MQSeries Integrator V2
FIX message adapter
Version 1.0

17th April 2001

Jim MacNair
MQSeries Sales Support
IBM
Somers, NY
USA

macnair@us.ibm.com

Take Note!

Before using this report be sure to read the general information under "Notices".

First Edition, April 2001

This edition applies to Version 1.0 of *MQSeries Integrator V2 – Fix message adapter* and to all subsequent releases and modifications unless otherwise indicated in new editions.

Table of Contents

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates.

Any reference to an IBM licensed program or other IBM product in this publication is not intended to state or imply that only IBM's program or other product may be used. Any functionally equivalent program that does not infringe any of the intellectual property rights may be used instead of the IBM product.

Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, New York 10594, USA.

The information contained in this document has not be submitted to any formal IBM test and is distributed AS-IS. The use of the information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item has been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

Trademarks and service marks

The following terms, used in this publication, are trademarks or registered trademarks of the IBM Corporation in the United States or other countries or both:

- IBM
- MQSeries
- MQSeries Integrator
- MQSI

The following terms are trademarks or registered trademarks of other companies:

Windows NT, Windows 2000, Visual Studio	Microsoft Corporation

Acknowledgments

The author would like to acknowledge the help that was received from a number of individuals. First, Malcolm Ayres, Peter Lambros and Phil Coxhead from the IBM Hursley Laboratories provided invaluable and tremendously useful information many times. They also displayed remarkable patience with the repeated questions and problems that they were constantly bombarded with. Second, Neil Kolban of the Dallas Systems Center was very helpful at numerous times, both for his expertise on both MQSeries Integrator as well as the Microsoft C++ development environment. Finally, the author wishes to thank the many people who also helped but who the author has unintentionally omitted from this brief mention.

Summary of Ammendments

Date	Changes
April 2001	Initial release

Preface

The FIX standard is commonly used within the finance (brokerage) industry, primarily for the exchange of electronic data related to securities transactions.

This SupportPac contains an adapter (parser and metadata) for FIX messages. The adapter is designed and written for the MQSeries Integrator Version 2.0.1 environment. The supplied parser is designed to operate in the Windows NT and Windows 2000 environment, and the IBM AIX environment. Steps have been taken to make the source code platform independent.

The SupportPac adds support to MQSI V2.0.1 for parsing of input messages and creation of output messages in the FIX format. Metadata is provided with this SupportPac for levels 4.2, 4.1 and 4.0 of the FIX standard. The SupportPac includes metadata and executable programs for the Windows NT/2000 and AIX environments, as well as documentation and the source programs.

The source programs and documentation are useful examples for anyone who is planning to write a parser.

All executable programs provided with this SupportPac have been compiled for the Windows environment using the Microsoft Visual C++ V6.0 compiler, and for the AIX environment using the IBM xlc_r compiler. The programs are largely written to the ANSI C standard, with some use the Microsoft Foundation classes in the Windows version. The necessary MFC support has been statically linked with the executable programs, so no additional DLLs should be required to run the supplied executable programs.

This document includes a general overview of the FIX message formats, and some additional details on how parsers work in an MQSeries Integrator Version 2.01 environment. These sections are informational and are not required to install and use this SupportPac.

A general understanding of MQSeries Integrator V2.0.1 is necessary to use this SupportPac. If the function of the parser is to be changed or customized by modifying the provided source code, then an in depth knowledge of MQSeries Integrator V2.0.1, as well as C programming, is required.

Chapter 1. Introduction to the FIX standards

The FIX organization is responsible for the maintenance and development of the Electronic Data Interchange (EDI) standards for the exchange of data related to securities markets. These standards allow industry partners to electronically exchange information such as policy/submission, claim and accounting data.

The FIX standards were originally developed approximately nine years ago, and have been continually maintained and enhanced since.

A brief introduction to FIX messages is contained in this document for the convenience of the reader. For additional information on the FIX standards, please visit the FIX web site (see below).

FIX message format

The FIX message format is designed to allow the exchange of securities related information between different organizations. The FIX message formats are defined and maintained by the FIX organization, which can be located on the web at:

http://www.fixprotocol.org

In their own words, "The message protocol, as defined, will support a variety of business functions. FIX was originally defined for use in supporting US domestic equity trading with message traffic flowing directly between principals. As the protocol evolved, a number of fields were added to support limited cross-border and fixed income trading. Similarly, the protocol was expanded to allow third parties to participate in the delivery of messages between trading partners. As subsequent versions of FIX are released, it is expected that functionality will continue to expand."

The actual FIX standard is contained in a Microsoft Word document. Each data element is assigned a numeric identifier, name and data type in the document. These names are used in the metadata files that are included with this SupportPac.

What does a FIX message look like?

A FIX message consists of a series of delimited tag and value pairs. A special non-printable ASCII character (SOH = #001) at the end of the value delimits each tag and value pair. The type and value are separated by an equal sign. Each individual item is of the format:

```
tag=value<SOH>
```

Each data item contains data of a specified type, which can include numbers (either integers or floating point), Boolean, date and/or time, string or character data or raw binary data. All values are given in a character format. For example, a Boolean value should contain either a "Y" or an "N". Integers should consist of numeric digits only, while floating point values can include a decimal point (but not an exponent).

Tags and values must not contain the delimiter character (SOH = 0x01). There is an exception however. Fields with a data type of "data" (raw data) may contain one or more delimiter characters within the data. All data elements of type data must be preceded by a length field, which specifies the length of the data. The length field must be of type integer.

Headers and Trailers

A FIX message must contain a standard FIX header and trailer. The header must begin with a "Begin String" data element (tag #8), which contains a constant to indicate the level of the standard used to create the message. This BodyLength (tag #9) and MsgType (tag #35) fields must be the second and third fields respectively, and follow immediately after the BeginString data element. For example, if the FIX 4.2 standard was used, the beginning of the message should look like the following, where <SOH> represents the delimiter character (#001):

```
8=FIX.4.2<SOH>9=nnn<SOH>35=X<SOH>
```

The last data element in the message should be the CheckSum (tag #10) data element. The data portion of the CheckSum data element must be three digits and must be followed by a delimiter character. For example, a CheckSum data element should look like the following:

```
10=nnn<SOH>
```

Any additional header fields must immediately follow the MsgType (tag #35) element, and any additional trailer elements must immediately precede the CheckSum element. The body elements can normally appear in any order. The only exception is fields within repeating data groups, which must occur in the same order as specified in the repeating group specification in the FIX standard.

Message Types

The data elements that are used in a FIX message will vary with the message type. For each message type, the data elements that are required and the data elements that are optional are specified in the FIX standard. The parser in this SupportPac does not enforce the presence of required fields beyond those that are needed to correctly parse a FIX message (basically, the first three header items plus the check sum item and any length fields that must precede raw data items).

What types of data are contained in FIX messages?

All elements in a FIX message are in character format (e.g. no binary, packed decimal data, etc), with the exception of the raw data type. Numeric data is represented as a string of numbers, possibly with a sign character for negative numbers and a decimal point for floating point numbers. If the number is negative, then a minus sign character precedes the numbers.

No extra blank, character return, line feed, tab or other characters may appear within the message.

Raw data fields are used to carry data that may contain any possible value in each byte of data. For example, encrypted data is defined as raw data, and may contain any possible value in any position, including delimiter characters, null characters, etc. Raw data fields must be preceded by a data element that specifies the length of the data in the raw data field. The length field must be in integer (digits) format, and has its own identifier, equal sign and delimiter. For example, if encrypted data is to be transmitted, the portion of the message containing the encrypted data should be in the following format:

```
90=nnn<SOH>91=<encrypted data bytes of length nnn><SOH>
```

An electronic signature, also of type raw data, should be in the following format:

```
93=nnn<SOH>89=<signature data of length nnn><SOH>
```

The FIX standard defines the following data types. All data types except for raw data are expressed in a character format, and may not contain delimiter (SOH) characters.

Compression and Encryption

FIX messages consist of normal character data. There is support within the standard for message encryption and encryption of certain fields. Certain fields in the header and trailer parts of the message must not be encrypted and are sent in clear text. Fields can be repeated within the encrypted section of the message. If a field is repeated in the encrypted section of the message, the encrypted form is considered more reliable. The type of encryption used is determined in a logon sequence and is sent in the EncryptMethod (tag #98) data element.

The parser contained in this SupportPac does not include support for any particular encryption or decryption routines. Rather, entry points are provided where a particular encryption and/or decryption routine could be inserted. The metadata for the parser indicates which fields contain encrypted data, and whether the encrypted data is the data for a single tag item or contains any number of encrypted tag and data items.

Any encryption and decryption routines to be used must be thread-safe, since the MQSI V2 environment is multi-threaded. They must also not interfere in the normal operation of MQSI V2. In general, this means not starting extra threads, not closing threads or making other changes to the runtime environment that are incompatible with MQSI V2.

The metadata provided includes flags set to indicate which data is to be encrypted and which is to be decrypted. A separate message domain of “FIXONLY” is supported which can be used when encryption and decryption is not supported. In this case, any raw data input fields will be rendered as byte arrays. For output, raw data fields should be specified as byte arrays.

FIX Metadata

The official FIX standards are stored in Microsoft Word documents. Each level of the standard is a separate level. The documents contain a description of every data tag defined in a level of the standard.

A text file has been created based on the document for FIX 4.2 and was used to create the metadata file shipped with this SupportPac. Files for FIX 4.1 and FIX 4.0 levels of the standard were also created and are included. A separate text file is used for support of repeated data items. This is also provided for all three supported levels of the FIX standard.

Versioning Support

FIX messages contain the standard version in the BodyString (tag #8) data element in the header portion of each message. The general format of the version is the characters FIX followed by a period, the version number, another period and the modification level. For example, a message that uses version 4 of the standard at modification level 2 would be specified as:

FIX.4.2

How versioning is handled by the Parser

The metadata for each version of the standard is contained in a single metadata file. The file name consists of the standard number that is contained in the standard FIX message header, with the periods removed. All of these metadata files have a file extension of “mtx”. For example, the metadata file for the 4.2 version of the FIX standard would be:

FIX42.mtx

Metadata Files

The metadata used by the FIX parser are kept in files. Every version is kept in a separate file. The metadata contains the type, data name and data type for each element defined in a particular version of the FIX standard, as well as other characteristics.

Chapter 2. Installation

SupportPac contents

The supplied zip file should be unzipped into a parent directory. Certain common files will be unzipped into this directory and the following subdirectories will be created under the parent directory:

- src
- utilsrc
- aix (and aix\msgcat)
- win
- sample

Prerequisites

This SupportPac provides a parser to be used with the IBM MQSeries Integrator Version 2.0.1 and above. For normal use, there are no other pre-requisite products other than those required by MQSeries Integrator Version 2.0.1 itself. If any changes are to be made to the parser or the related utilities, then an appropriate C compiler, such as Microsoft Visual C++ V6 or one of the supported C compilers for Unix, is required.

Supported Platforms

This SupportPac has been developed for and tested in the Windows 2000 and AIX environments. Windows NT V4 is also supported. Input and output messages can be in any single-byte code page. Input and output messages in Unicode or which contain double-byte characters are not supported.

