]
TIMESTAMP	GMTTIMESTAMP	-	INTERVAL[2]
TIMESTAMP	INTERVAL	+, -	TIMESTAMP[3]
GMTTIMESTAMP	TIMESTAMP	-	INTERVAL[2]
GMTTIMESTAMP	GMTTIMESTAMP	-	INTERVAL[2]
GMTTIMESTAMP	INTERVAL	+, -	GMTTIMESTAMP[3]
INTERVAL	INTEGER	*, /	INTERVAL[4]
INTERVAL	FLOAT	*, /	INTERVAL[4]
INTERVAL	DECIMAL	*, /	INTERVAL[4]
INTERVAL	TIME	+	TIME[3]
INTERVAL	GMTTIME	+	GMTTIME[3]
INTERVAL	DATE	+	DATE[3]
INTERVAL	TIMESTAMP	+	TIMESTAMP[3]
INTERVAL	GMTTIMESTAMP	+	GMTTIMESTAMP[3]

Table 7. Implicit CASTs for arithmetic operations (continued)

Left operand data type	Right operand data type	Supported operators	Result data type
Notes: 1. The operand which does not match the data type of the result is cast to the data type of the result before the operation proceeds. For example, if the left operand to an addition operator is an INTEGER, and the right operand is a FLOAT, the left operand is cast to a FLOAT before the addition operation is performed. 2. Subtracting a (GMT)TIME value from a (GMT)TIME value, a DATE value from a DATE value, or a (GMT)TIMESTAMP value from a (GMT)TIMESTAMP value results in an INTERVAL value representing the time interval between the two operands. 3. Adding or subtracting an INTERVAL from a (GMT)TIME, DATE or (GMT)TIMESTAMP value results in a new value of the data type of the non-INTERVAL operand, representing the point in time represented by the original non-INTERVAL plus or minus the length of time represented by the INTERVAL. 4. Multiplying or dividing an INTERVAL by an INTEGER, FLOAT or DECIMAL value results in a new INTERVAL representing the length of time represented by the original multiplied or divided by the factor represented by the non-INTERVAL operand. For example, an INTERVAL value 2 hours 16 minutes multiplied by an FLOAT value of 2.5 results in a new INTERVAL value of 5 hours 40 minutes. The intermediate calculations involved in multiplying or dividing the original INTERVAL are carried out in the data type of the non-INTERVAL, but the individual fields of the INTERVAL (such as HOUR, YEAR, etc.) are always integral, so some rounding errors may occur.			

Implicit CASTs for assignment

Values can be assigned to one of three entities:

- A message field (or equivalent in an exception or destination list)

  Support for implicit conversion between the MQSeries Integrator data types and the message (in its bitstream form) is dependent on the appropriate parser. For example, the XML parser casts everything as character strings before inserting them into the MQSeries message.

- A field in a database table

  MQSeries Integrator converts each of its data types into a suitable standard SQL C data type, as detailed in Table 8. Conversion between this standard SQL C data type, and the data types supported by each DBMS, is dependent on the DBMS. Consult your DBMS documentation for more details.

Table 8. Conversions from MQSeries Integrator to SQL data types

MQSeries Integrator data type	**SQL data type**
NULL, or unknown or invalid value	SQL_NULL_DATA
BOOLEAN	SQL_C_BIT
INTEGER	SQL_C_LONG
FLOAT	SQL_C_DOUBLE
DECIMAL	SQL_C_CHAR[1]
CHARACTER	SQL_C_CHAR
TIME	SQL_C_TIME
GMTTIME	SQL_C_TIME
DATE	SQL_C_DATE
TIMESTAMP	SQL_C_TIMESTAMP
GMTTIMESTAMP	SQL_C_DATE
INTERVAL	not supported[2]
BLOB	SQL_C_BINARY
BIT	not supported[2]

Notes:

1. For convenience, DECIMAL values are passed to the DBMS in character form.
2. There is no suitable standard SQL C data type for INTERVAL or BIT. These must be cast to another data type, such as CHARACTER, if it is necessary to assign them to a database field.

- A scalar variable.

  When assigning to a scalar variable: If the data type of the value being assigned and that of the target variable data type are different, then an implicit cast is attempted with exactly the same restrictions and behavior as specified for the explicit CAST function. The only exception, is when the data type of the variable is INTERVAL or DECIMAL.

  In both these cases, the value being assigned is first cast to a CHARACTER value, then an attempt is made to cast the CHARACTER value to an INTERVAL or DECIMAL. The reason for this is that INTERVAL requires a qualifier and DECIMAL requires a precision and scale: these must be specified in the explicit CAST, but must be obtained from the character string when implicitly casting. Therefore a further restriction is that when implicitly casting to an INTERVAL variable, the character string must be of the form ″INTERVAL ′<values>′ <qualifier>″ - the shortened ″<values>″ form that is acceptable for the explicit CAST is not acceptable here.

Data types of values from external sources

There are two external sources from which data can be extracted by ESQL:
- Message fields
- Database columns

The ESQL data type of message fields depends on the type of the message (XML, Neon, and so on), and the parser used to parse it. The ESQL data type of the value returned by a database column reference depends on the data type of the column in the database.

Table 9 shows which ESQL data types the various built-in DBMS data types (for DB2® (version shipped with the product), SQL Server Version 7.0, Sybase Version 12.0, and Oracle Version 8.1.5) are cast to when they are accessed by MQSeries Integrator.

Table 9. Implicit CASTS for database data types to MQSeries Integrator types

MQSeries Integrator	**DB2**	**SQL Server and Sybase**	**Oracle**
BOOLEAN		BIT	
INTEGER	SMALLINT INTEGER BIGINT	INT SMALLINT TINYINT	
FLOAT	REAL DOUBLE	FLOAT REAL	NUMBER()[1]
DECIMAL	DECIMAL	DECIMAL NUMERIC MONEY SMALLMONEY	NUMBER(P)[1] NUMBER(P,S)[1]
CHARACTER	CHAR VARCHAR CLOB	CHAR VARCHAR TEXT	CHAR NCHAR VARCHAR2 NVARCHAR2 ROWID UROWID LONG CLOB
TIME	TIME		
GMTTIME			
DATE	DATE		
TIMESTAMP	TIMESTAMP	DATETIME SMALLDATETIME	DATE
GMTTIMESTAMP			
INTERVAL			
BLOB	BLOB	BINARY VARBINARY TIMESTAMP IMAGE UNIQUEIDENTIFIER	RAW LONG RAW BLOB
BIT			
not supported	DATALINK GRAPHIC VARGRAPHIC DBCLOB	NTEXT NCHAR NVARCHAR	NCLOB BFILE

Notes:

1. If an Oracle database column with NUMBER data type is defined with an explicit precision (P) and scale (S), then it is cast to an ESQL DECIMAL value; otherwise it is cast to a FLOAT.

   For example, an ESQL statement like this:

   ```
   SET OutputRoot.xxx[]
    = (SELECT T.department FROM Database.personnel AS T);
   ```

   where ″Database.personnel″ resolves to a TINYINT column in a SQL Server database table, results in a list of ESQL INTEGER values being assigned to OutputRoot.xxx.

   By contrast, an identical query where ″Database.personnel″ resolved to a NUMBER() column in an Oracle database results in a list of ESQL FLOAT values being assigned to OutputRoot.xxx.

Predicates

The expression used to configure a Filter node must produce a boolean result. That means that in general it will consist of one kind of predicate. Many of the standard predicates are supported, for example `=`, `<>`, `<`, `>`.

Predicates can be combined using the AND, OR and NOT operators.

BETWEEN predicate

The standard default asymmetric form of the BETWEEN predicate is supported. This requires you to specify the smallest end-point value first, followed by the largest. You can use the ASYMMETRIC keyword, but in its absence the asymmetric form is implied.

If you prefer you can make the BETWEEN predicate symmetric by specifying the optional keyword SYMMETRIC after BETWEEN. In the symmetric form of the predicate, the order in which you specify the two end-point values is not significant. For example, the following two expressions are identical:

```
2 BETWEEN SYMMETRIC 1 AND 3
2 BETWEEN SYMMETRIC 3 AND 1
```

Both expressions return the value "TRUE".

LIKE predicate

The LIKE predicate searches for strings that have a certain pattern. The standard LIKE predicate for performing simple string-pattern matching is supported.

The pattern is specified by a string in which the percent (%) and underscore (_) characters can be used to have special meaning:

- The underscore character _ represents any single character.

  For example, the following predicate finds matches for 'IBM' and for 'IGI', but not for 'International Business Machines' or 'IBM Corp':

  ```
  Body.Trade.Company LIKE 'I__'
  ```
- The percent character % represents a string of zero or more characters.

  For example, the following predicate finds matches for 'IBM', 'IGI', 'International Business Machines', and 'IBM Corp':

  ```
  Body.Trade.Company LIKE 'I%'
  ```

If you want to use the percent and underscore characters within the expressions that are to be matched, you must precede these with an ESCAPE character, which defaults to the backslash (\) character.

For example, the following predicate finds a match for 'IBM_Corp'.

```
Body.Trade.Company LIKE 'IBM\_Corp'
```

You can specify a different escape character by using the ESCAPE clause on the LIKE predicate. For example, you could also specify the previous example like this:

```
Body.Trade.Company LIKE 'IBM$_Corp' ESCAPE '$'
```

IN predicate

An IN predicate of the following form is supported:

```
expression IN (expressiona, expressionb, ..., expressionk)
```

The IN predicate:

- Evaluates to TRUE if the comparison between the first expression and one of the expressions inside the parentheses evaluates to TRUE.
- Evaluates to FALSE if the comparison between the left-hand expression and all of the expressions inside the parentheses evaluate to FALSE.
- Evaluates to UNKNOWN if at least one comparison evaluates to UNKNOWN, and none evaluate to TRUE.

EXISTS predicate

You can use the EXISTS predicate to test whether a WHERE clause successfully matches any items of a repeating structure in the same way as you can use standard database SQL. The form of the EXISTS predicate is:

```
EXISTS(SELECT * FROM something WHERE predicate)
```

Comments

To make a single line comment in your ESQL, place `--` at the beginning of a line. Alternatively, you can start a multiple line comment with `/*` anywhere in ESQL. To terminate a multiple line comment use `*/`.

In arithmetic expressions you must take care not to initiate a line comment accidentally. For example, consider the expression:

```
1 - -2
```

Removing all white space from the expression results in:

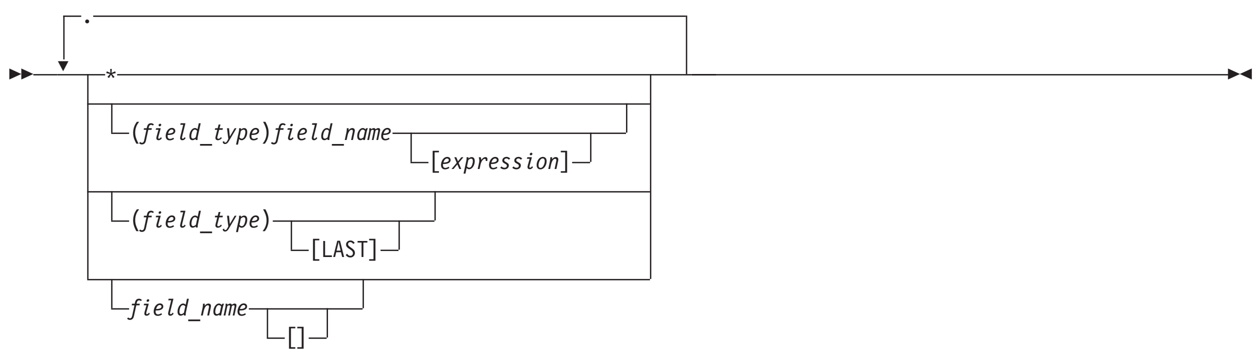
```
1--2
```

which is interpreted as the number 1, followed by a line comment.

Chapter 4. Field references

The full syntax for field references is defined:

Path element



So far, we have explained only those path elements consisting of a `field_name`. Here we will introduce the concept of a `field_type`, which could for example be: `XML.Tag` or `XML.Attr`.

The meaning of the first part of the path element is to define search parameters to find the correct syntax element. If only a field name is supplied, that is an instruction to search for elements that have a field name, regardless of the field type that they might have. Similarly, if a path element specifies only a field type, that is an instruction to search for elements that have the given element type, regardless of the name that they might have.

An asterisk in a path element indicates that all syntax elements should be searched, regardless of the field names or field types. These two options are discussed more in the following sections.

Initial correlation names

For an expression in a Filter node, or for a statement in a Database node, the following correlation names are defined by default:

Root Identifies the root of the message passing though the Filter node.

Body Identifies the last child of the root of the message, that is the "body" of the message. This is just an alias for `Root.*[LAST]`.

See "Anonymous field names" on page 61 for a description of how to use *.

Properties
Identifies the standard properties of the input message.

DestinationList
Identifies the structure which contains the destination list for the message passing through the node.

Field references

ExceptionList
: Identifies the structure which contains the current exception list that the node has access to.

For a Compute node, the initial correlation names are different because there are two messages involved, the input message and the output message. The initial correlation names for a compute name are as follows:

InputRoot
: Identifies the root of the input message

InputBody
: Identifies the "body" of the input message. Like "Body" in a Filter node this is just an alias for "InputRoot.*[LAST]"

InputProperties
: Identifies the standard properties of the input message.

InputDestinationList
: Identifies the structure which contains the destination list for the input message.

InputExceptionList
: Identifies the structure which contains the exception list for the message passing through the node.

OutputRoot
: Identifies the root of the output message.

OutputDestinationList
: Identifies the structure which contains the destination list for the output message. For a description of the format of a destination list, see "Exception and destination list tree structure" on page 8.

  Note that whilst this correlation name is always valid, it only has meaning when the "Compute Mode" property of the Compute node indicates that the Compute node is calculating the destination list.

OutputExceptionList
: Identifies the structure which contains the exception list which the Compute node is generating.

  Note that whilst this correlation name is always valid, it only has meaning when the "Compute Mode" property of the Compute node indicates that the Compute node is calculating the exception list.

  Note that in a Compute node there is no correlation name "OutputBody". New correlation names may be introduced by SELECT expressions (see "SELECT expression" on page 60), quantified predicates, and FOR statements.

Repeating fields

Messages are very likely to contain repeating fields, and these are supported by MQSeries Integrator Version 2.0.

Figure 4 defines a message with some repeating fields that illustrate some of these facilities (for example, Item). This message contains product order information, such as might appear in an invoice message, or an online bookshop purchase.

```
<Invoice>
 <Customer>
  <Name>Albert Einstein</Name>
  <InvoiceAddress>
   <Address>Patent Office</Address>
   <Address>Bern</Address>
   <Address>Switzerland</Address>
  </InvoiceAddress>
 </Customer>
 <Item>
  <Book>
   <Title>Principia Mathmatica</Title>
   <Author>Isaac Newton</Author>
   <ISBN>0-520-0881606</ISBN>
  </Book>
  <Price>60</Price>
  <Quantity>1</Quantity>
 </Item>
 <Item>
  <Book>
   <Title>A Brief History of Time</Title>
   <Author>Stephen Hawking</Author>
   <ISBN>0-553-175211</ISBN>
  </Book>
  <Price>7.99</Price>
  <Quantity>1</Quantity>
 </Item>
 <Item>
  <Stationary>pencil</Stationary>
  <Price>0.20</Price>
  <Quantity>200</Quantity>
 </Item>
 <Item>
  <Stationary>paper</Stationary>
  <Price>1.99</Price>
  <Quantity>100</Quantity>
 </Item>
</Invoice>
```

Figure 4. Repeating fields in a message

Array indices

If you know how many instances there are of a repeating field, and you want to access a specific instance of such a field, you can use an array index as part of a field reference. For example, if you wanted to filter on the first line of an address, to expedite the delivery of an order, you could write an expression such as:

```
Body.Invoice.Customer.InvoiceAddress.Address[1] = 'Patent Office'
```

The array index [1] indicates that it is the first instance of the repeating field that you are interested in (array indices start at 1). An array index such as this can be used at any point in a field reference, so you could, for example, filter on:

```
Body.Invoice."Item"[1].Quantity > 2
```

If you do not know exactly how many instances of a repeating field there are, you can look at the last instance, or a relative field (for example, the third field from the end). You can refer to the last instance of a repeat by using the special `LAST` array index, as in:

```
Body.Invoice."Item"[LAST].Quantity > 2
```

Alternatively, you can use the CARDINALITY function to determine how many instances of a repeating field there are, and use the result to refer to the second to last, for example. The following example shows how to do this:

```
Body.Invoice."Item"[CARDINALITY
               (Body.Invoice."Item"[]) - 2].Quantity > 2
```

In this case, the CARDINALITY function is passed a field reference that ends in `[]`. The meaning of this is "count all instances of the Item field". The [] at the end appears superfluous, because the context indicates that this is the meaning, but its presence is required. This makes the syntax consistent with other instances where it is necessary to refer to "all instances" of something. Remember that array indices start at 1, so the array index in the above example refers to the third-from-last instance of the `Item` field.

Note: If you are using a while loop to process elements of a message, then to improve performance, we recommend that you set a variable to the value of the CARDINALITY prior to entering the loop.

The quantified predicate

It is more likely that you do not know how many instances of a repeating field there are in a message. This is the situation that arises with the `Item` field in the example message in “Message referenced in examples” on page 113. In order to write a filter that takes into account all instances of the `Item` field, you need to use a construct that can iterate over all instances of a repeating field. The quantified predicate allows you to execute a predicate against all instances of a repeating field, and collate the results.

For example, you might want to verify that none of the items that were being ordered had quantity greater than 50. To do this you could write:

```
FOR ALL Body.Invoice.Purchases."Item"[] AS I (I.Quantity <= 50)
```

There are several things to note about this example. Firstly, you have to put double quotation marks around the `Item` in the field reference `Body.Invoice.Item[]`. This is because Item is a reserved word, and the double quotation marks are necessary to prevent it from being interpreted as a keyword and so giving a syntax error.

With the quantified predicate itself, the first thing to note is the "[]" on the end of the field reference after the "FOR ALL". The square brackets tell you that you are iterating over all instances of the `Item` field.

In some cases, this syntax appears unnecessary because you can get that information from the context, but it is done for consistency with other pieces of syntax.

The "AS" clause associates the name `I` with the current instance of the repeating field. This is similar to the concept of iterator classes used in some object oriented languages such as C++. The expression in parentheses is a predicate that is evaluated for each instance of the `Item` field.

A description of this example is:

1. Iterate over all instances of the field Item inside `Body.Invoice`.
2. For each iteration:
   a. Bind the name `I` to the current instance of `Item`.
   b. Evaluate the predicate `I.Quantity <= 50`. If the predicate:
      - Evaluates to TRUE for all of the instances of `Item`, return TRUE.
      - Is FALSE for any instance of `Item`, return FALSE.
      - For a mixture of TRUE and UNKNOWN, it returns UNKNOWN.

