Microsoft Message Analyzer v1.2



OPN Configuration Guide for Text Log Adapter v2

© 2015 Microsoft Corporation

Contents

[1. **Introduction** 3](#_Toc411941287)

[1.1. Reference Examples 3](#_Toc411941288)

[2. **Rendering Data From the ExampleTextLog.log File** 3](#_Toc411941289)

[3. **Specifying Message Definitions** 5](#_Toc411941290)

[3.1. LogEntry 5](#_Toc411941291)

[3.1.1. LogEntry Definition 5](#_Toc411941292)

[3.1.2. LogEntry Example 5](#_Toc411941293)

[3.2. EntryInfo Aspect 6](#_Toc411941294)

[3.2.1. EntryInfo Aspect Definition 6](#_Toc411941295)

[3.2.2. EntryInfo Example 7](#_Toc411941296)

[3.2.3. Multiline Log Entries Example 8](#_Toc411941297)

[3.3. EntryFieldInfo Aspect 9](#_Toc411941298)

[3.3.1. EntryFieldInfo Aspect Definition 9](#_Toc411941299)

[3.3.2. EntryFieldInfo Example 10](#_Toc411941300)

[3.4. DisplayInfo Aspect 10](#_Toc411941301)

[3.4.1. DisplayInfo Aspect Definition 10](#_Toc411941302)

[3.4.2. DisplayInfo Example 10](#_Toc411941303)

[4. **Mapping a Message Definition to a Log File Line** 11](#_Toc411941304)

[4.1. Definition Evaluation Algorithm 11](#_Toc411941305)

[4.2. Setting the Priority Level 11](#_Toc411941306)

[5. **Handling Unknown Log Entries** 11](#_Toc411941307)

[6. **Placing Configuration Files in the Default Location** 12](#_Toc411941308)

[7. **Modifying Configuration Files** 12](#_Toc411941309)

[8. **Testing Regular Expressions** 12](#_Toc411941310)

# Introduction

Microsoft Message Analyzer parses several types of text logs by default. To make this possible, each supported text log type has an associated OPN configuration file that defines how the parsing is accomplished and how data will be displayed. Whenever you are loading data from a supported text log (.log file) into Message Analyzer through a Data Retrieval Session, you can specify a predefined configuration file that will parse the data by selecting it from the **Text Log Configuration** drop-down list below the toolbar on the **Files** tab. Thereafter, the OPN definition for the log file parses the data and Message Analyzer loads it into the **Analysis Grid** viewer. However, you may have a text log with a unique or proprietary format that Message Analyzer does not support by default. If this is the case, you have the option to create your own OPN configuration file that will parse your log.

In order to create an OPN configuration file, you will need to develop message definitions for the log file entities for which you will display columns of data in the Message Analyzer **Analysis Grid** viewer. To create your message definitions, you will use Open Protocol Notation (OPN) and Regex notation. This document explains how to create a configuration file by using these technologies. However, before you dig into the development of a configuration file, you have the option to see firsthand how Message Analyzer handles data from a custom text log, so that you have an idea upfront of the kind of results you should pursue. To do this, you can deploy the configuration file and associated log file that are provided in this document as samples. Thereafter, you can load the log data by parsing it with the configuration file and displaying the results in Message Analyzer.

**More Information**

**To learn more** about loading data into Message Analyzer from the sample text log that is parsed by the sample OPN configuration file, see [Rendering Data From the ExampleTextLog.log File](#_Rendering_ExampleTextLog.log).

**To learn more** about OPN, see the [OPN Programming Guide](http://download.microsoft.com/download/3/E/8/3E845130-349C-4EFC-B634-C7DBD46140B7/OPN%20Programming%20Guide%20v0.4.3.docx).

## Reference Examples

The following attached **text log** and OPN configuration file are provided as examples that are referenced in this document:

 

# Rendering Data From the ExampleTextLog.log File

To see how the data in the ***ExampleTextLog.log*** file is rendered in **Message Analyzer,** perform the following steps:

1. Place the configuration file named “*ExampleTextLog.config*” in the following folder:

%LocalAppData%\Microsoft\MessageAnalyzer\OpnAndConfiguration\TextLogConfiguration\DevicesAndLogs\

1. Save the *ExampleTextLog.log* file in a designated location such as the following:

c:\users\%username%\Documents\MessageAnalyzer\Traces\

1. Next, open this log file from the Message Analyzer **File** menu by clicking **Quick Open** to display the **New Session** dialog.
2. From the **Text Log Configuration** drop-down list on the toolbar of the **Files** tab of the **New Session** dialog, select the **ExampleTextLog** configuration file and then click the **Start** button.

