MQSeries Integrator V2
ACORD AL3 message adapter

Version 1.0

31st October 2000

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, October 2000

This edition applies to Version 1.0 of **MQSeries Integrator V2 - ACORD AL3 message adapter** and to all subsequent releases and modifications unless otherwise indicated in new editions.

Table of Contents

Table of Contents iii
Notices v
Trademarks and service marks v
Preface vi
Acknowledgments vi
Introduction to the ACORD AL3 standards 1
ACORD AL3 message format 1
What does an ACORD AL3 message look like? 2
Transactions 2
What sorts of data are contained in ACORD messages? 2
Compression 3
ACORD AL3 Data Dictionary (DATADICT.ASC) 3
Versioning Support 3
How ACORD versioning is handled by the Parser 4
Metadata Files 4
Installation 5
SupportPac contents 5
Prerequisites 5
Supported Platforms 5
Installing the executable programs 5
Installing the metadata files and environment variable 6
Defining the message dictionary in the registry 6
Additional considerations 7
Using the Parser 7
General data structure 7
Naming of groups and data elements 7
Determining the names of the data items 8
Input Messages 8
Group Names 8
Field Names 9
Output Messages 10
Using the source code 11
Building the parser 11
Building the Message Catalog 11
Using the offline utilities 11
Error Messages 11
Parser Implementation 12
Parse Tree Structure 12
Handling of metadata files within the parser 12
Some more detailed design points 13
Character Compression 13
Parsing of input messages 13
Parsing of output messages 14
Handling of headers in output segments 14
System management messages 14
Recognition of management messages 14
Message types supported 14
System Management Message Formats 15
Flushing and monitoring the metadata cache 15
Capturing Statistics 15
Turning trace on and off and displaying trace status 16
Displaying the Level of the Executing Parser 17
Implementation considerations 17

Generation of names for Groups and Data Elements 18
Environment Variables 18
Customization of the supplied metadata files 19
Implementation details 19
Parser Initialization and Termination 19
Handling of Input Messages 19
Parser Context 20
Initialization functions 21
Parsing Routines 21
Termination Routines 22
Handling of Output Messages 22
Offline Utilities 22
Building the Metadata files 22
Editing entries in the REPEAT.DAT file 23
Problem Determination 23
Parser Exceptions 23
Debug version of the parser 24
Using the debug version 24
Reporting bugs 24
Hints and tips for writing a parser 25
What is a logical message and what is a wire format? 25
What do parsers do? 25
How do Parsers work? 25
What is "partial parsing"? 26
Parser Context 26
What happens if a parser encounters an error? 27
How do the completion bits found in message elements work? 27
What data types are supported and how are they stored internally? 28
Code pages and input buffers 29
Parser Utility Functions 29
Using the CciLog and CciThrowException utility functions 29
Creating a Message Dictionary 29
Calling the CciLog and CciThrowException functions. 30
Using Microsoft Foundation Classes (MFC) in a parser 30
What does the iFpIsHeaderParser parser function call do? 30
Appendix A – Group identifiers and long names 31
Appendix B - Error Message Details 36
Error Message Text and Likely causes 36
Message 10 (No Message Header Group found) 36
Message 11 (No Message Trailer Group found) 36
Message 12 (No Transaction Header Group found) 36
Message 20 (Invalid length (nn) in group header (bytes 4-6) for segment (xxxx)) 36
Message 21 (Message length (nn) does not match sum of group lengths (mm)) 36
Message 23 (Meta Data file not found for (xxxx) at offset (nn)) 36
Message 24 (Group length (mm) does not match metadata length (nn)) 37

Notices

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

Information contained in this report has not been submitted to any formal IBM test and is distributed “asis”. The use of this information and the implementation of any of the techniques is the responsibility of the reader. Much depends on the ability of the reader to evaluate these data and project the results to their operational environment.

The performance data contained in this report was measured in a controlled environment and results obtained in other environments may vary significantly.

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
- WordPerfect Corel, Ltd.
- ACORD ACORD

Preface

This SupportPac contains a parser written for MQSeries Integrator Version 2.0.1. The supplied parser is designed to operate in the Windows NT environment, although steps have been taken to make the source code platform independent. It adds support for parsing of input messages and creation of output messages in the ACORD AL3 format. The SupportPac includes executable programs for the Windows NT and Windows 2000 environments, including metadata files, as well as the source programs and documentation.

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

The ACORD AL3 standard is commonly used within the insurance industry, primarily for the exchange of electronic data.

All executable programs provided with this SupportPac have been compiled using the Microsoft Visual C++ V6.0 compiler. All source programs should be compiled with this compiler as well. The programs are largely written to the ANSI C standard, with some use the Microsoft Foundation classes. 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 ACORD AL3 message formats, and some additional details on how parsers work in an MQSeries Integrator Version 2.0 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.

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. Third, Mark Orlandi of the ACORD organization was very helpful with understanding many of the fine points of the ACORD AL3 standards. Finally, the author wishes to thank the many people who also helped but who the author has unintentionally omitted from this brief mention.

Introduction to the ACORD AL3 standards

The ACORD organization is responsible for the maintenance and development of the Electronic Data Interchange (EDI) standards for the exchange of data between property and casualty organizations, their agents and other trading partners. These standards allow industry partners to electronically exchange information such as policy/submission, claim and accounting data. These standards are known as AL3.

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

ACORD AL3 standards are available to ACORD members, EDI participants and subscribers. A brief introduction to ACORD messages is contained in this document for the convenience of reader. For additional information on using the ACORD AL3 standards, please see the ACORD web site (see below).

ACORD AL3 message format

The ACORD message format is designed to allow the exchange of insurance data between different organizations. The AL3 message formats are defined and maintained by the ACORD organization, which can be located on the web at:

http://www.acord.com

The actual ACORD standards are kept in WordPerfect documents. Each standard is assigned a three-digit number. Lines of business are used to organize the standards. The first digit of the standard number indicates the line of business. For example:

- 300 series - Personal lines
- 500 series - Commercial lines
- 600 series – Claims
- 700 series - Accounting
- 900 series - Common standards

Many of the standards define data elements (fields) and groups (records). A particular field is uniquely defined with a five-character name (reference name), and may appear in more than one data element group. If a field does appear in more than one place, its characteristics (such as length, data type, etc) should be the same. Each data element group has a four-character identifier, consisting of a number and three letters.