The above is a description of how the predicate is evaluated if the "ALL" keyword is used. An alternative is to specify "SOME", or "ANY", which are equivalent. In this case the quantified predicate returns TRUE if the sub-predicate returns TRUE for any instance of the repeating field. Only if the sub-predicate returns FALSE for all instances of the repeating field does the quantified predicate return FALSE. If a mixture of FALSE and UNKNOWN values are returned from the sub-predicate, an overall value of UNKNOWN is returned.

To illustrate this, the following examples are based on the message described in “Message referenced in examples” on page 113. In the following filter expression

```
FOR ANY Body.Invoice.Purchases."Item"[] AS I (I.Title = 'The XML Companion')
```

the sub-predicate evaluates to TRUE, however this next expression returns FALSE:

```
FOR ANY Body.Invoice.Purchases."Item"[] AS I (I.Title = 'C Primer')
```

because the ″C Primer″ is not included on this invoice. If in this instance some of the items in the invoice had not included a book title field, then the sub-predicate would have returned UNKNOWN, and the quantified predicate would have returned the value UNKNOWN.

Great care must be taken to deal with the possibility of null values appearing. You are therefore recommended to write this filter with an explicit check on the existence of the field, as follows:

```
FOR ANY Body.Invoice.Purchases."Item"[] AS I (I.Book IS NOT NULL AND
I.Book.Title = 'C Primer')
```

The "IS NOT NULL" predicate ensures that if an `Item` field does not contain a `Book`, a FALSE value is returned from the sub-predicate.

SELECT expression

Another way of dealing with arbitrary repeats of fields within a message is to use a SELECT expression. ESQL SELECT differs from database SQL SELECT:

- ESQL has THE and ITEM; SQL does not
- ESQL has no SELECT ALL
- ESQL has no SELECT DISTINCT
- ESQL has no GROUP BY or HAVING
- ESQL has no AVG column function

ESQL SELECT can be used to

- Access databases
- Make an output array which is a subset of an input array
- Make an output array which contains just the values of an input array
- Count the number of entries in an array
- Select the minimum or maximum value from a number of entries in an array

Suppose that you want to perform a special action on invoices that have a total order value greater that a certain amount. In order to calculate the total order value of an `Invoice` field, you need to multiply the `Price` fields by the `Quantity` fields in all of the `Items` in the message, and total the result. You can do this using a SELECT expression as follows:

```
(
 SELECT SUM( CAST(I.Price AS DECIMAL) * CAST(I.Quantity AS INTEGER) )
  FROM Body.Invoice."Item"[] AS I
) > 100
```

It is necessary to use CAST expressions to cast the string values of the fields `Price` and `Quantity` into the correct data types. The cast of the `Price` field into a decimal produces a decimal value with the "natural" scale and precision, that is, whatever scale and precision is necessary to represent the number.

The SELECT expression works in a similar way to the quantified predicate, and works in much the same way in which a SELECT works in standard database SQL. The FROM clause specifies what we are iterating over, in this case, all `Item` fields in `Invoice`, and establishes that the current instance of `Item` can be referred to using "`I`". This form of SELECT involves a column function, in this case the SUM function, so the SELECT is evaluated by adding together the results of evaluating the expression inside the SUM function for each `Item` field in the `Invoice`. As with standard SQL, NULL values are ignored by column functions, with the exception of the COUNT column function explained below, and a NULL value is returned by the column function only if there are no non-NULL values to combine.

The other column functions that are provided are MAX, MIN, and COUNT. The COUNT function has two forms which work in different ways with regard to NULLs. In the first form you use it much like the SUM function above, so, for example:

```
SELECT COUNT(I.Quantity)
FROM Body.Invoice."Item"[] AS I
```

This expression returns the number of `Item` fields for which the `Quantity` field is non-NULL. That is, the COUNT function counts non-NULL values, in the same way that the SUM function adds non-NULL values. The alternative way of using the COUNT function is as follows:

```
SELECT COUNT(*)
FROM Body.Invoice."Item"[] AS I
```

Using COUNT(*) counts the total number of `Item` fields, regardless of whether any of the fields is NULL. The above example is in fact equivalent to using the CARDINALITY function, as in:

```
CARDINALITY(Body.Invoice."Item"[])
```

In all of the examples of SELECT given here, just as in standard SQL, a WHERE clause could have been specified to provide filtering on the fields. Note that the SELECT, FROM and WHERE clauses are the only clauses supported. You cannot specify GROUP BY, HAVING, or ORDER BY, nor can you use the ALL or DISTINCT qualifiers in the SELECT clause.

Anonymous field names

It is possible to refer to the array of all children of a particular entity by using a path element of "*". So, for example:

```
InputRoot.*[]
```

is a path that identifies the array of all children of `InputRoot`. This is often used in conjunction with an array subscript to refer to a particular child of an entity by position, rather than by name. So, for example:

`InputRoot.*[LAST]`
Refers to the last child of the root of the input message, that is, the "body" of the message.

`InputRoot.*[1]`
Refers to the first child of the root of the input message.

This is the message properties.

It is useful to be able to find out the name of an entity that has been identified with a path of this kind. To do this, you can use the FIELDNAME function. This function takes a path as its only parameter and returns as a string the field name of the entity to which the path refers. Here are some examples of its usage:

`FIELDNAME(InputRoot.XML)`
Returns 'XML'.

`FIELDNAME(InputBody)`
Returns the name of the last child of InputRoot, which could be 'XML'.

`FIELDNAME(InputRoot.*[LAST])`
Returns the name of the last child of InputRoot, which could be 'XML'.

Field types for the XML parser

There are some instances when it is not enough to identify a field just by name and array subscript. Some message parsers have more complicated models to expose; it is to cope with these cases that an optional type can be associated with element. The message model exposed by the generic XML parser makes heavy use of this facility to deal with the more complicated XML features.

When a type is not present in a path element, it specifies that the type of the syntax element is not important. That is, a path element of "name" matches any syntax element that has a name of "name", regardless of the element type.

Repeating fields

In the same way that a path element can specify a name and not a type, a path element can specify a type and not a name. Such a path element matches any syntax element that has the specified type, regardless of name. An example of this is shown below:

```
FIELDNAME(InputBody.(XML.tag)[1])
```

This example returns the name of the first tag in the body of the message (assuming that it is an XML message). For an example of when it is necessary to use types in paths, consider the following generic XML:

```
<tag1 attr1='abc'>
  <attr1>123</attr1>
</tag1>
```

The path "InputBody.tag1.attr1" refers to the attribute called "attr1", because attributes appear before nested tags in a syntax tree generated by an XML parser. In order to refer to the tag called "attr1" it would be necessary to use a path "InputBody.tag1.(XML.tag)attr1". However, it would be advisable always to include types in these situations to be explicit about which entity is being referred to.

Field types for MQRFH2 headers

When you construct MQRFH2 headers in a compute node, there are two types of fields:

1. Fields in the MQRFH2 header structure (for example, Format and NameValueCCSID
2. Fields in the MQRFH2 NameValue buffer (for example mcd and psc)

To differentiate between these two possible field types, you must insert a value in front of the referenced field in the MQRFH2 field to identify its type (a value for the NameValue buffer is not required because this is the default). The value you must specify for the header structure is (MQRFH2.Field).

For example:

- To create or change an MQRFH2 Format field, specify the following ESQL:
  ```
  SET OutputRoot.MQRFH2.(MQRFH2.Field)Format = 'MQSTR   ';
  ```
- To create or change the psc folder with a topic:
  ```
  SET OutputRoot.MQRFH2.psc.Topic = 'department';
  ```
- To add an MQRFH2 header to an outgoing message that is to be used to make a subscription request:
  ```
  DECLARE I INTEGER;
  SET I = 1;
  WHILE I < CARDINALITY(InputRoot.*[]) DO
   SET OutputRoot.*[I] = InputRoot.*[I];
   SET I=I+1;
  END WHILE;
  SET OutputRoot.MQRFH2.(MQRFH2.Field)Version = 2;
  SET OutputRoot.MQRFH2.(MQRFH2.Field)Format = 'MQSTR';
  SET OutputRoot.MQRFH2.(MQRFH2.Field)NameValueCCSID = 1208;
  SET OutputRoot.MQRFH2.psc.Command = 'RegSub';
  SET OutputRoot.MQRFH2.psc.Topic = "InputRoot"."MRM"."topel";
  SET OutputRoot.MQRFH2.psc.QMgrName = 'DebugQM';
  SET OutputRoot.MQRFH2.psc.QName = 'PUBOUT';
  SET OutputRoot.MQRFH2.psc.RegOpt = 'PersAsPub';
  ```

For more information about the MQRFH2 header please see “The MQRFH2 parser” on page 127

Chapter 5. ESQL statements, expressions and functions

This chapter describes the syntax of ESQL statements, expressions and functions.

Statements and expressions

This first section describes how to use ESQL statements and expressions:

AND and OR

AND and OR are used as part of an expression.

Filter, Compute and Database expressions can involve more than one field comparison. For example:

- Using AND
  ```
  Body.Outertag.Innertag1 = 'ade' AND Body.Outertag.Innertag2 > 100
  ```
- Using OR
  ```
  Body.Outertag.Innertag1 = 'ade' OR Body.Outertag.Innertag2 > 100
  ```
- Using AND and OR
  ```
  Body.Outertag.Innertag1 = 'ade' OR (Body.Outertag.Innertag2 > 100
  AND Body.Outertag.Innertag3[1] = 'abc')
  ```

AS

AS is used as part of an expression.

For more complicated SELECT expressions, an AS clause can be used to give a name to the computed field as in:

```
SELECT T.Price, T.Quantity, T.Price * T.Quantity AS TotalValue
FROM Body.Invoice."Item"[] AS T
```

CASE

For multi-choice settings, you can use the CASE expression.

Both the simple and searched forms of the ESQL CASE expression are supported. You can only use CASE as an expression, not as a statement.

CASE

```
                                       ┌─ELSE NULL──────────────┐
►►──CASE──┬┤ searched-when-clause ├┬──┼────────────────────────┼──END───────────────────►◄
          └┤ simple-when-clause ├──┘  └─ELSE result-expression─┘
```

searched-when-clause:

```
├──WHEN search-condition──THEN──┬─result-expression─┬──────────────────────────────────┤
                                └─NULL──────────────┘
```

simple-when-clause:

```
├──expression──WHEN ──expression──THEN──┬─result-expression─┬──────────────────────────┤
                                        └─NULL──────────────┘
```

If you use the simple form, the value of the expression prior to the first WHEN keyword is tested for equality with the value of the expression following the WHEN keyword. The data type of the expression prior to the first WHEN keyword must therefore be comparable to the data type of each expression following a WHEN keyword.

The CASE expression must have at least one WHEN.

The ELSE is optional. A default ELSE expression is NULL.

A CASE expression is delimited by an END.

The following examples show CASE expressions used as part of a Filter expression:

```
Body.TestCase.Result = CASE SUBSTRING(Body.TestCase.Val1 FROM 1 FOR 1)
WHEN 'A' THEN 'Administration'
WHEN 'B' THEN 'Human Resources'
WHEN 'C' THEN 'Accounting'
WHEN 'D' THEN 'Design'
WHEN 'E' THEN 'Operations'
ELSE 'Manufacturing'
END

Body.TestCase.Result = CASE
WHEN CAST(Body.TestCase.Val1 AS INT) < 15 THEN 'SECONDARY'
WHEN CAST(Body.TestCase.Val1 AS INT) < 19 THEN 'COLLEGE'
END
```

The following two examples show CASE expressions as part of a Filter expression where the CASE is being used within a SELECT against an external database.

```
Body.TestCase.Val1 =
   THE (SELECT ITEM CASE SUBSTRING(B.broker_firstname FROM 1 FOR 1)
        WHEN 'D' THEN 'Dave' ELSE 'noname' END
       FROM Database.broker_details AS B
       WHERE B.broker_id = CAST(Body.TestCase.Val2 AS INT))

CAST(Body.TestCase.Val1 AS INT) =
  THE (SELECT ITEM C.cust_id FROM Database.customer_details AS C WHERE
       C.cust_id = CAST(Body.TestCase.Val2 AS INT) AND
       C.cust_status = CASE WHEN
         CAST(Body.TestCase.Val3 AS INT) = 1 THEN 'A'
         ELSE 'I' END)
```

Example of a searched when clause:

```
SET OutputRoot.XML.Invoice.StoreRecords.BuyTrends.MonthOfYear =
   CASE
      WHEN SUBSTRING(InputBody.Invoice.InvoiceDate FROM 6 FOR 2) = '01' THEN 'January'
      WHEN SUBSTRING(InputBody.Invoice.InvoiceDate FROM 6 FOR 2) = '02' THEN 'February'
      WHEN SUBSTRING(InputBody.Invoice.InvoiceDate FROM 6 FOR 2) = '03' THEN 'March'
      WHEN SUBSTRING(InputBody.Invoice.InvoiceDate FROM 6 FOR 2) = '04' THEN 'April'
      WHEN SUBSTRING(InputBody.Invoice.InvoiceDate FROM 6 FOR 2) = '05' THEN 'May'
      WHEN SUBSTRING(InputBody.Invoice.InvoiceDate FROM 6 FOR 2) = '06' THEN 'June'
      ELSE 'Second half of year'
   END;
```

Example of a simple when clause:

```
SET OutputRoot.XML.Invoice.StoreRecords.BuyTrends.MonthOfYear =
   CASE SUBSTRING(InputBody.Invoice.InvoiceDate FROM 6 FOR 2)
      WHEN '01' THEN 'January'
      WHEN '02' THEN 'February'
      WHEN '03' THEN 'March'
      WHEN '04' THEN 'April'
      WHEN '05' THEN 'May'
      WHEN '06' THEN 'June'
      ELSE 'Second half of year'
   END;
```

CAST

Please see “CASTs” on page 39 for more information.

DECLARE

The DECLARE statement declares a simple scalar variable that can be used to store some temporary value. The variable name cannot be a reserved word. The syntax of the declare statement is:

```
DECLARE variable_name data type;
```

where data type is one of the following:

DECLARE statement

```
►►──DECLARE──(variable name──┬─BIT─────────────────────────┬─)───────────────►◄
                             ├─BLOB────────────────────────┤
                             ├─BOOLEAN─────────────────────┤
                             ├─CHAR────────────────────────┤
                             ├─CHARACTER───────────────────┤
                             ├─DATE────────────────────────┤
                             ├─DECIMAL(precision, scale)───┤
                             ├─FLOAT───────────────────────┤
                             ├─GMTTIME─────────────────────┤
                             ├─GMTTIMESTAMP────────────────┤
                             ├─INT─────────────────────────┤
                             ├─INTEGER─────────────────────┤
                             ├─INTERVAL ───────────────────┤
                             ├─TIME────────────────────────┤
                             └─TIMESTAMP───────────────────┘
```

Declared variables have a NULL value until they are initialized. For an example of the DECLARE statement see the example in the description of the WHILE statement.

DELETE

A DELETE statement deletes rows from a table in an external database based on a search condition.

DELETE statement

```
►►──DELETE FROM──database.──┬───────────────┬──table_name──────────────►
                            └─schema_name.──┘

►──┬─────────────────────┬──┬─────────────────────────┬─────────────────►◄
   └─AS correlation_name─┘  └─WHERE search_condition──┘
```

A correlation name is created that can be used inside the search condition to refer to the values of columns in the table. This correlation name is either the name of the table (without the data source qualifier) or the explicit qualifier specified.

Example

Suppose that you have Database node that has been configured with a connection to a table SHAREHOLDINGS. The following statement could be written to configure the Database node:

```
DELETE FROM Database.SHAREHOLDINGS AS H
 WHERE H.ACCOUNTNO = Body.AccountNumber;
```

This will remove all rows from the SHAREHOLDINGS table where the ACCOUNTNO field in the table is equal to the AccountNumber in the message.

EVAL

EVAL expression

```
►►—EVAL—( expression )——————————————————————————————————————————————►◄
```

You can use EVAL in two ways:

1. As a complete ESQL statement.
2. As an expression that forms part of an ESQL statement.

EVAL takes one parameter in the form of an expression, and it evaluates this expression and casts the resulting value to a character string if it is not one already. The expression that is passed to EVAL must therefore be able to be represented as a character string.

After this first stage evaluation is complete, the behavior of EVAL depends on whether it is being used as a complete ESQL statement, or in place of an expression that forms part of an ESQL statement:

1. If it is a complete ESQL statement, the character string derived from the first stage evaluation is executed as if it were an ESQL statement.
2. If it is an expression that forms part of an ESQL statement, the character string is evaluated as if it were an expression and EVAL returns the result.

In the following examples A and B are integer scalar variables, and scalarVar1, operatorAsString are character string scalar variables.

The following statements are valid uses of EVAL:

- `SET OutputRoot.XML.Data.Result = EVAL(A+B);`

  The expression A+B is acceptable because, although it returns an integer value, integer values are representable as character strings, and the necessary cast is therefore performed before EVAL continues with its second stage of evaluation.
- `SET OutputRoot.XML.Data.Result  = EVAL('A' || operatorAsString || 'B');`
- `EVAL('SET ' || scalarVar1 || ' = 2;');`

  The semicolon included at the end of the final string literal is necessary because if EVAL is being used in place of an ESQL statement, then its first stage evaluation must return a string that represents a valid ESQL statement, including the terminating semicolon.

The real power of EVAL is that it allows you to dynamically construct ESQL statements or expressions. In the second and third valid examples shown, for example, the value of scalarVar1 or operatorAsString can be set according to the value of an incoming message field, or other dynamic value, thus allowing you to effectively control what ESQL is executed without requiring a potentially lengthy IF...THEN ladder.

However, you must consider the performance implications in using EVAL - dynamic construction and execution of statements or expressions is necessarily more time-consuming than simply executing pre-constructed ones. If performance is vital, you might find it preferable to write more specific, but faster, ESQL.

The following are not valid uses of EVAL:

- `SET EVAL(scalarVar1) = 2;`

  In this example, EVAL is being used to replace a field reference, not an expression.
- `SET OutputRoot.XML.Data.Result[] = EVAL((SELECT T.x FROM Database.y AS T));`

  In this example, the (SELECT T.x FROM Database.y) passed to EVAL returns a list, which is not representable as a character string.

