Portrait
Foundation

Logging Standards Guide

Edition 6.3

11 January 2013

Pitney Bowes
Software

Portrait Foundation
Logging Standards Guide

Acknowledgement of trademarks

About Portrait Software

Portrait Software is now part of Pitney Bowes Software Inc.

Portrait Software enables organizations to engage with each of their customers as individuals, resulting in improved customer profitability, increased retention, reduced risk, and outstanding customer experiences. This is achieved through a suite of innovative, insight-driven applications which empower organizations to create enduring one-to-one relationships with their customers.

Portrait Software was acquired in July 2010 by Pitney Bowes to build on the broad range of capabilities at Pitney Bowes Software for helping organizations acquire, serve and grow their customer relationships more effectively. The Portrait Customer Interaction Suite combines world leading customer analytics, powerful inbound and outbound campaign management, and best-in-class business process integration to deliver real-time customer interactions that communicate precisely the right message through the right channel, at the right time.

Our 300 + customers include industry-leading organizations in customer-intensive sectors. They include 3, AAA, Bank of Tokyo Mitsubishi, Dell, Fiserv Bank Solutions, Lloyds Banking Group, Merrill Lynch, Nationwide Building Society, RACQ, RAC WA, Telenor, Tesco Bank, T-Mobile, Tryg and US Bank.

Pitney Bowes Software Inc. is a division of Pitney Bowes Inc. (NYSE: PBI).

For more information please visit: http://www.pitneybowes.co.uk/software/

UK

Portrait Software
The Smith Centre
The Fairmile
Henley-on-Thames
Oxfordshire, RG9 6AB, UK

Email: support@portraitsoftware.com
Tel: +44 (0)1491 416778
Fax: +44 (0)1491 416601

America

Portrait Software
125 Summer Street
16th Floor
Boston, MA 02110
USA

Email: support@portraitsoftware.com
Tel: +1 617 457 5200
Fax: +1 617 457 5299

Norway

Portrait Software
Portrait Million Handshakes AS
Maridalsveien. 87
0461 Oslo
Norway

Email: support@portraitsoftware.com
Tel: +47 22 38 91 00
Fax: +47 23 40 94 99

About this document

Purpose of document

This document describes the features, standards, usage, and extensibility options for the Logging component of a system.

Intended audience

Those who wish to understand how to use, configure or extend Logging.

Related documents

Technical Architecture

Software release

Portrait Foundation 4.4 or later.

Contents

1 Introduction

The following sections describe:

- how Portrait Foundation logging is used by consumers and creators of log messages.
- features in terms of destinations, sources, catalogues, viewers, DLLs and controls.
- standards for use of logging
- how logging can be extended
- how logging can localised through message catalogues
- how logging is controlled within a system

2 Operation

Logging uses its registry based configuration to determines at runtime whether a call from the code to one of the `Log()` or `Trace()` methods will actually cause a message to be passed on to a logging destination.

Logging destinations are recorded under the `AIT\AMC\Logging\Destinations` registry key and include (at least) the following values:

Enabled – any non zero value indicates that this destination is enabled

Type – the type of destination e.g. Database, File, Debug

DLL – the DLL where the code for this destination type resides

Filter – the mechanism for selecting which messages get passed to this destination

Config – the type of property page that is used in the Portrait Management Console to configure this destination

Other properties are specific to individual destination types (e.g. the COM destination type requires the value 'Progid' naming the COM object to use).

Logging is regarded as being enabled if the registry key `AIT\AMC\Logging\Destinations` has a value `Enabled` that is non zero. This key is set by the Portrait Management Console according to whether any destination is enabled.

When logging is disabled all calls to logging return quickly as it is easy to determine that there is no processing necessary.

If logging is enabled then each log message must be checked to see if any enabled destination is configured to accept the specific log message. This process is longer than the disabled processing as comparisons between the log messages' attributes and destinations' filters needs to be made.

When configuring destinations, it is possible:

- to configure any number of destinations;
- to have many destinations of the same type;
- for many destinations to receive an individual log message.

3 Configuration

The Portrait Management Console performs configuration of logging settings. The help for this application describes how to capture configuration settings for individual destinations.

Logging settings are maintained in the SysConfig tables in the Portrait Foundation database and are written to the registry of a specific machine whenever SysConfig starts on that machine. Machines that do not have SysConfig running on them rely on the registry being setup manually or via the Portrait Management Console writing directly to the registry.