Installing the executable programs

The zip file should be unzipped into a parent directory. The following files should be created in the main directory:

- ia0c.pdf

The following files should be found in the ***win*** subdirectory:

- buildmtx.exe
- FIX40.mtx
- FIX41.mtx
- FIX42.mtx
- fixerr.dll
- fixparser.dbg
- fixparser.enc
- fixparser.lil
- MessageFlowsFix
- printmtx.exe

The following files should be found in the **aix** subdirectory:

- FIX40.mtx
- FIX41.mtx
- FIX42.mtx
- fixerr.h
- fixparser.dbg
- fixparser.lil
- Makefile

- buildmtx
- printmtx

The following files should be found in the ***msgcat*** subdirectory under the ***aix*** subdirectory:

- fixerr.cat
- fixerr.msg
- Makefile

The following files should be created in the ***utilsrc*** subdirectory :

- buildmtx.c
- buildmtx.dsp
- buildmtx.dsw
- comsubs.c
- comsubs.h
- crmtx.cmd
- data40.txt
- data41.txt
- data42.txt
- mtxstruc.h
- printmtx.c
- printmtx.dsp
- printmtx.dsw
- repeat40.txt
- repeat41.txt
- repeat42.txt

The following files should be created in the ***src*** subdirectory :

- buildmsg.cmd
- common.h
- context.h
- fixerr.h
- fixerr.mc
- fixparser.cpp
- fixparser.dsp
- fixparser.dsw
- fixparser.h
- fixparser.ncb
- fixparser.opt
- fixparser.plg
- fixsub.cpp
- fixsub.h
- metadata.cpp
- metadata.h
- metamem.h
- mgmtmsg.cpp
- mgmtmsg.h
- movelil.cmd
- moveonly.cmd
- moverel.cmd
- mtdstruc.h
- parsubs.cpp
- parsubs.h
- trace.cpp
- trace.h

Installation in the Windows NT/2000 environments

The parser executable (fix*parser.lil*) should be moved into the bin subdirectory of the MQSeries Integrator Version 2 root directory (default is c:\Program Files\IBM MQSeries Integrator 2.0.1\bin). The error message dictionary (fixerr.dll) should be moved to the messages subdirectory of the MQSeries Integrator root directory (default is c:\Program Files\IBM MQSeries Integrator 2.0.1\messages). The debug version of the parser (fixparser.dbg) should also be moved to the bin directory.

The MessageFlowsFix file contains two sample message flows that can be used to validate the proper functioning of the parser. If they are to be used or viewed, the message flows must be imported into a configuration manager using the import function of the control center, and then assigned to an execution group and deployed. The following local queues are used by the sample message flows and therefore must be defined if the sample message flows are to be used:

- FIX.IN
- FIX.OUT
- FIXML.IN
- FIXML.OUT
- FAILURE

The utility executables (buildmtx.exe and printmtx.exe) and related files should be moved to a program directory. This can be the same directory as the parser or it can be a separate directory.

Installing the metadata

A directory should be created on the broker system for the metadata files. The default for this directory name is “C:\Fix”. The metadata files (FIX42.mtx, FIX41.mtx and FIX40.mtx) should be moved to this directory. If another location is to be used, then the FIXMETADIR environment variable must be set.

Defining the message dictionary in the registry

Finally, an entry must be made in the Windows NT registry for the message dictionary. To do this with the registry editor, go to the Windows start button and select Run. Type regedit in the pop up edit box and press enter. The registry editor should start.

The registry editor should show five high level keys, with small plus signs next to them. Select the HKEY_LOCAL_MACHINE and press the small plus sign next to it. This should expand the entries under HKEY_LOCAL_MACHINE. In a similar fashion, select the following entries in order

SYSTEM->CurrentControlSet->Services->EventLog->Application

Highlight the Application entry and click the right mouse button. Select the following options:

New->Key

Enter “fixerr” (without the double quotes) as the name of the key. Select the new fixerr entry and click the right mouse button. Select the following options:

New->String Value

Change the name of this new entry to “EventMessageFile” by typing over the generated name. Click on the EventMessageFile value item and select Modify. Type in the fully qualified path name where the executable message file (fixerr.dll) was installed. For example, with a default MQSI Version 2 installation, this would be as follows:

“C:\Program Files\IBM MQSeries Integrator 2.0.1\messages\fixerr.dll”

Finally, click on the fixerr entry again and select the following options:

New->DWORD value

Change the generated name to “TypesSupported” and then select the new value and click the right mouse button. Select Modify. Change the value to 7. Close the registry editor. The message catalog should now be installed.

Installation in an AIX environment

The executable programs and metadata for the AIX environment are contained in the AIX subdirectory that is created when the main zip file is unzipped. The AIX files must be uploaded to the AIX system, using a utility like FTP. Be sure to upload the executable programs, message catalog and metadata files in binary mode and the text files in ASCII mode.

Installing the executable programs and message catalog

The parser executable programs must be uploaded to the lil subdirectory located under the MQSI Version 2 root. For example, if the MQSI V2 executable programs are located under the /usr/opt/mqsi directory, then the parser.lil and parser.dbg executable programs should be uploaded to a directory of /usr/opt/mqsi/lil.

The message catalog (fixer.cat) in the AIX\msgcat directory must be uploaded in binary to the /usr/opt/mqsi/messages directory.

Installing the metadata

The default directory name for the metadata files is /var/fix. The metadata files (FIX42.mtx, FIX41.mtx and FIX40.mtx) in the AIX subdirectory should be uploaded in binary to this directory. If a different directory is to be used, then the FIXMETADIR environment variable must be set to point to the chosen directory, and the files uploaded to that directory rather than /var/fix. The permissions on the directory must be set to allow the message broker runtime to read files in the directory. If the debug version of the parser is to be used, then the message broker runtime also needs write permission to the /tmp directory (or to another directory selected by the FIXTRACEFILE environment variable), so that it can write to its trace file. The offline metadata utility programs (buildmtx and printmtx) should also be uploaded in binary to the /var/fix directory.

Be aware that the metadata files contain binary and Unicode data and are therefore platform dependent. The metadata files for Windows platforms will not work in a Unix environment (and vice versa).

Working with the parser source programs on AIX

If the source programs are to be used in the AIX environment, then the source programs must be uploaded to a suitable source directory. The source programs are provided in two subdirectories. The first subdirectory (src) contains the source for the parser executable itself, while the second directory (utilsrc) contains the source for the offline utilities to build and display the metadata files. It is important that the source programs be uploaded in text rather than binary mode.

The following source files must be uploaded from the src subdirectory:

- fixparser.cpp as fixparser.c
- fixsub.cpp as fixsub.c
- metadata.cpp as metadata.c
- mgmtmsg.cpp as mgmtmsg.c
- parsubs.cpp as parsubs.c
- trace.cpp as trace.c

The file extensions of the seven source programs for the parser must be changed from “cpp” to “c”.

The following header files should be uploaded from the src subdirectory:

- common.h
- comsubs.h
- context.h
- fixparser.h
- fixsub.h
- metadata.h
- metamem.h
- mgmtmsg.h
- mtxstruc.h
- parsubs.h
- trace.h

All the header files except for fixerr.h and stdafx.h should be uploaded. . The fixerr.h file found in the AIX directory must also be uploaded in text mode. This file is used in place of the NT equivalent found in the src directory.

Copy the BipSampPluginUtil.c and BipSampPluginUtil.h files from the ${MQSI_ROOT}/sample/plugin directory into the same directory as the other source programs

The Makefile found in the AIX subdirectory should be uploaded as Makefile. The Makefile should be examined and any necessary changes made to match the directory names and compiler executable. The fixerr.h file found in the AIX directory should be uploaded. This file is specific to AIX and is different from the fixerr.h file found in the source subdirectory. Finally, the message catalog source (fixerr.msg) should be uploaded.

Once all of the source programs and the Makefile have been uploaded, the parser executable can be built. Please note that the Makefile that is provided has the _DEBUG compiler switch set. This will result in a debug build being produced. This variable definition should be removed from the Makefile for a production build. Please delete all intermediate object files before switching build types (e.g. perform a make clean or rm *.o). Once the new parser executable file (fixparser.lil), the broker runtime must be stopped and the parser executable moved to the lil directory.

Working with the metadata utility source programs on AIX

If the metadata utility programs are to be changed, then the following files should be uploaded in text mode from the utilsrc subdirectory:

- buildmtx.c
- printmtx.c
- comsubs.c
- comsubs.c
- comsubs.h
- mtxstruc.h
- data42.txt
- data41.txt
- data40.txt
- repeat42.txt
- repeat41.txt
- repeat40.txt

The utility programs can be compiled with a simple statement such as:

```
/usr/vac/bin/cc buildmtx.c –o buildmtx
```

Environment variable for the metadata files

A directory for the metadata files should be created and the corresponding zip file (metadata.zip) should be unzipped into this directory. The following environment variables should be set:

- FIXMETADIR – drive and directory containing metadata files.

If this variable is not set, then the parser will expect the metadata files to be located in a directory named “Fix” on the “C:” drive. For Unix systems, the metadata files are located in the /var/fix directory by default. The Windows and Unix versions of the metadata files are similar in structure and size but they are not the same.

Adding user defined tags to the metadata

The FIX standard reserves tag identifiers from 5000 to 9999 for user defined tag values. These tag values are generally unique to a particular enterprise. If user defined tags are used, then these tags should be added to the data4x.txt files that the metadata files are generated from and the metadata files should be rebuilt with the buildmtx offline utility.

The text input files contain one line for each tag item. Each line contains four or more values separated with commas. There is no comma at the end of the line. The first value is the tag identifier. The tag identifier should consist of four or five digits from 5000 through 99999. The second value should be the name that is to be used for the identifier. The name will be used to refer to the item when it is being used in a message flow. The name must be unique and must not be the same as any other tag, including tags defined in the FIX standard. The third field is the data type. The fourth field is the flags field. The fifth field is used for the flags2 value and the sixth field is used for raw data items to indicate the identifier of a length field tag that should be immediately preceding the raw data item. The fifth and sixth items are optional and are generally not used.

For a list of the allowed values for the data type and flags fields, see the section entitled “Using the offline utilities” later in this document.

Installation Verification

The following procedure should verify the proper installation and operation of the FIX adapter. Several simple test messages and two sample message flows have been provided with this SupportPac. The sample messages and flows verify that the adapter can properly handle incoming messages in FIX format and build output messages in FIX format.

The sample flow FIX2XML will take an input message in FIX format and produce a corresponding message in XML format. This flow will expect its input in the FIX.IN queue and will route output messages to the FIXML.OUT queue. The sample flow XML2FIX will take an input message in an XML format and produce a proper FIX message as output. This flow will look for input messages in the FIXML.IN queue and route output messages to the FIX.OUT queue. If errors are encountered in either flow, the input messages will be routed to the FAILURE queue.

The sample messages are provided as files in the **sample** subdirectory. They require a program to read the files and write the corresponding data to the suggested input queues. MQSeries SupportPac IH03 contains utilities that will perform this function.

To verify proper functioning of the adapter, perform the following steps:

1. Install the adapter.
2. Create the queues that are required by the sample message flows (FIX.IN, FIX.OUT, FIXML.IN, FIXML.OUT, FAILURE).
3. Import the sample message flows, assign them to an execution group and deploy the execution group.