The following example is acceptable because `'(SELECT T.x FROM Database.y AS T)'` is a character string literal, not an expression in itself, and therefore is representable as a character string.

```
SET OutputRoot.XML.Data.Result[]
 = EVAL('(SELECT T.x FROM Database.y AS T)');
```

The following example shows how to use EVAL to translate XML attributes to tags:

```
DECLARE thePath CHARACTER;
DECLARE newPath CHARACTER;
DECLARE theData CHARACTER;
DECLARE theChildName CHARACTER;
DECLARE theChildType INTEGER;
DECLARE J INTEGER;
DECLARE C INTEGER;
SET thePath = FIELDNAME(InputRoot.XML.*[1]);
EVAL('SET OutputRoot.XML.' || thePath || ' = CAST(InputRoot.XML.*[1] AS CHAR);');
WHILE (thePath IS NOT NULL) DO
EVAL('SET I = CARDINALITY(InputRoot.XML.' || thePath || '.*[]);');
SET J = 1;
WHILE ( J <= I ) DO
EVAL('SET theChildType = FIELDTYPE(InputRoot.XML.' || thePath || '.*[J]);');
EVAL('SET theChildName = FIELDNAME(InputRoot.XML.' || thePath || '.*[J]);');
      /* check for MQSIv2 reserved words (eg ITEM) */
IF UPPER(theChildName) IN ('ITEM') THEN
         SET theChildName = '"' || theChildName || '"';
  END IF;
END IF
   IF (theChildType = 0x01000000) THEN
VAL('SET C = CARDINALITY(OutputRoot.XML.' || thePath || '.' || theChildName || '[]) + 1;');
SET newPath = thePath || '.' || theChildName || '[' || CAST(C AS CHAR) || ']';
EVAL('SET theData = InputRoot.XML.' || newPath || ';');
EVAL('SET OutputRoot.XML.' || newPath || ' = theData;');
SET OutputRoot.XML.theStack.entry[CARDINALITY(OutputRoot.XML.theStack.*[])+1] = thePath || '.' || th
  END IF;
IF (theChildType = 0x03000000) THEN
EVAL('SET theData = InputRoot.XML.' || thePath || '.*[' || CAST(J AS CHAR) || '];');
EVAL('SET OutputRoot.XML.' || thePath || '.' || theChildName || '[LAST] = ''' || theData || ''';');
END IF;
SET J = J + 1;
END WHILE;
SET thePath = OutputRoot.XML.theStack.entry[1];
SET OutputRoot.XML.theStack.entry[1] = NULL;
END WHILE;
SET OutputRoot.XML.theStack = NULL;
```

IF

An IF statement controls execution of one set of statements or another based on the result of evaluating a predicate.

Note that if the control expression evaluates to UNKNOWN, the "else" statements are executed; UNKNOWN is treated the same as FALSE.

The IF statement takes one of the following forms:

```
IF condition THEN
  controlled statements
END IF;
```

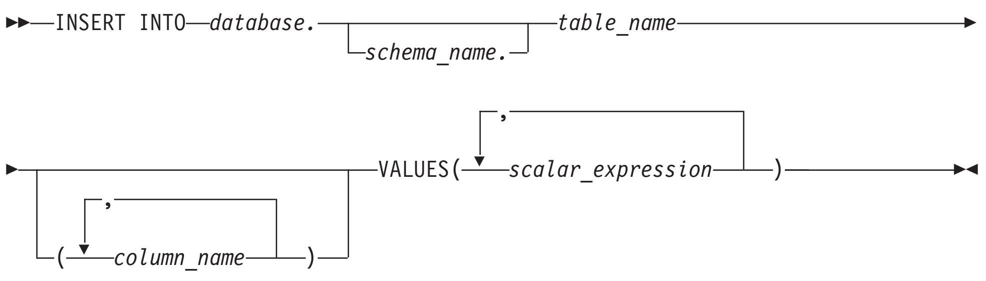
or:

```
IF condition THEN
  controlled statements 1
ELSE
  controlled statements 2
END IF;
```

INSERT

An INSERT statement can be used to add new rows to an external database.

INSERT statement



The optional column name list identifies a list of columns in the target table into which values are to be inserted. Any columns not mentioned in the column name list will have their default values inserted.

A run-time error can be generated if problems occur during the insert operation. For example the database table may have constraints defined which the insert operation may violate. In these cases, an attempt will be made to propagate the original message that was received by the node to the failure terminal on the node.

Example

The following example assumes that the dataSource property on the Database node has been configured and that the database identified by that datasource has a table called "TABLE1" with columns A, B, and C. Given a message that has the following generic XML body:

```
<A>
 <B>1</B>
 <C>2</C>
 <D>3</D>
</A>
```

the following INSERT statement will insert a new row into the table with the values (1, 2, 3).

```
INSERT INTO Database.TABLE1(A, B, C) VALUES (Body.A.B, Body.A.C, Body.A.D);
```

NULL with AND and OR

Below you will find examples of how NULL operates with AND and OR:

- Implied NULL comparison involving AND
  ```
  Body.Invoice.Customer.LastName = 'ade' AND Body.Wrong.Field > 100
  ```
  Evaluates to FALSE.
- Implied NULL comparison involving OR
  ```
  Body.Invoice.Customer.LastName = 'ade' OR Body.Wrong.Field > 100
  ```
  Evaluates to UNKNOWN.
- Explicit NULL comparison involving AND
  ```
  Body.Invoice.Customer.LastName = 'Smith' AND Body.Wrong.Field IS NULL
  ```
  Evaluates to TRUE.
- Explicit NULL comparison involving OR
  ```
  Body.Invoice.Customer.LastName = 'ade' OR Body.Wrong.Field IS NULL
  ```
  Evaluates to TRUE.

PASSTHRU

The PASSTHRU function allows the coding of ESQL statements which:

- By-pass the MQSeries Integrator 2.0 Broker Parser
- Go straight to the configured back-end database
- Execute a coded statement

The first parameter of PASSTHRU must be a valid ESQL expression containing your database syntax. PASSTHRU allows you to use database syntax not normally supported by ESQL.

The behavior of the PASSTHRU function depends on whether it is passed one, two, or more parameters. The first parameter of the PASSTHRU function must always be an ESQL expression that either is, or evaluates to, a string. You must use question marks in the string to denote where any parameter substitution is required.

If only one other parameter is passed, that parameter evaluates to one of the following:

- A single scalar value. If this is the case, it is inserted into the first parameter marker.
- A list of values. If this is the case, the list items are inserted in order into each of the parameter markers within the string.

If two or more other parameters are passed, each parameter is bound to the corresponding question mark in the statement string: that is, the first parameter is bound to the first question mark, the second parameter is bound to the second question mark, and so on.

Here are some examples that illustrate different ways of using the PASSTHRU statement:

```
SET OutputRoot.XML.Result.Data[] =
          PASSTHRU('SELECT * FROM user1.stocktable');
```

```
PASSTHRU('DELETE FROM user2.AccountData WHERE AccountId =
          ?', InputBody.Data.Account.Id);
```

```
SET OutputRoot.XML.Result.Data
    = PASSTHRU('SELECT AccountNum FROM user2.AccountData
      WHERE AccountId = ?', InputBody.Data.Account.Id);
```

```
SET OutputRoot.XML.Result.Data[]
  = PASSTHRU('SELECT AccountNum FROM user2.AccountData
             WHERE AccountId IN (? , ? , ?)',
           InputBody.Data.Account.Id[]);
```

```
PASSTHRU('INSERT INTO user1.stocktable (stock_id, quantity)
         values (?, ?)', InputBody.Transaction.Id,
                           InputBody.Transaction.Quantity);
```

You must take the following points into consideration when you construct string literals that you will use as the first parameter of the PASSTHRU function:

- If the ESQL statement that you want to execute against the database contains a single quote, you must escape the single quote when you define the string literal.

  For example, if you want to execute the following ESQL statement:

  ```
  INSERT INTO TABLE1 VALUES('abc', 'def')
  ```

  you can use the following PASSTHRU statement:

  ```
  PASSTHRU('INSERT INTO TABLE1 VALUES(''abc'', ''def'')');
  ```

  The use of double single quotation marks is required for a definition of a string literal containing a single quote.
- You must include trailing spaces in the individual string literals to avoid defining a string containing the text:

  ```
  'SELECT a, b, c FROM table1 WHERE d = 123'
  ```

Considerations when calling stored procedures

If you decide to use the PASSTHRU statement to call stored procedures, the following considerations must be noted:

- MQSeries Integrator 2.0 uses Open Database Connectivity (ODBC) to connect to databases. ODBC Version 1 provides support for Stored Procedure calls using the following ODBC escape sequence:

  `call` procedure name `[([`parameter`][,[`parameter`]]...)]}`.

  Only the escape sequence described above is supported for the input parameters of the PASSTHRU function in MQSeries Integrator 2.0.
- Using the **SQL CALL** facility, a database Stored Procedure can be called. This procedure behaves as if a sequence of in-line SQL statements are being executed
- Stored procedures can exist either
  - Individually (supported by both DB2 and Oracle). This would coded as follows:

    ```
    PASSTHRU('{call proc_insert_comp(?,?)}',InputBody.Test.Company,InputBody.Test.Price);
    ```

or

– As part of a collective which is accessed using a **Package** mechanism (supported by Oracle). This would be coded as follows:

```
PASSTHRU('{call share_management.add_share(?,?)}',
                    InputBody.Test.Company,InputBody.Test.Price);
```

- When writing Stored Procedures they can be either
  - **commital** (supported by Oracle): the procedure logic takes explicit commit and rollback actions.

    If a message flow rollback occurs, the database operations are committed. This is consistent with the behavior of the Database and Warehouse nodes which have a transaction attribute setting of commit.

    or
  - **non-commital** (supported by both DB2 and Oracle): the procedure logic does not take explicit commit and rollback action.

    If a message flow rollback occurs, the database operations are rolled back. This is consistent with the behavior of the Database and Warehouse nodes which have a transaction attribute setting of automatic.

Note: Stored Procedure calls, whether **commital** or **non-commital**, will affect any database operations (and subsequent outcome) if a message flow rollback occurs.

Limitations

There are some limitations when using PASSTHRU to call stored procedures. To illustrate the limitations, please consider the following example:

```
PASSTHRU('{call proc_delete_comp(?)}',InputBody.Test.Company);
```

1. MQSeries Integrator Version 2.0 only supports input parameters.
2. SqlMoreResults cannot be used by MQSeries Integrator Version 2.0 to retrieve result sets.

SELECT

Select statements are discussed in “SELECT expression” on page 60 and “Chapter 6. Complex SELECTs: ROWs and LISTs” on page 91.

SET

Used in the Compute and Database nodes, the general form of an assignment statement is either:

```
SET field_reference = expression;
```

or

```
SET variable = expression;
```

Set statements are all semicolon (";") terminated. The semicolon is a terminator, and not a separator, so it must appear at the end of every statement, even the last one.

The field reference or variable on the left of the assignment identifies either the field in the output message which is to be set, or an ESQL variable. It must start with "OutputRoot", or "OutputDestinationList", or "OutputExceptionList". The field referenced will be created if it doesn't already exist in the output message; if the field already exists in the output message, its value will be overwritten. Note that when array indices are used in the field reference, only one instance of a particular field will ever get created, so for example if you write as assignment statement starting:

```
SET OutputRoot.XML.Message.Structure[2].Field = ...
```

at least one instance of "Structure" must already exist in the message. That is, the only elements in the tree that are created are ones on a direct path from the root to the element identified by the field reference. A common example of Compute node will consist of a node which makes a modification to a message, either changing a field, or maybe adding a new field to the original message. Such a Compute node would be programmed by statements like the following:

```
SET OutputRoot = InputRoot;
SET OutputRoot.XML.Order.Name = UPPER(InputRoot.XML.Order.Name);
```

This example simply puts one field in the message into uppercase. The first statement constructs an output message which is a complete copy of the input message (as per the very first simple example). The second statement sets the value of the "Order.Name" field (which it is assumed the message flow writer knows will exist in the input message) to a new value, as defined by the expression on the right.

It is interesting to note what the effect is if the Order.Name field hadn't existed in the original input message. Because it didn't exist in the input message, it won't exist in the output message as generated by the first statement. The expression on the right of the second statement will return NULL, because the field referenced inside the UPPER function call does not exist). Assigning the NULL value to a field has the effect of deleting it if it already exists, and so the effect is that the second statement has no effect.

All the following result in a tree copy:

```
SET OutputRoot = InputRoot;
SET OutputRoot.MQMD = InputRoot.MQMD;
SET OutputRoot.XML.InputMessage = InputRoot;
```

Compute node

The Compute node and the Filter node share a common expression syntax. In its simplest form, a Compute node provides a way of building up a new message using a set of assignment statements. The expressions that appear on the right hand side of the assignment, that is, the source expressions, are expressions of exactly the same form as can appear in a Filter node. But, they are not restricted to returning single boolean values in the same way that a filter expression is.

A Compute node works by constructing a tree representation of a new message based on a list of assignment statements. A new message is always (at least conceptually) constructed, because the message passed to the node must be preserved in its original form (it is not permissible in a message flow to modify the information passed back "upstream"). The simplest possible Compute node simply constructs a new message as an exact copy of the input message. Such a Compute node would consist of the following statement

```
SET OutputRoot = InputRoot;
```

Because there are two messages involved in a Compute node, it is not sufficient to refer to "Root" as can be done in a Filter node where there is only one message. Instead you have to refer to "InputRoot" and "OutputRoot" in a Compute node. You can also refer to "InputBody" in a Compute node in the same way that you can refer to "Body" in a Filter node, though you cannot refer to "OutputBody", because there is no fixed concept of what the "body" of the output message is until the output message has been fully constructed.

The above example causes a complete copy of the input message to be propagated to the output terminal of the Compute node because when the right hand side of an assignment statement consists of a field reference, a complete recursive tree copy is performed to duplicate the tree representation of the input message. For more information about how you might use a Compute node see *MQSeries Integrator Using the Control Center*.

Copying messages between parsers: Compute node expressions can copy part of an input message to an output message. The results of such a copy depend upon the type of input and output parsers involved.

Like parsers: Where both the source and target messages have the same folder structure at root level, a **like-parser-copy** is performed. For example:

```
SET OutputRoot.MQMD = InputRoot.MQMD;
```

will result in all the children in the MQMD folder of the input message being copied to the MQMD folder of the output message.

Another example of a tree structure which will support a like-parser-copy is:

```
SET OutputRoot.XML.Data.Account = InputRoot.XML.Customer.Bank.Data;
```

Unlike parsers: Where the source and target messages have different folder structures at root level, it is not possible to make an exact copy of the message source. Instead, the **unlike-parser-copy** views the source message as a set of nested folders terminated by a leaf name-value pair. For example, copying the following message from XML to MRM:

```
<Name3><Name31>Value31</Name31>Value32</Name3>
```

will produce a name element ″Name3″, and a name-value element called ″Name31″ with the value ″Value31″.

Note: The second XML pcdata (Value32) cannot be represented and will be discarded.

The unlike-parser-copy scans the source tree, and copies folders, also known as name elements, and leaf name-value pairs. Everything else, including elements flagged as ″special″ by the source parser, will not be copied.

An example of a tree structure resulting in an unlike-parser-copy is:

```
SET OutputRoot.MRM.Data.Account = InputRoot.XML.Data.Account;
```

Note: If the algorithum used to make an unlike-parser-copy does not suit your tree structure, it might be necessary to further qualify the source field to restrict the amount of tree copied.

Using the compute node for data conversion: You can use the ESQL within a compute node to provide data conversion for code page and encoding of messages. You must set MQMD CCSID and Encoding fields of the output message, plus the CCSID and Encoding of any headers, to the required target value.

The following example illustrates what is required for a CWF message to pass from MQSeries Integrator to IMS on OS/390®.

1. You have defined the input message in XML and are using an MQRFH2 header. The header must be removed before the message is passed to IMS.
2. The message passed to IMS must have MQIIH header, and must be in the OS/390 codepage. This message is defined in the MRM and has identifier m_IMS1. The PIC X fields in this message must be defined as logical type string for EBCDIC <-> ASCII conversion to take place. If they are logical type binary, no data conversion occurs.
3. The message received from IMS is also defined in the MRM and has identifier m_IMS2. The PIC X fields in this message must be defined as logical type string for EBCDIC <-> ASCII conversion to take place. If they are logical type binary, no data conversion occurs.
4. The reply message must be converted to the Windows NT codepage. The MQIIH header is retained on this message.
5. You have created a message flow that contains:
   a. The outbound flow, **MQInput1 —> Compute1 —> MQOutput1**.
   b. The inbound flow, **MQInput2 —> Compute2 —> MQOutput2**.
6. You must set up the ESQL in Compute1 (outbound) node as follows (specifying the MessageSet id. that is created for you):

```
DECLARE I INTEGER;
SET I = 1;
WHILE I < CARDINALITY(InputRoot.*[]) - 1 DO
  SET OutputRoot.*[I] = InputRoot.*[I];
  SET I=I+1;
END WHILE;
SET OutputRoot.MQMD.CodedCharSetId = 500;
SET OutputRoot.MQMD.Encoding = 785;
SET OutputRoot.MQMD.Format = 'MQIMS   ';
SET OutputRoot.MQIIH.StrucId = 'IIH ';
SET OutputRoot.MQIIH.Version = 1;
SET OutputRoot.MQIIH.StrucLength = 84;
SET OutputRoot.MQIIH.Encoding = 785;
SET OutputRoot.MQIIH.CodedCharSetId = 500;
SET OutputRoot.MQIIH.Format = 'MQIMSVS ';
SET OutputRoot.MQIIH.Flags = 0;
SET OutputRoot.MQIIH.LTermOverride = '        ';
SET OutputRoot.MQIIH.MFSMapName = '        ';
SET OutputRoot.MQIIH.ReplyToFormat = 'MQIMSVS ';
SET OutputRoot.MQIIH.Authenticator = '        ';
SET OutputRoot.MQIIH.TranInstanceId = X'00000000000000000000000000000000';
SET OutputRoot.MQIIH.TranState = ' ';
SET OutputRoot.MQIIH.CommitMode = '0';
SET OutputRoot.MQIIH.SecurityScope = 'C';
SET OutputRoot.MQIIH.Reserved = ' ';
SET OutputRoot.MRM.e_elen08 = 30;
SET OutputRoot.MRM.e_elen09 = 0;
SET OutputRoot.MRM.e_string08 = InputBody.e_string01;
SET OutputRoot.MRM.e_binary02 = X'31323334353637383940';
SET OutputRoot.Properties.MessageDomain = 'MRM';
SET OutputRoot.Properties.MessageSet = 'DHCJOEG072001';
SET OutputRoot.Properties.MessageType = 'm_IMS1';
SET OutputRoot.Properties.MessageFormat = 'CWF';
```

7. You must set up the ESQL in Compute2 (inbound) node as follows (specifying the MessageSet id. that is created for you):

```
DECLARE I INTEGER;
SET I = 1;
WHILE I < CARDINALITY(InputRoot.*[]) DO
  SET OutputRoot.*[I] = InputRoot.*[I];
  SET I=I+1;
END WHILE;
SET OutputRoot.MQMD.CodedCharSetId = 437;
SET OutputRoot.MQMD.Encoding = 546;
SET OutputRoot.MQMD.Format = 'MQIMS   ';
SET OutputRoot.MQIIH.CodedCharSetId = 437;
SET OutputRoot.MQIIH.Encoding = 546;
SET OutputRoot.MQIIH.Format = '        ';
SET OutputRoot.MRM = InputBody;
SET OutputRoot.Properties.MessageDomain = 'MRM';
SET OutputRoot.Properties.MessageSet = 'DHCJOEG072001';
SET OutputRoot.Properties.MessageType = 'm_IMS2';
SET OutputRoot.Properties.MessageFormat = 'CWF';
```

You do not have to set any specific values for the MQInput1 node properties because the message and message set are identified in the MQRFH2 header, and no conversion is required.