The ACORD standards combine the data groups to form transactions. Each transaction represents a business event. An AL3 message consists of a message header group and a message trailer group, with one or more transactions contained between the header and trailer. Each transaction begins with a transaction request group.

Some standards do not have any data elements or groups, but rather discuss other aspects of the standard. For example, the 900 standard explains the meaning of the various data types used in data elements and the 910 standard provides details for the structure of transactions.

There are approximately 50 current Acord standards defined, of which approximately 40 define data elements and groups.

The metadata (schemas) for all ACORD AL3 standards are contained in a single text file (datadict.asc). The file is broken into fixed length columns, and includes a definition of each

data element within each data group. The first two data groups defined within the data dictionary file are the formats of the two types of entries contained within the dictionary itself. The characteristics of each field, such as the data type, length, offset within the group, and the standard in which the data element is defined, are specified.

What does an ACORD AL3 message look like?

An Acord AL3 message consists of one or more transactions. Each transaction will contain a number of individual segments (groups). Each individual group has a header. The first ten characters of the header are in a standard format, with the first four characters being the identifier and the next three characters being the length of the group. For all Acord defined messages, the identifier consists of a number followed by three letters. The next group can be found by adding the length to the offset of a particular group. The format of the rest of the group can be determined by using the appropriate metadata for the group. Some groups can occur more than once within an individual transaction.

The AL3 standards identify message layouts. The groups in a transaction have a hierarchical structure. For details on these structures, please refer to ACORD standard 921.

All Acord messages should start with a message header group (1MHG) as the first group in the message and contain a message trailer group (3MTG) as the last group. Individual transactions are contained between the header and trailer groups within the message. All transactions start with a transaction header group (2TRG), followed by a transaction control group (2TCG). The transaction header groups are then followed by a number of other groups that provide the data for the particular transaction. An individual Acord message can contain one or more transactions.

Transactions

An ACORD message consists of one or more transactions. A transaction represents a business event. Each transaction begins with a Transaction Request Group (2TRG), generally followed by a Transaction Control Group (2TCG). The rest of the transaction consists of a number of other groups. The particular groups that make up a transaction will vary depending on the type of transaction. There is a definite hierarchy but the parser supplied with this SupportPac does not verify whether the particular groups represent a valid transaction or not.

What sorts of data are contained in ACORD messages?

All data in an ACORD message is in character format (e.g. no binary, packed decimal data, etc). Numeric data is represented as a string of numbers, padded on the left with leading zeros. If the number can be signed, then a sign character follows the numbers. If the sign is a plus character or a blank, the number is assumed to be positive, and if the character is a minus sign, then the number is assumed to be negative. The number may also contain an assumed decimal point. There is no support for floating point (real) types of data.

The characters permitted in a particular field may be limited, depending on the definition of the field. Some possible options for the data in an individual data element (field) are alphabetic, alphanumeric, telephone numbers, and numeric data. Alphabetic data should contain only the 26 letters in upper case only. Alphanumeric data should consist of upper and lower case letters, numbers and certain special characters. Telephone numbers should contain only numbers and certain special characters. Numeric data can be signed or unsigned. An eight-character date format (YYYYMMDD) and a time format are also used.

Coded fields are also supported. Coded fields must contain one of a set of specifically defined codes. The codes consist of one or more characters. A specific type of coded fields is the yes/no field, which consists of a single character that should be either a “Y” (yes) or an “N” (no).

Individual fields (data elements) are contained in groups (message segments). Each group starts with a ten-byte header. Some groups have a twenty-byte header extension immediately after the header. The first four bytes of the header contain a segment identifier. The next three characters are a length field. The length field contains the length of the entire segment, including the header, as a three-digit number. The ninth character contains either a group version or a blank.

Compression

ACORD messages can be sent as normal uncompressed data. Two forms of compression are supported. There is no indicator in the message to indicate if the message data is in a compressed format.

The first form of compression is simple character compression. Areas of the message that contain repeated sequences of the same character are compressed to a four-byte sequence. The first character of the replacement sequence is a unique data character (hex “FA”) that is not valid in any other part of the message. The second and third bytes contain a length field, consisting of two hexadecimal characters that encode the length of the repeated characters sequence. The fourth character is the repeated character itself.

The second form of compression uses a package provided by an independent software vendor. The ACORD AL3 parser that is provided as part of this SupportPac does not support this form of compression.

ACORD AL3 Data Dictionary (DATADICT.ASC)

The official ACORD AL3 standards are stored as WordPerfect documents. A data dictionary file containing all the data elements and groups defined in the AL3 standards is available as part of the 910 standard. There are many different versions of the data dictionary file (25 in total) available on the CD distributed to members of the ACORD organization. Each data dictionary contains a definition of the ACORD standard at a different point in time. The first data dictionary contains the standard as defined in November of 1989, and the most recent dictionary contains the last defined AL3 standard as defined in March of 1999.

The metadata supplied with the parser contain definitions from all the data dictionaries supplied on the current ACORD CDROM.

Versioning Support

The ACORD AL3 standards support two types of versioning.

Each AL3 standard is represented by a three-digit number, and has a version and modification level. The version and modification levels are each single digit fields. Valid levels are the digits 1 through 9, followed by the letters A through Z.

When an ACORD message is built, the transaction control group, with an identifier of 2TCG, is used to indicate the levels of each standard used within a transaction. For example, if a 5BPI group were included in the message, then the 2TCG group would contain a field with the standard that defines the 5BPI group, in this case 920, followed by the version and modification level of the standard that was used. For example, if the version 5 and modification level 1 version of the 920 standard was used to create the 5BPI group in a transaction, then the 2TCG group would contain an entry of 92051.

Group versioning support was introduced about 1995. Each defined group can include a group version number in the group header. The group version is separate and distinct from the version and modification level of the standard of which the group is a part. If a group version is present, it takes priority over the standard version and modification level in the 2TCG group. When a change is made to any data element or group within a standard, a new

modification level or version of the standard is created. Groups that are not changed by a new version or modification of a standard retain their existing group versions.

If the group version is present in the header, it will take priority.

The group version number of each group that is changed with a new level of a standard is increased by one. Thus, if a group is not changed, than it will retain the same group version it had in the previous level of the standard. The group version number, if present, should be located in the 9th byte of the header.