4. Using the IH03 utilities (or another utility with equivalent function), read the testfix2.txt or the testfix4.txt file and send it to the FIX.IN queue.
5. Check that a message has been written to the FIXML.OUT queue. Read the message from the queue and save it as a temporary text file, using the IH03 utility (or equivalent).
6. Using the IH03 utilities (or equivalent), read the testfix2.xml or testfix4.xml file and write it to the FIXML.IN queue.
7. Check that a message has been written to the FIX.OUT queue. Read the message from the queue and save it as a temporary file, using the IH03 utility or equivalent. Examine the message and verify that the message is a valid FIX message.

Some additional test messages have been provided.

Testing encryption function

There are many types of encryption and decryption in common use. This SupportPac does not include support for any particular type of encryption. It is designed for the easy insertion of the desired encryption and decryption methods that are used by a particular installation.

However, it may be desirable to test the processing of encrypted messages without having to install a proper encryption and decryption method. A special version of the parser is provided in the test directory, which includes a fairly simple obfuscation routine that will simulate what encrypted data might look like and allow for the testing of message flows using messages containing encrypted data. Some sample messages are provided that have been built with this obfuscation routine used in place of any encryption methods. The special version of the parser is available only in a debug version with full tracing enabled.

To use this version of the parser to become familiar with the handling of encrypted data, the special version of the parser (fixparser.enc) contained in the test directory must have the extension renamed to “lil” and moved to the proper executable directory. The testfix7.txt file contains a sample message with a psuedo-encrypted section, while the testfix7.xml file contains an XML message with a proper structure to create a message with a pseudo-encrypted section.

Once these additional test messages have been used to gain a working knowledge of the use of encrypted data with the FIX adapter, the normal production version of the adapter should be reinstalled.

If this function is to be tested on the AIX environment, a version of the parser that supports both debug and encrypt must be built from the source programs. The Makefile used to build the parser must be modified to include an additional definition for _ENCRYPT_TEST.

Additional considerations

If the debug version of the parser is installed, additional environmental variables should be set, as described in the debugging section below.

If the source code for the parser is to be installed, then a directory for the source code and related files should be created. The appropriate zip file (source.zip) should then be unzipped into this directory.

If the metadata files are to be displayed, changed or rebuilt, then the corresponding zip file (utility.zip) should be unzipped into an executable directory.

Chapter 3. Using the Parser

Message Domains

Each incoming and outgoing message is assigned to a message domain. For incoming messages, the domain can be specified in the message in an RFH2 header, or can be specified in the defaults tab of the properties of an MQInput node. For output, the domain name is determined by the name of the body element.

The parser supplied with this SupportPac supports two additional domains, namely FIX and FIXONLY. The only difference between the two domains if that encryption and decryption support is suppressed in the FIXONLY domain, and any encrypted or decrypted fields will be rendered as byte arrays in the logical message tree.

General data structure

The FIX parser takes input messages in valid FIX formats and creates MQSeries Integrator V2 logical message tree structures that can then be processed by MQSeries Integrator message flows. Similarly, it will take a logical message tree created by a message flow and produce the data portion of an MQSeries message in a valid FIX format.

The parser will create the message tree from an input message in a certain specified format, and this format must be followed when a message tree is built in an MQSeries Integrator V2 message flow.

All logical message trees used within MQSeries Integrator V2 have a certain basic structure. There is a single high-level element known as the root element. The user data is found in the body of the message. The body has a single high-level element that is the last child of the root element. For a FIX message, the name of this element should be “FIX”, to match the message domain supported by the parser.

Naming of data elements

Each data element has an associated name in the metadata file. These names are the same that are used A second naming convention is also supported, based on the element type numbers used in the actual FIX messages. The alternate name is created by appending the element type number (without any leading zeros) to the letters “FX”.

The alternate names are supported to allow the parser to handle user data types that are not contained in the metadata, and to support versions of the standard for which no metadata file is available.

Determining the names of the data items

The contents of the metadata file can be displayed with the printmtx utility. This utility will create a list file from a metadata file. The list file will have a file name that is the same as the file name of the original metadata file, but with an extension of “msg”. The list file can be browsed with a text editor, such as the notepad utility.

Input Messages

The FIX parser will parse any inbound message that is in a valid FIX format. It will perform limited checking of the message format. The FIX parser will register with the execution broker for a message domain of “FIX”. It will attempt to parse any input message that is read by MQSeries Integrator V2 that is assigned to the FIX domain. The message domain can be specified in an RFH2 header in the message itself, or as a default property on the MQInput node of an MQSeries Integrator V2 message flow.

The parser will create a logical message tree that reflects the contents of the message. The name of the top-level element of the body will be “Fix”. The rest of the data for a particular message will be built as a logical data structure under this high level element.

The data translation option of the MQInput node should not be used, since a FIX message may contain non-character data in raw data fields. The message text is translated to Unicode, so any earlier translations are not required and merely increase overhead.

Input messages can be in either ASCII or EBCDIC. The message text including all tag names and character data must match the code page in the MQSeries message descriptor.

Encrypted Input

Encrypted input is identified in the metadata for an individual tag. The tag must have a data type of raw data and must be preceded by a tag that contains the length of the raw data. There are two kinds of encrypted data tags, namely tags where the data for the individual tag is encrypted and tags where the encrypted data consists of additional tag and value items.

If a tag contains additional tag and value items, the tag and the preceding length tag will not appear in the input message. The data will be decrypted and the tags and data in the decrypted data will be inserted into the logical message tree as if the item had not been encrypted.

Field Names

Two types of names for the data elements are supported.

The second type of field that is treated specially is the header field. All groups contain the same ten-byte header at the front of the group, and many groups contain an additional twenty-byte header extension. The individual data elements (fields) in the header are defined individually in the metadata files.

Output Messages

The message flow must create an output message in a format similar to the input messages as described above.

The data in the logical message tree should be in the same order as the individual elements are found in the group definition. When building an output message, the parser will build the message from left to right. It will use the top- level elements in the logical message tree to understand what group needs to be inserted into the output message and in what order.

The parser will attempt to locate the metadata file that is to be used for the output data based on the name of the root element for each group. If a long name is used for the name of this element, it will be looked up in the segment name data and converted to a four-character group identifier. If the name begins with an underscore character, then the second through fifth characters of the element name will be assumed to be the group identifier.

The parser will use the metadata definition when attempting to match the data elements in the output logical message tree to elements in the metadata file. The parser will use the metadata file to construct the output data area for the group. It will first initialize the output data area, using the initialization string found in the metadata file. It will then step through the individual fields in the metadata file, attempting to match each element to data in the logical message tree. When an element is found in the message tree that does not exist in the metadata, it will be ignored. If a field is not found in the logical message tree, the initialization value for the field will be used. It is important that the fields in the logical message tree are in the same order as the fields in the metadata file. If fields are not in the same order, then some of the values in the logical message tree will be skipped over when trying to find an earlier value, and the parser will not look at previous values in the message tree when attempting to match the later field in the metadata file.

The first two fields in the message header (group identifier and length), and the group version, are already filled complete in the initialization string contained in the metadata file, and thus may be omitted. Fields designated as reserved or deleted should similarly be omitted, although these fields will be treated as any other field if values are set in the logical message tree.

Any fields that are not found in the logical message tree will be set to either blanks or zeros, depending on the type of field.

Encrypted output

If encrypted output is desired, then the appropriate encryption routines must be added to the parser source and the parser must be rebuilt, as described in the Using the Source Code section of this document.

There are two types of encrypted output that can be produced. Either the data for a single field can be encrypted or a section of the message consisting of additional tags and values can be encrypted.

All encrypted fields must be of type raw data. Raw data fields require a length field as the preceding field in the FIX message. This raw data length field will have its own tag associated with it. In addition, one of the encryption flags must be set in the metadata entry for the raw data, and the identifier of the raw data length field must be indicated in the same metadata entry. Two encryption flags can be set in the flag field of the metadata for a raw data field, as follows:

- Individual field with encrypted data
- Encrypted portion of message, containing tag and value pairs in FIX format

Since the length of the encrypted data may be different than the length of the input clear text, the appropriate raw data length field will be generated and inserted in the message automatically. Therefore, no raw data length field is needed in the logical message tree. The tag for the length field must be indicated in the metadata for the raw data item itself.

If an individual field is to be encrypted, then the metadata for the particular FIX tag must have the FLAGS_DECRYPT_ONLY (#64 = 0x40) bit set in the flags. The value of the raw data item will be obtained from the logical message tree and translated to the appropriate output code page if necessary. The output data will then be passed to the encryption routine, which will return the encrypted data and the length of the encrypted data. The appropriate raw data length field and raw data field will then be inserted into the output message. For example, if the logical message tree contained the following item:

Fix. EncodedText='This is some text to be encoded'

The output would consist of two items and would be of the following format:

354=nnn<SOH>355=(encrypted data of length nnn)<SOH>

Please note that no parentheses are generated in the output message and that the <SOH> sequence represents the FIX delimiter character of 0x01.

If a portion of the message is to be encrypted, then the logical message should have a name element that is a parent of the portion of the message that is to be encrypted. The name of this parent element should match the name of the raw data item defined in the metadata (e.g. FX91 or SecureData) and the FLAGS_DECRYPT_PARSE (#32 = 0x20) bit should be set in the flags for this item. The logical message under this parent element should contain the normal items that are to be included in the encrypted portion of the message. The items that are children of the parent element will be built into a partial FIX message, and the results will then be passed to the encryption routine as a single data item. The encryption routine will return a single encrypted data item and the length of the encrypted item. The appropriate raw data length field and raw data field will then be inserted into the output message.

For example, assume that a logical message tree includes the following items:

```
Fix.SecureData
Fix.SecureData.MsgSeqNum='233'
Fix.SecureData.SendingTime='20010322-19:12:00'
Fix.SecureData.QuoteID='37829B33'
Fix.SecureData.QuoteReqID='99732/44'
Fix.SecureData.NoQuoteSets='2'
Fix.SecureData.QuoteSet.QuoteSetID='1'
Fix.SecureData.QuoteSet.UnderlyingSymbol='IBM'
Fix.SecureData.QuoteSet.TotQuoteEntries='2'
Fix.SecureData.QuoteSet.NoQuoteEntries='2'
Fix.SecureData.QuoteSet.QuoteEntry.QuoteEntryID='1'
Fix.SecureData.QuoteSet.QuoteEntry.MaturityMonthYear='200112'
Fix.SecureData.QuoteSet.QuoteEntry.StrikePrice='25.00'
Fix.SecureData.QuoteSet.QuoteEntry.PutOrCall='1'
Fix.SecureData.QuoteSet.QuoteEntry.BidPx='5.00'
Fix.SecureData.QuoteSet.QuoteEntry.OfferPx='5.25'
Fix.SecureData.QuoteSet.QuoteEntry.BidSize='10'
Fix.SecureData.QuoteSet.QuoteEntry.OfferSize='10'
Fix.SecureData.QuoteSet.QuoteEntry.QuoteEntryID='2'
Fix.SecureData.QuoteSet.QuoteEntry.MaturityMonthYear='200112'
Fix.SecureData.QuoteSet.QuoteEntry.StrikePrice='30.00'
Fix.SecureData.QuoteSet.QuoteEntry.PutOrCall='1'
Fix.SecureData.QuoteSet.QuoteEntry.BidPx='3.00'
Fix.SecureData.QuoteSet.QuoteEntry.OfferPx='3.25'
Fix.SecureData.QuoteSet.QuoteEntry.BidSize='10'
Fix.SecureData.QuoteSet.QuoteEntry.OfferSize='10'
Fix.SecureData.QuoteSet.QuoteSetID='2'
Fix.SecureData.QuoteSet.UnderlyingSymbol='DELL'
Fix.SecureData.QuoteSet.TotQuoteEntries='1'
Fix.SecureData.QuoteSet.NoQuoteEntries='1'
Fix.SecureData.QuoteSet.QuoteEntry.QuoteEntryID='1'
Fix.SecureData.QuoteSet.QuoteEntry.MaturityMonthYear='200206'
Fix.SecureData.QuoteSet.QuoteEntry.StrikePrice='57.25'
Fix.SecureData.QuoteSet.QuoteEntry.PutOrCall='1'
Fix.SecureData.QuoteSet.QuoteEntry.BidPx='2.00'
Fix.SecureData.QuoteSet.QuoteEntry.OfferPx='2.25'
Fix.SecureData.QuoteSet.QuoteEntry.BidSize='25'
Fix.SecureData.QuoteSet.QuoteEntry.OfferSize='25'
```

The output would consist of two items and would be of the following format:

```
90=nnn<SOH>91=(encrypted data of length nnn)<SOH>
```

Please note that no parentheses are generated in the output message and that the <SOH> sequence represents the FIX delimiter character of 0x01.