You must set values for Domain, set, type and format in the MQInput node for the inbound message flow (MQInput2). You do not need to set conversion parameters.

Using the compute node for message transformation: You can use the ESQL within a compute node to transform a message from one format to another.

For example, if you want to transform a generic XML message into an MRM message, you can:

1. Add the MRM message to Output Messages on the basic tab of the compute node properties dialog.
2. If you want to retain the headers of the message, select **Copy message headers**.
3. Select the **Use as message body** check box. This generates ESQL similar to:
   ```
   SET OutputRoot.Properties.MessageSet = 'DHOP5F709S001';
   SET OutputRoot.Properties.MessageType = 'test_message';
   ```

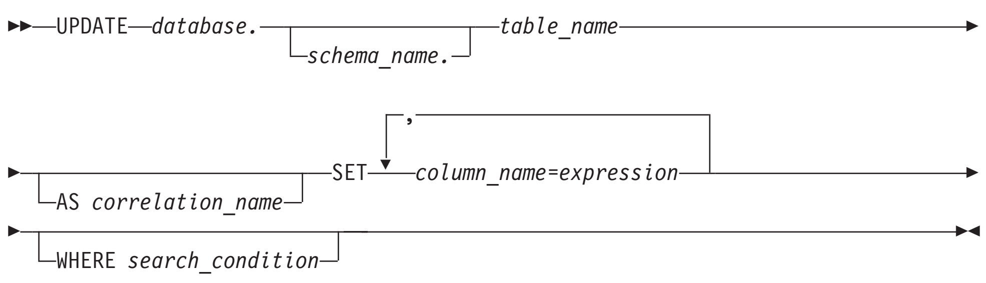
   Note that it is the message identifier that is required in the MessageType field.
4. Specify the output format of the message (this must be one of CWF, PDF, or XML). For example:
   ```
   SET OutputRoot.Properties.MessageFormat = 'CWF';
   ```
5. Specify the new message domain (in this transformation, this step is not necessary because MRM is the default, but you are recommended to include this for completeness):
   ```
   SET OutputRoot.Properties.MessageDomain = 'MRM';
   ```
6. Create ESQL statements to populate your output message, either manually or by using drag and drop to generate automatic mappings.

The same principles apply for other message transformations.

UPDATE

An UPDATE statement will update the values of specified columns in a table in an external database.

UPDATE statement



Example 1

This example updates the PRICE column of the row in the STOCKPRICES table whose COMPANY column matched the value given in the Company field in the message that the Database node is processing.

```
UPDATE Database.StockPrices AS SP
 SET PRICE = Body.Message.StockPrice
 WHERE SP.COMPANY =Body.Message.Company
```

Example 2

In this example, the "INV.QUANTITY" in the right hand side of the assignment refers to the previous value of the column before any updates have taken place.

```
UPDATE Database.INVENTORY AS INV
 SET QUANTITY = INV.QUANTITY - Body.Message.QuantitySold
 WHERE INV.ITEMNUMBER = Body.Message.ItemNumber
```

Example 3

This example shows multiple column updates.

```
UPDATE Database.table AS T
 SET column1 = T.column1+1,
     column2 = T.column2+1;
```

Compare the syntax to the way you assign to multiple fields in a Compute node:

```
SET field = expression;
```

Note also the form of the assignment: the column on the left of the assignment must be a single identifier. It must **not** be qualified with a table name or correlation name. In contrast, any column references to the right of the assignment **must** be qualified with a table name or correlation name.

WHILE

A WHILE statement executes a sequence of statements repeatedly while the controlling predicate evaluates to TRUE. The same caveats apply to using the WHILE statement as apply in any language, that is, it is up to you to ensure that the loop will terminate. Note that if the control predicate evaluates to UNKNOWN the loop terminates: UNKNOWN and FALSE are treated in the same way in this respect. The WHILE statement takes the following form:

```
WHILE predicate DO
  controlled statements
END WHILE;
```

For example:

WHILE

```
DECLARE I INTEGER;
SET I = 1;
WHILE I <= 10 DO
  SET I = I + 1;
END WHILE;
```

Note: Please note the position of the semicolons.

Functions

Most of the function descriptions here impose restrictions on the data types of the arguments that can be passed to the function. If the values passed to the functions do not match the required data types, errors are generated at node configuration time if is possible to detect the errors at that point, otherwise run-time errors are generated when the function is evaluated.

String manipulation functions

The following functions perform manipulations on all strings (bit, byte, and character) with the exception of UPPER and LOWER, which operate only on character strings.

In these descriptions, the term ’singleton’ is used to refer to a single part (bit, byte, or character) within a string of that type.

Concatenation

Two vertical bars, ||, can be used to concatenate variables in ESQL.

LENGTH

```
LENGTH( source_string )
```

The LENGTH function returns an integer value which gives the number of singletons in source_string. If the value of the source_string is a NULL value, the result of the LENGTH function is the NULL value.

Examples:

| DECLARE K INTEGER;

`SET K = LENGTH('Hello World!');` returns 12.

`SET K = LENGTH('');` returns 0.

| Using the “Message referenced in examples” on page 113:

```
| SET K = LENGTH(InputBody.Invoice.Customer.Billing.Address[2]);
```

| returns 15.

LOWER, LCASE

```
LOWER( source_string )
LCASE( source_string )
```

| An example of how to use LOWER:

```
| SET OutputRoot.XML.Invoice.Customer.Title =
|    LOWER(InputBody.Invoice.Customer.Title);
```

| The content of title in the input message will be ’mr’ when using “Message
| referenced in examples” on page 113.

The LOWER and LCASE functions both return a new character string which is the same length as the source string and which is identical to the input string, except that it has all uppercase letters replaced with the corresponding lowercase letters. If the source string is NULL, the return value is NULL.

LTRIM

```
LTRIM( source_string )
```

This function is equivalent to TRIM(LEADING ' ' FROM source_string). See "TRIM" on page 81 for more information.

OVERLAY

```
OVERLAY( source_string PLACING source_string2 FROM start_position )
OVERLAY( source_string PLACING source_string2 FROM start_position FOR string_length)
```

| Some examples of how to use OVERLAY:

```
| SET OutputRoot.XML.Invoice.Customer.LastName =
|    OVERLAY (InputBody.Invoice.Customer.FirstName PLACING
|                        InputBody.Invoice.Customer.LastName FROM 3);
```

| Results in LastName having the value 'AnSmith'

```
| SET OutputRoot.XML.Invoice.Customer.LastName =
|    OVERLAY (InputBody.Invoice.Customer.FirstName PLACING
|                        InputBody.Invoice.Customer.LastName FROM 1);
```

| Results in LastName having the value 'Smithw'

```
| SET OutputRoot.XML.Invoice.Customer.LastName =
|    OVERLAY (InputBody.Invoice.Customer.FirstName PLACING
|                        InputBody.Invoice.Customer.LastName FROM 3 FOR 2);
```

| Results in LastName having the value 'AnSmithew'

If any of the parameters are NULL, the result is a NULL value of the same data type as source_string. If string_length is not specified, string_length is equal to LENGTH(source_string2).

The result of the OVERLAY function is equivalent to:

```
SUBSTRING(source_string FROM 1 FOR start_position -1 ) || source_string2 ||
  SUBSTRING(source_string FROM start_position + LENGTH(source_string2))
```

(where || is the concatenation operator).

POSITION

The POSITION function returns an integer that gives the position of the first occurrence of one string (the `search_string`) in a second string (the `source_string`).

```
POSITION( search_string IN source_string )
```

If the value of either the search_string or the source_string is NULL, the result of the POSITION function is NULL. If the value of search_string has a length of zero, the result is one. If the search_string cannot be found, it returns 0.

For example:

```
POSITION('TQ_' IN Body.Trade.Company)
```

Using "Message referenced in examples" on page 113 the following example returns the value 9:

```
DECLARE K INTEGER;
 SET K = POSITION('Village' IN InputBody.Invoice.Customer.Billing.Address[2]);
```

RTRIM

```
RTRIM( source_string )
```

This function is equivalent to TRIM(TRAILING ' ' FROM source_string). See "TRIM" on page 81 for more information.

SUBSTRING

You can use the SUBSTRING function to extract a string of bits, bytes, or characters from within another string of that type. You can use the result of SUBSTRING, for example, to compare to a known value.

The format of the function is as follows:

```
SUBSTRING( source_string FROM start_position )
SUBSTRING( source_string FROM start_position FOR string_length )
```

If any of the parameters to the SUBSTRING function are NULL, the result is the NULL string (which is different from the empty string).

The following example:

```
SUBSTRING(Body.Trade.Company FROM 1 FOR 3) = 'TQ_'
```

compares the first three singletons of a string to a given value. The positions in the string start at 1, so the `FROM 1` clause indicates that the substring should start at the first singleton. The `FOR 3` clause indicates that three singletons are included in the substring. This has a similar result to using the LIKE predicate.

This second example returns the string 'World!':

```
SUBSTRING('Hello World!' FROM 7)
```

The SUBSTRING function is implemented using the following algorithm:

- Let C be the value of source_string. Let LC be the length of C and let S be the value of start_position.
- If string_length is specified, let L be the value of string_length and let E be S+L. Otherwise let E be LC+1.
- If E is less than S, the function returns a NULL value.
- If S is greater than LC, or if E is less than 1, the result of the SUBSTRING function is a zero length string.
- Otherwise Let S1 be the larger of S and 1. Let E1 be the smaller of E and LC+1. Let L1 be E1-S1.
- The result of the SUBSTRING function is a string containing the L1 singletons of C starting at number S1 in the same order that the singletons appear in C.

One more example:

```
SET OutputRoot.XML.Invoice.StoreRecords.BuyTrends.MonthOfYear =
   CASE SUBSTRING(InputBody.Invoice.InvoiceDate FROM 6 FOR 2)
      WHEN '01' THEN 'JANUARY'
      WHEN '02' THEN 'FEBRUARY'
      WHEN '03' THEN 'MARCH'
   ELSE 'Spring onwards'
      END;
```

| ## TRIM

The TRIM function is used to remove leading and trailing singletons from a string.

You can specify the TRIM function in any of the following formats:

```
 TRIM( trim_specification trim_singleton FROM source_string )
 TRIM( trim_specification FROM source_string )
 TRIM( trim_singleton FROM source_string )
 TRIM( source_string )
```

where `trim_specification` is one of LEADING, TRAILING, or BOTH. If `trim_specification` is not specified, BOTH is assumed. If `trim_singleton` is not specified, a default singleton is assumed. This default depends on the data type of `source_string`:

character	' ' (space)
byte	X'00'
bit	B'0'

TRIM returns a string value of the same data type and content as `source_string` but with any leading or trailing singletons that are equal to `trim_singleton` removed (depending on the value of `trim_specification`). If any of the parameters are the NULL value, the TRIM function returns a NULL value of the same data type as `source_string`.

The FROM keyword is not required, and is in fact prohibited if neither a trim specification, for example LEADING or TRAILING, nor a trim singleton, is specified.

If you have a field in a message that is padded at the end with an unknown number of 'x' characters, and you want to compare the body of the character string to a literal value, you could use the following example:

```
TRIM(TRAILING 'x' FROM Body.Trade.Company) = 'Uncertain'
```

If you want to strip 'x' characters from the beginning and end of the string, you could write:

```
TRIM('x' FROM Body.Trade.Company) = 'Uncertain'
```

By default, blanks are stripped from a character string, and you can therefore leave out the character altogether, as follows:

```
TRIM(LEADING FROM Body.Market.Sector) = 'Target'
```

To strip blanks from the beginning and end of a character string, you could write:

```
TRIM(Body.Market.Sector) = 'Target'
```

It is often unnecessary to strip trailing blanks from character strings before comparison because the rules of character string comparison mean that trailing blanks are not significant.

The following examples illustrate additional function:

```
TRIM(TRAILING 'b' FROM 'aaabBb') returns 'aaabB'
TRIM('  a  ') returns 'a'
TRIM(LEADING FROM '  a  ') returns 'a  '
TRIM('b' FROM 'bbbaaabbb') returns 'aaa'
```

UPPER, UCASE

```
UPPER(source_string)
UCASE(source_string)
```

An example of how to use UPPER:

```
SET OutputRoot.XML.Invoice.Customer.LastName =
   UPPER(InputBody.Invoice.Customer.LastName);
```

The content of LastName in the output message is 'SMITH'.

The UPPER and UCASE functions both return a new character string which is the same length as the source character string and which is identical to the input string, except is has all lowercase letters replaced with the corresponding uppercase letters. If the source string is NULL, the return value is NULL.

Numeric functions

Numeric functions in ESQL work in the same way as standard SQL operators.

ABS

```
ABS( expression )
ABSVAL( expression )
```

The argument must be a numeric value. The function returns the absolute value of the argument, that is, a number without a sign. The argument can be NULL. If the argument is NULL, the function returns a NULL value.

This example shows how ABS could give someone's approximate age:

```
SET OutputRoot.XML.Invoice.Customer.Age
= ABS(1900 + CAST(SUBSTRING(InputBody.Invoice.Customer.DOB FROM 7 FOR 2)
                        AS INTEGER) - (EXTRACT (YEAR FROM CURRENT_DATE)));
```

which results in Age being set to 30.

BITAND

```
BITAND(expression1, expression2, ...)
```

The BITAND function takes two or more parameters that must result in integer values, and returns the result of performing the bitwise AND of the binary representation of the numbers.

For example:

```
DECLARE I1 INTEGER;
DECLARE I2 INTEGER;
SET I1 = 12;
SET I2 = 7;
SET OutputRoot.XML.Invoice.XMLAND = BITAND(I1, I2);
```

Results in XMLAND being set to 4.

BITNOT

```
BITNOT(expression)
```

The BITNOT function takes one parameter which must result in an integer value and returns the result of performing the bitwise complement of the binary representation of the number.

For example:

```
DECLARE I1 INTEGER;
DECLARE I2 INTEGER;
SET I1 = 12;
SET I2 = 7;
SET OutputRoot.XML.Invoice.XMLNOT = BITNOT(I2);
```

Results in XMLNOT being set to -8.

BITOR

```
BITOR(expression1, expression2, ...)
```

The BITOR function takes two or more parameters that must result in integer values, and returns the result of performing the bitwise OR of the binary representation of the numbers.

For example:

```
DECLARE I1 INTEGER;
DECLARE I2 INTEGER;
SET I1 = 12;
SET I2 = 7;
SET OutputRoot.XML.Invoice.XMLOR = BITOR(I1, I2);
```

Results in XMLOR being set to 15.

BITXOR

```
BITXOR(expression1, expression2, ...)
```

The BITXOR function takes two or more parameters that must result in integer values, and returns the result of performing the bitwise XOR of the binary representation of the numbers.

For example:

```
DECLARE I1 INTEGER;
DECLARE I2 INTEGER;
SET I1 = 12;
SET I2 = 7;
SET OutputRoot.XML.Invoice.XMLXOR = BITXOR(I1, I2);
```

Results in XMLXOR being set to 11.

CEIL

```
CEIL( expression )
CEILING( expression )
```

Returns the smallest integer value greater than or equal to the expression. The argument can be any numeric data type. If the argument is NULL, the result is the NULL value.

For example, using the sample message in“Message referenced in examples” on page 113:

```
SET OutputRoot.XML.Invoice.Total =
   CAST(CEILING((SELECT SUM(CAST (T.UnitPrice AS DECIMAL) * CAST(T.Quantity AS INTEGER))
   FROM InputBody.Invoice.Purchases."Item"[ ] AS T)) AS DECIMAL);
```

Results in Total being set to 159.

FLOOR

```
FLOOR( expression )
```

Returns the largest integer value less than or equal to the expression. The argument can be any numeric data type. If the argument is NULL, the result is the NULL value.

For example, using the sample message in“Message referenced in examples” on page 113:

```
SET OutputRoot.XML.Invoice.LowestPrice =
   CAST(FLOOR((SELECT MIN(CAST(T.UnitPrice AS DECIMAL))
   FROM InputBody.Invoice.Purchases."Item"[ ] AS T))
     AS DECIMAL);
```

Results in LowestPrice being set to 27.

Notes:

1. There is a restriction that FLOAT data type results from a DECIMAL argument unless the outer CAST in the above example is present.
2. All brackets are needed

MOD

```
MOD(expression1, expression2)
```

Returns the remainder of the first argument divided by the second argument. The result is negative only if first argument is negative. The arguments must have integer data types. The function returns an integer. If any argument is NULL, the result is the NULL value.

For example:

```
SET OutputRoot.XML.Invoice.Moddedl=
    MOD(CAST(InputBody.Invoice.InvoiceNo AS INTEGER),
        CAST(InputBody.Invoice.Payment.Valid AS INTEGER));
```

In the case of the example in “Message referenced in examples” on page 113 the above gives:

300524 divided by 1200 = 250 and remainder 524,
so the result of this example is 524.

ROUND

```
ROUND(expression1, expression2)
```

If expression2 is a positive number, ROUND returns the expression1 rounded to expression2 placed right of the decimal point. If expression2 is negative, expression1 is rounded to the absolute value of expression2 placed to the left of the decimal point. Expression1 can be any built-in numeric data type. Expression2 is an integer. A decimal argument is converted to a FLOAT for processing by the function. The result of the function is INTEGER if the first argument is INTEGER, FLOAT if the first argument is FLOAT, and DECIMAL if the first argument is DECIMAL. If any argument is NULL, the result is the NULL value.

To illustrate the use of ROUND, see the following examples:

- When considering the UnitPrice (27.95) of the first book in “Message referenced in examples” on page 113:

  ```
  SET OutputRoot.XML.Invoice.Rounded1 =
      ROUND(CAST (InputBody.Invoice.Purchases."Item"[1].UnitPrice AS DECIMAL), 3);
  ```

  Gives a result of 27.95

```
SET OutputRoot.XML.Invoice.Rounded2 =
    ROUND(CAST (InputBody.Invoice.Purchases."Item"[1].UnitPrice AS DECIMAL), -1);
```

Gives a result of 30.00

```
SET OutputRoot.XML.Invoice.Rounded3 =
    ROUND(CAST (InputBody.Invoice.Purchases."Item"[1].UnitPrice AS DECIMAL), +0);
```

Gives a result of 28.00

- This example can be used to compare the ROUND and TRUNCATE functions. For more information about TRUNCATE see "TRUNCATE".

  ```
  ROUND(893,-2)
  ```

  Gives a result of 900.