Group version numbers do not appear to be entirely consistent. There are cases where more than one group version will exist in the same version and modification level of a standard. There are also cases where a group version number appears to have been skipped, in that the particular group version does not appear in any of the data dictionary files provided on the ACORD CDROM.

How ACORD versioning is handled by the Parser

Each group has one or more metadata files associated with it. A metadata file is created for each version of every group. The file name consists of the four-character group identifier (such as 5BPI or 2TCG) plus a one or two character extension. If the group has a group version number, then the group version number is appended to the group identifier to form a five-character metadata file name. If there is no group version number, then the two character standard version and modification level is appended to form a six-character file name. All of these metadata files have a file extension of “mtd”.

To determine the version number for a particular segment, the segment header is first examined. If it contains a segment version character, then this is used. If this character is not present (blank), then the segment identifier must be mapped to a particular standard. The levels of each standard used to build a message are contained in the 2TCG segment. This segment can hold up to 20 standards and their corresponding version and modification levels. Once the standard that a segment is part of is identified, then the 2TCG segment is searched for the version and modification level used to build the message. This standard version and modification level is then used to find the appropriate level for the particular segment.

The version levels for the message header (1MHG) and trailer (3MTG) are specified in the message header, and the levels for the transaction headers (2TRG and 2TCG) are specified in the beginning of the transaction header (2TRG).

Metadata Files

Metadata for ACORD groups are kept in files. Every version for each group is kept in a separate file.

The metadata files must be installed in a directory on a disk accessible to the MQSI Version 2 broker where the parser will run. An environment variable with a name of ACORDMETADIR should be set to the drive and directory where the meta data files are installed. If this variable is not set, then the parser will look for the metadata files on the “C:” drive in a directory named “\ACORD”.

In addition to the individual metadata files for each group, three additional files are created.

The first file is a cross-reference file between group identifiers and ACORD standard numbers. This file is named “*STDFILE.MTE*”. It contains an entry for each group version for a particular group. It is used to identify which group version number to use for a particular standard version and revision level. It is also used to identify which standard a particular group belongs to.

The second file is the default group version file. This file is named "*DEFFILE.MTE*". It is used when no group or standard versions are specified in the group header or in the transaction control group. It contains an entry for each group identifier, to indicate what metadata file to use if no group version or standard revision and modification levels are given in the message.

The third file is the group name file. The normal ACORD names for a particular group consist of a number and a three-character acronym. The names are short and may therefore be a bit cryptic. Therefore, an alternative name that is longer and hence more descriptive is provided. When a message is parsed, the group identifier will be used to find a longer and more descriptive name for the top-level item corresponding to the group. If the name is not found in the group name file, then the four-character group name is used, with a leading underscore character added to the beginning of the name. The underscore character is necessary because names for data elements in the logical message model may not begin with a number. For output operations, either the long name or short name may be used for any group.

The three special metadata files contain character data only and can be edited with a character editor such as notepad. The metadata files for the individual groups contain binary data and cannot be edited or changed with a simple text editor. A utility program is provided to display the contents of these metadata files.

Installation

SupportPac contents

The supplied zip file should be unzipped into a temporary directory. The following files will be created.

- exec.zip
- metadata.zip
- source.zip
- utilsrc.zip

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 Microsoft Visual C++ V6 is required.

Supported Platforms

This SupportPac has been developed for and tested in a Windows NT (Windows 2000) environment.

Installing the executable programs

The executable zip file should be unzipped into a temporary directory. The following files should be created from the zip file:

- segparse.lil
- acorderr.dll
- segparse.dbg
- MessageFlowsAcord
- acordasc.exe
- printmtd.exe
- procobol.exe

- repeat.dat
- defnames.dat

The parser executable (*segparse.lil*) should be copied 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 (acorderr.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 (segparse.dbg) should also be moved to the bin directory.

The MessageFlowsAcord contains three sample message flows that can be used to validate the proper functioning of the parser. They 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:

- ACORD.IN
- ACORD.OUT
- ACORDMGT.IN
- ACORDMGT.OUT
- ACORDXML.IN
- ACORDXML.OUT
- FAILURE

The utility executables 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 files and environment variable

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:

- ACORDMETADIR – 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 “Acord” on the “C:” drive.

If fields which are defined as either deleted or reserved for future use are to be ignored on input parsing operations, the ACORD_IGNORE_FILLER environment variable should be set to a “1” (without the quotes).

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 register 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 “acorderr” (without the double quotes) as the name of the key. Select the new acorderr 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 (acorderr.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\acorderr.dll”

Finally, click on the acorderr 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.

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.

Using the Parser

General data structure

The ACORD AL3 parser takes input messages in valid ACORD AL3 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 AL3 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 an ACORD AL3 message, the name of this element should be “ACORD”, to match the message domain supported by the parser.

Naming of groups and data elements

Each group within the message will have a top-level element that is a child of the body element. The name of the top-level element for each group will be the name of the first entry in the corresponding metadata file. This name is derived from the description field of the group in the most recent ACORD data dictionary file (19990316 datadict.asc). In some cases, the name was shortened to be less than thirty characters. In cases where the

description field did not yield a unique name for the group, the name was changed by the metadata generation utility. The mapping between a particular group identifier and the corresponding data element name is contained in a special metadata file (segname.mte). This file is generated by the metadata generation utility.

All of the individual data elements for a particular group are contained in a structure under the corresponding group element that they belong to. The data elements are expected to be in the same order as they appear in the data dictionary file. The names of the data elements are the names as they appear in the metadata file for the corresponding group. If a particular group appears more than once in the AL3 message, then there will be multiple instances of the particular group in the logical message tree.

Determining the names of the data items

There are two ways to determine the names that have been assigned to the individual data elements in each group. The first method is to use the metadata display utility program (printmtd.exe) to create a list file from a metadata file. The name of the metadata file will be the four-character group name followed by either the group version number or the standard version and revision numbers. The list file should be browsed with an editor, such as the notepad utility.

The second method is to create a valid AL3 input message and pass it through a message flow that uses the Acord parser provided with this SupportPac. The output should then be transformed to XML and written to a queue or a trace node should be inserted in the message flow to dump out the contents of the body of the parsed message.