3.1 Filters

Each destination has a filter expression associated with it. A log message will be passed to the destination if the message passes the criteria expressed in the filter. The log message can be filtered on four criteria:

Message Type

Which has the following values:

- Trace – 1
- Log – 2

Severity

Which has the following values:

- Info – 0
- None – 1
- Warning – 2
- Error – 3
- Severe – 4
- Fatal – 5

Facility

These are defined in the `LogFacility.h` file in the SDK. This Facility should not be confused with the HRESULT Facility bits in the HRESULT.

Category

These are defined in the `LogCategory.h` file in the SDK.

Filter Expression

A filter expression can contain any number of filters, each specified as per the following:

A single filter is specified as:

```
[ MessageType Criteria : Severity Criteria : Facility Criteria : Category Criteria ]
```

All Criteria (for example the MessageType Criteria) are expressed as a list of numbers enclosed within brackets. If all values should pass the criteria a single value of '*' can be used instead of a list.

If the values within the brackets are to be excluded by the filter then the open bracket should be preceded by '!'. Some examples are:

`!(*)` – Exclude all values

`(1,3)` – Include values 1 and 3

`(*)` – Include all values

Filter mechanics

The Criteria specifications within an individual filter are **AND**ed together. A message must match the Message Criteria ***and*** the Severity Criteria ***and*** the Facility Criteria ***and*** the Category Criteria in order to pass the filter.

[(2):!(0,1):(*):(*)]

Will only pass Log (2) messages that are not of severity Information (0) and are not severity None (1).

Each filter within a filter specification is regarded as a being **Or**ed together. A message will be logged if it passes any of the filters. E.g.

[!(1,2):(*):(*):(*)] [(*):(*):(*):(*)]

Will log everything because the second filter is always passed regardless of whether the first filter is passed.

The Portrait LogViewer has a Filter Tool that can be used to parse and construct filter expressions. This provides the ability to generate the filter string by selecting the 4 parts of the filter by symbolic name rather than by numbers.

3.2 Updates

Updates to the configuration are reflected immediately in the processes on the machine in which the changes have been performed. Note that it is possible to change the configuration with the Portrait Management Console and NOT elect for the changes to be written to the registry directly.

4 Message Catalogues

4.1 Messages

Any `Log()` call that uses a severity other than `AMCNONE` could be potentially sent to the Eventlog destination. These messages must have an entry in a message catalogue that provides the text and formatting for the message.

The catalogue lookup is performed on the HRESULT code passed into logging. As `Trace()` calls do not have an HRESULT parameter these are assigned a default id and share a generic 'the following was traced' message text. However, it is questionable as to whether `Trace()` calls should be sent to the Eventlog. The default message catalogue is `BaseMessagesU.dll`; this catalogue contains the HRESULTS defined for the base Portrait Foundation code.

It is possible to define different message catalogue to be used with different COM Facilities.

The message catalogue to be used is determined by the HRESULT Facility code of the message logged as described below:

```
//  The HRESULT is in this form:
//
//  Values are 32 bit values layed out as follows:
//
//   3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
//   1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
//  +---+-+-+-----------------------+-------------------------------+
//  |Sev|C|R|     Facility          |               Code            |
//  +---+-+-+-----------------------+-------------------------------+
//
//  where
//
//      Sev - is the severity code
//
//          00 - Success
//          01 - Informational
//          10 - Warning
//          11 - Error
//
//      C - is the Customer code flag
//
//      R - is a reserved bit
//
//      Facility - is the facility code
//
//      Code - is the facility's status code
//
```

The facility code is looked up in the `AIT\AMC\Logging\Facilities` registry key. If the facility number is specified under this key it is used. If not, the key `Default` is used. The value of `Source` specified in the selected key is used in the `::ReportEvent()` API call as the `Source` parameter to identify the resource DLL containing the message text.

The `Default` entry specifies a source of `PortraitBase`. The following Windows Eventlog registry settings provides the connection to the desired message catalogue:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Eventlog\Application\PortraitBase
```

4.2 Log statements

A log statement is typically of the form:

```
return CamcLogger::Log( AMC_LOGCAT_LOGGING,
    AMC_LOGGING_LOCATION,
    AMCERROR,
    E_LOGGING_ACTION_FAILED,
    "Action '%s' failed HRESULT = %8.8x",
    strAction,
    hr );
```

This would be matched with the following message in the catalogue. Note that matching is done on the Severity, Message Id, and HRESULT Facility parts of the HRESULT.