SQRT

```
SQRT(expression)
```

Returns the square root of the expression. The argument can be any built-in numeric data type. It has to be converted to a FLOAT number for processing by the function. The result of the function is a FLOAT. If the argument is NULL, the result is the NULL value.

For example:

```
SET OutputRoot.XML.Invoice.Root2 =
 SQRT(CAST (InputBody.Invoice.Purchases."Item"[1].Quantity AS DECIMAL));
```

Results in Root2 being set to 1.414213562373095E+0

TRUNCATE

```
TRUNCATE(expression1, expression2)
```

If expression2 is positive, TRUNCATE returns expression1 truncated to expression2 places right of the decimal point.

If expression2 is negative, TRUNCATE returns expression1 truncated to the absolute value of expression2 places to the left of the decimal point.

Expression2 must evaluate to an INTEGER

Expression1 can be any built-in numeric data type. Decimal values are converted to double-precision floating-point numbers for processing by the function.

If expression1 is an INTEGER, the result is an INTEGER.

If expression1 is a FLOAT or a DECIMAL, the result is a FLOAT.

If either expression evaluates to NULL, TRUNCATE returns NULL.

Here is are some examples of how to use TRUNCATE:

- This example uses the message in "Message referenced in examples" on page 113:

  ```
  SET OutputRoot.XML.Invoice.Trunc =
  TRUNCATE(CAST (InputBody.Invoice.Purchases."Item"[1].UnitPrice AS DECIMAL), 1);
  ```

  Expression1 evaluates 27.95, expression2 equals 1, so TRUNCATE returns 27.9.

- This example can be used to compare the ROUND and TRUNCATE functions. For more information about ROUND see “ROUND” on page 84.

  ```
  ROUND(893,-2)
  ```

  Gives a result of 800.

Datetime functions

You can use arithmetic operators to perform various natural calculations on Datetime values. For example, you can calculate the difference between two dates as an interval, or you can add an interval to a timestamp.

Adding an interval to a Datetime value

The simplest operation you can perform is to add an interval to, or subtract an interval from, a Datetime value. For example, you could write the following expressions:

```
DATE '2000-01-29' + INTERVAL '1' MONTH
TIMESTAMP '1999-12-31 23:59:59' + INTERVAL '1' SECOND
```

Adding or subtracting two intervals

Two interval values can be combined using addition or subtraction. The two interval values must be of compatible types. For example, it is not valid to add a year-month interval to a day-second interval. So the following example is not valid:

```
INTERVAL '1-06' YEAR TO MONTH + INTERVAL '20' DAY
```

The interval qualifier of the resultant interval is sufficient to encompass all of the fields present in the two operand intervals. For example:

```
INTERVAL '2 01' DAY TO HOUR + INTERVAL '123:59' MINUTE TO SECOND
```

would result in an interval with qualifier DAY TO SECOND, because both day and second fields are present in at least one of the operand values.

Subtracting two Datetime values

Two Datetime values can be subtracted to return an interval. In order to do this an interval qualifier must be given in the expression to indicate what precision the result should be returned in. For example:

```
(CURRENT_DATE - DATE '1776-07-04') DAY
```

would return the number of days since the 4th July 1776, whereas:

```
(CURRENT_TIME - TIME '00:00:00') MINUTE TO SECOND
```

would return the age of the day in minutes and seconds.

Scaling intervals

An interval value can be multiplied by or divided by an integer factor:

```
INTERVAL '2:30' MINUTE TO SECOND / 4
```

Extracting fields from Datetimes and intervals

You can extract individual fields from datetime values and intervals using the EXTRACT function. For example, you could extract the number of seconds from the current time with the expression:

```
EXTRACT(SECOND FROM CURRENT_TIME)
```

You can use any of the keywords YEAR, MONTH, DAY, HOUR, MINUTE, and SECOND in the EXTRACT function, but you can only extract a field that is present in the source value. Either a parse-time or a run-time error is generated if the requested field does not exist but this depends on how early the error can be detected. Other examples include:

```
EXTRACT(YEAR FROM CURRENT_DATE)
EXTRACT(HOUR FROM LOCAL_TIMEZONE)
```

The following Datetime functions allow you to manipulate fields according to date and time values:

CURRENT_DATE

CURRENT_DATE returns a date value representing the current date in local time. That is, it is equivalent to CAST(CURRENT_TIMESTAMP AS DATE).

```
CURRENT_DATE
```

The CURRENT_DATE function is not a true function in that no parentheses are necessary. All calls to CURRENT_DATE within the processing of one node are guaranteed to return the same value.

CURRENT_TIME

The CURRENT_TIME function returns a non-GMT time value representing the current local time. That is, it is equivalent to CAST(CURRENT_TIMESTAMP AS TIME).

```
CURRENT_TIME
```

The CURRENT_TIME function is not a true function in that no parentheses are necessary. All calls to CURRENT_TIME within the processing of one node are guaranteed to return the same value.

CURRENT_TIMESTAMP

The CURRENT_TIMESTAMP function returns a non-GMT timestamp value representing the current local time.

```
CURRENT_TIMESTAMP
```

The CURRENT_TIMESTAMP function is not a true function in that no parentheses are necessary. All calls to CURRENT_TIMESTAMP within the processing of one node are guaranteed to return the same value.

CURRENT_GMTDATE

The CURRENT_GMTDATE function returns a date value representing the current date in the GMT time zone. It is equivalent to CAST(CURRENT_GMTTIMESTAMP AS DATE).

```
CURRENT_GMTDATE
```

The CURRENT_GMTDATE function is not a true function in that no parentheses are necessary. All calls to CURRENT_GMTDATE within the processing of one node are guaranteed to return the same value.

CURRENT_GMTTIME

The CURRENT_GMTTIME function returns a GMT time value representing the current time in the GMT time zone. It is equivalent to CAST(CURRENT_GMTTIMESTAMP AS TIME).

```
CURRENT_GMTTIME
```

The CURRENT_GMTTIME function is not a true function in that no parentheses are necessary. All calls to CURRENT_GMTTIME within the processing of one node are guaranteed to return the same value.

Datetime functions

CURRENT_GMTTIMESTAMP

The CURRENT_GMTTIMESTAMP function returns a GMT timestamp value representing the current time in the GMT time zone.

```
CURRENT_GMTTIMESTAMP
```

The CURRENT_GMTTIMESTAMP function is not a true function in that no parentheses are necessary. All calls to CURRENT_GMTTIMESTAMP within the processing of one node are guaranteed to return the same value.

LOCAL_TIMEZONE

The LOCAL_TIMEZONE function returns an interval value which represents the local time zone displacement from GMT.

```
LOCAL_TIMEZONE
```

The LOCAL_TIMEZONE function is not a true function in that no parentheses are necessary. The value returned is an interval in hours and minutes representing the displacement of the current time zone from Greenwich Mean Time. The sign of the interval is such that a local time could be converted to a time in GMT by subtracting the result of the LOCAL_TIMEZONE function.

Miscellaneous functions

You can also use the CARDINALITY, FIELDNAME, FIELDTYPE, BITSREAM, COALESCE, and NULLIF functions, as described below.

BITSTREAM

The BITSTREAM function returns a BLOB representing the actual bit stream of the portion of the message specified.

```
BITSTREAM(path)
```

This function is typically used in message warehouse scenarios, where the bit stream of a message needs to be stored in a database. The function returns the bit stream of the physical portion of the message, identified by the path parameter. It does not return the bit stream representing the actual syntax element identified. So for example the following two calls would return the same value:

```
BITSTREAM(Root.MQMD);
BITSTREAM(Root.MQMD.UserIdentifier);
```

CARDINALITY

```
CARDINALITY(array)
```

Returns the number of elements in the argument array, or the number of children of a parent in the message tree.

For more information about the cardinality function, please see 58.

COALESCE

COALESCE returns the first argument that is not NULL. The arguments are evaluated in the order in which they are specified, and the result of the function is the first argument that is not NULL. The result is NULL only if all the arguments are NULL. The arguments must be compatible. The COALESCE function can be used to provide a default value for the value of a field which might not exist in a message. For example, the expression:

```
COALESCE(Body.Salary, 0)
```

would return the value of the Salary field in the message if it existed, or 0 (zero) if that field did not exist.

FIELDNAME

```
FIELDNAME(path)
```

Returns the name of the field that the argument path identifies as a string. If the path identifies a nonexistent entity, NULL is returned.

FIELDTYPE

```
FIELDTYPE(path)
```

Returns the type of the field that the argument path identifies as an integer. If the path identifies a nonexistent entity, NULL is returned. The result of this function will typically be compared with a symbolic constant defined by a parser which represents a type value. Note that this is not the data type of the field that the path identifies.

NULLIF

The NULLIF function returns a NULL value if the arguments are equal; otherwise, it returns the value of the first argument. The arguments must be comparable. The result of using NULLIF(e1,e2) is the same as using the expression

```
CASE WHEN e1=e2 THEN NULL ELSE e1 END
```

Note that when e1=e2 evaluates to unknown (because one or both arguments is NULL), CASE expressions consider this not true. Therefore, in this situation, NULLIF returns the value of the first argument.

Miscellaneous functions

Chapter 6. Complex SELECTs: ROWs and LISTs

The previous examples of the SELECT expression, in “SELECT expression” on page 60, have all involved column functions, because these are a common form in filter expressions. However, more general subselects can also be used. These operate in a similar way to standard SQL selects, but the "result sets" that are generated from the different forms need some discussion. As a way of illustrating the various forms that a SELECT clause can take, consider the following examples of assigning the results of database queries to fields in a message using a Compute node.

ROW and LIST constructors

The ROW and LIST constructor functions can be used to explicitly generate rows or lists of values which can be assigned to fields in an output message. A ROW consists of a sequence of named values. When assigned to a field reference it results in the creation of that sequence of named values as child fields of the referenced field. A LIST consists of a sequence of unnamed values. When assigned to an array field reference (indicated by [] suffixed to the last element of the reference), each value is assigned in sequence to an element of the array. A ROW cannot be assigned to an array field reference; a LIST cannot be assigned to a non-array field reference. Consider the following examples.

ROW constructor:

Example 1:

```
SET OutputRoot.XML.Data = ROW('granary' AS bread,
                'riesling' AS wine,
                'stilton' AS cheese);
```

produces:

```
<Data>
   <bread>granary</bread>
   <wine>riesling</wine>
   <cheese>stilton</cheese>
</Data>
```

Example 2:

Given the XML input message body:

```
<Proof>
   <beer>5</beer>
   <wine>12</wine>
   <gin>40</gin>
</Proof>
```

the ESQL:

```
SET OutputRoot.XML.Data = ROW(InputBody.Proof.beer,
                InputBody.Proof.wine AS vin,
                   (InputBody.Proof.gin * 2) AS special);
```

produces:

```
<Data>
   <beer>5</beer>
   <vin>12</vin>
   <special>80</special>
</Data>
```

Note: Note that because the values in this case are derived from field references, which already have names, it is not necessary to explicitly provide a name for each element of the row, although you may if you wish.

A List constructor:

Example 1:

Given the XML message input body:

```
<Car>
   <size>big</size>
   <colour>red</colour>
</Car>
```

The ESQL:

```
SET OutputRoot.XML.Data.Result[] = LIST{InputBody.Car.colour,
                                         'green',
                                         'blue'};
```

produces:

```
<Data>
   <result>red</result>
   <result>green</result>
   <result>blue</result>
</Data>
```

Note: In the case of a LIST there is no explicit name associated with each value - rather, the values are assigned in sequence to elements of the message field array specified as the target of the assignment. Also note that curly braces rather than parentheses are used to surround the LIST items

.

Example 2:

Given the XML input message body:

```
<Data>
   <Field>Keats</Field>
   <Field>Shelley</Field>
   <Field>Wordsworth</Field>
   <Field>Tennyson</Field>
   <Field>Byron</Field>
</Data>
```

The ESQL:

```
--Copy the entire input message to the output message,
--including the XML message field array as above
SET OutputRoot = InputRoot;
SET OutputRoot.XML.Data.Field[] = LIST{'Henri','McGough','Patten'};
```

produces:

```
<Data>
   <Field>Henri</Field>
   <Field>McGough</Field>
   <Field>Patten</Field>
</Data>
```

Note: The previous members of the Data.Field[] array have been discarded. Assigning a new list of values to an already existing message field array causes all the elements in the existing array to be removed before the new ones are assigned.

ROW and LIST combined

A ROW may validly be an element in a LIST. For example:

```
SET OutputRoot.XML.Data.Country[] =
         LIST{ROW('UK' AS name,'pound' AS currency),
              ROW('US' AS name, 'dollar' AS currency),
                                  default'};
```

produces:

```
<Data;>
   <Country>
      <name>UK</name>
      <currency>pound</currency>
   </Country>
   <Country>
      <name>US</name>
      <currency>dollar</currency>
   </Country>
   <Country>default</Country>
</Data>
```

Notes:

1. ROW and non-ROW values can be freely mixed within a LIST.
2. A LIST cannot validly be a member of a ROW. Only named scalar values can be members of a ROW.

ROW and LIST comparisons

ROWs and LISTs can validly be compared against other ROWs and LISTs.

Example 1:

```
IF ROW(InputBody.Data.*[1],InputBody.Data.*[2]) =
                 ROW('Raf' AS Name,'25' AS Age) THEN ...
IF LIST{InputBody.Data.Name, InputBody.Data.Age} = LIST{'Raf','25'} THEN ...
```

The following XML input message body would cause both the IF expressions in both of the above statements to evaluate to TRUE:

```
<Data>
   <Name>Raf</Name>
   <Age>25</Age>
</Data>
```

Note that in the comparison between ROWs both the name and the value of each element are compared, whereas in the comparison between LISTs only the value of each element is compared. In both cases the cardinality and sequential order of the LIST or ROW operands being compared must be equal in order for the two operands to be equal. In other words, all the following are false because either the sequential order or the cardinality of the operands being compared do not match:

```
ROW('alpha' AS A, 'beta' AS B) =
             ROW('alpha' AS A, 'beta' AS B, 'delta' AS D)
ROW('alpha' AS A, 'beta' AS B) =
             ROW('beta' AS B,'alpha' AS A)
LIST{1,2,3} = LIST{1,2,3,4}
LIST{3,2,1} = LIST{1,2,3}
```

Example 2:

Consider this ESQL:

```
IF InputBody.Places =
   ROW('Ken' AS first, 'Bob' AS second, 'Kate' AS third) THEN ...
```

The following XML input message body would cause the above IF expression to evaluate to TRUE:

```
<Places>
   <first>Ken</first>
   <second>Bob</second>
   <third>Kate</third>
</Places>
```

Note: The presence of an explicitly constructed ROW as one of the operands to the comparison operator results in the other operand also being treated as a ROW.

This should be contrasted with a comparison such as:

```
IF InputBody.Lottery.FirstDraw = InputBody.Lottery.SecondDraw THEN ...
```

which compares the value of the FirstDraw and SecondDraw fields, not the names and values of each of FirstDraw and SecondDraw’s child fields constructed as a ROW. Thus an XML input message body such as:

```
<Lottery>
   <FirstDraw>wednesday
      <ball1>32</ball1>
      <ball2>12</ball2>
   </FirstDraw>
   <SecondDraw>saturday
      <ball1>32</ball1>
      <ball2>12</ball2>
   </SecondDraw>
</Lottery>
```

would not result in the above IF expression being evaluated as TRUE, because the values ’wednesday’ and ’saturday’ are being compared, not the names and values of the ball fields.

Example 3:

Consider this ESQL:

```
IF InputBody.Cities.City[] = LIST{'Athens','Sparta','Thebes'} THEN ...
```

The following XML input message body would cause the IF expression to evaluate to TRUE:

```
<Cities>
<City>Athens</City>
<City>Sparta</City>
<City>Thebes</City>
</Cities>
```

Two message field arrays may be compared together in this way, for example:

```
IF InputBody.Cities.Mediaeval.City[] =
                    InputBody.Cities.Modern.City[] THEN ...

IF InputBody.Cities.Mediaeval.*[] = InputBody.Cities.Modern.*[] THEN ...

IF InputBody.Cities.Mediaeval.(XML.tag)[] =
                    InputBody.Cities.Modern.(XML.tag)[] THEN ...
```

The following XML input message body would cause the IF expression of the first and third of the statements above to evaluate to TRUE:

```
<Cities>
   <Mediaeval>1350
      <City>London</City>
      <City>Paris</City>
   </Mediaeval>
   <Modern>1990
      <City>London</City>
      <City>Paris</City>
   </Modern>
</Cities>
```

However the IF expression of the second statement would evaluate to FALSE, because the *[] indicates that all the children of Mediaeval and Modern are to be compared, not just the (XML.tag)s - so in this case the values 1350 and 1990, which form nameless children of Mediaeval and Modern, are compared as well as the values of the City tags.

Note that the IF expression of the third statement above would evaluate to TRUE with an XML input message body such as:

```
<Cities>
   <Mediaeval>1350
      <Location>London</Location>
      <Location>Paris</Location>
   </Mediaeval>
   <Modern>1990
      <City>London</City>
      <City>Paris</City>
   </Modern>
</Cities>
```

LISTs are composed of unnamed values - so it is the values of the child fields of Mediaeval and Modern that are compared, not their names.

Examples of complex SELECTs

Implications of the item order within the SELECT clause

Using the Control Center, create a message flow consisting of an MQInput node wired to a Compute node, wired to an MQOutput node. Configure the queue names on the MQInput node and MQOutput node to point to suitable queues, and set the Message Domain attribute on the Defaults tab of the MQInput node property editor to be "XML". Configure the Compute node using the following ESQL statements:

```
SET OutputRoot.MQMD = InputRoot.MQMD;
SET OutputRoot.XML.Test.Result[] =
(SELECT T.Field4, T.Structure1 FROM InputBody.Test.Input[] AS T);
```

Examples of complex SELECTs

Deploy the message flow to a suitable broker, and then send a simple trigger message like the following to the input queue:

```
<Test>
 <Input>
  <Field1>value1</Field1>
  <Structure1>
   <Field2>value2</Field2>
   <Field3>value3</Field3>
  </Structure1>
  <Field4>value4</Field4>
 </Input>
 <Input>
  <Field1>value5</Field1>
  <Structure1>
   <Field2>value6</Field2>
   <Field3>value7</Field3>
  </Structure1>
  <Field4>value8</Field4>
 </Input>
</Test>
```

You should receive the following message on the output queue:

```
<Test>
 <Result>
  <Field4>value4</Field4>
  <Structure1>
   <Field2>value2</Field2>
   <Field3>value3</Field3>
  </Structure1>
 </Result>
 <Result>
  <Field4>value8</Field4>
  <Structure1>
   <Field2>value6</Field2>
   <Field3>value7</Field3>
  </Structure1>
 </Result>
</Test>
```

The order in which the tags appear inside the Result tag reflects the order in which the items appeared in the select clause, not the order in which the fields appeared in the original message. Also, the Structure1 fields are copied in their entirety from the input message: that is, a tree copy has been performed. You can, of course, rename the fields by using an AS clause after some or all of the items in the SELECT clause.