Input Messages

The ACORD AL3 parser will parse any inbound message that is in a valid AL3 format. It will perform limited checking of the message format. The AL3 parser will register with the execution broker for a message domain of “ACORD”. It will attempt to parse any input message that is read by MQSeries Integrator V2. 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 “ACORD”. The rest of the data for a particular group will be built as a logical data structure under this high level element.

It is recommended that the data translation option of the MQInput node not be used. 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 must match the code page in the MQSeries message descriptor.

The parser will ignore any carriage return or line feed characters that are found between group entries or at the end of the message. The ACORD standard does not state whether such sequences are valid or not. However, some messages seem to contain these sequences between groups or at the end of the message, and therefore the parser will ignore them if they are found. In the case of EBCDIC messages, the line feed character is assumed to be a x’25’ character.’ The parser will also translate any binary zero characters found in the input message to blanks.

Group Names

The ACORD AL3 parser supports two types of names for groups. The first type of name is a short name, which consists of the four-character group identifier preceded by an underscore character. Longer and more meaningful names have also been assigned to all groups.

These names are documented in Appendix A. The longer name mappings are kept in one of the special metadata files (segname.mte). This file is in a text format and can be edited with a text editor. However, the long names used during input parsing operations are contained in the individual metadata files. If the long names are changed, then the individual metadata files must be rebuilt.

By default, the parser will use the longer names for all parsing of input messages. It will accept either the longer names or the short names for output messages. Use of long and short names in a single message is supported.

Field Names

There are no generally agreed names for the data elements that are defined within the ACORD standards. Within the data dictionary file that is provided as a part of standard 010, there is a five-character reference name and a sixty-character description. The reference name is short and hence rather cryptic. The reference name can also be used more than once in a single group. The description is long enough but unfortunately not a good candidate for variable names in many cases. In the future, when a new XML based standard is available, a mapping may be available that would provide more descriptive names.

The algorithm used by the offline utilities, which generate the metadata files, is to create a field name by using the reference name. If the reference name is used more than once in the individual group, the generated name is made unique by appending the sequence number of the data element within the group to the end of the reference name. The long name used for the group itself is contained in a separate file that is used by the utility.

There are two types of fields that are treated specially. First, fields which are reserved for future use or which have been deleted (reference names ZZZZZ and ZZDEL respectively) are flagged in the meta data file as filler fields and can be ignored on input parsing operations. An environment variable can be set to control this behavior (see the section on environment variables).

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.

The following names are assigned to the individual header and, if present, the header extension fields.

Name	Offset	Length	Comments
Identifier	0	4	
Length	4	3	
Format	7	1	
Version	8	1	
Reserved	9	1	Filler
ProcLevel	10	2	
Iteration	12	4	
Ref_Identifier	16	4	
Ref_ProcLevel	20	2	
Ref_Iteration	22	4	
ActionFlag	26	3	
Spare	29	1	Filler

In certain cases, data elements (fields) or groups of fields can occur more than one time in an individual group. In these cases, it is more logical to treat these fields as a repeating element or structure rather than assign each individual field a unique name. Therefore, the metadata files will indicate a repeating field or structure rather than a series of individual fields.

Output Messages

The message flow must create an output message in a similar format as the input message.

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.

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.

Building the parser

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.

A directory for the source files should be created and the zip file containing the source files for the parser (segparse.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 segparse workspace should then be opened.

Before the project can be built successfully, the locations of the include and library files 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.

Two 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

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.

Using the offline utilities

Error Messages

There are certain situations where the parser will generate an error. For example, if the message does not appear to be a valid ACORD AL3 message, then an exception will be raised and the message will be rejected. For example, if there is no Message Header or Trailer Group, or if one of the length fields in a group header is invalid, then the parser will raise an exception. An error message will be written to the event log. The Windows event

viewer should be used to view the error information. For a detailed list of error messages, see the appendix below.

Parser Implementation

Parse Tree Structure

ACORD messages have a definite structure. Each message begins with a message header group and ends with a message trailer group. There are one or more transactions contained between the message header and trailer. Each transaction consists of several groups that contain the data for a particular business event. Each transaction begins with a transaction header, which is usually followed by a transaction control group. Additional data groups provide the necessary data.

The parse tree structure that is built reflects this structure. The first child of the body is a message header (MessageHdr) element. Information from the message header group (1MHG) is inserted as children of this element. The last child of the body is a message trailer (MessageTlr) element. This element contains information from the message trailer group (3MTG). The data for each transaction in the message is stored under a transaction element (Tran). The transaction element is a child of the body element. The body will have one child for each individual transaction in the message plus one child for the message header and one child for the trailer data.

The message header element is named “MessageHdr”. The name of each transaction element will vary, reflecting the type of transaction. It consists of the following fields:

- MsgOrig Message address (origination)
- MsgDest Message address (destination)
- ContractNo Contract number
- UserPw Password
- SysType System type code
- RevLevel Interface software revision level
- SeqNo Message sequence number
- TransDate Transaction date (13 char - older standards)
- CntUnitCd Count unit code
- SpecHndlg Special handling
- VERNO Message standard revision level
- NetRefNo Network reference number
- NetResvd Network reserved for future use
- TransDate Transaction date (15 char - more recent standards)

The message trailer element is named “MessageTlr” and contains the following data elements (fields):

- MsgLen Message length
- AddlDataFlag Additional data flag
- CommTextFlag Communications text flag
- CommText Communications text

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. The files are contained in a directory based on the version number of the ACORD message segment.

The internal structure of the file is in five parts. The first part of the file is a metadata header. It contains overall characteristics of the segment and the meta data 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)
- Segment length (Maximum)
- Number of variables
- Offset of name and Unicode name tables
- Offset of initialization string

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 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 optional. If it is present, it contains an initialization string with the initial values for the data area. This is used when an output data area is built, to increase processing efficiency.

Initialization strings for ACORD defined groups will only contain space and zero characters, except for the header. The group identifier, length and group version fields in the header will be filled in.

Some more detailed design points

Character Compression

An ACORD message may contain long sequences of a single character. To increase transmission efficiency, the repeated characters are replaced with a four-byte character sequence. The replacement sequence begins with a unique character (hex “FA”), followed by a two-character length field and a single occurrence of the replaced character. The length field consists of two hexadecimal characters, with the largest allowed value being “E6” (230). For example, if a sequence of 17 spaces was found in the output message, it would be replaced by four characters, namely a “FA” character, two occurrences of the digit “1”, and a space character.