The data from the ExampleTextLog.log file should display in the **Analysis Grid** viewer.

1. In the **Field Chooser Tool Window**, navigate to the **ExampleTextLog** node, as shown in Figure 1, expand a subnode such as **EventHeader** or **SummaryText,** and add one or more message fields as columns to the **Analysis Grid**, by right-clicking the field and selecting **Add as Column** from the context menu. For example, you could add the **EventId** field, **Level** field, and others as columns.

**Note**: If **Field Chooser** is not displayed, select it from the **Tool Windows** drop-down list on the Ribbon of the Message Analyzer **Home** tab to display it in the default location.



**Figure 1 Field Chooser Tool Window**

1. In the **Analysis Grid** viewer columns that you specified in **Field Chooser**, confirm that Message Analyzer parsed the text log messages by observing the correct field values in the associated columns.

After you create an OPN configuration file for your log, follow the previously specified procedure to place your configuration and log files in the appropriate locations and then load the log data into Message Analyzer. After you display the data columns for the fields of interest in the **Analysis Grid** viewer, you should compare those field values with the actual data specified in your log file to verify the accuracy of your message definitions.

In the sections that follow, you will learn how to create your message definitions.

# Specifying Message Definitions

## LogEntry

The LogEntry definition is used to describe what is common across all lines of log file data.

### LogEntry Definition

All messages that you define inherit from LogEntry, as indicated by the following definition:

// A base message for all log messages:

 message LogEntry { }

**Note**: You can have an arbitrary number of derived messages in your inheritance chain based on LogEntry.

### LogEntry Example

The following message definition inherits from the base message LogEntry. As an example, this particular definition is specifying the common elements for all lines of data to be displayed in Message Analyzer, based on fields in the sample text log attached to this document. All subsequent message definitions inherit from this one:

// LogEntry is the message definition that is common to all messages:

 messageEventHeader with

 EntryInfo { Regex =

@"(?<TS>[-\d\w:\.]+),?\s\*(?<EventId>\d+),?\s\*(?<Code1>\d+),?\s\*(?<Level>[a-zA-

 Z]+),?\s\*(?<EventCode>0x[\da-fA-F]+)"} **:** LogEntry

 {

 DateTimeTS with EntryFieldInfo { IsTimestamp = true };

 uint EventId;

 ushort Code1;

 string Level where ValidationCheck(value == null ? true : !(value == "Warning" ||

 value == "Error"), this, "Error level is " + value);

 uint EventCode with EntryFieldInfo{Base = NumericBase.Hexadecimal};

 }

**Note**: When your text log fields contain different number bases, you can use the following NumericBase enum values to specify them; the default value is Decimal:

Decimal = 0,

Hexadecimal = 1,

Octal = 2,

Binary = 3

**More Information**

**To learn more** about the required EntryInfo aspect in the above example message definition, see the [EntryInfo Aspect](#_EntryInfo) section.

**To learn more** about the EntryFieldInfo aspect, see the [EntryFieldInfo](#_EntryFieldInfo) section.

## EntryInfo **Aspect**

Each message definition must include an EntryInfo aspect. If a message **definition** does not have an EntryInfo aspect defined, that message **definition** is ignored by **Message Analyzer**.

**Note**: An aspect in OPN is similar in purpose to an attribute in the C# language.

### EntryInfo Aspect Definition

The EntryInfoaspect is defined as follows:

// The EntryInfo aspect signifies that a given message declaration

// should be used for log parsing:

 aspect EntryInfo

 {

// A regular expression for parsing a log entry into an OPN message.

// This is a mandatory field that locates and assigns log entry values

// to variables:

 string Regex;

// A tag which uniquely identifies whether a log entry should be parsed into

// a given OPN message. Although it is an optional field, it is recommended

// to have it, as it significantly increases parsing performance:

 string Tag;

// A function delegate that provides a way to parse multiline log

// entries. If you specify the optional Multiline field, the configuration

// should keep passing log lines one by one until the function returns false:

 bool(string, LineContext) Multiline;

// The following is an example of the signature format of the called function

// for Multiline:

 bool HandleMultilineConfig(string line, LineContext context) { ... }

// The priority of an entry helps resolve any overlapping regular

// expressions among entries (when a field matches more than one Regex definition).