```
;//
;// Messsage:   800a0067 E_LOGGING_ACTION_FAILED
MessageId=103
Severity=Warning
Facility=Logging
SymbolicName=E_LOGGING_ACTION_FAILED
Language=English
Message:'Logging action '%20' failed with the HRESULT %21'%r
AdditionalInfo:'%13'%r
HRESULT:%12, Source:%9(%10), Facility:0x%1s, Type:%2, Category:%3, Severity:%4,
MachineName:'%5', Process:%6, Thread:%7, Time:'%11', ActivityToken:'%8'%r
Data is the serialised LogMessage
.
;MESSAGE_SHORT_TEXT( E_LOGGING_ACTION_FAILED, L"Logging action failed" )
;//
;//************************************************
```

It is important to note:

- The text logged in the Windows Eventlog will be the whole text from below '`Language=English`' to the trailing full stop.
- The content of this text is entirely configurable in the message catalogue source and may bear no resemblance to the format string in the logging call (for reasons of internationalisation).
- Do not use Severity=Error , use Warning (e.g. E_FAIL is of type Warning) and Informational.
- The parameters to the message text (%20 and %21 in this case) are the parameters passed into the above `Log()` call e.g. `strAction` and `hr`.
- The message text that appears in 'simple' viewers (e.g. Debug View) will appear as Additional Info (parameter %13)
- The `MESSAGE_SHORT_TEXT macro` provides a simple definition of the message for error lookup utilities e.g. Log Viewer
- The message text can use any (or none) of the attributes of the message in any order. These are parameters %1 to %19.
- The message text can use any (or none) of the arguments to the format string (e.g. `strAction` and `hr` above) These are parameters %20 and above.
- The %parameters are defined as follows:

Table 1 – Message parameter values

Parameter	Value
%1	Facility
%2	Message Type
%3	Severity
%4	Category
%5	Machine Name
%6	Process Id
%7	Thread Id
%8	Activity Token
%9	File Name
%10	Line Number
%11	Time Stamp
%12	HRESULT
%13	Additional Info
%14 to %19	Reserved
%20	First argument for the format string (or format string if there are no arguments)
%21	Second argument for the format string
%22 and above	2nd and above argument for the format string

4.3 More info

Details of use of the message compiler can be found by searching for "Message Files" under
`Platform SDK: Debugging and Error Handling`

5 Destinations

The following destination types, which are implemented in `LogLibu[d].dll`, are available:

Eventlog

Log messages are formatted and presented to the Windows Eventlog. The text for the message is extracted from the appropriate message catalogue. This provides the ability to have messages and their parameters presented in different languages or using alternative texts.

Debug

This destination calls `::OutputDebugString()` with a simply formatted textual representation of the Log message. Little interpretation or expansion of the message parameters is done for reasons of speed.

Database

This destination writes the table `amc_log_message` in the currently configured database. This destination was developed for use with the Model Diagnosis Tool (MDT).

Pipe

This destination will send log messages to a Named Pipe. The Log Viewer application is capable of reading a specific named pipe as its source of data. This application is designed to provide:

File

This destination will send messages to a file in Portrait Foundation log file format.

MDT Interactive

This destination will send messages to the Portrait Model Diagnosis Tool application via the Portrait Log Message Concentrator.

MDT Recorded

This destination will send messages to a the database for later interrogation by the Portrait Model Diagnosis Tool

COM

This destination can be configured to load a named COM object and to pass on messages to this destination. This is the mechanism for using destinations developed in COM.

Wraparound and BlackBox

These specialised destinations are not supported via the Portrait Management Console but are diagnostic aids that can be configured manually in the registry.

Wraparound provides a very fast and low impact in memory logging that wraps around after a configured number of entries have been written. This information can be interrogated by analysing memory dumps or by attaching WinDBG.

BlackBox provides the ability to have log messages emitted only if a failure occurs during processing on the Process Server.

Please contact Portrait Support for further details.

6 Message Suppression

After a release has been made it is not possible to change code to suppress spurious logging. The stated aim is to only have logging written to the Eventlog in live systems that are actual errors that need actioning. To this end it is now possible to create a list of source files and line numbers that are the source of logging that is to be ignored. A file containing these file and line numbers can be converted by support into a .plx file that can be installed into "PST\Portrait\Common\Config\LogFilter.plx". Once installed this file will suppress logging from the identified files. Generation of the .plx file is a support responsibility because control must be exercise over what messages are suppressed.