Output of repeating fields

For output purposes, repeating fields can use either a hierarchical structure or they can be specified in the desired order as children of the body element (or an encrypted parent element if they are part of an encrypted section of the message).

If a hierarchical structure is used, then no output will be generated from the parent element, and the name of the parent element is ignored. For consistency sake, it is desirable to use the same structure and repeating field names that are used when input messages are parsed, but this is not necessary, since any hierarchical structure does not carry over to the output message.

Chapter 4. Using the source code

Source code for the parser itself, the related message dictionary, and the supporting metadata utilities are provided as part of this SupportPac. None of these materials are required to use this SupportPac.

Adding encryption and decryption routines

Subroutine calls are made to allow for the easy insertion of encryption and/or decryption subroutines within the message parser itself. There are four points in the code where allowances are made for an encryption or decryption routine to be inserted. The first call is in the initialization routine for the parser object. This is a good point to acquire any encryption keys or other data that the encryption routines require. A pointer field (iEncryptionData) in the message context is available to allow for the encryption routine to acquire storage for its exclusive use whenever a parser object is created. This should be done in the cpiCreateContext method contained in the fixparser.cpp program. If any such storage is acquired, it must be released in the cpiDeleteContext method contained in the fixparser.cpp program. As an alternative, the definition of the context area itself can be altered to include any required fields that the encryption and/or decryption routines require.

The other two subroutine calls are related to encryption and decryption of individual fields. Two types of encrypted fields are recognized. The first type of field is a single encrypted field. The second type is where a group of fields has been encrypted as a single element (e.g. SecureData tag). If the field contains tags and data for many fields encrypted as a single raw data field, then the contents must be parsed into the individual fields after the single raw field has been decrypted. In a similar manner, output fields can contain a single item or they can contain multiple tag and data items.

On input, the metadata is used to identify encrypted fields, and to tell whether they contain a single data item or a group of tag and data items. This information is stored in the flags field in each individual data item. When the metadata indicate that a field is encrypted, it will be passed to the decrypt routine for decryption. If the data item contains multiple tag items, then the decrypted field will be broken into individual fields.

On output, the metadata is used to identify fields that contain compressed data. If an individual data item is to be compressed, the encrypt routine will be called, and a raw data item and its corresponding length item will be generated in the output message. If a field is to contain multiple tag and data items, then the tag and data items should be children of a single name element. The name element should have the encrypt bit set to indicate that multiple tag and data items are to be generated and then encrypted.

On output, encrypted data will be translated to the desired output code page and then encrypted, unless the data type is byte array. On input, encrypted data will be decrypted and the data will be assumed to be in the code page indicated in the MQSeries message descriptor (MQMD).

Building the parser (Windows NT/2000)

A directory for the source files should be created and the zip file containing the source files for the parser (fixparser.zip) should be unzipped into this directory. The Microsoft Visual C++ Version 6 visual studio should be started, and the open workspace option under the file menu should be selected. Navigate to the directory that the source files were unzipped into. The workspace file for the fixparser workspace should then be opened.

Before the project is opened, the BipSampPluginUtil.c and BipSampPluginUtil.h files should be copied from the <MQSI_root>\examples\plugin directory to the same directory as the other source files.

Before the project can be built successfully, the locations of all include and library files in the project properties should be checked and changed if necessary. These files are located under the MQSeries Integrator Version 2 root directory. Select the version of the project that is to be built (release or debug). At this point it should be possible to build the project.

Command files are provided to move the resulting executables to the necessary MQSeries Integrator directory. The drive and path names used in these command files should be checked and if necessary corrected before using these command files.

Building the Message Catalog (Windows NT/2000)

A command file (buildmsg.cmd) is provided with the necessary steps to build the Windows NT message dictionary from the source provided. This step may require installation of part of the Microsoft development environment. The drive and path names used in the command file should be checked and if necessary corrected before execution of this command file.

Building the Message Catalog (Unix)

A Makefile if provided in the AIX\msgcat directory for building the message catalog on AIX. This step is only necessary if the messages in the catalog are changed. Otherwise, the pre-built message catalog (fixer.cat) can be used. To build the message catalog without using the supplied Makefile, use the following commands to build the message catalog and rename the generated include file. The include file should then be moved to the parser source directory, replacing the existing include file.

```
runcat fixerr  fixerr.msg
mv fixerr_msg.h fixerr.h
```

Chapter 5. Error Messages

There are certain situations where the parser will generate an error and reject the message. For example, if the message does not appear to be a valid FIX message, then an exception will be raised and the message will be rejected. For example, if there is no BeginString element (tag #8) at the beginning of the message, then the parser will raise an exception. When a parser exception is raised, an error message will be written to the event log. The Windows event viewer should be used to view the error information. For Unix systems, the Syslog facilities are used.

For a detailed list of error messages, see Appendix C.

Chapter 6. Customization of the supplied metadata files

Metadata files for FIX V4.2, FIX V4.1 and FIX V4.0 are supplied with this SupportPac. The metadata files have been created from text files that are also supplied. The fix standards documents are NOT supplied with this SupportPac. They are available to view or download at the FIX web site.

A utility program is used to build the actual metadata files from the text definition files. This program is included to allow the metadata files to be changed.

Using the offline utilities

An offline utility (buildmtx.exe) is provided to build the metadata files from input text files. Another offline utility (printmtx.exe) is provided to display the contents of a metadata file.

The utility that creates the metadata files uses two text files as input and creates a metadata file and a message file as output. The first input text file contains one line for each data tag that is defined in the standard. Each line contains four to six fields, separated by commas. The first field is the tag identifier and the second field is the long name that will be used when an input message is parsed. The third field is the type of data, expressed as a string and the fourth field is a flags field. The fifth and sixth fields are optional. The fifth field contains the value of the flags2 field. The sixth field is only used for raw data items and contains the tag identifier of the corresponding length item. No embedded blanks or other extra characters or white space is allowed. Comment lines are allowed and must start with an asterisk (‘*’) character in the first position of the line.

The following values are allowed for the data type:

- Int
- Float
- Qty
- Price
- Priceoffset
- Amt
- Boolean
- String
- Multiplevaluestring
- Currency
- Exchange
- UTCTimestamp
- UTCTimeonly
- UTCDate
- Localmktdate
- Data
- Month-year
- Day-of-month

The following bits are defined within the flags byte:

- 1 - field is count field for a repeating item
- 2 – field is first field in a repeating sequence
- 4 – field contains the length of a raw data item
- 8 – field is message length tag (tag #9)
- 16 – field is message type tag (tag #35)
- 32 – field is encrypted and contains other tag and value items
- 64 – field is encrypted and contains data for an individual item

The following bits are defined within the flags2 byte:

- 1 – suppress field on input
- 2 – suppress field on output

The second input file contains one line for each repeating sequence defined in a message type. The line contains a minimum of five fields and may contain many more. The fields are separated by commas. The first field is the message type as defined in the FIX standard and is contained in the MsgType (tag #35) item. The second field is the tag identifier of the of the count field and the third field is the tag identifier of the first data item in the repeating sequence. The fourth field is a name that will be used as a higher-level qualifier for all elements in a repeated sequence. The rest of the fields contain the tag identifiers of all tags that are considered part of the repeated sequence. A repeated sequence is considered to end when a tag is found in the message that is not part of the repeated sequence set or when the message trailer is found. No entry is necessary for repeated sequences that contain only the first sequence item.

Chapter 7. Parser Implementation

Parse Tree Structure

FIX messages begin with a BeginString element followed by a BodyLength element and a MsgType element. The last element in a FIX message must be a CheckSum element. Additional data elements provide the necessary data. The elements follow one another within the message. There is no hierarchy within the message.

When a logical message tree is built from an input message by the FIX parser, the structure is relatively flat. A top-level body element is assigned with the name “FIX”. Each data item in the message is then added in sequence as a child of the body, except for the CheckSum element. The value in the CheckSum item is verified but no element is added to the logical message.

There are two exceptions to the relatively flat structure normally used for FIX messages. The exceptions are for items defined as type multiple-string and for repeating data items.

All items are added as name-value items with the value in character format, except for multiple string items and elements whose data is in raw data format. Any items with input data in a raw data format will have its data stored as a byte array rather than a character string. This is necessary because the data may contain characters that are not allowed in character fields, such as binary zeros.

Multiple string items contain several different items within the same data element. In order to treat the data as individual items belonging to the same element, a top-level name element is created for the multiple string item. The multiple string data is then broken into one or more values and these values are then added as children of the name element. To make the individual value items easier to reference in a message flow, each value item is given an arbitrary name of “VAL”. Therefore, the second value item in an ExecInst (tag #18) would be referenced as:

```
Body.ExecInst.Val[2]
```

To find out how many individual value items are contained in the above example, the following expression could be used:

```
CARDINALITY(Body.ExecInst.Val[])
```

Repeating data items

The FIX standard specifies certain data elements as repeating data items. A repeating data item allows a sequence of individual items in a specific order to occur more than once in a single message. Furthermore, repeating data items can be nested in that a repeating data item can contain another data item that is itself a repeating data item.

Handling of metadata files within the parser

A structure is created to hold the data from up to 200 files and initialized in the bipGetParserFactory function. This function is called during parser initialization. A pointer to the structure is maintained in the module that handles metadata file processing (metadata.cpp) and available to the main parser module. This optimizes performance by allowing modules in the other parser modules to directly reference metadata.

For output, walk through the copybook structure, matching parent elements to metadata elements. For each parent, process all the children of the parent before moving to the next sibling of the parent. The order of processing should be in the same order as the copybook.

Each data group (segment) is represented by a metadata file, which includes the characteristics of each field as well as the segment overall. In addition, for performance reasons, an initialization string is included at the end of the file. The file name is the name of the four-character segment identifier plus a version number.