// A lower value defines a higher priority. The default value is 0xFFFFFFFF:

 uint Priority;

 }

The following describes the EntryInfoaspect fields:

* **EntryInfo.Regex** – This is a required field specifying a regular expression with capture variables. Names of capture variables must map one-to-one to the message fields (comparison is case sensitive). Based on a field type, **Message Analyzer** converts a sequence of characters captured by a variable into a corresponding type and assigns it to the field. Message fields assigned through Regex capture variables can be primitive types only, such as int, guid, string, and so on. They cannot be of collection types (arrays, sets, maps) or composite types. If this restriction is violated, then a corresponding field will not be populated with any data and its message will get a parser error attached at runtime with an appropriate error message.

**More Information**

**To learn more** about regular expressions, see the[Regular Expression Language Quick Reference](http://msdn.microsoft.com/en-us/library/hs600312.aspx) on MSDN.

**To learn more** about a tool that you can use to test regular expressions, seethe [.Net Regular Expression Tester](http://derekslager.com/blog/posts/2007/09/a-better-dotnet-regular-expression-tester.ashx). For an example test, see [Testing Regular Expressions](#_Testing_Regular_Expressions) in this document.

* **EntryInfo.Tag** – is an optional field. It contains a string type that uniquely identifies text log entries for the corresponding **OPN** message. For example, if you specify the Tag field in your configuration file with a particular string value and that value is found in a log file entry, then that log entry should be parsed into a message. If Tag is not specified, then the Regex field is used to figure out whether a log entry should be parsed into a message (see [Mapping a Message Definition to a Log File Line](#_How_a_Configuration) for more information).
* **EntryInfo.Multiline** – is an optional field that you can specify to handle logs which contain entries that use multiple lines. It should be [assigned to a function](#Section3_2_2function_assignment_example) that returns true while valid lines of log data are being evaluated in multiline entries, in which case, their string values will not be zero-length or null. When a log line evaluates to a zero-length string or null, the function should return false to signify the end of a particular multiline log entry. A boolean value that signifies that a multiline log entry is being processed, in addition to the means to maintain state is provided by the type LineContext, the definition for which consists of the following:

// The context of a log line is used to signify custom parsing of a multiline log

// entry:

 type LineContext

 {

// Indicates that a multiline handler callback was invoked for the log line

// that triggered multiline parsing:

 bool IsInitialLine;

// An optional collection of key/value pairs that remembers state information

// that could be used to process multiline log entries. For example, you might use a

// counter to maintain the number of lines consumed in multiline parsing, or the code

// could remember previously identified entities or segments of log lines:

 map<string, any> Data;

// This field is reserved for internal use and should be not accessed:

 any Reserved;

 }

**Note**: If you have a text log with entries that consist of multiple lines and you do not specify the Multiline delegate in EntryInfo, the parsing engine might treat each line of the log as a separate message or ignore them altogether.

* **EntryInfo.Priority** – is an optional field that you can use to establish the order in which message definitions are processed, which can minimize any issues that might occur with overlapping Regex configuration entries. For example, a single string in a log file might match multiple regular expressions, therefore, you can specify Priority field values in your message definitions to determine the most relevant definition to use when parsing a log file line. Note that the lower the value of Priority, the higher the processing priority is. For an example of how to use the Priority field, see [EntryInfo Example](#_EntryInfo_Example).

**More Information**

**To learn more** about using the Priority field, see [Mapping a Message Definition to a Log File Line](#_How_a_Configuration).

### EntryInfo Example

The following example shows how to use EntryInfo.Tag, EntryInfo.Regex, and DisplayInfo in a message definition that is included in the sample configuration file attached to this document:

message TransitionStateChangedEvent with EntryInfo

{ Tag = "TransitionStateChangedEvent", Regex = @"TransitionStateChangedEvent,\s\*FirstField:\s\*(?<FirstField>[\d\w\_]\*?),\s\*SecondField:\s\*(?<SecondField>[\d\w\_]\*?),\s\*ComputerID:\s\*(?<ComputerID>[\d\w]{8}-[\d\w]{4}-[\d\w]{4}-[\d\w]{4}-[\d\w]{12}),\s\*SequenceID:\s\*(?<SequenceID>[\d\w]{8}-[\d\w]{4}-[\d\w]{4}-[\d\w]{4}-[\d\w]{12}),\s\*OrigTS:\s\*(?<OrigTS>.\*)", Priority = 1 },

DisplayInfo { ToText = TransitionStateChangedEventToText } : EventHeader

{

// Field values are assigned by Regex:

 string FirstField;

 string SecondField;

 guid ComputerID;

 guid SequenceID;

 string OrigTS;

 static string TransitionStateChangedEventToText(any data)

 {

 var e = data as TransitionStateChangedEvent;

 return "Example of " + e.FirstField + " different " + e.SecondField;