Use of the ITEM keyword

The following example shows the use of the ITEM keyword, which selects one item and creates a single value. (Example 1 shows a structure that creates a single field.)

```
SET OutputRoot.MQMD = InputRoot.MQMD;
SET OutputRoot.XML.Test.Result[] =
(SELECT ITEM T.Field1 FROM InputBody.Test.Input[] AS T);
```

Sending the same trigger message will result in a message on the output queue which looks like this:

```
<Test>
 <Result>value1</Result>
 <Result>value5</Result>
</Test>
```

Comparing this message to the one which is produced if the ITEM keyword is omitted:

```
<Test>
 <Result>
  <Field1>value1</Field1>
 </Result>
 <Result>
  <Field1>value5</Field1>
 </Result>
</Test>
```

illustrates the effect of the ITEM keyword. The evaluation of the ESQL expressions happens independently of any information about the schema of the target message. In the case of generating a generic XML message there is no message schema for the message being generated, so the structure of the message that is generated must be defined entirely by the ESQL.

Effects of the THE keyword

The two examples above have both specified a list as the source of the SELECT in the FROM clause (so the field reference had a [] at the end), and so in general the SELECT will generate a list of results. Because of this it was necessary to specify a list as the target of the assignment (thus the "Result[]" as the target of the assignment). However, often you will know that the WHERE clause that you specify as part of the SELECT will only return TRUE for one item in the list. In this case the "THE" keyword can be used to indicate this. The following shows the effect of using the THE keyword

```
SET OutputRoot.MQMD = InputRoot.MQMD;
SET OutputRoot.XML.Test.Result =
THE (SELECT T.Field4, T.Structure1 FROM InputBody.Test.Input[]
 AS T WHERE T.Field1 = 'value1');
```

The "THE" keyword means that the target of the assignment becomes "OutputRoot.XML.Test.Result" (the "[]" is no longer necessary, or even allowed). This results in the following message:

```
<Test>
 <Result>
  <Field4>value4</Field4>
  <Structure1>
   <Field2>value2</Field2>
   <Field3>value3</Field3>
  </Structure1>
 </Result>
</Test>
```

Examples of complex SELECTs

Projection

Using selects for projection:

```
SET OutputRoot.XML.Projection =
 (SELECT M.field1,
   M.field2,
   CAST(M.field3 AS INTEGER) *CAST(M.field4 AS INTEGER) AS field5
 FROM InputBody.Message AS M);
```

equivalent to:

```
SET OutputRoot.XML.Projection.field1 = InputBody.Message.field1;
SET OutputRoot.XML.Projection.field2 = InputBody.Message.field2;
SET OutputRoot.XML.Projection.field5 =
    CAST(InputBody.Message.field3 AS INTEGER)
  * CAST(InputBody.Message.field4 AS INTEGER);
```

Multiple items in the FROM clause

The FROM clause is not restricted to having one item. Specifying multiple items in the FROM clause has the usual "joining" effect that it does in standard SQL. For example:

```
SELECT A.a, B.b
FROM InputBody.Test.A[], A.B[]
```

In this case, the following message:

```
<Test>
  <A>
   <a>1</a>
   <B>
    <b>2</b>
   </B>
   <B>
    <b>3</b>
   </B>
  </A>
  <A>
   <a>4</a>
   <B>
    <b>5</b>
   </B>
   <B>
    <b>6</b>
   </B>
  </A>
 </Test>
```

produces the following output:

```
<Test>
 <Result>
  <a>1</a>
  <b>2</b>
 </Result>
 <Result>
  <a>1</a>
  <b>3</b>
 </Result>
 <Result>
  <a>4</a>
  <b>5</b>
 </Result>
 <Result>
  <a>4</a>
  <b>6</b>
 </Result>
</Test>
```

Joining items in the FROM clause

You can join between a list and a non-list, two non-lists, and so on.

```
OutputRoot.XML.Test.Result1[] =
  (SELECT ... FROM InputBody.Test.A[], InputBody.Test.b);
OutputRoot.XML.Test.Result1 =
  (SELECT ... FROM InputBody.Test.A, InputBody.Test.b);
```

Note carefully the location of the "[]" in each case. Of course, any number of items can be specified in the FROM list, not just one or two, and in each case if any of the items specify "[]" to indicate a list of items, the SELECT will generate a list of results (the list may contain only one item, but the SELECT can potentially return a list of items), and so the target of the assignment must specify a list (so must end in "[]" or else the THE keyword must be used if is known that the WHERE clause will guarantee that only one combination is matched.

Using SELECT to return a scalar value

A SELECT with a column function is not the only form of SELECT that can be used in a scalar expression. You can make a SELECT return a scalar value by issuing both the THE and ITEM keywords as in:

```
1 + THE(SELECT ITEM T.a FROM Body.Test.A[] AS T WHERE T.b = '123')
```

Selecting from a list of scalars

Selecting from a list of scalars, consider the sample message:

```
<Test>
 <A>1</A>
 <A>2</A>
 <A>3</A>
 <A>4</A>
 <A>5</A>
</Test>
```

and the ESQL statements

```
SET OutputRoot.MQMD = InputRoot.MQMD;
SET OutputRoot.XML.Test.A[] = (SELECT A FROM InputBody.Test.A[]
WHERE CAST(A AS INTEGER) BETWEEN 2 AND 4);
```

Chapter 7. Querying external databases

Queries against external databases can be done in much the same way as can be done in, for example, embedded SQL.

In order to include a query against an external database in a Filter or Compute node, the node must be configured with the connection information for the database. This consists of an ODBC datasource name. It is up to the MQSeries Integrator or database administrator to ensure that a suitable ODBC datasource has been created on the systems on which the brokers, to which the message flows are deployed, are running.

The connection to the database is performed using the database user ID and password supplied on the **mqsicreatebroker** command that created the individual broker. The MQSeries Integrator or database administrator must therefore ensure that user has sufficient database privileges to query the required database tables. If not, a run-time error is generated by the broker when it attempts to process a message and attempts to connect to the database for the first time.

Whilst the standard SQL SELECT syntax is supported for queries to an external database, there are a number of points to be borne in mind. It is necessary to prefix the name of the table with the keyword "Database" in order to indicate that the SELECT is to be targeted at the external database, rather than at a repeating structure in the message.

Therefore the basic form of database SELECT is:

```
SELECT ...
FROM Database.TABLE1
WHERE ...
```

If necessary a schema name can be given:

```
SELECT ...
FROM Database.SCHEMA.TABLE1
WHERE ...
```

where SCHEMA is the name of the schema in which the table TABLE1 is defined.

References to column names must be qualified with either the table name or the correlation name defined for the table by the FROM clause. So, where you could normally execute a query such as:

```
SELECT column1, column2 FROM table1
```

it is necessary to write one of the following two forms:

```
SELECT T.column1, T.column2 FROM Database.table1 AS T

SELECT table1.column1, table1.column2 FROM Database.table1
```

This is necessary in order to distinguish references to database columns from any references to fields in a message which may also appear in the SELECT:

```
SELECT T.column1, T.column2 FROM Database.table1
  AS T WHERE T.column3 = Body.Field2
```

Querying external databases

The standard ‘select all’ SQL option is supported in the SELECT clause. If you use this option, you must qualify the column names with either the table name or the correlation name defined for the table. For example:

```
SELECT T.* FROM Database.Table1 AS T
```

The following examples illustrate how the results sets of external database queries are represented in MQSeries Integrator. The results of database queries are assigned to fields in a message using a Compute node.

| For more information on how to use the Database node please see *MQSeries*
| *Integrator Using the Control Center*.

Examples of external database queries

Create a database table

Create a message flow consisting of an MQInput node wired to a Compute node, wired to an MQOutput node. Configure the queue names on the MQInput node and MQOutput node to point to suitable queues, and set the Message Domain attribute on the Defaults tab of the MQInput node property editor to be "XML".

Create a database table called USERTABLE with two char(6) data type columns (or equivalent), called Column1 and Column2. Insert two rows into the table so that it looks like this:

	Column1	**Column2**
Row 1	value1	value3
Row 2	value2	value4

Add a database table input to the Compute node by clicking the **Add** input button on the properties pane of the node and entering the ODBC Data Source Name and table name. The user id and password specified when you created the broker is used for accessing the database, therefore you must ensure that this id and password pair have appropriate permissions within the DBMS.

You are also recommended to ensure that you include the schema name when you create a table, and as the second component of the database table reference (for example, `Database.user1.USERTABLE`) in the Compute node ESQL you specify. This avoids potential confusion that some databases might encounter.

For example, if you create your database table as user id `user1`, but specified user id `user2` when you created the broker, you might find that the broker attempts to access table `user2.USERTABLE`, which does not exist, rather than `user1.USERTABLE`, which does.

You can vary the names of the fields produced by explicitly listing the columns that you want to extract. How you do this depends partly on your database system. Most database systems are not case sensitive with regard to database names. In other words, even though a column might be called "COLUMN1", you can refer to it in a SELECT as "column1".

Configure the Compute node using the following ESQL statements:

```
SET OutputRoot = InputRoot;
SET OutputRoot.XML.Test.Result[] =
  (SELECT T.Column1, T.Column2 FROM Database.USERTABLE AS T);
```

To trigger the SELECT, you must send in a trigger message with an XML body that is of the following form:

```
<Test>
 <Result>
  <Column1>value1</Column1>
  <Column2>value2</Column2>
 </Result>
 <Result>
  <Column1>value3</Column1>
  <Column2>value4</Column2>
 </Result>
</Test>
```

The exact structure of the XML is not important, but the enclosing tag must be `<Test>`. If it is not, the ESQL statements will result in top-level enclosing tags being formed, which is not valid XML.

Create a table in a case sensitive database system

If the database system is case sensitive, you must use an alternative approach. This approach is also necessary if you want to change the name of the generated field to something different:

```
SET OutputRoot = InputRoot;
SET OutputRoot.XML.Test.Result[] =
  (SELECT T.COLUMN1 AS Column1, T.COLUMN2 AS Column2
  FROM Database.USERTABLE AS T);
```

This example produces the same message as Example 1 above.

Use of the ITEM keyword

Suppose that the Compute node were configured using the following ESQL statements:

```
SET OutputRoot = InputRoot;
SET OutputRoot.XML.Test.Result[] =
 (SELECT ITEM T.Column1 FROM Database.USERTABLE AS T);
```

The same trigger message will produce the following message:

```
<Test>
 <Result>value1</Result>
 <Result>value3</Result>
</Test>
```

The following message is produced if the ITEM keyword is omitted:

```
<Test>
 <Result>
    <Column1>value1</Column1>
 </Result>
 <Result>
  <Column1>value3</Column1>
 </Result>
</Test>
```

Comparing this to the previous generated message illustrates the effect of the ITEM keyword. The evaluation of the ESQL expressions happens independently of any information about the schema of the target message. In the case of generating a generic XML message, there is no message schema for the message being generated, so the structure of the message that is generated must be defined entirely by the ESQL.

Use of the WHERE clause

This example illustrates the use of the WHERE clause:

```
SET OutputRoot = InputRoot;
SET OutputRoot.XML.Test.Result =
 THE (SELECT ITEM T.Column1 FROM Database.USERTABLE AS T
 WHERE T.Column2 = 'value2');

<Test>
 <Result>value1   </Result>
</Test>
```

Appendix A. ESQL Components

Special Characters in ESQL

;	semicolon	End of ESQL statement
.	period	Message hierarchy separator / decimal point
=	equals	Variable comparison
>	greater than	Variable comparison
<	less than	Variable comparison
[]	square brackets	Array subscript
′	single quote	Delimit string constant
||	double vertical bar	Concatenation
()	round brackets	Expression delimiter
″	double quote	Delimit field or message hierarchy name
*	asterisk	All subscripts / multiply
+	plus	Arithmetic add
-	minus	Arithmetic subtract / date separator
/	forward slash	Arithmetic divide
_	underscore	LIKE single wild card
%	percent	LIKE multiple wild card
\	backslash	LIKE escape character
:	colon	Time separator
,	comma	List separator
<>	less than greater than	Not equals
–	double minus	ESQL comment
/* */	slash & asterisk	ESQL comment
?	questionmark	Substitution variable in PASSTHRU

Data types used in ESQL

BIT	GMTTIME
BLOB	GMTTIMESTAMP
BOOLEAN	INT or INTEGER
CHAR or CHARACTER	INTERVAL
DATE	TIME
DECIMAL	TIMESTAMP
FLOAT	

Arithmetic operations supported in ESQL

+	plus	*	multiplied by
-	minus	/	divided by

Here are some examples of how they can be used:

```
DECLARE X1 = INTEGER;
DECLARE X2 = INTEGER;
DECLARE X3 = INTEGER;
DECLARE X4 = INTEGER;
DECLARE X5 = INTEGER;
SET X1 = 8;
SET X2 = X1 + 2;
SET X3 = X2 - X1;
SET X4 = X3 * X2;
SET X5 = X1 / X3;
```

ESQL comparison operators

=	equals	<	less than
<>	not equals	>=	greater than or equals to
>	greater than	<=	less than or equals to

These operators are used in
- WHILE and IF statements when using the Compute node
- The Filter nodes
- The Database node

Initial correlation names

Table 10. Initial correlation names

Body	InputRoot
DestinationList	OutputDestinationList
ExceptionList	OutputExceptionList
InputBody	OutputRoot
InputDestinationList	Properties
InputExceptionList	Root
InputProperties	

Reserved words used in ESQL

Note: * denotes words which are not used in MQSeries Integrator Version 2.0.2, but are words reserved for future releases.

ABS	ELSEIF*	MONTH
ABSVAL	END	NOT
ALL	ESCAPE	NULL
AND	EVAL	NULLIF
ANY	EXISTS	OR
AS	EXTRACT	OVERLAY
ASYMMETRIC	FALSE	PASSTHRU
BETWEEN	FIELDNAME	PLACING
BIT	FIELDTYPE	POSITION
BITAND	FLOAT	REPEAT*
BITNOT	FOR	ROUND
BITOR	FROM	ROW
BITSTREAM	GMTTIME	RTRIM
BITXOR	GMTTIMESTAMP	SECOND
BLOB	HOUR	SELECT
BOOLEAN	IF	SET
BOTH	IN	SOME
BY*	INSERT	SQRT
CARDINALITY	INT	SUBSTRING
CASE	INTEGER	SUM
CAST	INTERVAL	SYMMETRIC
CEIL	INTO	THE
CEILING	IS	THEN
CHAR	ITEM	TIME
CHARACTER	ITERATE	TIMESTAMP
COALESCE	LAST	TO
COUNT	LCASE	TRAILING
CURRENT_DATE	LEADING	TRIM
CURRENT_GMTDATE	LEAVE	TRUE
CURRENT_GMTTIME	LENGTH	TRUNCATE
CURRENT_GMTTIMESTAMP	LIKE	UCASE
CURRENT_TIME	LIST	UNKNOWN
CURRENT_TIMESTAMP	LOCAL_TIMEZONE	UNTIL*
DATE	LOOP*	UPDATE
DAY	LOWER	UPPER
DECIMAL	LTRIM	VALUES
DECLARE	MAX	WHEN
DELETE	MIN	WHERE

DO	MINUTE	WHILE
ELSE	MOD	YEAR

Reserved words used in ESQL

Appendix B. Examples

Message referenced in examples

The following message is used in many of the examples throughout this book:

```
<Invoice>
<InvoiceNo>300524</InvoiceNo>
<InvoiceDate>2000-12-07</InvoiceDate>
<InvoiceTime>12:40:00</InvoiceTime>
<TillNumber>3</Till>
<Cashier StaffNo="089">Mary</Cashier>
<Customer>
   <FirstName>Andrew</FirstName>
   <LastName>Smith</LastName>
   <Title>Mr</Title>
   <DOB>20-01-70</DOB>
   <PhoneHome>01962818000</PhoneHome>
   <PhoneWork/>
   <Billing>
      <Address>14 High Street</Address>
      <Address>Hursley Village</Address>
      <Address>Hampshire</Address>
      <PostCode>SO213JR</PostCode>
   </Billing>
</Customer>
<Payment>
   <CardType>Visa</CardType>
   <CardNo>4921682832258418</CardNo>
   <CardName>Mr Andrew J. Smith</CardName>
   <Valid>1200</Valid>
   <Expires>1101</Expires>
</Payment>
<Purchases>
   <Item>
      <Title Category="Computer" Form="Paperback" Edition="2">The XML Companion
</Title>
      <ISBN>0201674866</ISBN>
      <Author>Neil Bradley</Author>
      <Publisher>Addison-Wesley</publisher>
      <PublishDate>October 1999</PublishDate>
      <UnitPrice>27.95</UnitPrice>
      <Quantity>2</Quantity>
   </Item>
   <Item>
      <Title Category="Computer" Form="Paperback" Edition="2">A Complete Guide
to DB2 Universal Database</Title>
      <ISBN>1558604820</ISBN>
      <Author>Don Chamberlin</Author>
      <Publisher>Morgan Kaufmann Publishers</Publisher>
      <PublishDate>April 1998</PublishDate>
      <UnitPrice>42.95</UnitPrice>
      <Quantity>1</Quantity>
   </Item>
   <Item>
      <Title Category="Computer" Form="Hardcover" Edition="0">JAVA 2 Developers
Handbook</Title>
      <ISBN>0782121799</ISBN>
      <Author>Philip Heller, Simon Roberts </Author>
      <Publisher>Sybex, Inc.</Publisher>
      <PublishDate>September 1998</PublishDate>
      <UnitPrice>59.99</UnitPrice>
      <Quantity>1</Quantity>
```

```
        </Item>
     </Purchases>
     <StoreRecords/>
     <DirectMail/>
     <Error/>
     </Invoice>
```

Using a trace to view a message structure

When looking at messages, if the tree structure is not clear, you can generate a trace record, using the MQSeries Integrator Trace node to show it. To do this create, assign, and deploy a simple message flow containing a:

1. Message from MQSeries
2. MQInput node
3. Trace node with a Destination property of file, a File Path property set to a fully-qualified file and path name where the trace record will be written, and a Pattern property of ${Root}
4. MQOutput node

See *MQSeries Integrator Using the Control Center* for more information on how to do this.