The internal structure of the file is in four parts. The first part of the file is a metadata header. It contains overall characteristics of the segment and the metadata file, including the version number of the file. The header format is as follows:

- Length of the header
- Metadata file format version (0 for this version)
- Number of variables
- Number of repeated tags
- Offset of name and Unicode name tables
- Repeated tags table

The next part of the file is the variable table. The third and fourth parts of the file are variable name tables. The first table contains field names as ASCII characters and the second contains the names in Unicode. The Unicode versions of the names are present to increase processing efficiency. The last part of the file is the repeated item table. There is one entry for every repeated sequence.

Some more detailed design points

Parsing of input messages

When an MQInput node receives an input message, the cpiParseBuffer routine of the parser is called. This routine will decompress the message, if necessary, and translate the message to Unicode. By translating the message to Unicode, the parser can accept either ASCII or EBCDIC input messages. It will validate that the message appears to be in an FIX format and will then assume ownership of the body of the message.

The first time that a field in the body of a FIX message is referenced by a node, MQSeries Integrator V2.01 calls a routine within the FIX parser that will begin parsing the message. This routine builds the logical message tree based on the contents of the incoming message.

The message is parsed one element at a time. A type tag contained in the individual element identifies each data element.

Parsing of output messages

The output message is built from the logical message tree. Each child of the body element and each child of a name element must have a name that is a valid name. This can be a short name (letters “FX” followed by the tag identifier) or a long name that is defined in the metadata.

Handling of headers in output segments

Header fields are optional for groups created in the logical message tree. When a group is added to the output message, the identifier and length of the segment will be filled in automatically. The group version number will also be filled in, matching the metadata file that is used to create the output group (segment). If any of these fields are found in the output parse tree, then the values in the parse tree will be used.

Chapter 8. System management messages

System management messages are special messages that the parser recognizes and which cause the parser to perform some special action. Normally, the parser expects to receive messages in a valid FIX format. However, there are certain actions that the parser might need to perform, such as purging cached metadata or identifying the level of the parser that is currently executing. Special system management messages are used to tell the parser to perform a system management action rather than perform its usual message parsing functions.

Recognition of management messages

System management messages must have the MessageSet property is set to the character string "*SYSTMGMT*". The MessageType property should be set to one of the values in the next section. The contents of the message should also be set as described below, and the domain should be "*FIX*". The contents of the message should then follow the rules outlined below, rather than the normal FIX standards.

Message types supported

The following system management message types are supported:

- FLUSH
- FLUSHALL
- CACHSTAT
- STATS
- DUMPSTAT
- TRACEON
- TRACEOFF
- TRACSTAT
- GETLEVEL

System Management Message Formats

All messages will start with a four digit level number, beginning in the first position and padded on the left with zeros, and a four-digit modification level, padded on the left with zeros. An eight-digit message type follows. Any data provided with the message follows the level number header and will vary by the particular message type and the individual request. If the format of the data is changed in the future, the version and modification level will also be changed. Any parser should ignore messages that are at a higher level than the parser is designed to handle. The parser can choose to process lower modification levels.

Flushing and monitoring the metadata cache

The FLUSH and FLUSHALL commands are used to remove in memory copies of metadata files. The next time the metadata are needed, the metadata will be reloaded from the metadata file on disk. This allows metadata to be changed without having to stop and restart a message broker or execution group.

For a FLUSH command, the individual files to be removed from the cache will be contained in eight character file names following the header. If a file is found, the cached data will be freed and the name will be changed to all x'BB' characters, and the use counter will be set to zero. This will remove the metadata entry from the cache. The next time the file is used, it will be reloaded from disk. A FLUSHALL request is similar, except the entire cache will be flushed.

No additional message data is needed for a FLUSHALL request. For a FLUSH request, the identifiers of the particular FIX metadata files that are to be flushed should be provided, as a series of eight character fields. For example, to flush the entry in the metadata cache for the FIX 4.2 standard, the characters FIX42 should appear in the list of files to be purged.

With all flush attempts, the defaults, group names and standards file data will also be flushed.

The CACHSTAT command will report on the current status of the metadata cache. It will not remove any entries from the cache.

All three messages will build a parse tree with one entry for each active entry in the cache. If desired, this message should be transformed to an output format such as XML and written to a queue.

Capturing Statistics

When a STATS or DUMPSTAT command is received, then the relevant statistics will be parsed rather than the message data. The DUMPSTAT command will also attempt to write the statistics to the file pointed to by the “FIX_STAT_FILE” environmental variable. The message data can then be processed in a standard message flow. If the statistics data is to be written to an output node, the message domain must be changed to a parser, which can output arbitrary data, such as XML. The statistics can also be written to a file by using a trace node.

The following parse tree (shown in an XML like format) will be built by a system management message containing a STATS request.

```
  <FIX>
      <statistics>
          <cache>
              <filesOpened>nnn</filesOpened>
              <filesRemoved>nnn</filesRemoved>
              <filesMax>nnn</filesMax>
              <useCounter>nnn</useCounter>
          </cache>
          <messages>
              <inMessageCount>nnn</inMessageCount>
              <inTagCount>nnn</inTagCount>
              <invalidMsgCount>nnn</invalidMsgCount>
              <maxMessageSize>nnn</maxMessageSize>
              <minMessageSize>nnn</minMessageSize>
              <averageMessageSize>nnn</averageMessageSize>
              <outMessageCount>nnn</outMessageCount>
              <averageParseTime>nnn</averageParseTime>
              <maxParseTime>nnn</maxParseTime>
              <minParseTime>nnn</minParseTime>
              <writeCount>nnn</writeCount>
              <outTagCount>nnn</outTagCount>
              <invalidOutputCount>nnn</invalidOutputCount>
              <averageOutputSize>nnn</averageOutputSize>
              <maxOutputSize>nnn</maxOutputSize>
              <minOutputSize>nnn</minOutputSize>
              <averageWriteTime>nnn</averageWriteTime>
              <maxWriteTime>nnn</maxWriteTime>
              <minWriteTime>nnn</minWriteTime>
        </messages>
    </statistics>
</FIX>
```

To access the statistics, a system management message should be sent to the input queue of a special message flow. The message domain should be set to FIX, so that the standard FIX parser will process the message.

Turning trace on and off and displaying trace status

If the debug level of the parser is installed, a local trace function is provided which will write detailed trace entries to a file. This trace capability is unique to the debug version of the parser and is separate and distinct from the MQSI Version 2 trace capability. The name and location of this trace file, and the initial settings of the various traces, can be controlled with environment variables. If the debug version of the parser is being used, it may be desirable to dynamically turn the trace function on and off.

To turn trace on or off, the MessageType parameter of the message should be set to TRACEON (padded on the right with a space) or TRACEOFF.

The message data should include the standard sixteen characters of header information, followed by one or more eight-character entries. Each can set either a particular trace on or off, or can set all trace functions on or off. To turn all trace types on or off, the trace message should contain a single eight-character entry after the header information, with the characters "*TRACEALL*". The following character sequences can be used to affect only a particular type of trace, such as module entries and exits or input parsing details:

- TRPARSE
- TRWRITE
- TRMGMT
- TRMODULE

All entries should be padded on the right with spaces to a length of eight characters.

An entry of TRACEOFF will turn all traces off. This is useful when only selected traces are to be turned on, since it allows all traces to first be turned off and then the selected trace functions to be turned on individually.

The TRACSTAT command will report on the current status of the trace. It will not change the current trace options.

All three messages will build a parse tree with one entry for each trace type. If desired, this message should be transformed to an output format such as XML and written to a queue.

Displaying the Level of the Executing Parser

In some cases, it may be desirable to know what level of the parser is currently running. If a message is received with a MessageSet of "*SYSTMGMT*" and a MessageType of "*GETLEVEL*", then a logical message tree will be built identifying the levels of the main modules used to build the parser. The parse tree will be of the following format:

```
<FIX>
    <ModuleLevels>
        <FixParse>nnn</FixParse>
        <Fixsub>nnn</Fixsub>
        <Metadata>nnn</Metadata>
        <Mgmtmsg>nnn</Mgmtmsg>
        <Parsubs>nnn</Parsubs>
        <Trace>nnn</Trace>
    </ModuleLevels>
</FIX>
```

This data should be written out or otherwise processed by a message flow to provide the desired information.

Implementation considerations

There are a number of areas where implementation decisions must be made. For example, how much data checking and validation should be done on input and output messages? If the data in a logical message field is longer than the corresponding field, should the data just be truncated or should an exception be thrown. Many of these questions do not have clear answers. Design choices were made to reduce the complexity and offer the most flexibility.

The parser does a minimal amount of validation on input and output messages. The purpose of the input validation is to ensure that the input message is a well-formed FIX message that can be parsed properly. The following characteristics are checked:

- The message must have a BeginString item in the front of the message and end with a CheckSum item.
- The value in the CheckSum item is verified against the message contents.
- Tags must consist of only the digits zero through nine.
- Tags must be from one to five digits long.
- Raw data items, such as an encrypted section of the message, must have a length field as the previous data item.
- The length field is verified to be of the proper data type, that the length is greater than zero and does not extend beyond the end of the message.
- The character immediately following the data is a data delimiter.

If any of the above checks fail, then the message is rejected and the parser raises an error (parser exception).

While more extensive checking and validation of individual data items could be performed, this would result in data editing functions being moved out of applications and into the parser. It is usually better to perform data editing functions in the applications that process the message rather than a data transformation and routing engine.

Individual field values are generally not checked for valid data. This includes the characters used in things like integer fields as well as the data values in coded fields. If checking of each field were to be performed, the overhead would be considerable and the parser would become much more complex.

Minimal checking is performed on output messages as well, as follows:

- Data element names must have a corresponding entry in the metadata.
- A metadata file must exist for the level of the FIX standard indicated in the output message tree.

On output, the BodyLength field is calculated automatically and inserted as the second data item in the output message. The CheckSum field is also calculated automatically and inserted as the last item in the message.

Generation of names for Data Elements

Each data element is assigned a unique numeric identifier by the FIX standard. The identifier is not particularly suitable for a data name, since it starts with a number and is not meaningful in any useful way. It is desirable to have a longer and more descriptive name for each data element.

Meaningful names have been assigned for each data element in the FIX standard. The long name and the associated data identifier are contained in a text file (namedata.txt). This file is used by the utility that creates the individual metadata files.

A short name is also supported. The short name begins with the uppercase characters “FX” with the data identifier appended. The metadata for each data element can be looked up using either the longer name or the shorter name.

Chapter 9. Environment Variables

An environment variable is used to point to the location of the metadata files used by the parser. The variable is FIXMETADIR. If this variable is not set, then a default value of “C:\FIX” is used.

If the debug version of the parser is being used, then a trace file will be produced. The default location of the trace file is c:\Fix\Parser.trc (/tmp/Parser.trc on Unix). The location and name of this file can be changed with the FIXTRACEFILE environment variable. The initial trace settings can also be set with the following environment variables:

- FIXTRACEALL
- FIXTRACEPARSER
- FIXTRACEWRITE
- FIXTRACEMODULE
- FIXTRACEMGMT

If statistics are to be written to a file, the “FIX_STAT_FILE” should be set to point to a fully qualified path and file name. Statistics can be forced out at any time with a system management message and will also be written out when the broker is stopped. (N.B. There is a bug in MQSeries Integrator V2.0.1 that prevents this from happening. This bug is supposed to be fixed in CSD1.)