 }

}

### Multiline Log Entries Example

The example in this section shows how to enhance EntryInfo for text logs with multi-line entries by calling a multiline handler function. The associated delegate declaration/assignment and the handler function signatures are given below:

// Format for declaring an entry that assigns a multiline handler:

 Message M with EntryInfo { Multiline = HandleMultiline } { ... }

// The associated function signature would be similar to the following:

 Bool HandleMultiline(string line, LineContext context) { ... }

The following provides an implementation example of the Multiline delegate function, as included in the sample OPN configuration file that is attached to this document:

message MultilineEvent with EntryInfo

 { Tag = "Multiline", Regex = @"(?<Content>Multiline,.\*)",

 Multiline = HandleMultiLine, Priority = 1 }: EventHeader

 {

 string Content;

 bool HandleMultiLine(string line, LineContext context)

 {

 if (context.IsInitialLine)

 {

 // The initial log line with the entity “Multiline” is already matched by

 // the regular expression and assigned to the “Content” string variable,

 // therefore return true to indicate that there are more lines to process

 // for a multiline log entry:

 return true;

 }

 else

 {

 // The value of the ‘line’ starting with "\_" is considered the continuation

 // of the current message (see ExampleTextLog.log):

 if (line.Count > 0 && line[0] == '\_')

 {

 // Remove the underscore and concatenate the rest of the current log

 // line, and do this for each call until no lines are left in the

 // multiline entry:

 Content += line.Segment(1);

 return true;

 }

 else

 {

 // End of message:

 return false;

 }

 }

 }

 override string ToString()

 {

 // Return results for this message:

 return "Multiline example: " + Content;

 }

}

## EntryFieldInfo Aspect

A message definition can include an EntryFieldInfo aspect that enables you to represent additional fields of information in a text log.

### EntryFieldInfo Aspect Definition

The EntryFieldInfo aspect is defined as follows:

// The EntryFieldInfo aspect enables you to provide additional information in your

// message definitions for fields in a log file:

 aspect EntryFieldInfo

 {

// Indicates whether a particular field contains a message timestamp.

// This flag can be set to true for a single message field of type DateTime only.

// Note: this flag can be overridden anywhere in the inheritance chain of derived

// message types:

 bool IsTimestamp;

// Indicates whether a timestamp field contains local or UTC time.

// The default value is false (UTC time). This flag can be used only for fields

// that are marked as timestamps:

 bool IsLocalTime;

// Indicates the numeric base of a value, which can be Decimal, Hexadecimal, Octal,

// or Binary. The default is Decimal if this member is unspecified:

 NumericBase Base;

 }

### EntryFieldInfo Example

An example of how to use the EntryFieldInfo aspect is shown in the field declarations section of [LogEntry Example](#_LogEntry_Example).

## DisplayInfo Aspect

A message definition can include a DisplayInfo aspect that enables you to customize the display of information in Message Analyzer **Analysis Grid** data columns after you load your log data.

### DisplayInfo Aspect Definition

The DisplayInfo aspect is defined as follows:

 aspect DisplayInfo

 {

// Defines the name of a function that is responsible for providing a textual

// representation of the entity that this aspect is attached to:

 optional string ToText;

 }

As a member of DisplayInfo, you can assign ToText to a function that returns custom message text that you want to display in the **Analysis Grid** data columns. When Message Analyzer is ready to display your message, it simply calls the ToString() method to provide your custom string value, that is, if ToText was specified.

**More Information**

**To learn more** about the DisplayInfo aspect, see section 5.16.4 of the [OPN Programming Guide](http://download.microsoft.com/download/3/E/8/3E845130-349C-4EFC-B634-C7DBD46140B7/OPN%20Programming%20Guide%20v0.4.3.docx).

### DisplayInfo Example

The message definition specified in [EntryInfo Example](#_EntryInfo_Example) shows how DisplayInfo calls the function TransitionStateChangedEventToText.

# Mapping a Message Definition to a Log File Line

The first criteria for evaluating and matching log line data is based on the [Priority](#PriorityInteger_EntryInfo) values that you specify in your message definition entries. Entries with a lower Priority value are evaluated first, as these are of the highest priority. If Priority is unspecified, an entry is considered to be of the lowest priority. Prior to checking message definitions with the algorithm defined in section 4.1, the message definitions are sorted by their Priority values, from the highest to lowest, to ensure that the foreach algorithm evaluates message definitions in prioritized order.