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 ACORD AL3 format and will then assume ownership of the body of the message.

The first time that a field in the body of an ACORD message is referenced by a node, MQSeries Integrator V2.0 calls a routine within the ACORD 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 group (segment) at a time. Each segment is identified by the value of the first four characters. A three-character length field follows the identifier. The segment length is checked to ensure that it matches the metadata, and is used to find the start of the next segment. Parser exceptions are thrown if the identifier is not recognized or the length is invalid (less than ten or longer than the remaining characters in the message).

An environment variable can be used to control whether fields that are marked as deleted or reserved for future use are ignored or added to the message tree.

Parsing of output messages

The output message is built from the logical message tree. Each child of the body element must have a name that is a valid group (segment) identifier with an underscore character added to the beginning.

For each identified segment, an output segment of the appropriate length is built. Each field within the segment is then initialized to either blanks or zeroes, depending on the data type. The first seven characters of the segment header (the identifier and length) and the segment version are also filled in automatically. If these fields are found in the logical message tree, the data from the message tree will override these defaults. Each element in the message tree under the segment element is then examined. If the name of the element matches the name of an element in the metadata, then the data in the corresponding field is updated. If the name does not match the name of a field in the metadata file, then the field is ignored. Fields in the logical message tree must be in the same order as the fields 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.

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 ACORD AL3 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 "*SYSMGMT*". 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 "*ACORD*". The contents of the message should then follow the rules outlined below, rather than the normal ACORD 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 ACORD groups that are to be flushed should be provided, as a series of four character fields. For example, to flush all entries in the metadata cache for the 5BPI group, then the characters 5BPI should appear in the list of group identifiers to be purged. All metadata entries that start with the specified group identifier will be flushed from memory.

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 “ACORD_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.

```
<ACORD>
   <statistics>
      <cache>
         <filesOpened>nnn</filesOpened>
         <filesRemoved>nnn</filesRemoved>
         <filesMax>nnn</filesMax>
         <useCounter>nnn</useCounter>
      </cache>
      <messages>
         <messageCount>nnn</messageCount>
         <compressedmessages>nnn</compressedmessages>
         <segmentCount>nnn</segmentCount>
         <writeCount>nnn</writeCount>
         <invalidMsgCount>nnn</invalidMsgCount>
         <maxMessageSize>nnn</maxMessageSize>
         <minMessageSize>nnn</minMessageSize>
         <averageMessageSize>nnn</averageMessageSize>
         <averageUncompressedSize>nnn</averageUncompressedSize>
         <writeBufferCount>nnn</writeBufferCount>
         <maxOutputMsg>nnn</maxOutputMsg>
         <minOutputMsg>nnn</minOutputMsg>
         <averageOutputMsg>nnn</averageOutputMsg>
         <displayFieldsTruncated>nnn</displayFieldsTruncated>
         <numericFieldsTruncated>nnn</numericFieldsTruncated>
         <averageParseTime>nnn</averageParseTime>
         <maxParseTime>nnn</maxParseTime>
         <minParseTime>nnn</minParseTime>
         <averageWriteTime>nnn</averageWriteTime>
         <maxWriteTime>nnn</maxWriteTime>
         <minWriteTime>nnn</minWriteTime>
      </messages>
   </statistics>
</ACORD>
```

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 ACORD, so that the standard ACORD 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 ACORD 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:

```
<ACORD>
    <ModuleLevels>
        <Acordsub>nnn</Acordsub>
        <Cobsubs>nnn</Cobsubs>
        <Metadata>nnn</Metadata>
        <Mgmtmsg>nnn</Mgmtmsg>
        <Miscsubs>nnn</Miscsubs>
        <Parsubs>nnn</Parsubs>
        <SegParse>nnn</SegParse>
        <Trace>nnn</Trace>
    </ModuleLevels>
</ACORD>
```

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. Extra fields in the logical message model (e.g. fields which do not match a field in the output message) will be ignored. Fields of type yes/no fields in the ACORD standard will be initialized to a blank character.

Many of these questions do not have clear answers. Design choices were made to reduce the complexity and offer the most flexibility.

Field values are not checked for valid data. This includes the characters used in things like alphabetic or text 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. Some simple checking is performed, primarily to ensure that the data represents a well-formed ACORD message. In particular, input messages are checked as follows:

- The message begins with a 1MHG group and ends with a 3MTG group.
- The second group is a transaction group (2TRG).
- All group identifiers are recognized.
- The length of each group is valid and matches the metadata length.

For a group length to be considered valid, it must be at least ten and less than the number of remaining characters in the message.

For output messages, the following checks are performed:

- All children of the body element and any transaction elements (“Tran”) have names that are valid AL3 groups.
- The first child must be a message header (“MessageHdr” or “_1MHG”) and the last child is a message trailer (“MessageTlr” or “_3MTG”).
- The second child of the body is a transaction element (“Tran”).

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

Data truncation will never occur on an input message. On output, however, the length of the data in a variable can exceed the length of the corresponding ACORD AL3 data element. In this case, truncation will occur. All truncations will be counted and are available in the statistics. In addition, if tracing is enabled, an entry will be written to the trace file for each field that is truncated.

Generation of names for Groups and Data Elements

Each group is assigned a unique four-character identifier by the ACORD standard. The identifier consists of a number and three letters, for all groups defined in the standard. The identifier is not particularly suitable for a data name, since it starts with a number and has only three meaningful characters. It is desirable to have a longer and more descriptive name for each group.

Long group names have been assigned for each group. The long name and the associated group identifier are contained in a text file (defnames.dat). This file is used by the utility that creates the individual metadata files from the ACORD data dictionary.

The long names are based on the description field associated with each group. In all cases, the long name has been limited to less than thirty characters. The word “Group” has been removed from the end of any long name in which it would have otherwise appeared, and capitalization has been made consistent and many words have been abbreviated. Long group names that would have duplicated the long names of other groups have been made unique. These longer names have been used in the generation of all metadata files. The long names can also be used when a message flow generates a logical message structure that will be turned into an ACORD message.

A short name is also supported. The short name begins with an underscore character and includes four more characters. If the group name begins with an underscore, then the group identifier should be in the next four characters of the element name. This identifier is used to locate the appropriate metadata.