For example, if you were to receive a simple XML message on an MQSeries queue, whose message originated in the message flow from an MQInput node, and which had an MQRFH2 header (see Table 19 on page 127 for more information) like the message below:

```
<Trade type='buy'
 Company='IBM'
 Price='200.20'
 Date='2000-01-01'
 Quantity='1000'/>
```

You could use the Trace node to produce the following tree representation:

```
Root
      Properties
         CreationTime=GMTTIMESTAMP '1999-11-24 13:10:00'
          (a GMT timestamp field)

  ... and other fields ...

      MQMD
         PutDate=DATE '19991124'
          (a date field)

         PutTime=GMTTIME '131000'
          (a GMTTIME field)

  ... and other fields ...

      MQRFH
          mcd
          msd='xml'
             (a character string field)

         .. and other fields ...

      XML
         Trade
          type='buy'
          (a character string field)

         Company='IBM'
          (a character string field)

         Price='200'
          (a character string field)

         Date='2000-01-01'
          (a character string field)

         Quantity='1000'
          (a character string field)
```

Example exception list

Figure 5 illustrates one way in which an exception list can be constructed.

```
ExceptionList {
    RecoverableException = {                         1
        File     = 'f:/build/argo/src/DataFlowEngine/ImbDataFlowNode.cpp'
        Line     = 538
        Function = 'ImbDataFlowNode::createExceptionList'
        Type     = 'ComIbmComputeNode'
        Name     = '0e416632-de00-0000-0080-bdb4d59524d5'
        Label    = 'mf1.Compute1'
        Text     = 'Node throwing exception'
        Catalog  = 'MQSeries Integrator2'
        Severity = 3
        Number   = 2230
        RecoverableException = {                     2
            File     = 'f:/build/argo/src/DataFlowEngine/ImbRdlBinaryExpression.cpp'
            Line     = 231
            Function = 'ImbRdlBinaryExpression::scalarEvaluate'
            Type     = 'ComIbmComputeNode'
            Name     = '0e416632-de00-0000-0080-bdb4d59524d5'
            Label    = 'mf1.Compute1'
            Text     = 'error evaluating expression'
            Catalog  = 'MQSeries Integrator2'
            Severity = 2
            Number   = 2439
            Insert   = {
               Type = 2
               Text = '2'
            }
            Insert   = {
               Type = 2
               Text = '30'
            }
            RecoverableException = {                 3
                File     = 'f:/build/argo/src/DataFlowEngine/ImbRdlValueOperations.cpp'
                Line     = 257
                Function = 'intDivideInt'
                Type     = 'ComIbmComputeNode'
                Name     = '0e416632-de00-0000-0080-bdb4d59524d5'
                Label    = 'mf1.Compute1'
                Text     = 'Divide by zero calculating '%1 / %2''
                Catalog  = 'MQSeries Integrator2'
                Severity = 2
                Number   = 2450
                Insert   = }
                   Type = 5
                   Text = '100 / 0'
                }
            }
        }
    }
}
```

Figure 5. Exception list structure

Notes:

1. The first exception description **1** is a child of the root. This identifies error number 2230, indicating an exception has been thrown. The node that has thrown the exception is also identified (*mf1.Compute1*).

Example exception list

2. Exception description **2** is a child of the first exception description **1**. This identifies error number 2439.
3. Exception description **3** is a child of the second exception description **2**. This identifies error number 2450, which indicates that the node has attempted to divide by zero.

Exception handling paths will base their decisions on the number of exception conditions on:

- The message number, which identifies the type of exception that has occurred.
- The label, which is the known name of the object in which the exception occurred.

Figure 6 illustrates an extract of ESQL to show how you can set up a Compute node to use the exception list. The ESQL loops through the exception list to the last (nested) exception description, and extracts the error number. This error relates to the original cause of the problem and normally provides the most precise information. Subsequent action taken by the message flow can be decided by the error number retrieved in this way.

```
/* Error number extracted from exception list */
DECLARE Error INTEGER;
/* Current path within the exception list */
DECLARE Path CHARACTER;

/* Start at first child of exception list */
SET Path = 'InputExceptionList.*[1]';

/* Loop until no more children */
WHILE EVAL( 'FIELDNAME(' || Path || ') IS NOT NULL' ) DO

  /* Check if error number is available */
  IF EVAL( 'FIELDNAME(' || Path || '.Number) IS NOT NULL' ) THEN
    /* Remember only the deepest error number */
    SET Error = EVAL( Path || '.Number' );
  END IF;

 /* Step to last child of current element (usually a nested exception list */
 SET Path = Path || '.*[LAST]';

END WHILE; /* End loop */
```

Figure 6. Retrieving the exception error code

Appendix C. MQSeries message header parsers

The following sections define the element names, types, and attributes for each of the supported MQSeries headers.

The following parsers are described:

The MQCFH parser

The Root name for this parser is ″MQPCF″. Table 11 lists the elements native to the MQCFH header.

Table 11. MQCFH parser element names, types, and attributes

Element Name	**Element Data Type**	**Element Attributes**
Type	INTEGER	Name Value
StrucLength	INTEGER	Name Value
Version	INTEGER	Name Value
Command	INTEGER	Name Value
MsgSeqNumber	INTEGER	Name Value
Control	INTEGER	Name Value
CompCode	INTEGER	Name Value
Reason	INTEGER	Name Value
ParameterCount	INTEGER	Name Value

For further information about this header and its contents, see the *MQSeries Programmable System Management* book.

The MQCIH parser

The Root name for this parser is "MQCIH". Table 12 lists the elements native to the MQCIH header.

Table 12. MQCIH parser element names, types, and attributes

Element Name	Element Data Type	Element Attributes
Format	CHARACTER	Name Value
Version	INTEGER	Name Value
Encoding	INTEGER	Name Value
CodedCharSetId	INTEGER	Name Value
Flags	INTEGER	Name Value
ReturnCode	INTEGER	Name Value
CompCode	INTEGER	Name Value
Reason	INTEGER	Name Value
UOWControl	INTEGER	Name Value
GetWaitInterval	INTEGER	Name Value
LinkType	INTEGER	Name Value
OutputDataLength	INTEGER	Name Value
FacilityKeepTime	INTEGER	Name Value
ADSDescriptor	INTEGER	Name Value
ConversationalTask	INTEGER	Name Value
TaskEndStatus	INTEGER	Name Value
Facility	BLOB	Name Value
Function	CHARACTER	Name Value
AbendCode	CHARACTER	Name Value
Authenticator	CHARACTER	Name Value
Reserved1	CHARACTER	Name Value
ReplyToFormat	CHARACTER	Name Value
RemoteSysId	CHARACTER	Name Value
RemoteTransId	CHARACTER	Name Value
TransactionId	CHARACTER	Name Value
FacilityLike	CHARACTER	Name Value
AttentionId	CHARACTER	Name Value
StartCode	CHARACTER	Name Value
CancelCode	CHARACTER	Name Value
NextTransactionId	CHARACTER	Name Value
Reserved2	CHARACTER	Name Value
Reserved3	CHARACTER	Name Value
CursorPosition	INTEGER	Name Value
ErrorOffset	INTEGER	Name Value
InputItem	INTEGER	Name Value
Reserved4	INTEGER	Name Value

The MQDLH parser

The Root name for this parser is ″MQDLH″. Table 13 lists the elements native to the MQDLH header.

Table 13. MQDLH parser element names, types, and attributes

Element Name	Element Data Type	Element Attributes
Format	CHARACTER	Name Value
Version	INTEGER	Name Value
Encoding	INTEGER	Name Value
CodedCharSetId	INTEGER	Name Value
Reason	INTEGER	Name Value
DestQName	CHARACTER	Name Value
DestQMgrName	CHARACTER	Name Value
PutApplType	INTEGER	Name Value
PutApplName	CHARACTER	Name Value
PutDate	TIMESTAMP/CHARACTER	Name Value
PutTime	TIMESTAMP/CHARACTER	Name Value

The MQIIH parser

The Root name for this parser is ″MQIIH″. Table 14 lists the elements native to the MQIIH header.

Table 14. MQIIH parser element names, types, and attributes

Element Name	Element Data Type	Element Attributes
Format	CHARACTER	Name Value
Version	INTEGER	Name Value
Encoding	INTEGER	Name Value
CodedCharSetId	INTEGER	Name Value
Flags	INTEGER	Name Value
LTermOverride	CHARACTER	Name Value
MFSMapName	CHARACTER	Name Value
ReplyToFormat	CHARACTER	Name Value
Authenticator	CHARACTER	Name Value
TranInstanceId	BLOB	Name Value
TranState	CHARACTER	Name Value
CommitMode	CHARACTER	Name Value
SecurityScope	CHARACTER	Name Value
Reserved	CHARACTER	Name Value

The MQMD parser

The Root name for this parser is ″MQMD″. Table 15 lists the orphan elements adopted by the MQMD header.

Table 15. MQMD parser orphan element names, types, and attributes

Element Name	Element Data Type	Element Attributes
SourceQueue	CHARACTER	Name Value
Transactional	CHARACTER	Name Value

Table 16 lists the elements native to the MQMD header.

Table 16. MQMD parser native element names, types, and attributes

Element Name	Element Data Type	Element Attributes
Format	CHARACTER	Name Value
Version	INTEGER	Name Value
Report	INTEGER	Name Value
MsgType	INTEGER	Name Value
Expiry	INTEGER/TIMESTAMP	Name Value
Feedback	INTEGER	Name Value
Encoding	INTEGER	Name Value
CodedCharSetId	INTEGER	Name Value
Priority	INTEGER	Name Value
Persistence	INTEGER	Name Value
MsgId	BLOB	Name Value
CorrelId	BLOB	Name Value
BackoutCount	INTEGER	Name Value
ReplyToQ	CHARACTER	Name Value
ReplyToQMgr	CHARACTER	Name Value
UserIdentifier	CHARACTER	Name Value
AccountingToken	BLOB	Name Value
ApplIdentityData	CHARACTER	Name Value
PutApplType	INTEGER	Name Value
PutApplName	CHARACTER	Name Value
PutDate	TIMESTAMP/CHARACTER	Name Value
PutTime	TIMESTAMP/CHARACTER	Name Value
ApplOriginData	CHARACTER	Name Value
GroupId	BLOB	Name Value
MsgSeqNumber	INTEGER	Name Value
Offset	INTEGER	Name Value
MsgFlags	INTEGER	Name Value
OriginalLength	INTEGER	Name Value

The MQMDE parser

The Root name for this parser is "MQMDE". Table 17 lists the elements native to the MQMDE header.

Table 17. MQMDE parser element names, types, and attributes

Element Name	Element Data Type	Element Attributes
Format	CHARACTER	Name Value
Version	INTEGER	Name Value
Encoding	INTEGER	Name Value
CodedCharSetId	INTEGER	Name Value
Flags	INTEGER	Name Value
GroupId	BLOB	Name Value
MsgSeqNumber	INTEGER	Name Value
Offset	INTEGER	Name Value
MsgFlags	INTEGER	Name Value
OriginalLength	INTEGER	Name Value

The MQRFH parser

The Root name for this parser is ″MQRFH″. Table 18 lists the elements native to the MQRFH header.

Table 18. MQRFH parser element names, types, and attributes

Element Name	Element Data Type	Element Attributes
Format	CHARACTER	Name Value
Version	INTEGER	Name Value
Encoding	INTEGER	Name Value
CodedCharSetId	INTEGER	Name Value
Flags	INTEGER	Name Value

Other name value elements might be present that contain information as parsed from or destined for the option buffer. See the Rules and Format header documentation for specific names and values.

The MQRFH2 parser

The Root name for this parser is ″MQRFH2″. Table 19 lists the elements native to the MQRFH2 header.

Table 19. MQRFH2 parser element names, types, and attributes

Element Name	Element Data Type	Element Attributes
Format	CHARACTER	Name Value
Version	INTEGER	Name Value
Encoding	INTEGER	Name Value
CodedCharSetId	INTEGER	Name Value
Flags	INTEGER	Name Value
NameValueCCSID	INTEGER	Name Value

Other name and child name value elements might be present that contain information as parsed from or destined for the option buffer. See the Rules and Format header section in the MQSeries Integrator documentation for further details.

The MQRMH parser

The Root name for this parser is ″MQRMH″. Table 20 lists the elements native to the MQRMH header.

Table 20. MQRMH parser element names, types, and attributes

Element Name	**Element Data Type**	**Element Attributes**
Format	CHARACTER	Name Value
Version	INTEGER	Name Value
Encoding	INTEGER	Name Value
CodedCharSetId	INTEGER	Name Value
Flags	INTEGER	Name Value
ObjectType	CHARACTER	Name Value
ObjectInstanceId	BLOB	Name Value
SrcEnv	CHARACTER[1]	Name Value
SrcName	CHARACTER[2]	Name Value
DestEnv	CHARACTER[3]	Name Value
DestName	CHARACTER[4]	Name Value
DataLogicalLength	INTEGER	Name Value
DataLogicalOffset	INTEGER	Name Value
DataLogicalOffset2	INTEGER	Name Value

Notes:

1. This field represents both SrcEnvLength and Offset
2. This field represents both SrcNameLength and Offset
3. This field represents both DestEnvLength and Offset
4. This field represents both DestNameLength and Offset

The MQSAPH parser

The Root name for this parser is ″MQSAPH″. Table 21 lists the elements native to the MQSAPH header.

Table 21. MQSAPH parser element names, types, and attributes

Element Name	Element Data Type	Element Attributes
Format	CHARACTER	Name Value
Version	INTEGER	Name Value
Encoding	INTEGER	Name Value
CodedCharSetId	INTEGER	Name Value
Flags	INTEGER	Name Value
Client	CHARACTER	Name Value
Language	CHARACTER	Name Value
HostName	CHARACTER	Name Value
UserId	CHARACTER	Name Value
Password	CHARACTER	Name Value
SystemNumber	CHARACTER	Name Value
Reserved	BLOB	Name Value

The MQWIH parser

The Root name for this parser is "MQWIH". Table 22 lists the elements native to the MQWIH header.

Table 22. MQWIH parser element names, types, and attributes

Element Name	Element Data Type	Element Attributes
Format	CHARACTER	Name Value
Version	INTEGER	Name Value
Encoding	INTEGER	Name Value
CodedCharSetId	INTEGER	Name Value
Flags	INTEGER	Name Value
ServiceName	CHARACTER	Name Value
ServiceStep	CHARACTER	Name Value
MsgToken	BLOB	Name Value
Reserved	CHARACTER	Name Value

The SMQ_BMH parser

The Root name for this parser is "SMQ_BMH". Table 23 lists the elements native to the SMQ_BMH header.

Table 23. SMQ_BMH parser element names, types, and attributes

Element Name	Element Data Type	Element Attributes
Format	CHARACTER	Name Value
Version	INTEGER	Name Value
Encoding	INTEGER	Name Value
CodedCharSetId	INTEGER	Name Value
ErrorType	INTEGER	Name Value
Reason	INTEGER	Name Value
PutApplType	INTEGER	Name Value
PutApplName	CHARACTER	Name Value
PutDate	TIMESTAMP/CHARACTER	Name Value
PutTime	TIMESTAMP/CHARACTER	Name Value

The BLOB parser

The Root name for this parser is ″BLOB″. Table 24 lists the elements native to the BLOB header.

Table 24. BLOB parser element names, types, and attributes

Element Name	Element Data Type	Element Attributes
BLOB	BLOB[1]	Name Value
UnkownParserName	CHARACTER[2]	Name Value

Notes:

1. This field contains the remaining unparsed bitstream from the message. It is represented as a BLOB and may be manipulated as such.
2. This field (if present) contains the class name of the parser that would have been chosen in preference to the BLOB parser. This information is used by the header integrity routine (described in "Maintaining header integrity" on page 6) to ensure that the semantic meaning of the message is preserved.

Appendix D. Notices

This information was developed for products and services offered in the United States. IBM may not offer the products, services, or features discussed in this information in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user’s responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this information. The furnishing of this information does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the information. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this information at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Notices

Trademarks

Glossary of terms and abbreviations

This glossary defines MQSeries Integrator terms and abbreviations used in this book. If you do not find the term you are looking for, see the index or the *IBM Dictionary of Computing*, New York: McGraw-Hill, 1994.

This glossary includes terms and definitions from the *American National Dictionary for Information Systems*, ANSI X3.172-1990, copyright 1990 by the American National Standards Institute. Copies may be ordered from the American National Standards Institute, 11 West 42 Street, New York, New York 10036. Definitions are identified by the symbol (A) after the definition.

A

Access Control List (ACL). The list of principals that have explicit permissions (to publish, to subscribe to, and to request persistent delivery of a publication message) against a topic in the topic tree. The ACLs define the implementation of topic-based security.

ACL. Access Control List.

AMI. Application Messaging Interface.

Application Messaging Interface (AMI). The programming interface provided by MQSeries that defines a high level interface to message queuing services. See also *MQI* and *JMS*.

B

blob. Binary Large OBject. A block of bytes of data (for example, the body of a message) that has no discernible meaning, but is treated as one solid entity that cannot be interpreted. Also written as BLOB.

broker. See *message broker*.

broker domain. A collection of brokers that share a common configuration, together with the single Configuration Manager that controls them.

C

callback function. See *implementation function*.

category. An optional grouping of messages that are related in some way. For example, messages that relate to a particular application.

check in. The Control Center action that stores a new or updated resource in the configuration or message respository.

check out. The Control Center action that extracts and locks a resource from the configuration or message respository for local modification by a user. Resources from the two repositories can only be worked on when they are checked out by an authorized user, but can be viewed (read only) without being checked out.

collective. A hyperconnected (totally connected) set of brokers forming part of a multi-broker network for publish/subscribe applications.

configuration. In the broker domain, the brokers, execution groups, message flows and message sets assigned to them, topics and access control specifications.

Configuration Manager. A component of MQSeries Integrator that acts as the interface between the configuration repository and an executing set of brokers. It provides brokers with their initial configuration, and updates them with any subsequent changes. It maintains the broker domain configuration.

configuration repository. Persistent storage for broker configuration and topology definition.

connector. See *message processing node connector*.

content-based filter. An expression that is applied to the content of a message to determine how the message is to be processed.

context tag. A tag that is applied to an element within a message to enable that element to be treated differently in different contexts. For example, an element could be mandatory in one context and optional in another.

Control Center. The graphical interface that provides facilities for defining, configuring, deploying, and monitoring resources of the MQSeries Integrator network.

D

datagram. The simplest form of message that MQSeries supports. Also known as *send-and-forget*. This type of message does not require a reply. Compare with *request/reply*.

| **debugger.** A facility on the *Message Flows* view in the
| Control Center that enables message flows to be
| debugged.

deploy. Make operational the configuration and topology of the broker domain.

destination list. A list of internal and external destinations to which a message is sent. These can be nodes within a message flow (for example, when using the RouteToLabel and Label nodes) or MQSeries queues (when the list is examined by an MQOutput node to determine the final target for the message).

distribution list. A list of MQSeries queues to which a message can be put using a single statement.

Document Type Definition (DTD). The rules that specify the structure for a particular class of SGML or XML documents. The DTD defines the structure with elements, attributes, and notations, and it establishes constraints for how each element, attribute, and notation can be used within the particular class of documents. A DTD is analogous to a database schema in that the DTD completely describes the structure for a particular markup language.