7 LogListControl

The rendering of Log events in an application can be performed by this control (`AIT.AMC.Logging.LogListControl.1`) . The MDT and the Log Viewer use this control in their user interface. The control only needs wiring up to a Log Source in order to render the messages that the source provides.

8 Sources

In order to allow 'consumers' of log messages to be abstracted from the destination that has captured them, the concept of a Log Source has been devised.

A Log Source is characterised by implementing the interface `IamcLogSource`. These are packaged as COM objects and are used by consumers of log messages. Log Sources 'advise' their consumers of log messages via the outbound interface `IamcLogDataEventSink` which only has one method `OnNewData()`

For example the LogListControl accepts input from Log Source; in the case of the MDT, its embedded LogListControl uses the Database Log Source. In the case of the Log Viewer its embedded LogListControl can have its source set as any of the source types Pipe Source, Database Source, COM source, or File Source.

The current Log Sources, which are implemented in `LogLibu[d].dll`, are:

Pipe Source will read messages to a Pipe destination from a named pipe

COM Source will read messages to a COM object

Database source will read messages from amc_log_message in the Portrait Foundation database sent there by the Database destination

File source will read raw log messages from a disc file. The Log Viewer file\open processing uses this

8.1.1 Message Collection

As a buffer between a source and the consumer the COM object `AIT.AMC.Logging.MessageCollection.1` has been created. Its purpose is to collect messages from a source and buffer them for replaying or direct access from clients.

Log Sources typically push a message to their consumer when it arrives and have no 'memory' of any messages that have passed though them. A message collection interposed between a consumer and a source will add the ability to randomly access previously received messages.

This object supports `IamcLogSource` (because it is a source of log messages to a consumer) AND `IamcLogDataEventSink` (because it accepts messages from the source it is buffering).

9 Viewer

This application is designed as a different approach to viewing log messages to applications such as Debug View. It has the following features.

Filters

The application collects messages and allows a number of filters and highlighting to be applied to the data that is viewed. As these can be applied and removed it is possible to view the same collection of messages in many different ways.

Expanded Definitions

The viewer provides textual representations of the following fields:

- HRESULT (both AIT and Microsoft defined)
- Severity
- Category
- Facility
- Type

OutputDebugString

The application can capture data sent via `::OutputDebugString()` as per Debug View. This allows these messages to be recorded along with messages sent to this destination. It is a valid mode of operation to not send data to the Viewer via a 'Viewer' type destination but via a 'Debug' type destination.

Error Lookup

The application contains an error lookup feature that will retrieve the textual representation of HRESULT (both AIT and Microsoft defined) and `::GetLastError()` codes.

Log Filter Parse / Construct

Log filters can be parsed into their symbolic names from the encoded form required by the Portrait Management Console. Filters can be constructed from their symbolic names also.

Remote Viewing

The viewer can be configured to receive log messages from any number of different machines.

Speed

The viewer can accept messages over twice as fast Debug View.

Save/Load

It can save and load raw Log Messages from Portrait Foundation log files (`.plx`) and can open System Event Log files (.evt & .xml). Extended (.evtx) files cannot be loaded, for later operating systems please select save as Xml File from the Windows Event Viewer.

10 Usage Standards

10.1 Which call

The following is a guide to how to determine which logging call should be used in particular circumstances. The calls to `Log()` and `Trace()` below do not show the full signature of the method calls but emphasise the elements of the call that should be present.

Use the next section 'Choose a call' to determine the appropriate call to make. If your particular requirements match a statement then use the logging call that is recommended after that statement. Use the first statement that matches; don't attempt to match your requirements to subsequent statements. Indentation in the text implies dependency on the preceding condition at a lower level of indentation.

10.1.1 Choose a call

If you are being called during Logging's internal processing and want to record an error then use the following. There are very few places when this call is appropriate; the most likely case is in the code for a log destination. As this call does not get filtered, it is not suitable for `Trace()` activity.