Environment Variables

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

If statistics are to be written to a file, the “ACORD_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.)

Some additional environment variables can be set to override handling of certain specific situations that may be encountered in the parsing of ACORD messages. They are as follows:

> ACORD_IGNORE_FILLER – This environment variable is used to control the treatment of fields marked as filler fields in the metadata files. If this variable is set to a "1" (without the quotes), the contents of any filler fields will be ignored and no elements will be added to the logical message tree during parsing of an input message. If this variable is not set or is set to any other value, these fields will be treated as any other field and the input data contained in them will be accessible to any message flows. The flag is generally set by a utility that generates the metadata files, when a field is a deleted field or is reserved for future use. Fields are marked as filler fields if they are tagged as deleted or reserved for future use in the ACORD metadata dictionary.

Customization of the supplied metadata files

A complete set of metadata files is supplied with the SupportPac. The metadata files have been created from the data dictionary files that are delivered on the Acord CDROM. The data dictionary files are NOT supplied with this SupportPac.

Two utility programs are provided to allow the metadata files to be changed. The first utility will process an ACORD data dictionary file and produce the corresponding metadata files. The second utility will process an individual COBOL copybook and generate a metadata file that corresponds to the copybook. Full source code for both programs is provided. Please see the utilities section later in this document for details of using these utility programs.

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 ACORD format. Each group must have a recognized four-character identifier followed by a three-character group length field. Subsequent groups must follow immediately after the preceding group.

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.
eParent	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. If the message was compressed, this is the size of the message before being decompressed.
iLength	Length of the message area to be parsed. If the message was compressed, this is the length after decompression.
iDataPtr	Pointer to the input data buffer.
iBuffer	Pointer to the data to be parsed. If the data was not compressed, this is the data in the original input buffer. If the data was compressed, then this is a pointer to the buffer is acquired to hold the decompressed data.
iOffset	Number of bytes that have already been processed in the message.
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.
iEncodeInt	Integer encoding format. This field indicates if integers are encoded using PC format (iEncodeInt = 1) or host (iEncodeInt = 0).
iEncodeIPD	Packed decimal encoding format. This field indicates if packed decimal fields are encoded using PC format (iEncodePD = 1) or host (iEncodePD = 0).
iFiller	Indicator if filler fields should be included in the parse tree on input operations. This indicator is filled in based on the ACORD_IGNORE_FILLER environment variable.
iMsgType	This field will be set to one if the message is a special management message.
iInTran	This field will be set to one if the parser is currently parsing a group which is part of a transaction. Transactions are started with a transaction control group, and continue until the next transaction control group or the message trailer is found. The message header and message trailer should be the only groups that are not children of a transaction.
hdrVersion	Version number of the header and trailer groups. This is taken from the corresponding field in the 1MHG message group.

trnVersion Version number of the transaction header and control groups. This value is taken from the corresponding field in the 2TRG group.

msgVersion Version information that is found in the 2TCG group.

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 (iDataPtr) 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 (iLength). It initializes the parent (iParent) 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.

This routine will next check if the input message has been compressed. If it was compressed, then a new buffer will be allocated to hold the decompressed message and the message will be decompressed. The message format will now be checked to make sure that the input message is in fact an ACORD AL3 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 acordsub.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

Offline Utilities

Three offline utilities are provided to assist in building the necessary metadata files. The first utility will process the ACORD data dictionary and produce the corresponding metadata files. The second utility will produce a formatted text file showing the contents of a metadata file. The third utility will process a COBOL copybook and produce a corresponding metadata file. The utilities are named ACORDASC, PRINTMTD and PROCOBOL respectively.

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.

The last utility program is provided in case an individual user has produced their own COBOL copybooks to represent ACORD groups. It is not normally used to build the metadata files used by the parser.

In general, the offline utilities are not required, since this SupportPac includes a complete set of metadata files for all versions of the ACORD data dictionaries that are provided by ACORD.

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 ACORD data areas, or different data naming conventions are desired.

The metadata files are built using the ACORD data dictionary command line utility (ACORDASC.EXE). A command file is provided that will process all data dictionary files provided on the ACORD CDROM. These files are located in the “Data Dictionary” directory.

The utility uses a repeated data element file and a group name file that contains long names to be used for group names. A repeated data element is a data element with the same reference name that has more than one occurrence in the same group, and which would normally be represented as multiple occurrences of the same variable. There are cases where the same reference name is used in a group but the variables are not really occurrences of the same item. Therefore, a control file is necessary.

The repeated data element file (REPEAT.DAT) is used in the processing of variables that are repeated in a particular group. If a variable’s reference name and sequence number matches an entry in the repeated data file, then this variable will be treated as a group entry rather than individual data elements. The result will be a metadata structure with multiple occurrences of a data item rather than as individual data items with unique names. A version of this file is provided with the SupportPac and the file normally does not have to be changed. A description of the layout of this file is given below. The REPEAT.DAT and DEFNAMES.DAT files should be in the same directory as the ACORDASC.EXE utility when the utility is executed.

The input to the dictionary processing utility is an ACORD data dictionary file. The dictionary file is processed, and a metadata file is produced for each data group (segment) encountered. In addition to the individual metadata files, a defaults file and a version standard mapping file are produced. The parser uses the version standard mapping file to determine the proper metadata file to use for a particular data group, based on the version and modification level of the standards in the Transaction Control Group.

The first step in using the dictionary processing utility is to create a directory and copy the executable program (ACORDASC.EXE), the repeating data file (REPEAT.DAT), group names file (defnames.dat), the command file (BUILDMTD.CMD) and the ACORD data dictionary files from the ACORD CDROM into a directory. The command file should then be executed, and all the metadata files for the parser will then be created. All the files created will have an extension of either “MTD” or “MTE”. If necessary, the metadata files should then be moved or copied to the desired execution directory. The ACORDMETADATA environment variable should be set to point to this directory.

Editing entries in the REPEAT.DAT file

The repeating items file (REPEAT.DAT) is used as input to the utility program that processes ACORD data dictionary files and creates the corresponding metadata files. This file is a text file. Each line consists of a variable length text string. The first part of the string is in a fixed format, and there is an optional variable name at the end of the line. Each line must end with a carriage return and line feed (CRLF) sequence.