DTD. Document Type Definition

E

e-business. A term describing the commercial use of the Internet and World Wide Web to conduct business (short for electronic-business).

element. A unit of data within a message that has business meaning, for example, street name

element qualifier. See *context tag*.

ESQL. Extended SQL. A specialized set of SQL statements based on regular SQL, but extended with statements that provide specialized functions unique to MQSeries Integrator.

exception list. A list of exceptions that have been generated during the processing of a message, with supporting information.

execution group. A named grouping of message flows that have been assigned to a broker. The broker is guaranteed to enforce some degree of isolation between message flows in distinct execution groups by ensuring that they execute in separate address spaces, or as unique processes.

Extensible Markup Language (XML). A W3C standard for the representation of data.

| **external reference.** A reference within a message set to a component that has been defined outside the current message set. For example, an integer that defines the length of a string element might be defined in one message set but used in several message sets.

F

| **field reference.** A sequence of period-separated values that identify a specific field (which might be a structure) within a message tree. An example of a field reference might be something like `Body.Invoice.InvoiceNo`.

filter. An expression that is applied to the content of a message to determine how the message is to be processed.

format. A format defines the internal structure of a message, in terms of the fields and order of those fields. A format can be self-defining, in which case the message is interpreted dynamically when read.

G

graphical user interface (GUI). An interface to a software product that is graphical rather than textual. It refers to window-based operational characteristics.

I

implementation function. Function written by a third-party developer for a plug-in node or parser. Also known as a *callback function*.

input node. A message flow node that represents a source of messages for the message flow.

installation mode. The installation mode can be Full, Custom, or Broker only. The mode defines the components of the product installed by the installation process on Windows NT systems.

J

Java™ Database Connectivity (JDBC). An application programming interface that has the same characteristics as **ODBC** but is specifically designed for use by Java database applications.

Java Development Kit (JDK). A software package that can be used to write, compile, debug, and run Java applets and applications.

Java Message Service (JMS). An application programming interface that provides Java language functions for handling messages.

Java Runtime Environment (JRE). A subset of the Java Development Kit (JDK) that contains the core executables and files that constitute the standard Java platform. The JRE includes the Java Virtual Machine, core classes and supporting files.

JDBC™. Java Database Connectivity.

JDK™. Java Development Kit.

JMS. Java Message Service. See also *AMI* and *MQI*.

JRE. Java Runtime Environment.

L

local error log. A generic term that refers to the logs to which MQSeries Integrator writes records on the local system. On Windows NT, this is the Event log. On UNIX® systems, this is the syslog. See also *system log*. Note that MQSeries records many events in the log that are not errors, but information about events that occur during operation, for example, successful deployment of a configuration.

M

message broker. A set of execution processes hosting one or more message flows.

messages. Entities exchanged between a broker and its clients.

message dictionary. A repository for (predefined) message type specifications.

| **message domain.** The value that determines how the
| message is interpreted (parsed). The following domains
| are recognized:
| • MRM, which identifies messages defined using the
| Control Center
| • NEONMSG[3], which identifies messages created using
| the NEONFORMATTER user interfaces.
| • XML, which identifies messages that are self-defining
| • BLOB, which identifies messages that are undefined

| You can also create your own message domains: if you
| do so, you must supply your own message parser.

message flow. A directed graph that represents the set of activities performed on a message or event as it passes through a broker. A message flow consists of a set of message processing nodes and message processing node connectors.

message flow component. See *message flow*.

message parser. A program that interprets a message bitstream.

message processing node. A node in the message flow, representing a well defined processing stage. A message processing node can be one of several primitive types or can represent a subflow.

message processing node connector. An entity that connects the output terminal of one message processing node to the input terminal of another. A message processing node connector represents the flow of control and data between two message flow nodes.

3. The message domain NEON is also recognized for compatibility with previous releases.

message queue interface (MQI). The programming interface provided by MQSeries queue managers. The programming interface allows application programs to access message queuing services. See also *AMI* and *JMS*.

message repository. A database holding message template definitions.

| **message repository manager (MRM).** A component of
| the Configuration Manager that handles message
| definition and control. A message defined to the MRM
| has a message domain set to MRM.

message set. A grouping of related messages.

message template. A named and managed entity that represents the format of a particular message. Message templates represent a business asset of an organization.

message type. The logical structure of the data within a message. For example, the number and location of character strings.

metadata. Data that describes the characteristic of stored data.

MQI. Message queue interface.

| **MQIsdp.** MQSeries Integrator SCADA device
| protocol. A lightweight publish/subscribe protocol
| flowing over TCP/IP.

MQRFH. An architected message header that is used to provide metadata for the processing of a message. This header is supported by MQSeries Publish/Subscribe.

MQRFH2. An extended version of MQRFH, providing enhanced function in message processing.

| **MQSeries Everyplace.** A generally available MQSeries
| product that provides proven MQSeries reliability and
| security in a mobile environment.

| **MRM.** Message Repository Manager.

multilevel wildcard. A wildcard that can be specified in subscriptions to match any number of levels in a topic.

N

node. See *message processing node*.

O

ODBC. Open Database Connectivity.

Open Database Connectivity. A standard application programming interface (API) for accessing data in both relational and non-relational database management systems. Using this API, database applications can access data stored in database management systems on a variety of computers even if each database management system uses a different data storage format and programming interface. ODBC is based on the call level interface (CLI) specification of the X/Open SQL Access Group.

output node. A message processing node that represents a point at which messages flow out of the message flow.

P

plug-in. An extension to the broker, written by a third-party developer, to provide a new message processing node or message parser in addition to those supplied with the product. See also *implementation function* and *utility function*.

point-to-point. Style of messaging application in which the sending application knows the destination of the message. Compare with *publish/subscribe*.

POSIX. Portable Operating System Interface For Computer Environments. An IEEE standard for computer operating systems (for example, AIX® and Sun Solaris).

predefined message. A message with a structure that is defined before the message is created or referenced. Compare with *self-defining message*.

primitive. A message processing node that is supplied with the product.

principal. An individual user ID (for example, a log-in ID) or a group. A group can contain individual user IDs and other groups, to the level of nesting supported by the underlying facility.

property. One of a set of characteristics that define the values and behaviors of objects in the Control Center. For example, message processing nodes and deployed message flows have properties.

publication node. An end point of a specific path through a message flow to which a client application subscribes. A publication node has an attribute, subscription point. If this is not specified, the publication node represents the default subscription point for the message flow.

publish/subscribe. Style of messaging application in which the providers of information (publishers) are decoupled from the consumers of that information (subscribers) using a broker. Compare with *point-to-point*. See also *topic*.

publisher. An application that makes information about a specified topic available to a broker in a publish/subscribe system.

Q

queue. An MQSeries object. Message queuing applications can put messages on, and get messages from, a queue. A queue is owned and maintained by a queue manager. Local queues can contain a list of messages waiting to be processed. Queues of other types cannot contain messages: they point to other queues, or can be used as models for dynamic queues.

queue manager. A system program that provides queuing services to applications. It provides an application programming interface (the MQI) so that programs can access messages on the queues that the queue manager owns.

R

retained publication. A published message that is kept at the broker for propagation to clients that subscribe at some point in the future.

request/reply. Type of messaging application in which a request message is used to request a reply from another application. Compare with *datagram*.

rule. A rule is a definition of a process, or set of processes, applied to a message on receipt by the broker. Rules are defined on a message format basis, so any message of a particular format will be subjected to the same set of rules.

S

| **SCADA.** Supervisory, Control, And Data Acquisition.

self-defining message. A message that defines its structure within its content. For example, a message coded in XML is self-defining. Compare with *pre-defined message*.

send and forget. See *datagram*.

setup type. The definition of the type of installation requested on Windows NT systems. This can be one of **Full**, **Broker only**, or **Custom**.

shared. All configuration data that is shared by users of the Control Center. This data is not operational until it has been deployed.

signature. The definition of the external characteristics of a message processing node.

single-level wildcard. A wildcard that can be specified in subscriptions to match a single level in a topic.

stream. A method of topic partitioning used by MQSeries Publish/Subscribe applications.

subscriber. An application that requests information about a specified topic from a publish/subscribe broker.

subscription. Information held within a publication node, that records the details of a subscriber application, including the identity of the queue on which that subscriber wants to receive relevant publications.

subscription filter. A predicate that specifies a subset of messages to be delivered to a particular subscriber.

subscription point. An attribute of a publication node that differentiates it from other publication nodes on the same message flow and therefore represents a specific path through the message flow. An unnamed publication node (that is, one without a specific subscription point) is known as the default publication node.

Supervisory, Control, And Data Acquisition. A broad term, used to describe any form of remote telemetry system used for gathering data from remote sensor devices (for example, flow rate meters on an oil pipeline) and for the near real time control of remote equipment (for example, pipeline valves).

system log. A generic term used in the MQSeries Integrator messages (BIPxxx) that refers to the local error logs to which records are written on the local system. On Windows NT, this is the Event log. On UNIX systems, this is the syslog. See also *local error log*.

T

terminal. The point at which one node in a message flow is connected to another node. Terminals enable you to control the route that a message takes, depending whether the operation performed by a node on that message is successful.

topic. A character string that describes the nature of the data that is being published in a publish/subscribe system.

topic based subscription. A subscription specified by a subscribing application that includes a topic for filtering of publications.

topic security. The use of ACLs applied to one or more topics to control subscriber access to published messages.

topology. In the broker domain, the brokers, collectives, and connections between them.

transform. A defined way in which a message of one format is converted into one or more messages of another format.

U

Uniform Resource Identifier. The generic set of all names and addresses that refer to World Wide Web resources.

Uniform Resource Locator. A specific form of URI that identifies the address of an item on the World Wide Web. It includes the protocol followed by the fully qualified domain name (sometimes called the host name) and the request. The Web server typically maps the request portion of the URL to a path and file name. Also known as Universal Resource Locator.

URI. Uniform Resource Identifier

URL. Uniform Resource Locator

User Name Server. The MQSeries Integrator component that interfaces with operating system facilities to determine valid users and groups.

utility function. Function provided by MQSeries Integrator for the benefit of third-party developers writing plug-in nodes or parsers.

W

warehouse. A persistent, historical datastore for events (or messages). The **Warehouse** node within a message flow supports the recording of information in a database for subsequent retrieval and processing by other applications.

wildcard. A character that can be specified in subscriptions to match a range of topics. See also *multilevel wildcard* and *single-level wildcard*.

wire format. This describes the physical representation of a message within the bit-stream.

W3C. World Wide Web Consortium. An international industry consortium set up to develop common protocols to promote evolution and interoperability of the World Wide Web.

X

XML. Extensible Markup Language.

Bibliography

This section describes the documentation available for all current MQSeries Integrator products.

MQSeries Integrator Version 2.0.2 cross-platform publications

The MQSeries Integrator cross-platform publications are:

- *MQSeries Integrator Introduction and Planning*, GC34-5599
- *MQSeries Integrator Using the Control Center*, GC34-5602
- *MQSeries Integrator Messages*, GC34-5601
- *MQSeries Integrator Programming Guide*, SC34-5603
- *MQSeries Integrator Administration Guide*, SC34-5792
- *MQSeries Integrator ESQL Reference*, SC34-5923

These books are all available in hardcopy.

You can order publications from the IBMLink™ Web site at:

`http://www.ibm.com/ibmlink`

In the United States, you can also order publications by dialing 1-800-879-2755.

In Canada, you can order publications by dialing 1-800-IBM-4YOU (1-800-426-4968).

For further information about ordering publications contact your IBM authorized dealer or marketing representative.

MQSeries Integrator Version 2.0.2 platform-specific publications

| Each MQSeries Integrator product provides one
| platform-specific installation guide, which is
| supplied in hardcopy.

MQSeries Integrator for AIX Version 2.0.2
: *MQSeries Integrator for AIX Installation Guide*, GC34-5841

MQSeries Integrator for HP-UX Version 2.0.2
: *MQSeries Integrator for HP-UX Installation Guide*, GC34-5907

MQSeries Integrator for Sun Solaris Version 2.0.2
: *MQSeries Integrator for Sun Solaris Installation Guide*, GC34-5842

MQSeries Integrator for Windows NT Version 2.0.2
: *MQSeries Integrator for Windows NT Installation Guide*, GC34-5600

| MQSeries Everyplace | publications

| If you intend to connect MQSeries Everyplace
| applications to message flows that include the
| MQSeries Everyplace message flow nodes, you
| will find the following publications useful:

| - *MQSeries Everyplace for Multiplatforms Version 1.1 Introduction*, GC34-5843
| - *MQSeries Everyplace for Multiplatforms Version 1.1 Programming Guide*, SC34-5845
| - *MQSeries Everyplace for Multiplatforms Version 1.1 Programming Reference*, SC34-5846
| - *MQSeries Everyplace for Multiplatforms Version 1.1 Native Client Information*, SC34-5880

| You can find these books on the MQSeries Web
| site (see "MQSeries information available on the
| Internet" on page 143). Translated versions of
| these books are also available in some languages
| from the same Web site.

NEONRules and NEONFormatter Support for MQSeries Integrator publications

| The following publications are supplied on the
| product CD in PDF format, and are installed with
| the Documentation component.

- *NEONRules and NEONFormatter Support for MQSeries Integrator User's Guide*
- *NEONRules and NEONFormatter Support for MQSeries Integrator System Management Guide*

- *NEONRules and NEONFormatter Support for MQSeries Integrator Programming Reference for NEONRules*
- *NEONRules and NEONFormatter Support for MQSeries Integrator Programming Reference for NEONFormatter*
- *NEONRules and NEONFormatter Support for MQSeries Integrator Application Development Guide*

These books are provided in US English only.

Softcopy books

All the MQSeries Integrator books are available in softcopy formats.

Portable Document Format (PDF)

All books in the MQSeries Integrator library are supplied in US English only in a searchable PDF library on the product CD.

You can install the library as follows:

- On AIX, invoke `install –d` and select the documentation fileset. After installation, run the command `mqsidocs`. This launches Acrobat Reader and opens the PDF package.
- On HP-UX, invoke `swinstall –d` and select `MQSI-DOCS` from the menu. After installation, run the command `mqsidocs`. This launches Acrobat Reader and opens the PDF package.
- On Sun Solaris, invoke `pkgadd –d` and select `mqsi-docs` from the menu. After installation, run the command `mqsidocs`. This launches Acrobat Reader and opens the PDF package.
- On Windows NT, select the `Online Documentation` component on a custom installation, or do a full installation. After installation, select *Start—>Programs—>IBM MQSeries Integrator 2.0—>Documentation*.

In addition, PDF files for books that have been translated are installed into the location `mqsi_root/bin/book/pdf/<locale>` (on UNIX) or `mqsi_root\bin\book\pdf\<locale>` (on Windows NT) where `<locale>` is one of the following:
- **de_DE** for German
- **en_US** for US English
- **es_ES** for Spanish
- **fr_FR** for French
- **it_IT** for Italian
- **ja_JP** for Japanese
- **ko_KR** for Korean
- **pt_BR** for Brazilian Portuguese
- **zh_CN** for Simplified Chinese
- **zh_TW** for Traditional Chinese

An index file (in HTML format) that provides a link to each book is supplied for each language. For example, the French index file is called `indexfr.htm`. The files are stored in the following directory:
- On UNIX, `<mqsi_root>/docs/`
- On Windows NT, `<mqsi_root>\bin\book`

Each index file has an entry for every book: if a particular book has not been translated into the appropriate language for that index file, a link to the English PDF is included. You can use any Web browser to view the index file. On Windows NT, you can also access the index file through the *Start* menu.

The PDF file names for the English books are shown in Table 25.

Table 25. File names of MQSeries Integrator book PDFs

Book title	File name
MQSeries Integrator for AIX Installation Guide	bipaac04.pdf
MQSeries Integrator for HP-UX Installation Guide	bipcac00.pdf
MQSeries Integrator for Sun Solaris Installation Guide	bip7ac03.pdf
MQSeries Integrator for Windows NT Installation Guide	bipyac03.pdf
MQSeries Integrator Introduction and Planning	bipyab02.pdf
MQSeries Integrator Administration Guide	bipyag04.pdf
MQSeries Integrator Using the Control Center	bipyar03.pdf
MQSeries Integrator ESQL Reference	bipyae00.pdf
MQSeries Integrator Programming Guide	bipyal02.pdf
MQSeries Integrator Messages	bipyao02.pdf

The fifth character of the file name indicates the language of the book (**a** indicates US English). You can deduce the file names of translated books by using the following substitutions for the fifth character:
- **g** for German
- **s** for Spanish
- **f** for French

- **i** for Italian
- **j** for Japanese
- **k** for Korean
- **b** for Brazilian Portuguese
- **z** for Simplified Chinese
- **t** for Traditional Chinese

PDF files can be viewed and printed using the Adobe Acrobat Reader.

If you cut and paste examples of commands from PDF files to a command line for execution, you must check that the content is correct before you press `Enter`. Some characters might be corrupted by local system and font settings.

If you need to obtain the Adobe Acrobat Reader, or would like up-to-date information about the platforms on which the Acrobat Reader is supported, visit the Adobe Systems Inc. Web site at:

```
http://www.adobe.com/
```

PDF versions of all current MQSeries Integrator books are also available from the MQSeries product family Web site at:

```
http://www.ibm.com/software/mqseries/
```

MQSeries information available on the Internet

The MQSeries product family Web site is at:

```
http://www.ibm.com/software/mqseries/
```

By following links from this Web site you can:

- Obtain latest information about the MQSeries product family.
- Access the MQSeries books in HTML and PDF formats.
- Obtain information about complementary offerings by following these links:
  - IBM Business Partners
  - Partner Offerings (within *Related links*)
- Download an MQSeries SupportPac™.

Index

Sending your comments to IBM

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM.

Feel free to comment on what you regard as specific errors or omissions, and on the accuracy, organization, subject matter, or completeness of this book.

Please limit your comments to the information in this book and the way in which the information is presented.

To make comments about the functions of IBM products or systems, talk to your IBM representative or to your IBM authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate, without incurring any obligation to you.

You can send your comments to IBM in any of the following ways:

- By mail, to this address:

  User Technologies Department (MP095)
  IBM United Kingdom Laboratories
  Hursley Park
  WINCHESTER,
  Hampshire
  SO21 2JN
  United Kingdom
- By fax:
  - From outside the U.K., after your international access code use 44–1962–816151
  - From within the U.K., use 01962–816151
- Electronically, use the appropriate network ID:
  - IBM Mail Exchange: GBIBM2Q9 at IBMMAIL
  - IBMLink: HURSLEY(IDRCF)
  - Internet: idrcf@hursley.ibm.com

Whichever method you use, ensure that you include:

- The publication title and order number
- The topic to which your comment applies
- Your name and address/telephone number/fax number/network ID.

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC34-5923-00