## Definition Evaluation Algorithm

The following pseudo code expresses the evaluation algorithm that is used to match a **configuration file message definition** to a line of log data:

foreach message definition

if EntryInfo.Tag is defined and the current log line contains its value, then use RegEx in this message definition to parse this log line;

else if EntryInfo.Tag is not defined and the log line matches EntryInfo.Regex, then use this message definition to parse this log line;

else, try the next message definition;

## Setting the Priority Level

It is important to consider how you set the Priority value so that the appropriate message definition in your configuration file is used for parsing a particular line of data in a log file. If a message definition specifies a certain Priority value, and the message meets the criteria of either the if or else if statements in the above algorithm, then that message is the only one that will be used to parse the line of log data, even if a particular string in the log file matches more than one regular expression statement in the configuration file.

For example, if the line of data in the log contains the value specified by EntryInfo.Tag in the message definition currently under evaluation, then the first if statement that checks for EntryInfo.Tag exclusively determines that the current message definition will be used to parse the log line. Thereafter, the EntryInfo.Regex statements in that message definition are used to locate the log file data entities, capture their values, and assign them to fields that the message definition declares. As in the else if statement of the above algorithm, if EntryInfo.Tag is not defined but the EntryInfo.Regex statement in the message definition currently under evaluation finds a match in the log file line, then EntryInfo.Regex exclusively determines that the current message definition will be used for the remaining processing. If EntryInfo.Regex does not find a match, then the next *prioritized* message definition is tested by the same algorithm criteria. Therefore, because of the evaluation logic provided by the algorithm, you can specify appropriate values for the Priority member in your message definitions to avoid conflicts between Regex matches.

# Handling Unknown Log Entries

Sometimes **Message Analyzer** does not know how to parse certain log entries. This can occur when a **configuration file** does not have a **message definition** capable of parsing a log entry with the given format or because the definition is incorrect. In these cases, the message will be dropped.

# Placing Configuration Files in the Default Location

Configuration files reside in the following location:

**%LocalAppData%\Microsoft\MessageAnalyzer\OpnAndConfiguration\TextLogConfiguration\DevicesAndLogs\**

You must place any configuration file that you create in this location. This makes it possible for Message Analyzer to locate your configuration file and populate it to the **Text Log Configuration** drop-down list from where you can select it in Data Retrieval Session configuration. If you do not place your configuration file in this location, Message Analyzer will be unable to parse your text log as intended, although the log data will display in a more simplified format.

# Modifying Configuration Files

Compilation of all OPN **configuration files** occurs whenever you start **Message Analyzer**. Therefore, if you modify a **configuration file**, place the modified file in the location specified in the previous section and then restart **Message Analyzer** to recompile your OPN configuration source. This will ensure that the changes you made will take effect.

# Testing Regular Expressions

The following is an example of how you can test the validity of a regular expression. On the [.Net Regular Expression Tester](http://derekslager.com/blog/posts/2007/09/a-better-dotnet-regular-expression-tester.ashx) site, enter the following hypothetical log file line in the **Source** text box of the **Input** pane:

2011-08-11T20:19:10.140, 4277275, 067, Information, 0x00002201, Event: TransitionEvent, FirstField: substituting, SecondField: fields, ComputerID: 03223777-0000-0000-0000-0015177d99d4, SequenceID: 03223928-0000-0000-0000-0015177d98f0, OrigTS: 8/11/2011 8:19:10 PM

Next, enter the following regular expression (no quotes) in the **Pattern** text box of the **Input** pane to parse the first three fields (FirstField, SecondField, and ComputerID) and display the parsed results in the format <*fieldvalue*> (FieldName):

TransitionEvent,\s\*FirstField:\s\*(?<FirstField>[\d\w\_]\*?),\s\*SecondField:\s\*(?<SecondField>[\d\w\_]\*?),\s\*ComputerID:\s\*(?<ComputerID>[\d\w]{8}-[\d\w]{4}-[\d\w]{4}-[\d\w]{4}-[\d\w]{12})

A successful match might look similar to the following, as displayed in the **Result** pane on the site:

*Found 1 match:*

1.TransitionEvent, FirstField: substituting, SecondField: fields, ComputerID: 03223777-0000-0000-0000-0015177d99d4 has 3 groups:

1. substituting (FirstField)

2. fields (SecondField)

3. 03223777-0000-0000-0000-0015177d99d4 (ComputerID)

*String literals for use in programs:*

C#

@"TransitionStateChangedEvent,\s\*FirstField:\s\*(?<FirstField>[\d\w\_]\*?),\s\*SecondField:\s\*(?<SecondField>[\d\w\_]\*?),\s\*ComputerID:\s\*(?<ComputerID>[\d\w]{8}-[\d\w]{4}-[\d\w]{4}-[\d\w]{4}-[\d\w]{12})"