The fixed portion of the line contains the following fields:

- 4 character group identifier
- 5 character reference name
- A single character of either ‘S’ for a single repeating variable or ‘C’ for a group of repeating variables.
- 4 digit number with the number of occurrences
- 4 digit sequence number of the first variable
- 4 digit sequence number of the last variable for a single repeating variable or a count of the number of variables for a group
- Variable name (up to 30 characters)

Problem Determination

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

Parser Exceptions

The ACORD AL3 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 therefore 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.

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 generally 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.

- ACORDTRACEFILEDEF – location and name of trace file.

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

- ACORDTRACE
- ACORDTRACEPARSER
- ACORDTRACEWRITE
- ACORDTRACEMODULE
- ACORDTRACEMGMT

The ACORDTRACE 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 location of “\ACORD\parser.trc” on the “C:” drive 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 (SegParse.lil) should be renamed with a different extension (e.g. SegParse.rel) and the debug version of the parser should be renamed from SegParse.dbg to SegParse.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.

Hints and tips for writing a parser

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 ACORD AL3 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 ACORD AL3 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 “acorderr.mc”. More documentation for the message dictionary formats is contained in the Microsoft SDK for Windows NT.

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 (acorderr.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 A – Group identifiers and long names

1MHG	MessageHdr
2ACI	AgencyCompanyInfo
2FID	FileTransferIdent
2FTT	FileTransferTlr
2GCG	OmittedGroupCtrl
2TAG	TransAgency
2TCG	TransCtrl
2TRG	TransHdr
3MTG	MessageTlr
4CRG	CtrlReq
4CRR	CtrlReqRej
4MAK	MessageAck
4NMK	NegativeMessage
4PTG	PhaseTerminator
4ROC	ReceiveOptCtrl
4SOF	Signoff
4SOG	Signon
4SOR	SignonRejection
4STG	StreamTerminator
5ACT	PolicyActivity
5AOI	AddlOthInt
5ATT	Attachments
5BCI	BldConst
5BIS	BasInsuredInfo
5BNG	BinderNotification
5BPI	BasPolicyInfo
5CAR	CommlLinesAuto
5CER	CertHolder
5CFG	CommonFactgroup
5CHG	Charges
5CLD	MiscCov
5CNR	CancelNonRenewReinstate
5COV	CovAndAdj
5CVG	CommlLinesCov
5DCV	CovInfo
5DRV	Driver
5DSF	DiscountSurchargeFact
5DSP	BatchDownloadSetup
5ELG	ExtendedLine
5FCI	FloodCommunityInfo
5FOR	Forms
5GIG	GenInfo
5GNA	GenNameAddr
5GPP	GenLiabilityPriPol
5GUR	MiscUnitAtRisk
5IIG	DwellingInspection
5ISI	InsuredsSuppl
5LAG	LocAddr
5LHG	LossHistory
5MED	MedStatement
5NID	NameIdentifier
5OIC	OthInsWithCompany
5PAY	PayPlanInfo
5PCG	PolicyCov
5PIG	PackageInfo
5PIN	PremiumInfo
5PMI	PayMethodInfo
5PPH	PriPolHistory

5PPI	PerPropItem
5PPS	PerPropSchedule
5PSU	PrintImageElement
5PTL	LongLine
5PTS	ShortLine
5PUL	PerUmbLiability
5QIG	QuoteInfo
5REP	DwellingReplCost
5RMK	Remarks
5SLC	SubLoc
5SNG	SupplName
5SSG	StateSummary
5TAI	TexasAddlInsured
5UQG	UnderwritingQuestion
5VEH	Vehicle
5YRI	YearlyReportingInfo
6ADR	AddlRestrictions
6ATF	BusinessAutoTruckFiling
6AUR	BusinessAutoTruckUW
6BAU	BusinessAutoTruckNotOwned
6BFR	BusinessAutoTruckFinResp
6BKH	BoeckhHighvaluedDwelling
6BKS	BoeckhSquareFootMethod
6BMA	BMOccupancyRateTable
6BMB	BMObjectCovered
6BMC	BMCovAdj
6BRT	BusinessAutoTruckSuplRating
6BVS	BusinessAutoTruckVehSupl
6CAS	BusinessAutoTruckCoverAutoSym
6CCG	LossNoticeCov
6CCI	CrimeCtrlInfo
6CCV	CrimeCov
6CDI	DriverSupplement
6CEC	CrimeEmployeesClass
6CFS	ClaimFinancialSummary
6CIE	InjuryInfo
6CKY	ClaimKeys
6CLI	CrimeLocInfo
6CMP	CrimeMessengerProt
6CMS	CrimeMoneyandSecurities
6COC	Occurrence
6COI	LossNoticeOthIns
6COL	CommlPropCauseloss
6COM	CommunicationsIdInfo
6CPA	ClaimPaySummary
6CPC	CommlLinesPriPol
6CPD	ClaimPayDetail
6CPH	CommlPropPriPol
6CPK	CommlPropBlanket
6CPL	PropLoss
6CPN	CommlPropUW
6CPO	CommlLinesPolicy
6CPP	PremisesProdOth
6CPS	CrimePremisesSafeProt
6CPU	CommlProp
6CRM	CrimeUnitAtRisk
6CSV	CrimeSafeVaultInfo
6CUA	CLUmbExcessAddl
6CUB	CLUmbExcessVehicle
6CUC	CareCustodyCtrl