Chapter 10. Implementation details

Parser Initialization and Termination

There are also two entry points that are used during initialization and one used during termination.

The BipCreateParserFactory entry point is called when the execution group initializes. It will specify the message domain that the parser will handle. Metadata initialization is also performed, including the loading of three global metadata files. If the debug version of the parser is being used, trace initialization is also performed.

The cpiCreateContext entry point is called when a thread is created to handle a particular message flow.

The cpiDeleteContext entry point is called when a thread terminates. The key functions are to release any memory that was acquired in cpiCreateContext or during the processing of the last message processed. Statistics are also written to a statistics file and are reset to zero.

Handling of Input Messages

Input messages must be in a valid FIX format. Each item must have a recognized one-digit to five-digit identifier for each data item. The standard items all have an identifier of one to three numeric digits (user defined items can have a five-digit identifier). The identifier is used to look up a descriptive name for each data item.

The CheckSum data element is verified but is not added to the logical message tree. All other data elements are added to the parse tree.

There are five entry points used within the parser for the parsing of input messages.

When a message is received in an MQInput node of a message flow, an instance is created to process this message. The MQInput node will determine the parser to assign to the body of the message. It creates a root element for the body and assigns it a name based on the parser domain.

Parser Context

The context area for a processing of a particular message is allocated in the cpiCreateContext routine. The intent is that the context will be allocated once for each thread that is started to service a particular message flow, and that the context will be reused by each succeeding message. There is a bug in MQSeries Integrator version 2.0.1 that results in this routine being called once per message. This will cause a memory leak. A fix for this problem is supposed to be made available in the first CSD.

The context area contains the following fields:

eBody	Pointer to the root element of the body.
eCurrentElement	Pointer to the current parent element. Elements with values will be added as children of this element.
iSize	Size of the input message buffer area.
iLength	Length of the message area to be parsed.
iInput	Pointer to the raw input data buffer.

iBuffer	Pointer to the data to be parsed, translated to Unicode. This pointer will be used for identification of all tags and for all data values except raw data items.
iIndex	Number of bytes that have already been processed in the message.
iRemaining	Number of bytes left in the output buffer.
iCodePage	Code page of the input or output message. This field is taken from the CodedCharSetId field in the message properties.
iEncoding	Numeric encoding format. This field is taken from the Encoding field in the message properties.
iPrevTag	Pointer to the tag identifier of the previous item, translated to Unicode.
iPrevData	Pointer to the data portion of the previous item, translated to Unicode.
iPrevDataType	Data type of the previous item, from the metadata entry.
iMsgType	This field will be set to one if the message is a special management message.
iCurrentCharacter	The value of the character at the offset being parsed, in the local code page of the parser system.
sMsgType	Message type from the MsgType tag (#35).
sLevelName	Version string from the BeginString tag (#8). This field is used to find the metadata for the message.
iRepeat	Pointer to a repeating data structure, which is used to retain information while processing a set of repeating items.
msgDomain	Message domain to be used for this message.
msgSet	Value of the MessageSet item in the properties folder.
msgType	Value of the MessageType item in the properties folder.
iStartTimeHigh and iStartTimeLow	The time a particular thread was started.
TotalParseTime	The total time that the parser has taken with this message. This value will be incremented each time another part of the message is parsed. This value will be added to the statistics when the thread ends or when the thread parses a new message.

Initialization functions

The bipCreateParserFactory entry point is called when an execution group process is started. The main function of this entry point is to perform basic parser wide initialization. The parser will initialize the metadata structures in memory, and will load three specific metadata files (defaults, group names and standards mapping). If the debug version of the parser is being loaded, the trace function will also be initialized.

The cpiCreateContext entry point is then called. This routine allocates and initializes memory for use during the processing of a particular input message. The allocated memory is initialized to binary zeros.

The cpiParseBuffer entry point is called after the cpiCreateContext entry routine has finished. This routine will initialize the parser context, and will get the code page and numeric data format from the message properties. If an alternate buffer area has been allocated for the previous message, the area will be freed. This routine will assume ownership of the rest of the message. This routine performs several functions, including the initialization of the fields in the context area. First, it initializes a pointer (eBody) in the context area to point to the body root element. It sets context variables to point to the original message data area (iInput) and the length of the input message (iSize). It checked if the message has been compressed and if so allocates a new area for the message and decompresses the message. It then sets a pointer to the message data (iBuffer), and sets the message length field (iSize). It initializes the parent (eCurrentElement) and current element (iCurrentElement) pointers to null values, to indicate that no parsing of the message has taken place. The code page of the input message is determined from the message properties and saved in the iCodePage field.

The message format will now be checked to make sure that the input message appears to be a FIX message. Finally, the routine returns with a length equal to the remaining buffer size, indicating that the parser has taken ownership of the remaining part of the message.

Parsing Routines

Parsing of a message is done when data within the message is first referenced in a message flow. There are four entry points that can be called, depending on the particular node routine that was called. The entry points are cpiParseFirstChild, cpiParseLastChild, cpiParseNextSibling and cpiParsePreviousSibling. Each of these routines will call the parseNextSegment routine in the fixsub.cpp module until the appropriate element completion flag has been set.

The message will be parsed from left to right (beginning of the message to the end of the message).

Termination Routines

The cpiDeleteContext routine is called when a parser thread terminates. A thread will usually terminate when the execution group process ends. This routine will check if an alternate input buffer has been allocated, and if so, will release it. The routine then releases the context area itself.

Handling of Output Messages

Three of the required data items for a FIX message are generated automatically when an output message is built from a logical message tree. These fields are the BeginString (tag #8) item, the BodyLength (tag #9) and the CheckSum (tag #10) items.

If the BeginString item is found as the first child of the body element, then the value contained in the parse tree will be used. Otherwise, a default value of “FIX.4.2” is used, to indicate that the message will be at the Version 4.2 level. This value also determines the metadata that is used to process the rest of the logical message tree.

The BodyLength and CheckSum items will be automatically calculated and inserted in their proper locations within the output message. These items are difficult to calculate within a message flow, and may change if the message is encrypted during output. The BodyLength element will be ignored if it is present, since the length will be calculated and this element inserted automatically. The CheckSum item should not be used, since this element will be automatically appended at the end of the message. If the item is found in the logical message tree, it will also be included in the output message, resulting in two CheckSum items.

All other items in the message will be generated based on the names and values contained in the logical message tree. If the name of an item begins with the letters “FX”, it will be assumed to contain the tag identifier following the first two letters. For example, if an element has a name of “FX47”, and a value of “3”, then the following data would be inserted in the output message:

```
47=3<SOH>
```

All other names will be looked up in the metadata file associated with the given level of the FIX standards. If it is found, then the tag identifier associated with the data name is found and used to build the FIX output. For example, if an element in the logical message tree has a name of “OrdType” and a value of “A”, then the following data would be inserted in the output message:

```
40=A<SOH>
```

All data in the parse tree will be translated to character data in the specified code page, except for data that is specified as raw data in the metadata or data that is held as ByteArray type data in the logical message tree.

Neither the names nor the data values within data items in the logical message tree should contain delimiter characters (either an equal sign or a 0x01 data delimiter character). The delimiter characters will be inserted in the output message as needed.

Chapter 11. Offline Utilities

Two offline utilities are provided to assist in building the necessary metadata files. The utilities are named “buildmtx” and “printmtx” respectively. The first utility will process a text file and produce the corresponding metadata file. The second utility will produce a formatted text file showing the contents of a metadata file.

The print utility is a program that can be used to produce a listing of the contents of a particular metadata file. It is provided because the contents of the metadata files are in binary and therefore it can be difficult to understand their contents.

In general, the offline utilities are not required, since this SupportPac includes a complete set of metadata files for the commonly used versions of the FIX standard.

Building the Metadata files

This section documents how the metadata files that are provided with the parser were built. Since pre-built versions of the metadata files are provided with this SupportPac, there is generally no need to build the metadata files. This procedure would only be required when modifications are to be made to the standard FIX data areas, different data naming conventions are desired or user defined data elements are to be added to the metadata files.

The metadata files are built using the BUILDMTX command line utility (BUILDMTX.EXE). A command file is provided that will process the text files supplied with this SupportPac.

The input to the metadata processing utility is two simple text files. The first text file contains the characteristics of the individual tag and value items. The second text file is used to identify repeating data items.

The metadata files contain binary and Unicode data and are therefore platform dependent. The metadata files must be built on the platform that they will be used on. Metadata files intended to be used on Windows platforms must be built on Windows and metadata files intended to be used on Unix platforms must be built on Unix platforms.

Chapter 12. Problem Determination

There are many types of problems that can arise. Some of the more common problems are discussed below.

Broker will not start

If your broker is running on Windows NT or windows 2000, look in the application log using the event viewer. If you are running on Unix, look in the /etc/syslog.conf file. Near the bottom there should be an entry for a user.debug, with an associated file name. Look in this file for messages that explain why the broker is not starting. If there is any indication that the parser may be causing a problem when it is loaded during the initialization of the broker, the parser executable (fixparser.lil) should be moved out of the <MQSI_ROOT>/lil directory.

Parser Exceptions

The FIX parser does not perform a thorough check of the contents of a message. However, a message must meet certain minimum criteria for the parser to be able to handle the message. If the message does not meet the minimum criteria, then the parser cannot process the message and will raise an exception. This will normally cause the message flow to fail and the message will usually wind up on some sort of failure or dead letter queue.

When a parser exception is raised, an entry is written to the application log. In the Windows NT environment, this log can be viewed using the event viewer. When a message is not properly processed, and the message flow appears to fail, the event log is usually the first place to look.

The parser must be able to find and use the proper metadata for the version of the group or standard that was used to build a message. A message flow can fail if the wrong version is specified or if a version is not specified for a group. If a message flow fails and the event log contains a message indicating that either a metadata file could not be found or that the group length in the metadata file did not match the length in the group header, the group version in the group header and standard version in the transaction control group should be checked.

Performance

Make sure that the release build is being used. The debug version of the parser writes extensive trace data to a trace file and performance will be significantly degraded. Check the file size of the fixparser.lil file in the {MQSI_ROOT}\bin directory to be sure that the release build is being used.

Debug version of the parser

A special debug version of the parser is provided with this SupportPac. This version of the parser contains a detailed tracing facility. This special trace is written to a text file. The trace is quite detailed and therefore can be quite large. Therefore, this version of the parser should **not** be run in a production environment.

Using the debug version

If the debugging version of the parser is installed, the following environment variable should be set.

- FIXTRACEFILE – location and name of trace file.

The following environment variables can be used to control the initial setting of the trace options.

- FIXTRACE
- FIXTRACEPARSER

- FIXTRACEWRITE
- FIXTRACEMODULE
- FIXTRACEMGMT

The FIXTRACE variable controls the overall setting of the trace as on or off. This variable must be set to a ‘1’ if tracing is to be enabled when the parser starts. The other variables allow for limiting the type of trace information that is collected. At least one of these trace functions should be set to a ‘1’.

If none of the environment variables are set, then tracing of all types will be enabled. If the trace file location is not specified in an environment variable, then the default file and location of “\FIX\parser.trc” on the “C:” drive for Windows and /tmp/parser.trc for Unix will be used. If this directory does not exist, no trace output will be produced.