```
CamcLogger::EVTLog()
```

or

```
CamcLogger::ODSLog()
```

`ODSLog` passes the call on to `::OutputDebugString()` and `EVTLog` records the message in the Application Event Log.

If are you returning a non zero HRESULT to your caller then:

Please note that you should not normally be returning an HRESULT that you have received to your caller. You should be returning HRESULTS that your caller could reasonably have expected to be part of the contract with the called component. Using E_FAIL is also deprecated because it does not pass on any information other than a binary 'something went wrong' and it also cannot be logged in a meaningful way to the Eventlog.

If you know the severity of the failure then use the following. However, it is often not possible to determine the impact of a failure on your caller e.g. failure to open a file can be catastrophic or merely require a dialog box. Note that any relevant information not available to the caller should be included in the format string:

```
return CamcLogger::Log( AMCWARNING | AMCERROR | etc , E_XYZ, … )
```

If you have any relevant information that is not available to your caller about the nature of the error (`::GetLastError()` codes etc) then use:

```
return CamcLogger::Log( AMCNONE, E_XYZ, "Action failed GetLastError() = %d", nErr )
```

If this code is likely to called infrequently (it is safest to assume it is not when in doubt) and you regard a log message as useful to problem determination (bear in mind that you have no extra relevant information or you would have stopped at the above test) use:

```
return CamcLogger::Log( AMCNONE, E_XYZ, "Action failed" )
```

When none of the above applies, then it will be the responsibility of the caller to decide whether to record your 'failure' and/or act upon it. In these circumstances use:

```
return E_XYZ;
```

If are you trying to record a success which may be recorded in the Eventlog of a system in production then use:

```
CamcLogger::Log( AMCINFO, S_SERVICEHOST_STARTED,… );
```

In all other circumstances, it is likely that the call to use is:

```
CamcLogger::Trace(…)
```

10.2 Deprecated calls

10.2.1 Log ()

The two calls below are calls to `Log()` style functions that do not allow an HRESULT to be passed in. This prohibits them from being used in accordance with the current standards (the "Which call" analysis above will never result in usage of this style of call).

These calls should no longer be used.

```
Static void Log( long lCategory, LPCWSTR szFilename, long lLineNumber,
        BSTR ActivityToken, Severity severity, LPCWSTR format, ...);
static void Log( long lCategory, LPCWSTR szFilename, long lLineNumber,
    Severity severity, LPCWSTR format, ...);
```

Note that, with the exception of `Log( AMCINFO )` all `Log()` calls should be setting the returned HRESULT of the method they are in, for example:

```
return CamcLogger::Log(…)
```

or

```
hr =  CamcLogger::Log(…)
```

If that is not the case then `Trace()` probably should be being used.

10.2.2 DestroyLogger()

`DestroyLogger()` need not be called any more. Logging shutdown and startup is now more process based than module based.

10.3 Dos and Don'ts

10.3.1 Please don't

Do not call `Log()` then `Trace()` for the same event

Do not use `ODS` or `::OutputDebugString()` or similar

Do not use `AMCHERE`

Do not cause C++ exceptions to be thrown (e.g. via `Check()`) during normal processing.

Do not return `E_FAIL`

Do not use `AMCWARNING` or above as the severity if you don't expect someone to take some remedial action.

Do not use the following if you don't not know the contents of a buffer.

```
CamcLogger::Log( …, strBuffer );
```

Use:

```
CamcLogger::Log( …, L"%s", strBuffer );
```

10.3.2 Please do

Do use unique message codes

Do update the appropriate message catalogue

Do use `AMCNONE` if you do not know whether the event is an error requiring action (let your caller decide)

Do include information only available to you in `Log()` calls

If you cannot think of a category, create one matching the Facility and use that

Do use `WillLog()` if your logging call is going to be expensive to construct the parameters.

11 Extending Logging

11.1 Adding a new Destination

New destinations can be added to logging by the following process. As of Manhattan destinations are not `::CoCreateInstance()` able ATL COM objects but are COM objects implemented as a C++ classes derived from `CamcLogDestination`.

For those who wish to develop a destination in a language, other than C++, that supports COM then there is a destination – `CAmcLoginggCOMDestination` that will hand off messages to a COM object. The COM object must implement an interface (`IamcLogDestination`) that is, to all intents and purposes, the same as the `CamcLogDestination` interface. The one difference is documented below.

As described below Destinations are packaged in DLLs and are located and called without COM. It is not necessary for any DLL containing destinations to be COM DLLs (e.g. they need not be registered with Regserver32 and they do not export class factories).

11.1.1 Configuration

Logging will attempt to locate destinations configured in the registry under `AIT\Amc\Logging\Destinations` with the value `Enabled` non-zero. It will load the DLL specified in the `DLL` value of the destination's key and locate the entrypoint `DllGetLogDestination()`. This entrypoint will be called to factory an instance of the destination of the type specified in the key's `Type` value.

The signature of the `DllGetLogDestination()` is