6CUE	CLUmbExcessCov
6CUL	CLUmbExcessLoc
6CUR	ScapCrimeUnitAtRisk
6CUU	CLUmbExcessUnderlying
6CVA	PerAutoCov
6CVC	BusinessAutoTruckCoverAdj
6CVF	DwellingFireCov
6CVH	HomeownersDwellingFireCover
6CVL	VehicleLossInfo
6DBC	DirBillCommStmtSum
6DBD	DirBillCommStmtDet
6DBS	DirBillPayStatus
6DIS	DriverDiscount
6DRA	AddlDriverRestriction
6EMP	WCompEmpl
6FRU	DwellingFireSuppl
6FSR	DwellingLocInfo
6GCM	GenLiabilityClaimsMade
6GGA	DealersPhysicalDamage
6GGI	GarageandDealersInfo
6GGK	GarageKeepersLiability
6GGL	GarageLiability
6GGP	GaragePhysicalDamage
6GIC	ContractorsUnderwriting
6GLC	GenLiability
6GLS	GenLiabilityStateRating
6GLT	GenLiabilityTransition
6GPC	ProdCompletedOperations
6GPD	ProdDetail
6GSU	GlassSignUnderwriting
6GUW	GenLiabilityUnderwriting
6HRU	HomeownersDwellingFireRateUW
6ICR	InlandMarineComputerRoom
6IMA	CommInlandMarineUnitAtRisk
6IMB	CommlScheduledItem
6IMC	CommlInlandMarine
6IMD	InlandMarineMediaData
6IPI	InvolvedParty
6IRU	PerInlandMarine
6LUR	AllLocUnitRisk
6MEM	MemoHdr
6MHT	MobileHomeTlr
6MRG	MemoRouting
6MSA	MarshallSwiftSeriesData
6MVR	MotorVehReqResp
6PAD	AcctCurrDetail
6PAL	PolicyAcctLine
6PAS	AcctCurrSummary
6PCV	CommlPropCov
6PDA	PerAutoDriving
6PDC	AcctCurrDiscrepancy
6PDR	PerAutoDriver
6PDS	PerAutoSpecial
6PGS	CommlPropGlassSign
6PIM	PerInlandMarineCov
6PIS	PerAutoInsured
6POI	CommlPropOth
6PRP	PerUmbPriPol
6PSL	CommlPropSubLoc
6PSO	CommlPropSubjOpt

6PSR	CommlPropSpecific
6PST	CommlPropState
6PTI	ProtInfo
6PUM	PerUmbCov
6PUR	ScapPropUnitAtRisk
6PUS	PerUmb
6PVH	PerAutoVehicle
6RCA	ReplCostAddl
6RPT	ReportOrderingInfo
6SAM	SuretyJointVentureAmounts
6SBA	SuretyBondUnitAtRisk
6SBB	SuretyBondCov
6SBC	SuretyBondPolicyInfo
6SBF	BeneficiariesInfo
6SBL	SuretyPremiumBilling
6SBR	BasBondReq
6SBT	BankruptcyTrusteeReceiver
6SBX	BasBondExecution
6SCN	ContractSuretyBond
6SCO	CollateralInfo
6SDV	CommlLinesDriver
6SFI	FiduciaryBondInfo
6SIG	InsInfo
6SJU	JudicialBondInfo
6SLI	ScapLocInfo
6SLP	LicensePermitBondInfo
6SMB	SnowmobileLiability
6SMI	MiscBondInfo
6SOI	CommlPropSubjIns
6SPC	SpecialtyLinesCommon
6SPG	PrincipalInfo
6SPI	SmallCommlAccts
6SPL	ScapPolicyLimits
6SPO	PublicOfficialBond
6SRG	ContractSuretyBidResults
6SRT	SuretyRatingInfo
6SSR	StateInfo
6STR	SuretyTermsInfo
6SUB	SubsidiaryInfo
6SUC	ScapUnderlyingCov
6SUU	ScapUmbUnitAtRisk
6SWC	SelfInsuredWorkers
6TEQ	TruckersEquipment
6TTI	TruckersTlrInterchange
6TTR	TruckersTerminal
6TZO	BusinessAutoTruckZone
6VIO	MvrAccidentviolations
6WAL	WaterUnitLiability
6WAS	WaterUnitSuppl
6WCA	WCompAssigned
6WCC	WCompPremium
6WCD	WCompDefCover
6WCH	WCompPolicy
6WCI	WCompIndivual
6WCL	WCompInjury
6WCP	WCompPolicyOld
6WCR	WCompRating
6WCS	WCompState
6WCT	WCompOth
6WCU	WCompUnique

6WCV	WCompCoverAdj
6YDQ	YoungDriverQuestionnaire
8CDD	CompanyUniqueDataDictionary
8DED	SingleFieldEditDef
8DEF	TemporaryFieldDef
8DER	RelationalEditDef
8TBL	TableDef
9ADR	AddlDriverInfo
9AOI	AddlIntExt
9BIS	BasInsuredExt
9CKY	ClaimKeysExt
9COC	OccurrenceExt
9CVL	VehicleLossInfoExt
9EMP	WCompEmplExt
9IPI	InvolvedPartyExt
9WCL	WcompInjuryExt

Appendix B - Error Message Details

The parser may produce the following error messages. The messages will be recorded in the Windows NT event viewer. 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 Message Header Group found)

This message will be produced if not ACORD message header group (1MHG) segment is found at the beginning of the message. This group is required as the first group for all ACORD messages. This message generally indicates that the message is not a valid ACORD AL3 message.

Message 11 (No Message Trailer Group found)

This message indicates that no message trailer group (3MTG) was found at the end of the message. This group is required as the last group in all ACORD messages. This is usually an application error in the sending application. An incorrect length in a group header can also cause it.

Message 12 (No Transaction Header Group found)

The second group in all ACORD messages must be a Transaction Header Group (2TRG). This error will occur if this group is not a transaction header. There can be more than one 2TRG group in a message. Each 2TRG group starts a new transaction. This message usually indicates a problem with the sending application.

Message 20 (Invalid length (nn) in group header (bytes 4-6) for segment (xxxx))

This indicates that the length field in the group header is invalid. It is either less than ten or contains non-numeric characters. The three bytes following the group identifier must contain the length of the group segment. This length must be at least ten and not more than the number of bytes that remain in the message, after the beginning of this group. An incorrect length in the previous group header can also cause this message.

The offset given is the beginning of the group and the length is the value that was calculated from the three characters in the header that should contain the length. This error usually indicates a problem with the sending application.

Message 21 (Message length (nn) does not match sum of group lengths (mm))

The length of the complete input message does not match the sum of the lengths given in the group headers. There are several things that can cause this error. The most common cause is an error in one of the length fields in a group header. Data truncation can also cause this problem. The most likely cause of this problem is an error in the originating application.

Message 23 (Meta Data file not found for (xxxx) at offset (nn))

No metadata file was found for the indicated message group. This can indicate an invalid group identifier. Invalid or unrecognized group version numbers or standard level and revision numbers will also cause this error.

Message 24 (Group length (mm) does not match metadata length (nn))

This message indicates that the length of the metadata for this group does not match that specified in the group header. An invalid length in the group header can cause this problem. An invalid group version number or standard level and revision number can also cause this error to occur.

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