To install the debug version of the parser, the message broker must first be stopped. If the standard installation instructions have been followed, the executables for both versions of the parser should be located in the <MQSI_root>\bin directory. The normal release version of the parser (fixparser.lil) should be renamed with a different extension (e.g. fixparser.rel) and the debug version of the parser should be renamed from fixparser.dbg to fixparser.lil. The broker should then be restarted and the desired message(s) processed. Once the messages have been processed, the broker should be stopped again and the files renamed to their original names. The broker can now be restarted.

The trace function can also be turned on and off by using system management messages.

Reporting bugs

Although no official support is provided, the author is interested in hearing of any problems or suggestions for improvement for this SupportPac. If a bug is suspected, please send an email with a problem description. If possible, please attach a file with a copy of the message so that the author can reproduce the problem locally. The author’s email address is on the front cover of this document.

Appendix A - Hints and tips for writing a parser

This section contains a discussion of some areas that are not covered in the Programming Guide or other standard product documentation.

What is a logical message and what is a wire format?

A logical message is the interpreted version of a message that the message flow elements (nodes) process. The logical message data consists of the data as individual fields. A wire format is the actual data from an MQSeries message. A parser is used to convert a message from one format to the other.

What do parsers do?

In MQSeries Integrator V2, parsers provide the function needed to interpret incoming messages and create a logical message based on the data within the message. The logical message usually consists of the individual data fields within the message. Parsers are also responsible for creating an output message based on the data found within the logical message.

In some cases, parsers rely on external data representations stored in some kind of metadata repository. For example, the IBM supplied MRM parser stores information about the message formats it can recognize in a repository stored in a relational database. In other cases, the message format itself is self-defining and no metadata is required to parse a message.

How do Parsers work?

A parser is initially loaded when an MQSeries Integrator version 2 message broker is started. The broker in turn starts one or more execution groups. Each execution group operates as a separate operating system process, running a module called "DataFlowEngine". Each execution group loads all modules found in the <MQSI root>\bin directory with an extension of "LIL". The parser modules are built as DLLs. The execution group then calls an entry point within the parser (bipGetParserFactory), which completes its initialization process and indicates what types of messages (domain) the parser will process. The parser is now loaded and ready to process messages.

The execution group then loads any messages flows and starts an active thread for each MQInput node within each message flow. Each thread issues an MQGet with wait for each input queue.

When a message arrives in the queue, the MQGet completes and the MQInput node begins to process the message. It first starts another thread (if the number of threads for the message flow is less than the maximum allowed) to issue another MQGet to the input queue. The thread then creates a root element for the logical message, and starts to identify the various parts of the message.

The MQInput node creates a child element of the root for the message properties and MQMD. It then identifies any additional parts of the message and creates a child element of the root for each additional section of the message (generally message headers, such as an RFH2 header) and the body of the message (the user data). The parser for the body of the message is identified by the domain value in the RFH2 header. If there is no RFH2 header, then the default domain specified in the MQInput node defaults property is used. If there is no default domain specified in the MQInput node defaults, then the message body is treated as a blob.

The cpiCreateContext entry point is called once when a thread is initialized. The purpose is to acquire a storage area for any context that is to be saved during parsing of a message. This is primarily of use for a partial parser, which will be called repeatedly to parse a complete message.

The cpiParseMessage entry point is called during the initial processing of a message by the MQInput node. A primary function of this entry point is to allow the parser to determine which part of the message the parser will assume ownership of, and prepare to process the message. The parser should defer parsing of the message until a particular part of the message needs to be parsed. When a part of the body of the message is referred to in the message flow, and the message must be parsed, one of the parsing entry points, such as cpiParseFirstChild or cpiParseNextSibling, is called. None of the other major sections of the message are parsed at this time, and the elements for each section are NOT marked as complete.

The message is then propagated to the Out terminal of the MQInput node. When a field within the body of the message is referenced within the message in a later node (such as a filter node or a compute node), and an attempt is made to retrieve either a child or a sibling of a message element which is not marked as complete, one of four entry points within the parser will be called. The four entry points are cpiParseFirstChild, cpiParseLastChild, cpiParseLeftSibling and cpiParseRightSibling. Each entry point is passed the address of the element that the message flow was attempting to navigate from. The particular routine should then complete enough of the parse tree and set the appropriate completion in the referenced element for the message flow to continue processing.

The cpiDeleteContext function is only called when the thread is finished. It should release any memory acquired by the cpiCreateContext function.

What is "partial parsing"?

The MQSeries Integrator Version 2 broker is written to support what is called partial parsing. Since an individual message may contain hundreds or even thousands of individual fields, the parsing operation can require considerable memory and processor resources to complete. Since an individual message flow may only reference a few of these fields, or possibly none at all, it is inefficient to parse every input message completely. For this reason, MQSeries Integrator Version 2 has been designed to allow for parsing of messages on an as needed basis. This does not prevent a parser from processing the entire message all at once, and some parsers are written to do exactly this.

Rather than parse the entire message contents and build a complete logical message, the broker waits until a part of the message is referenced, and then invokes the parser to parse that part of the message. This will reduce the overhead when a large part of a message is not referenced in a message flow. To understand how this works, one must be familiar with MQSeries Integrator Version 2 nodes, and how they refer to fields within the message. Nodes refer to fields within the message using hierarchical names. The name begins at the root of the message and then proceeds down the message tree until the particular element is located. If an element is encountered without the completion bits set, and further navigation from this element is required, then the appropriate parser entry point will be called to parse the necessary part of the message. The relevant part of the message should be parsed, and appropriate elements added to the logical message tree, and the element in question should then be marked as complete. If the element is not marked as complete, a looping condition can then arise.

Parser Context

When a parser is called up to parse a message, it is useful to have an area where information about the specific message that is currently being parsed can be kept. This is particularly useful when a parser is parsing a message incrementally (partial parsing) and must remember how much of the message it has already parsed. MQSeries Integrator version 2 provides a facility for a parser to acquire an area of storage and associate it with a particular message. This area of storage is called the parser context. There is one context maintained for each thread that has parsed a message that has required the use of a particular parser.

To understand the parser context, some understanding of the threading model used within MQSeries Integrator Version 2 is required.

MQSeries Integrator Version 2 uses a multi-process multi-thread architecture. Every execution group defined within a broker runs as a separate operating system process. Threading is used within individual execution groups. When a message flow is assigned to an execution group, and the execution group is started, one or more threads will be started to process messages associated with that message flow. First, a thread is started for each MQInput node within the flow. These threads issue an MQGet with wait against the queue specified in the MQInput node. There is a parameter (additionalInstances) that can be set on the message flow that controls the number of additional threads that the particular message flow can use to process more than one message at a time. These additional threads are also started, but they wait on a semaphore.

When a message arrives on an input queue, the MQGet is satisfied and the thread starts to process the message. Prior to exiting the MQInput node, the thread will check if there are any additional instances in the pool for this message flow. If there are, one of these threads will be posted and will issue an MQGet with wait. The current thread will then process the message. When it completes the current message, it will check if another thread has issued an MQGet with wait. If another thread has issued the MQGet, then this thread will then rejoin the thread pool and wait on a semaphore. If there is no thread with an outstanding MQGet, the current thread will reissue the MQGet.

The input node identifies the parser(s) needed to parse the input message. If a parser is required that has not been used before by the particular thread, then an instance of the parser object will be instantiated for that thread. This parser object will be retained for the duration of the thread. The threads are usually retained until the execution group (or broker) is stopped. The thread will also be stopped if the message flow is changed and a deployment operation is initiated.

Whenever the message passes through a Compute node (or Extract node), a new message tree will be created. When the body element of the new message tree is created (using a call such as cniCreateElementAsFirstChildUsingParser), an owning parser is created for the body of the message. This parser will be used to create an output buffer from the logical message tree data when required (generally as a result of a later MQOutput node). Any parser objects that are created to handle subsequent message trees will be destroyed when the particular instance (message) completes.

When a parser object is created, the cpiCreateContext entry point will be called. The parser should acquire any storage that it needs and return the address when this routine is finished. This storage will be retained and reused for the life of the parser object. When the parser object is destroyed, the cpiDeleteContext entry point is called.

What happens if a parser encounters an error?

If a parser encounters invalid data or other types of errors, it has two basic options. It can ignore the error or it can create (throw) an exception and cause the message flow to be terminated.

How do the completion bits found in message elements work?

Every element in a parse tree has five logical pointers. The pointers are to the parent, previous sibling, next sibling, first child and last child. The parser builds the parse tree structure by adding elements as either the first child or the last child of a previous element. The appropriate pointers of all surrounding elements are adjusted when a new element is added to the parse tree. The pointers are always valid. This means they will either point to an element that is a sibling or child of the element, or they will have a null address.

When a node needs to find an element in the tree, it must navigate to the desired element, starting at the root element. The node can use any of the five pointers to locate the desired element. For example, if a node needs to locate the element that corresponds to InputBody.A.B, it could accomplish this is in the following manner.

First, the root element must be found, using the cniRootElement function. The node would then use the cniLastChild function to get a pointer to the last child of the root element. This would be the body element. The node must now locate the child of the body whose name is A. To do this, it would use the cniFirstChild to locate the first child of the body. It would then search through the children of the body, using the cniNextSibling function as needed, looking for an element whose name is A. When the A element is located, the cniFirstChild function would again be called to locate the first child of element A. The children of element A would then be searched using the cniNextSibling function until an element with a name of B is located. The desired element has now been located.

MQSeries Integrator version 2 supports late or partial parsing, to reduce the overhead in certain common situations. This means that the parse tree will only be built when it is needed. To support partial parsing, the parser must be able to indicate where the parse tree is complete and where it is not. To do this, two bits are available within each element. The parser to indicate whether the first child pointer is complete, and whether the last child pointer is complete set the bits. To be considered complete, the first child pointer must point to the element that is really the first child of the current element, and the last child complete bit indicates that the last child pointer is pointing to the element that is truly the last child of the current element.

When the various node navigation functions (such as cniFirstChild or cniNextSibling) are called, they look at the corresponding completion bits to determine if they need to invoke the parser before they return the result. The cniFirstChild function will call the parser until the completePrevious bit is set, and will then return the first child pointer from the given element. The cniLastChild will call the parser if the completeNext bit is not set, and after this bit is set by the parser, will return the last child pointer from the given element. The cniPreviousSibling function will check the completePrevious bit in the parent of the given element, and if that bit is not set, will call the parser. After the parser sets the completePrevious bit in the parent, then the previous sibling pointer from the given element will be returned. In a similar manner, cniNextSibling will check the completeNext bit in the parent, and if necessary, invoke the parser. When this bit is set by the parser, the next sibling pointer will be returned.

Most parsers are will operate from left to right (from the beginning of a message to the end). If the first child has been parsed, or if there are no children, then the completePrevious bit of the parent should be set. If the last child of an element has been parsed, or if there are no children, then the completeNext bit of the parent should be set.

What data types are supported and how are they stored internally?