```
extern "C"
STDAPI DllGetLogDestination(
   const WCHAR* DestinationType,
   CamcLogDestination** ppDestination
)
```

This function must be added to the `.def` file for the DLL as an export e.g.

```
EXPORTS
        DllCanUnloadNow     @1 PRIVATE
        DllGetClassObject   @2 PRIVATE
        DllRegisterServer   @3 PRIVATE
        DllUnregisterServer  @4 PRIVATE
   DllGetLogDestination @5 PRIVATE
```

11.1.2 Interface

The C++ interface is defined in `Destination.h` and matches the COM interface (`IamcLogDestination`) except as identified below.

```
////////////////////////////////////////////////////////////////////////
// Function:        CamcLogDebugDestination::Initialise
// Parameters:      [i] DestinationName – Name of the LogDestination
//                                        we're being asked to implement
//                  [i] pConfig – pointer to source of configuration info
// Returns:         S_OK – if successful
// Description:     This method is called once only in the lifetime of a
//                  LogDestination and it gives the LogDestination an opportunity
//                  to perform any necessary initialisation.
////////////////////////////////////////////////////////////////////////
HRESULT CamcLogDebugDestination::Initialise(
                        const WCHAR* DestinationName,
                        CamcLoggerConfig* pConfig )
```

The Logging will call this method when the destination has been loaded because it is enabled.

The `CamcLoggerConfig` object can be used to subscribe to notifications of when the Logging registry hive changes. However please note that Logging will `Unitialise()` and `delete` then `Initialise`() a new instance of destination whenever any logging settings change effectively rendering use of the `pConfig` pointer useless.

```
////////////////////////////////////////////////////////////////////////
// Function:        CamcLogDebugDestination::UnInitialise
// Parameters:      none
// Returns:         S_OK – if successful
// Description:     This method is called once only in the lifetime of a
//                  LogDestination and it gives the LogDestination an opportunity
//                  to perform any necessary uninitialisation before being
//                  destroyed.
////////////////////////////////////////////////////////////////////////
HRESULT CamcLogDebugDestination::UnInitialise()
```

This method is called prior to delete (via reference counting).

```
////////////////////////////////////////////////////////////////////////
// Function:        CamcLogDebugDestination::Log
// Parameters:      [i] pMessage – LogMessage to be logged
// Returns:         S_OK – if successful
// Description:     This method is called whenever the LogDestination is being
//                  asked to process a message which has met its filter criteria.
////////////////////////////////////////////////////////////////////////
HRESULT CamcLogDebugDestination::Log( CamcLogMessage* pMessage )
```

This method is the opportunity for the destination to process the provided log message. The COM definition of the interface has a BSTR parameter rather than a `CamcLogMessage*`. This `BSTR` represents the serialised form of the log message and the original message can be created using the following code fragment.

```
CamcLogMessage LogMessage;
LogMessage.InitialiseFromString( pData );
```

11.2 Adding a new Message Catalogue

The following steps are required to provide a new catalogue a specific HRESULT Facility.

- Create a message catalogue project base on AMC2000\System\Logging\LogMessages\BaseMessages
- Create a subkey with an arbitrary name under the following key naming the DLL produced by the above project `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Eventlog\Application\`
- Create a subkey under the key `AIT\AMC\Logging\Facilities` . The name of this key is arbitrary. It must have two values:

  `Id` – The numerical value of the HRESULT Facility
  `Source` – The name of the subkey created above.

  This key relates the HRESULT Facility code to the application key named in the previous step.

The new message catalogue will now be used by the Eventlog destination for messages with the specified HRESULT Facility code.