The logical message model supports many types of data, as defined within the ESQL standard. Data types include character, integer, decimal, floating point, boolean and date/time formats. The input numeric data representation for integer and packed decimal data is determined from the MQMD encoding parameter, and the output data format is determined from the encoding parameter in the message properties (first child of the root). Internally, character data is stored as unicode characters, while integers are stored as 64 bit values using the encoding sequence native to the platform on which MQSeries Integrator Version 2 is running (e.g. for Windows NT, this would be "little endian", whereas if the broker were to run on an RS/6000 processor under AIX, it would use "big endian" format internally). Decimal data is stored as characters in either little endian or big endian order. No conversion of floating point data is provided. Date and time values are stored as data structures.

Code pages and input buffers

The code page of the input buffer is contained in the properties. The input buffer should be translated to Unicode before any processing and the subsequent processing should be done using Unicode. This makes the parser independent of the code page of the incoming message. The code page of the data in the input buffer is contained in the properties.

Parser Utility Functions

What is the difference between similarly named node and parser functions, such as cniNextSibling and cpiNextSibling? The parser functions will not cause the invocation of a parser, and hence are recommended to use within parser routines. The node utility functions will invoke a parser if the completion bits are not set. If a parser were to use the node utility functions on a part of the parse tree that it is responsible for, then the parser could be called recursively and a loop could result.

There is one instance where the node functions must be used. If a parser needs to access the message properties, the individual fields under the main properties element may not have been created. If the parser utility functions are used for this navigation, then the desired element may not be found. If the node utility functions are used, the properties parser will be invoked as needed to complete the properties section of the parse tree. In fact, this function is used in several places within the provided FIX parser.

Using the CciLog and CciThrowException utility functions

The CciLog and CciThrowException functions write an entry into the Windows NT event log. The CciThrowException function will generate an exception. The exception may be handled by the message flow or, if not handler is present, it may cause the message flow to be terminated. The CciLog function will write an event into the event log and execution of the message flow will then continue.

The CciThrowException function requires a parameter that indicates the type of error. Most errors that are detected with the contents of a particular message should use an error type of CCI_PARSER_EXCEPTION. This will result in a runtime error being raised within the particular instance that is being processed, and will generally result in the message being rejected. The message is usually placed on some kind of failure or dead letter queue. If an error type of CCI_FATAL_EXCEPTION is used, then the entire execution group will be brought down. This exception type should only be used for serious errors that are likely to affect the entire execution group, such as memory corruption.

All errors thrown by the provided FIX parser are of the parser exception type.

Creating a Message Dictionary

Both the CciLog and CciThrowException functions require a Windows message dictionary to be created and registered in the Windows registry.

The first step in creating a message dictionary is to create the source input for the messages themselves. This SupportPac includes a message dictionary, including the source code for the messages. The source file for the message dictionary is called “fixerr.mc”. More documentation for the message dictionary formats is contained in the Microsoft SDK for Windows NT. A sample command file is included with this SupportPac (buildmsg.cmd) that will create the fixerr.dll message dictionary and copy it to the execution directory. Directory names and drive locations may have to be changed to match a given installation.

A registry entry is also needed in the system that the parser will execute on. Instructions on creation of this registry entry are included in the installation part of this document.

Calling the CciLog and CciThrowException functions.

A number of parameters must be passed to either the CciLog or CciThrowException functions. Two of the parameters are the name of a message dictionary and the message identifier within the message dictionary. An include file with definitions of the message identifiers is generated by the message compiler and should be included in the source program using the message dictionary. The message dictionary name is a character string with a null termination.

An error message defined in a message dictionary can include additional parameters in the message definition that are to be filled in at execution time. All such parameters must be character strings. The last parameter passed to either of the above functions should be a parameter of zero. This indicates the end of the execution time parameters list. Even if there are no run time parameters, a single parameter with a value of zero should be passed to the above utility functions.

The message dictionary format for an insertion is a percent sign followed by a number, which in turn is followed by an exclamation point, the letter “s” and a second exclamation point. The number indicates which parameter should be used. The first parameter that is passed on the utility function call is matched to the insertion sequence identified by the number two. If the parameter number in the message is the digit one (e.g. the insertion sequence of %1!s!), then a character sequence consisting of the broker name and execution group name is inserted. The first parameter passed on the utility function call will replace the sequence %2!s!, the second parameter will replace the sequence %3!s!, and so on.

Microsoft documentation should be used for further information on the format of a message dictionary. A sample message dictionary is also provided (fixerr.mc).

Using Microsoft Foundation Classes (MFC) in a parser

The Microsoft Foundation Classes (MFC) can be used in a parser. However, their use can cause link edit errors. Two steps should to avoid these problems. First, the _USRDLL definition in the preprocessor section of the C/C++ tab in the project properties should be removed if present. The second step is to explicitly add the appropriate MFC library to the beginning of the Object/Library modules area in the general section of the Link tab.

What does the iFpIsHeaderParser parser function call do?

MQSeries Integrator Version 2 sets the format field in the MQMD to the parser domain name if the cpiParserType routine returns a value of zero. If this routine returns a non-zero (TRUE) value, then the format field is not set to the domain name of the parser. It can then be set by a message flow. If this routine is not implemented in the parser, then the format field is always set to the name of the parser domain.

Appendix B – Repeating field names

Some FIX message types can contain multiple occurrences of certain tag items, and some of the repeated tag items can themselves have multiple occurrences. Without a hierarchical structure in the logical message tree, it can be difficult or impossible to know which repeated items belong to which higher-level items.

To provide a hierarchical structure, some additional high-level names are required. These additional higher-level names are contained in the repeating data section of the metadata files. The list below indicates the high-level names that are used when an input message is parsed and a logical message tree is constructed. The first column indicates the FIX message type, the second column indicates the tag identifier for the first item in the repeating sequence and the third column indicates the high-level name that is assigned for the sequence.

MsgType	Tag	Element name
6	216	RoutingInfo
B	216	Routing
B	58	Textlines
C	216	Routing
C	58	Textlines
R	55	Symbol
i	302	QuoteSet
i	299	QuoteEntry
Z	299	QuoteEntry
b	302	QuoteSet
b	299	QuoteEntry
V	55	Symbol
W	269	MarketData
X	279	RefreshData
c	311	Security
d	311	Security
D	79	Allocation
8	375	ContraBroker
G	79	Allocation
J	11	Orders
J	32	IndExec
J	79	AllocGroup
J	137	MiscFee
k	399	BidDesc
k	66	BidComp
l	12	BidComp
E	11	Orders
E	79	AllocAcct
m	55	StrikePrice
N	11	Orders
A	35	MsgTypes

Appendix C - Error Message Details

The parser may produce the following error messages. Error messages will be recorded in the application log and can be viewed with the Windows NT event viewer. On Unix platforms, the Syslog facility is used.

A detailed description of the error, as well as the common causes of the message, is provided below. In all cases, the message will be preceded by the broker and execution group names.

Error Message Text and Likely causes

Message 10 (No BeginString item found)

This message will be produced if no type “8” element is found at the beginning of the message. This group is required as the first group for all FIX messages. This error generally indicates that the message is not a valid FIX message.

Message 11 (No delimiter (0x01) for BeginString data found)

The data portion of the BeginString item should be terminated with a delimiter (0x01) character. No such delimiter was found within the first 30 bytes of data following the BeginString identifier. This is usually an application error in the application that created the message.

Message 12 (No ending delimiter (0x01) found)

This message indicates that no delimiter character was found at the end of the message. This delimiter character is required as the last character in all FIX messages. This is usually an application error in the sending application.

Message 13 (No CheckSum (tag #10) found)

The last data item in a FIX message should be a check sum item (tag #10). The check sum item should occupy the last 7 characters of the message and must be in clear (unencrypted) text. The first three bytes are the identifying tag (10) and an equal sign. The next three characters are digits that represent the check sum (which is a number in the range of 0 through 255) and the final character is an item delimiter (0x01). This message indicates that the identifier (“10=”) was not found starting seven characters before the end of the input message.

Message 14 (Calculated check sum does not match checksum item)

The last data item in a FIX message should be a check sum item (tag #10). The check sum item should occupy the last 7 characters of the message and must be in clear (unencrypted) text. The first three bytes are the identifying tag (10) and an equal sign. The next three characters are digits that represent the check sum (which is a number in the range of 0 through 255) and the final character is an item delimiter (0x01). The check sum is calculated by adding (modulo 256) the character value of each individual character from the beginning of the message (BeginString identifier “8”) to the character just before the beginning of the check sum item itself.

Message 20 (No identifier found preceding equal sign)

A data item was found with no data identifier preceding the equal sign. Each data item within a FIX message must be of the following format:

Identifier=value<SOH>

Where identifier is a numeric identifier that indicates the name of the item, value represents the value of this item and the terminator character (<SOH>) indicates the end of one item and the beginning of the next item. The terminator character (<SOH) is a single byte with a value of binary one and does not represent a printable ASCII character. Identifiers should not have leading zeros and can be from one to three digits long. User defined identifiers can be up to five digits long.

Message 21 (Element name (XXX) not found in metadata file)

The element name contained in the message was not found in the metadata. The element name may be invalid or misspelled, or the element name may not be valid at the level of FIX standard that is being used to create this output message.

Message 22 (No metadata found for this fix level - XXX)

No metadata file was found that matched the fix level specified in the BeginString (tag #8) element in the logical message tree. If no BeginString element is missing, then the default value of FIX42 is used. This error can also be caused if the metadata directory is specified incorrectly. Check if the FIXMETADIR environment variable is specified and if it points to the expected location. The default metadata directory is C:\fix on Windows platforms and /var/fix on Unix platforms. Metadata files should have a file extension of “mtx” and should have a file name of the format FIXnn, where nn is the version and modification level of the FIX standard that the metadata is supposed to match.

Message 23 (Tag length too long (> 5 characters))

The length of a tag cannot be more than 5 digits long and must be delimited with an equal sign. If an equal sign is not found within 6 characters, this error is raised. It can be due to a missing equal sign or an invalid tag value. Tag values should not have leading zeros.

Message 24 (Invalid character found in tag)

Tags must consist of from one to five digits. This error indicates that a character other than a numeric digit was found in a tag. Tags must be delimited with an equal sign. This error can also be caused by a missing or invalid equal sign or an invalid code page value in the MQSeries message descriptor (MQMD).

Message 30 (Invalid data type for length of raw data field)

Raw data items are used for data that can contain any data characters, including the normal delimiter character used to indicate the end of the data portion of an individual item. Although a delimiter character must be used after the data in a raw data field, the delimiter character cannot be used to indicate the length of the data. Therefore, a length item must immediately precede the raw data item, to indicate the length of the raw data. This length is used to determine the end of the data contained in a raw data item. This length field should be of type integer. If the field immediately preceding a raw data field is not of type integer, then that field is assumed to be something other than the required length field and therefore the required length field is assumed to be missing.

Message 31 (Length field at offset (nn) evaluates to zero)

A length field must immediately precede a raw data item. The length field must be an integer and must have a length that is greater than zero. The field immediately before the raw data item has a value of zero or a negative value.

Message 32 (Length field for raw data field exceeds remaining buffer)

This message indicates that the length field for a raw data item is greater than the remaining characters in the message. This usually indicates a problem with the application that created the message data.

Message 33 (Delimiter at end of raw data field missing)

This message indicates that the delimiter character that must follow the data portion of the raw data item is missing. This usually indicates a problem with the application that created the message data. It can also indicate a problem with the length field that is contained in the field just before the raw data item.

----- End of Document ----